PAULO ANTÔNIO VERONEZ JÚNIOR

MÉTODO DE RECUPERAÇÃO DA ARQUITETURA PARA A COMPREENSÃO DE SOFTWARES LEGADOS

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dr. Kechi Hirama

São Paulo
2011

FICHA CATALOGRÁFICA M2011C

Veronez Júnior, Paulo Antônio

Método de Recuperação da Arquitetura para a Compreensão de Softwares Legados. / P. A. Veronez Júnior – São Paulo, 2011.

120p.

Monografia (MBA em Tecnologia de Software) – Escola Politécnica da Universidade de São Paulo. Programa de Educação Continuada em Engenharia.

1. Manutenção de software 2. Compreensão de software 3. Engenharia reversa 4. Recuperação de arquitetura de software 5. Arquitetura de software I. Universidade de São Paulo. Escola Politécnica. Programa de Educação Continuada em Engenharia II.t.

DEDICATÓRIA

Dedico este trabalho a meus pais
pelo apoio incondicional, sem o qual
eu não chegaria até aqui.

AGRADECIMENTOS

A Deus por me manter firme em busca de meus objetivos.

Ao meu orientador Dr. Kechi Hirama pela orientação e dedicação oferecida.

Aos demais professores e amigos do curso de Tecnologia de Software com os quais pude aprender muito durante a realização do mesmo.

A minha namorada Alessandra e a todos que direta ou indiretamente colaboraram para a realização deste trabalho.

RESUMO

A manutenção do software legado é um processo importante para que o mesmo consiga continuar atendendo aos seus objetivos com eficácia, eficiência e qualidade durante seu ciclo de vida. A compreensão do software é uma das atividades mais importantes desse processo e é essencial para seu sucesso, evitando que as modificações comprometam a arquitetura e a qualidade do software.

No entanto, nem sempre o software que receberá manutenção possui documentação atualizada para que seja compreendido adequadamente pelos participantes do processo de manutenção. Nesses casos, é necessária a aplicação de técnicas de engenharia reversa para que as características e comportamentos do software sejam identificados a partir dos artefatos disponíveis.

Este trabalho tem como objetivo a proposição de um método de recuperação da arquitetura de software que, através da análise do software e da construção de suas visões arquiteturais, facilite a compreensão deste pelos *stakeholders* do processo de manutenção. O presente trabalho também mostra a aplicação deste método e análise dos resultados obtidos.

PALAVRAS-CHAVE: manutenção de software, compreensão de software, engenharia reversa, recuperação de arquitetura de software, arquitetura de software.

ABSTRACT

The maintenance of legacy software is an important process to allow it to continuously attending to its aims with effectiveness, efficiency and quality during its life cycle. The software comprehension is one of the most important activities of this process and it is essential for its success avoiding that the modifications compromise the architecture and the quality of the software.

However, not always the software that will pass through a maintenance process possesses updated documentation so it must be understood adequately by the participants of the maintenance process. In these cases, the use of reverse engineering techniques is necessary to identify the characteristics and behaviors of software from the available devices.

This work proposes a method of software architecture reconstruction that eases the understanding of the software for the maintenance process stakeholders through the analysis of software and the construction of its architectural views. This work also shows the application of the method and the analysis of the obtained results.

KEYWORDS: software maintenance, software comprehension, reverse engineering, software architecture reconstruction, software architecture.

LISTA DE ILUSTRAÇÕES

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

ARMIN	Architecture Reconstruction and Mining
RSF	Rigi Standard Format
SEI	Software Engineering Institute
UML	Unified Modeling Language

SUMÁRIO

1. INTRODUÇÃO

1.1 Motivações

O padrão IEEE Std 1219-1998 (IEEE, 1998) define a manutenção de software como a modificação de um produto de software após sua entrega para corrigir defeitos, aperfeiçoar o desempenho ou outros atributos ou adaptar o produto a um ambiente modificado.

A manutenção é um processo importante no ciclo de vida do software para que o mesmo consiga continuar atendendo aos seus objetivos com eficácia, eficiência e qualidade durante seu ciclo de vida. Atualmente, a quantidade de softwares legados mantidos pelas organizações é imensa e continua crescendo, o que torna necessário a alocação de mais recursos para a atividade de manutenção desses softwares.

A manutenção de software consome a maior parte dos recursos financeiros utilizados no ciclo de vida do software (IEEE, 2004; SEACORD; PLAKOSH; LEWIS, 2003). Enquanto os projetos de desenvolvimento têm duração típica de alguns meses até poucos anos, os processos de manutenção normalmente duram vários anos, período pelo qual os softwares legados são mantidos em funcionamento. Atualmente, uma das grandes preocupações das organizações que fornecem e consomem os softwares é a diminuição dos custos empregados na manutenção do software.

Apesar da manutenção de software ser parte integrante de seu ciclo de vida, ela não tem recebido a mesma atenção de outras etapas deste ciclo. Historicamente, as organizações tem se preocupado muito mais com o desenvolvimento do software do que com sua manutenção (IEEE, 2004).

Os processos de manutenção seguidos pelas empresas são, em grande parte, adaptações dos processos de desenvolvimento. A carência de novas soluções para

os problemas da manutenção leva as organizações a enfrentar sérios problemas em seus processos de manutenção, contribuindo para o aumento do esforço empregado e do custo das atividades de manutenção.

Devido a esses fatores, muitas empresas têm optado por criar grupos especializados em manutenção de softwares legados ou terceirizar tais atividades. Na maioria desses casos, os responsáveis pela manutenção dos softwares legados não têm conhecimento suficiente para iniciar a manutenção do software de imediato. Estes precisam primeiramente compreender o software e seu funcionamento antes de iniciar as atividades de manutenção.

A compreensão do software que passará por manutenção emprega grande parte do esforço e do custo do processo. Além disso, é extremamente importante para que as modificações feitas durante o processo sejam feitas com qualidade e sem comprometer a arquitetura ou a qualidade do software. Em muitos casos, a falta ou a desatualização da documentação do software consiste em mais um grande desafio nesta fase.

A recuperação da arquitetura de software é um processo de engenharia reversa utilizado para a compreensão do software. Este processo, pode se utilizar de uma grande variedade de análises e de diversas maneiras de representação da arquitetura recuperada do software em estudo. Existe ainda um grande número de ferramentas que auxiliam os responsáveis pela manutenção no processo de recuperação da arquitetura. Escolher quais análises serão realizadas, quais ferramentas serão utilizadas e qual a maneira que a arquitetura recuperada será representada pode ser uma tarefa difícil devido a essa variedade. É necessário, portanto, estudar e avaliar tais análises e ferramentas, bem como as diversas formas de representação da arquitetura, antes de escolher quais serão utilizadas nessa tarefa, com o objetivo de escolher as que fornecem os melhores resultados para a compreensão do software.

1.2 Objetivo

Este trabalho tem como objetivo a proposição de um método de recuperação da arquitetura baseado no método ARMIN (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003). O método proposto é composto pelas análises mais adequadas à obtenção de informações e à identificação de padrões arquiteturais do software em estudo e pela melhor representação da arquitetura dos softwares legados a fim de facilitar a compreensão dos mesmos.

Através do estudo dos principais modos de representação da arquitetura e das principais análises utilizadas na recuperação da arquitetura dos softwares, serão identificadas quais fornecem os melhores resultados para as tarefas em questão. A partir dessa identificação será determinada qual a combinação de representações e análises é a mais efetiva para a compreensão dos softwares legados.

O método proposto pode ser aplicado nas atividades de compreensão dos softwares legados que necessitam de manutenção quando esta atividade for requerida.

1.3 Justificativas

A recuperação da arquitetura é um método de engenharia reversa que tem como objetivo a construção de visões de arquitetura a partir dos artefatos disponíveis de um software já construído. A recuperação da arquitetura realiza várias análises sobre esses artefatos a fim de extrair informações que serão utilizadas para a construção dessas visões e identificar os padrões arquiteturais existentes no software. Existem na literatura alguns trabalhos (BASS; CLEMENTS; KAZMAN, 2002; KAZMAN; O'BRIEN; VERHOEF, 2003; LI, 2009; LÖWE et al., 2002; SEACORD; PLAKOSH; LEWIS, 2003) que avaliam e comparam tais análises a fim de identificar as vantagens e desvantagens de cada uma delas. Contudo, esses estudos não visam identificar quais são as análises que proporcionam os melhores resultados para a recuperação da arquitetura de software com o objetivo de compreender os softwares legados.

A *Unified Modeling Language* (UML) tem sido utilizada como o padrão para descrever as características dos softwares. Há, no entanto, diversas formas de representação da arquitetura dos softwares e diversas maneiras de se documentar essa arquitetura. Existem diversas abordagens (BASS; CLEMENTS; KAZMAN, 2003; CLEMENTS et al., 2010; HOFMEISTER; NORD; SONI, 2000; KRUCHTEN, 1995; LÖWE et al., 2002; ROZANSKI; WOODS 2005; SEACORD; PLAKOSH; LEWIS, 2003) na literatura para a representação da arquitetura dos softwares, cada uma caracterizada por um conjunto distinto de visões arquiteturais. No entanto, há uma carência de estudos para determinar quais visões da arquitetura proporcionam os melhores resultados para a compreensão dos softwares legados que necessitam de manutenção.

Apesar da recuperação da arquitetura ser um processo muito utilizado para a compreensão dos softwares legados, faltam esforços no sentido de determinar quais métodos são a mais efetivos para a compreensão do software. Existem estudos (BUSS et al., 1994; O'BRIEN; STOERMER; VERHOEF, 2002) cujo foco é a comparação e a análise da eficácia de alguns métodos existentes para a recuperação da arquitetura. No entanto, ainda são necessários estudos com o objetivo de propor novos métodos ou novas combinações de análises e representações para aumentar a eficácia da recuperação da arquitetura.

A contribuição deste trabalho é a proposição de um método de recuperação da arquitetura dos softwares legados que seja mais efetivo que os atuais para a compreensão desses softwares. Esse método será composto pela representação da arquitetura do software considerada como a mais efetiva para a tarefa em questão e pelas análises do software adequadas a essa representação.

1.4 Estrutura do Trabalho

O Capítulo 1 – INTRODUÇÃO – apresenta as motivações, o objetivo, as justificativas e a estrutura do trabalho.

O Capítulo 2 – REVISÃO BIBLIOGRÁFICA – apresenta os conceitos básicos da manutenção de software e da atividade de compreensão do software como parte desse processo. Apresenta ainda uma breve introdução sobre a arquitetura de software, com destaque para as representações das arquiteturas de software. Por fim, apresenta o método de recuperação da arquitetura de software que será utilizado como base do método a ser proposto.

O Capítulo 3 – AVALIAÇÃO E SELEÇÃO DAS ANÁLISES E REPRESENTAÇÕES DA ARQUITETURA DO SOFTWARE – apresenta um estudo das principais visões utilizadas para a representação da arquitetura de software. Este estudo busca definir a melhor combinação de visões com o objetivo de facilitar a compreensão do software pelos responsáveis pela manutenção. Este capítulo também apresenta um estudo das principais análises realizadas sobre os softwares em estudo com o objetivo de obter um conjunto de informações que seja adequado para a construção das visões da arquitetura desejadas e de identificar os padrões arquiteturais presentes nesses softwares. Finalmente, apresenta as considerações e justificativas da escolha dessas visões e análises.

O Capítulo 4 – MÉTODO DE RECUPERAÇÃO DA ARQUITETURA PARA A COMPREENSÃO DE SOFTWARES LEGADOS – apresenta o método de recuperação da arquitetura proposto composto pela combinação da representação da arquitetura e das análises consideradas como as mais efetivas para a compreensão do software.

O Capítulo 5 – APLICAÇÃO PRÁTICA E ANÁLISE – apresenta a aplicação do método em um software e a análise dos resultados obtidos.

O Capítulo 6 – CONSIDERAÇÕES FINAIS – apresenta os resultados obtidos e as contribuições do presente trabalho. Por fim, sugere novos estudos e possíveis aplicações do método proposto neste trabalho.

2. REVISÃO BIBLIOGRÁFICA

2.1 Manutenção de Software

O desenvolvimento de software resulta em uma entrega de um produto de software que satisfaz às necessidades do usuário. Uma vez em operação, defeitos são descobertos, o ambiente operacional muda e novos requisitos surgem. Conseqüentemente, o software precisa mudar ou evoluir (IEEE, 2004).

De acordo com o padrão IEEE Std 1219-1998 (IEEE, 1998), a manutenção de software é definida como a modificação de um produto de software após sua entrega para corrigir defeitos, aperfeiçoar o desempenho ou outros atributos ou adaptar o produto a um ambiente modificado.

A manutenção é um dos cinco processos primários que podem ser executados durante o ciclo de vida do software (IEEE, 2008). É necessária para que o software consiga atender aos seus objetivos com eficácia, eficiência e qualidade durante seu ciclo de vida. A manutenção também pode ser dirigida a necessidade de atender a demanda dos usuários por atualizações e aperfeiçoamentos.

Tais necessidades estão em concordância com a primeira lei de Lehman: "Um grande programa em utilização deve submeter-se a mudanças contínuas ou torna-se, progressivamente, menos útil" (LEHMAN; BELADY, 1985 apud SEACORD; PLAKOSH; LEWIS, 2003).

Apesar de sua importância, a manutenção de software não tem recebido a mesma atenção de outras fases do ciclo de vida. Historicamente, as organizações tem se preocupado muito mais com o desenvolvimento do software do que com sua manutenção (IEEE, 2004).

Este cenário está mudando à medida que as organizações se esforçam para conter seus investimentos em desenvolvimento, mantendo as operações dos softwares

existentes pelo máximo de tempo possível. Atualmente, a quantidade de softwares legados mantidos pelas organizações é imensa e continua crescendo, o que torna necessário a alocação de mais recursos para a atividade de manutenção desses softwares.

Outro fator que contribuiu para a mudança deste cenário foi o paradigma de código aberto, que trouxe atenção adicional ao problema da manutenção, no sentido de manter artefatos de software desenvolvidos por terceiros (IEEE, 2004).

A manutenção de software consome a maior parte dos recursos financeiros utilizados no ciclo de vida do software (IEEE, 2004; SEACORD; PLAKOSH; LEWIS, 2003). Enquanto os processos de desenvolvimento têm duração típica de alguns meses até poucos anos, os processos de manutenção normalmente duram vários anos, período pelo qual os softwares legados são mantidos em funcionamento.

A manutenção é aplicável a softwares desenvolvidos em qualquer modelo de desenvolvimento. As atividades da manutenção de software começam com a instanciação do processo de manutenção, que pode ocorrer antes da entrega do software, e termina com a retirada do software de operação. O objetivo deste processo é modificar o software mantendo sua integridade.

O software pode mudar devido a ações corretivas e não-corretivas. A manutenção pode ser realizada para corrigir defeitos; melhorar o desempenho, a manutenibilidade ou outros atributos de qualidade; proporcionar interface com outros sistemas; adaptar o software para que diferentes tipos de hardware, software, sistemas e equipamentos de telecomunicação possam ser utilizados; entre outros motivos.

O padrão ISO/IEC 14764 – IEEE Std 14764-2006 (IEEE, 2006), define quatro categorias de manutenção:

1. Manutenção corretiva: modificação reativa de um produto de software realizada após a entrega para corrigir defeitos descobertos.

2. Manutenção preventiva: modificação de um produto de software após a entrega para detectar e corrigir defeitos ocultos antes que eles se manifestem como erros de operação.
3. Manutenção adaptativa: modificação de um produto de software realizada após a entrega para mantê-lo utilizável um ambiente modificado ou em modificação.
4. Manutenção perfectiva: modificação de um produto de software após a entrega para melhorar seu desempenho ou a facilidade de manutenção.

Independentemente da categoria, o processo de manutenção inclui as seguintes fases (IEEE, 1998):

- Planejamento da manutenção
- Identificação do problema ou modificação
- Análise
- Projeto
- Implementação
- Teste de regressão ou de sistema
- Teste de aceitação
- Entrega

A manutenção de software apresenta desafios técnicos e gerenciais únicos para os seus responsáveis. Entre estes desafios estão a manutenibilidade, a análise de impacto e a compreensão do software que receberá manutenção (IEEE, 2004). No entanto, muitos dos processos de manutenção adotados pelas organizações são adaptações dos processos de desenvolvimento. Tal fato faz com que as organizações enfrentem sérios problemas, contribuindo para o aumento do esforço empregado e do custo do processo.

A manutenibilidade, que é uma das características de qualidade de software, é a facilidade que com que o software pode ser mantido, aperfeiçoado, adaptado ou corrigido para satisfazer os requisitos especificados. A manutenibilidade é difícil de conseguir porque, normalmente, suas características não são levadas em conta

durante o processo de desenvolvimento e durante o próprio processo de manutenção. Este fato normalmente resulta em softwares complexos e com documentação incompleta, desatualizada ou até inexistente, que é a principal causa das dificuldades da análise de impacto e da compreensão do software.

O fato da complexidade do software aumentar durante a manutenção está de acordo com a segunda lei de Lehman que diz que quando um programa é continuamente modificado, sua complexidade, que reflete a deterioração de sua estrutura, aumenta a menos que algum esforço seja feito para mantê-la ou reduzi-la (LEHMAN; BELADY, 1985 apud SEACORD; PLAKOSH; LEWIS, 2003).

A análise de impacto é a análise completa das conseqüências de uma determinada modificação no software existente. A análise de impacto identifica os sistemas e produtos de software afetados por uma requisição de modificação de software e faz uma estimativa dos recursos necessários para concluir a modificação. Os responsáveis pela manutenção precisam ter um profundo conhecimento da estrutura do software e de suas características para realizar a análise de impacto (IEEE, 2004).

A compreensão do software é a atividade de entendimento do software por parte dos responsáveis pela manutenção. Os responsáveis pela manutenção precisam de um tempo considerável para ler e entender os softwares antes de iniciar o processo de modificação. Pesquisas indicam que 40% a 60% do esforço da manutenção é empregado na compreensão do software que receberá manutenção (IEEE, 2004).

Esta atividade é extremamente importante para que as modificações feitas durante o processo sejam feitas da forma correta e sem comprometer a arquitetura ou a qualidade do software. Também é necessária para produzir e fornecer documentação adequada para as atividades seguintes do processo de manutenção e fornece as bases para uma análise de impacto adequada.

Durante a compreensão do software, os responsáveis pela manutenção devem obter dos desenvolvedores o conhecimento necessário para as atividades de manutenção.

Em muitos casos, entretanto, o contato com os desenvolvedores é restrito ou não é possível, o que cria um grande desafio para os responsáveis pela manutenção (IEEE, 2004). Grupos especializados em manutenção ou empresas terceirizadas normalmente não possuem um amplo contato com os desenvolvedores e apóiam a compreensão do software na aplicação de técnicas de engenharia reversa.

A documentação do software, quando clara e concisa, também auxilia os responsáveis pela manutenção na compreensão do software. Entretanto, muitas vezes não há documentação disponível, ou ainda, esta é de baixa qualidade ou está desatualizada após atividades anteriores de manutenção. Nesses casos, a documentação necessita ser criada ou atualizada durante o processo de compreensão do software para que possa auxiliar nas fases seguintes do processo de manutenção.

Na realidade, a única fonte de informações confiável a respeito do software é o código fonte, ou em alguns casos, os arquivos binários (CANFORA; DI PENTA, 2008).

De acordo com o padrão IEEE Std 1219-1998 (IEEE, 1998), a engenharia reversa é definida como o processo de extração de informações – incluindo a documentação – do sistema de software a partir do código fonte.

A compreensão do software utiliza técnicas de engenharia reversa para construir representações do software em vários níveis de abstração, de acordo com os objetivos da compreensão, desde representações em nível de código até o nível do domínio da aplicação. O conhecimento de cada nível de abstração tem como base o conhecimento obtido em níveis inferiores de abstração (SEACORD; PLAKOSH; LEWIS, 2003).

A engenharia reversa é passiva, ou seja, não modifica o software nem resulta em um novo software (IEEE, 2004; CANFORA; DI PENTA, 2008). Seu objetivo é produzir uma visão do software em diferentes níveis de abstração, seja para auxiliar

na compreensão do software e a para criação ou atualização da documentação do mesmo.

As técnicas de engenharia reversa utilizadas na compreensão do software baseiam-se na análise do software em diversos níveis de abstração como representações textuais, textos pré-processados, árvores de sintaxe, gráficos de fluxo de dados ou controle, descrições arquiteturais, modelos conceituais, entre outros.

Entre as análises utilizadas para a compreensão de software estão a análise manual; a análise estática; a análise dinâmica; o particionamento (*slicing*) do software; a identificação de padrões comportamentais, semânticos e estruturais; a identificação de estruturas; a identificação de abstrações; a agregação de hierarquias e a localização de conceitos. Estas análises são detalhadas e estudadas no capítulo 3 – AVALIAÇÃO E SELEÇÃO DAS ANÁLISES E REPRESENTAÇÕES DA ARQUITETURA DO SOFTWARE – a fim de identificar quais são mais efetivas para a compreensão do software.

2.2 Arquitetura de Software

Segundo Bass; Clements e Kazman (2003), a arquitetura de um software ou sistema computacional é a estrutura ou as estruturas do sistema, que compreendem os elementos do software, as propriedades externas visíveis desses elementos e os relacionamentos entre eles.

A arquitetura é uma abstração do software que define quais são seus elementos e contém informações sobre como esses elementos se relacionam entre si e com o ambiente externo ao software, omitindo as informações que não são relevantes para esses relacionamentos. A arquitetura apresenta ainda os comportamentos destes elementos visíveis sob o ponto de vista externo ao software ou de outros elementos do software, a organização dos elementos dentro do sistema e a distribuição desses elementos em possíveis módulos ou subsistemas. A arquitetura ainda está relacionada com outras características do software como funcionalidade, desempenho, reuso, entre outras.

A arquitetura de software constitui um modelo compreensível de como o software está estruturado a fim de alcançar seus objetivos e de como seus elementos interagem de forma a alcançar esses objetivos. Este modelo pode ser transferido para outros softwares para que estes possuam funções e atributos de qualidade similares, promovendo o reuso dessa arquitetura ou de elementos ou estruturas dessa arquitetura.

A arquitetura de um software é uma importante ferramenta de comunicação entre os indivíduos interessados no software. Portanto, a arquitetura deve ser suficientemente abstrata para ser facilmente compreensível por indivíduos com habilidades técnicas diferentes, mesmo para os grandes sistemas de software. Conseqüentemente, a arquitetura pode ser, e freqüentemente é, utilizada como uma introdução do software para novos membros do projeto.

Ao mesmo tempo, a arquitetura deve ser uma representação consistente e completa do software para ser utilizada como modelo de análise do projeto e como base para negociação, consenso e comunicação entre os interessados no software. Além disso, a arquitetura pode ser utilizada como instrumento de verificação de conformidade da implementação do sistema ou das modificações decorrentes da manutenção do mesmo. Essas atividades são realizadas por meio do processo de recuperação da arquitetura de software.

A arquitetura também é importante pelo fato de representar as primeiras decisões sobre o projeto do software. Essas decisões são as mais difíceis de serem tomadas e as mais difíceis de serem modificadas em estágios posteriores do desenvolvimento. Também é importante citar que os efeitos dessas decisões são perceptíveis durante todo o ciclo de vida do software.

A definição inicial da arquitetura geralmente ocorre após a revisão e a validação dos requisitos do software, permitindo que a arquitetura forneça as bases para o gerenciamento do projeto. A partir da definição da arquitetura é possível o planejamento das fases seguintes do desenvolvimento, a divisão da execução do

trabalho entre grupos distintos e a obtenção de estimativas de custo e de cronograma mais acuradas.

Apesar de sua grande importância, uma boa arquitetura de software sozinha não garante funcionalidade ou qualidade ao software. Decisões em todos os estágios do ciclo de vida afetam a qualidade do software. Portanto, uma arquitetura é necessária, mas não suficiente para assegurar a qualidade de um software.

Para que a arquitetura cumpra suas funções durante o ciclo de vida do software, esta deve ser documentada de forma que os interessados no software consigam compreendê-la e utilizá-la efetivamente em suas atividades. Para que estes objetivos sejam alcançados, a arquitetura deve ser escrita sob o ponto de vista de seus futuros usuários e não de seus criadores. Identificar quem serão os usuários dessa arquitetura e a forma com que eles a utilizarão é essencial para que o arquiteto consiga transmitir as características do software de forma efetiva para os usuários da arquitetura.

A documentação da arquitetura é a principal maneira para apresentar e prover uma introdução sobre o software para novos desenvolvedores, patrocinadores e outros que necessitem ter uma visão geral sobre o software (BASS; CLEMENTS; KAZMAN, 2003).

Apesar das muitas inovações na área da arquitetura de software, ainda não há um consenso sobre a melhor maneira de descrever a arquitetura de software (IEEE, 2000). A UML, no entanto, tem se tornado uma notação padrão para documentar as arquiteturas de software proporcionando resultados satisfatórios. A maior contribuição da UML é a apresentação das visões primárias do software (BASS; CLEMENTS; KAZMAN, 2003).

Os sistemas de software modernos são muito complexos para que sejam compreensíveis como um todo. Por isso, é comum a representação de suas estruturas separadamente. Uma visão é uma representação coerente do software

sob um ponto de vista definido, que apresenta as características relevantes e omite as não relevantes para este ponto de vista.

As visões são modelos do software com a função de detalhar determinados aspectos do software como a organização lógica, a distribuição das funcionalidades, a distribuição física do software na plataforma a ser utilizada, entre outros aspectos. Cada visão é dependente de um ponto de vista que define as convenções sob as quais a visão é criada. Embora sejam diferentes, as visões de um software não são independentes, pois elementos de uma visão farão parte de outras visões sendo necessário descrever as relações entre as elas.

Não há ainda um consenso sobre quais visões da arquitetura devem ser documentadas. Da mesma forma que algumas visões só se aplicam a alguns tipos de sistemas, os usuários da arquitetura podem estar interessados em apenas algumas dessas visões para a realização de suas atividades.

Os pontos de vista e, conseqüentemente, as visões que serão documentadas devem ser selecionadas de acordo com os futuros usuários dessa arquitetura (IEEE, 2000).

Em 1995, Philippe Kruchten (KRUCHTEN, 1995) publicou um artigo muito influente no qual descreveu o conceito da arquitetura compreendendo estruturas separadas. O autor recomendou a concentração em quatro dessas estruturas: lógica, de processo, de desenvolvimento e física. Kruchten recomendou o uso de casos de uso para verificar se as estruturas não são conflitantes entre si e que seu conjunto de fato representa um software que atende a seus requisitos (BASS; CLEMENTS; KAZMAN, 2003).

Escolher um conjunto de visões úteis para os futuros usuários é uma das maiores responsabilidades do arquiteto. Um estudo sobre as visões da arquitetura e suas representações é feito no capítulo 3 – AVALIAÇÃO E SELEÇÃO DAS ANÁLISES E REPRESENTAÇÕES DA ARQUITETURA DO SOFTWARE.

Na próxima seção, é feita uma apresentação do processo de recuperação da arquitetura de software que será a base do método a ser proposto neste trabalho.

2.3 Recuperação da Arquitetura de Software

A recuperação da arquitetura é o processo pelo qual a arquitetura de um software existente é obtida a partir da análise detalhada dos artefatos deste software com o auxílio de ferramentas de apoio. Este processo resulta em um modelo arquitetural do software composto por diversas visões de sua arquitetura.

A recuperação da arquitetura é um processo interpretativo, interativo e iterativo; não um processo automático. A análise do software legado é feita com o auxílio de ferramentas que extraem as informações do software e auxiliam na criação representações do software em sucessíveis níveis de abstração. O resultado deste processo são representações arquiteturais do software em estudo. Em alguns casos, no entanto, a criação de uma representação arquitetural não é possível devido à ausência de estruturas arquiteturais coerentes no software em estudo (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003). Essa ausência pode ser causada por más práticas de desenvolvimento ou de manutenção, que resultam em um software com estruturas arquiteturais corrompidas ou mesmo sem tais estruturas. No entanto, mesmo nesses casos extremos, a recuperação da arquitetura fornece uma idéia de como o software está estruturado.

As representações arquiteturais obtidas podem ser utilizadas de diversas maneiras. Se não existe documentação do software ou esta está desatualizada, a representação arquitetural obtida pode ser utilizada como a base para a criação ou atualização da documentação da arquitetura deste software (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003). Por este motivo, a recuperação da arquitetura está intimamente relacionada com a compreensão do software que consiste no entendimento de seus artefatos em seus diversos níveis de abstração (SEACORD; PLAKOSH; LEWIS, 2003).

As representações arquiteturais podem ser utilizadas ainda como ponto de partida para a reengenharia do sistema, como meio de se identificar componentes para reuso ou para estabelecer uma linha de produto de software baseada na arquitetura (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003). Também pode ser utilizada durante o processo de desenvolvimento do software com a finalidade de verificar a conformidade da implementação com o projeto do software. A verificação de conformidade também pode ser feita durante o processo de manutenção do software a fim de se certificar que as modificações realizadas no software respeitam o projeto do mesmo.

A recuperação da arquitetura de software é um processo de engenharia reversa, portanto é passiva, ou seja, não modifica o software em estudo e não resulta em um novo software (IEEE, 2004). O processo envolve a compreensão dos artefatos e das informações extraídas em seus diferentes níveis de abstração. O conhecimento obtido dos níveis mais baixos consiste nas bases para a formação e compreensão dos níveis mais altos de abstração. O processo de recuperação da arquitetura termina quando um modelo suficientemente acurado do software em estudo é construído (SEACORD; PLAKOSH; LEWIS, 2003).

Devido à complexidade do processo e da variedade de atividades e habilidades exigidas, é recomendado que o processo de recuperação da arquitetura conte com o suporte de engenheiros de software familiarizados com técnicas de construção de compiladores e de mineração de dados (SEACORD; PLAKOSH; LEWIS, 2003). É importante também contar com o suporte de alguém com conhecimento substanciais sobre o software para auxiliar na formulação e validação das hipóteses sobre a arquitetura a ser recuperada. Também é útil que se definam os objetivos da recuperação da arquitetura a fim de eles guiem suas atividades evitando extração de informações e construção de visões desnecessárias.

Devido ao tamanho e complexidade dos softwares atuais, é praticamente impossível realizar as atividades do processo manualmente, portanto, o auxílio de ferramentas se torna essencial. Devido à diversidade de linguagens de programação nas quais os softwares são implementados, não há uma ferramenta universalmente aplicável

às atividades de recuperação da arquitetura. Normalmente, um conjunto de ferramentas é necessário para a realização das atividades do processo e, em alguns casos, é necessário adaptar ou criar ferramentas para realizar algumas atividades.

O processo de recuperação da arquitetura deve ser aberto, ou seja, permitir a utilização de novas análises e ferramentas quando necessário. O processo deve permitir que a introdução de novas ferramentas no processo seja feita sem impacto na utilização das ferramentas existentes ou nos resultados obtidos por essas (KAZMAN; O'BRIEN; VERHOEF, 2003).

Existem diversos métodos propostos na literatura para a recuperação da arquitetura de softwares legados. Esses métodos incluem a recuperação manual, auxiliada por ferramentas de suporte, baseada em linguagens de pesquisa, baseada em mineração de dados, entre outras. Uma introdução sobre esses métodos pode ser encontrada nos trabalhos de O'Brien; Stoermer e Verhoef (2002), Kazman; O'Brien e Verhoef (2003) e Guo; Atlee e Kazman (1999).

Um dos métodos que atende às exigências descritas acima é o ARMIN – *Architecture Reconstruction and Mining* – que substituiu o método Dali (KAZMAN; CARRIÈRE, 1997) de recuperação da arquitetura. O ARMIN permite que diversas ferramentas sejam utilizadas para a extração de informações do código fonte ou de outros artefatos do software em estudo, o que o torna independente de ferramentas e da linguagem de programação utilizada na implementação do software (KAZMAN; O'BRIEN; VERHOEF, 2003).

Devido às características apresentadas acima, o SEI – *Software Engineering Institute da Universidade Carnegie Mellon dos EUA* – (O'BRIEN; STOERMER, 2003) tem utilizado os métodos Dali e ARMIN como métodos para recuperação da arquitetura em uma ampla variedade de domínios e com várias finalidades, entre elas a compreensão de dependências arquiteturais para a reengenharia de software, a avaliação da conformidade da implementação de sistemas, a redocumentação de sistemas e a avaliação do potencial de criação de linhas de produção (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003).

O método de recuperação da arquitetura ARMIN é composto por quatro fases:

1. Extração de Informações: obtenção das informações dos vários artefatos do software disponíveis.
2. Construção do Banco de Dados: conversão dos dados em formato padrão e carregamento do banco de dados.
3. Fusão das Visões: combinação das informações obtidas para a geração de visões de baixo nível do software.
4. Composição de Visões Arquiteturais: construção de abstrações e representações arquiteturais do software em estudo por meio de atividades de visualização e interação e definição de *scripts* e interpretação.

Como pode ser visto na Figura 1, o método ARMIN é altamente iterativo (BASS; CLEMENTS; KAZMAN, 2003).

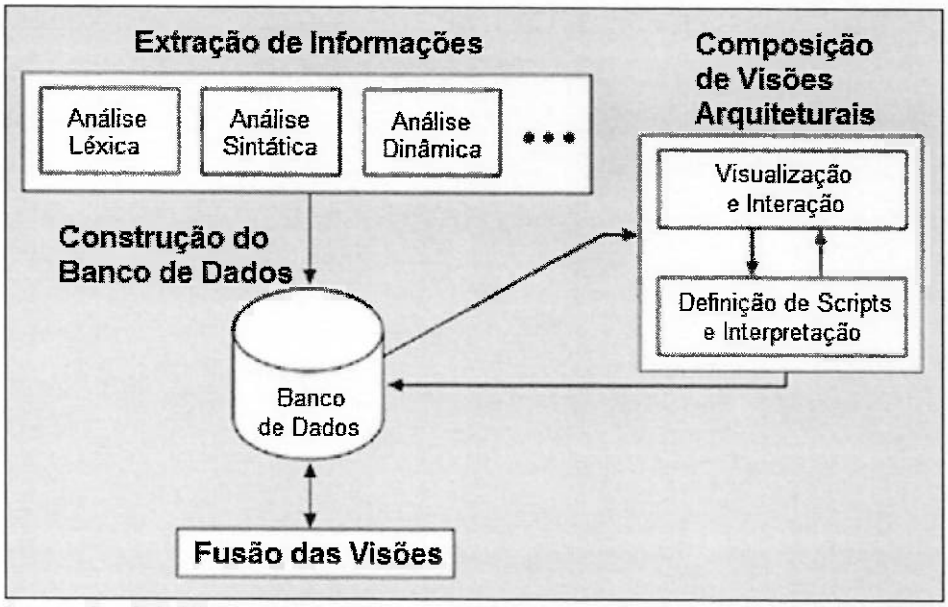

Figura 1 - Atividades de recuperação da arquitetura do método ARMIN Adaptado de Bass; Clements e Kazman (2003).

A seguir, é dada uma breve descrição das fases apresentadas, com atenção especial para as fases de Extração de Informações e Composição de Visões Arquiteturais, que são o objeto de estudo do presente trabalho.

2.3.1 Extração de Informações

A fase de Extração de Informações compreende a análise dos artefatos do software disponíveis, a extração de informações destes artefatos e a construção de visões fundamentais do software em estudo.

As informações são extraídas dos artefatos do software na forma de um conjunto de elementos e relações entre esses elementos (O'BRIEN; STOERMER, 2003). Esse conjunto de informações é carregado em um banco de dados e será utilizado na construção das visões arquiteturais do software.

Os artefatos disponíveis para extração de informações podem ser de diversos tipos e níveis de abstração bem como códigos fonte, bibliotecas, arquivos de *build*, arquivos executáveis, arquivos de *log*, entre outros.

Existem diversos tipos de ferramentas que realizam a extração de informações dos artefatos do software. A escolha das ferramentas adequadas dependerá do tipo de artefato a ser analisado e do tipo de linguagem de programação utilizada na construção do software. Entre as ferramentas mais comuns utilizadas para a extração de informações estão:

- Analisadores sintáticos (*parsers*)
- Analisadores de árvores sintáticas abstratas (*abstract syntax tree*)
- Analisadores léxicos
- Analisadores dinâmicos (*profilers*)

A Tabela 1 apresenta os elementos e as relações tipicamente extraídas dos artefatos do software em estudo.

Cada uma das relações entre os elementos constitui uma das visões fundamentais do software. O conjunto de elementos e relações a ser extraído vai depender do software a ser analisado, dos objetivos da recuperação da arquitetura e do conjunto de ferramentas utilizado.

Tabela 1 - Elementos e relações tipícos em um software
Adaptado de: Kazman; O'Brien e Verhoef (2003) e O'Brien e Stoermer (2003)

Elemento Primário	Relação	Elemento Secundário	Descrição
Diretório	has_file	Arquivo	Diretório possui arquivo armazenado
Arquivo	includes	Arquivo	Inclusão de arquivo
Arquivo	defines_fn	Função	Definição de uma função em um arquivo
Arquivo	defines_var	Variável	Definição de uma variável em um arquivo
Função	calls	Função	Chamada de função
Função	access_ [read|write]	Variável	Acesso de uma função a uma variável (leitura ou escrita)
Classe	defines_var	Variável	Definição de uma variável em uma classe
Classe	has_subclass	Classe	Classe possui subclasse
Classe	defines_fn	Função	Definição de um método em uma classe

As visões fundamentais extraídas do software podem ser classificadas como estáticas ou dinâmicas. Visões estáticas são aquelas obtidas a partir da análise de artefatos do software disponíveis antes de sua execução. Visões dinâmicas são aquelas obtidas através da análise de artefatos gerados durante a execução do software. As visões estáticas e dinâmicas são complementares, pois podem conter diferentes tipos de informações sobre o software. Informações como polimorfismo e ponteiros para funções, por exemplo, podem não existir nas visões estáticas. Visões estáticas e dinâmicas são então fundidas na fase de Fusão das Visões a fim de criar visões mais completas e acuradas do sistema.

2.3.2 Construção do Banco de Dados

A fase de Construção do Banco de Dados consiste na conversão dos diversos dados extraídos para um formato padrão e seu carregamento no banco de dados que é utilizado para armazenar as informações do software em estudo.

A maioria das ferramentas de análise e extração tem a capacidade de armazenar os dados extraídos em arquivos de texto. O conjunto de informações extraídas pelas diversas ferramentas deve ser convertido para um formato padrão e carregado no banco de dados. Um formato muito utilizado pelo ARMIN é o RSF – *Rigi Standard*

Format.. A conversão é feita por meio de *scripts* que convertem os dados extraídos pelas ferramentas para o RSF (O'BRIEN; STOERMER, 2003).

Os arquivos RSF têm o seguinte formato: *relação <elemento 1> <elemento 2>*. A Figura 2 apresenta um trecho de um arquivo RSF.

```
contains     String.BaseNodePtr.Map.cc        StringBaseNodePtrMap::clear
contains     String.BaseNodePtr.SplayMap.cc   StringBaseNodePtrSplayMap::leftmost
calls        StringBaseNodePtrSplayMap::OK    StringBaseNodePtrSplayMap::leftmost
calls        StringBaseNodePtrSplayMap::OK    StringBaseNodePtrSplayMap::succ
calls        StringBaseNodePtrSplayMap::OK    StringBaseNodePtrSplayMap::succ
calls        StringBaseNodePtrSplayMap::OK    StringBaseNodePtrMap::error
defines_var  Application::ImportMapping       Application::ImportMapping//pmMapping
defines_var  Application::DrawTree            Application::DrawTree//prend
includes     String.BaseNodePtr.Map.cc        String.BaseNodePtr.Map.h
```

Figura 2 - Trecho de um arquivo RSF
(KAZMAN; O'BRIEN; VERHOEF, 2003)

Também podem ser carregados no banco de dados os dados resultantes de pesquisas já realizadas nos dados a fim de facilitar o acesso a esses dados quando forem requeridos para futuras pesquisas (KAZMAN; O'BRIEN; VERHOEF, 2003).

2.3.3 Fusão das Visões

A fase de Fusão das Visões é a manipulação das visões fundamentais a fim de criar visões mais completas e acuradas do software. O objetivo desta fase é encontrar conexões entre as visões obtidas a fim de encontrar informações complementares sobre o software e validar as informações já obtidas.

Uma manipulação muito comum nesta fase é a fusão das visões estáticas e dinâmicas obtidas do software, já que ambas as visões podem não conter todas as informações arquiteturais do software. A fusão dessas visões proporciona maior completude e confiabilidade às informações obtidas.

A Figura 3 exibe as diferenças entre as informações das visões estáticas e dinâmicas.

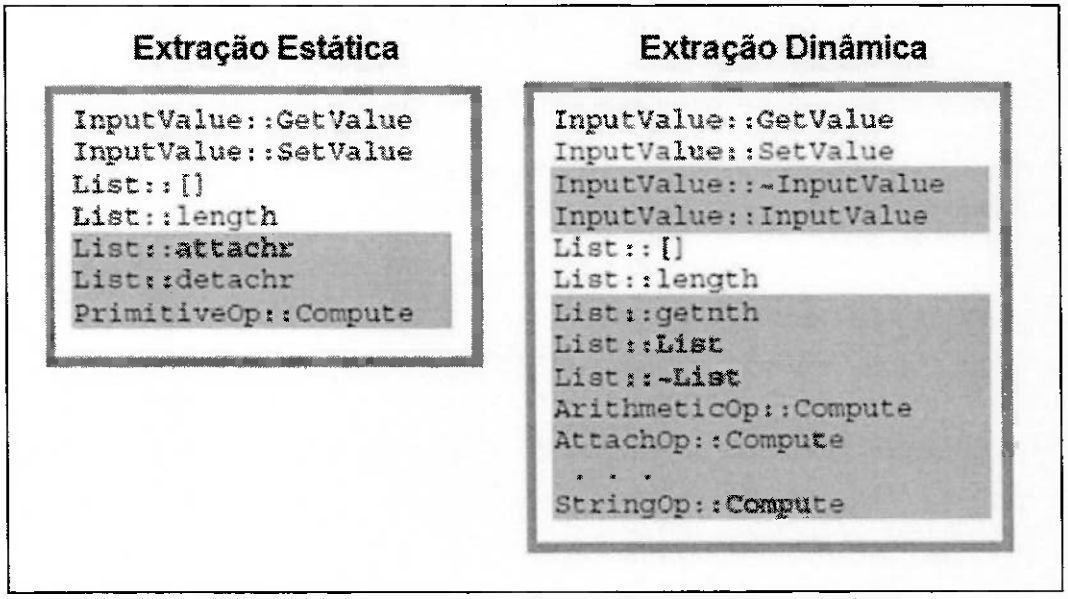

Figura 3 - Diferenças entre as visões estática e dinâmica (KAZMAN; O'BRIEN; VERHOEF, 2003)

Neste exemplo, a visão dinâmica mostra que o método *getnth* da classe *List* é chamado durante a execução do software. No entanto, este método não foi identificado na extração de informações estáticas do software. Da mesma forma, os métodos construtores e destrutores das classes *InputValue* e *List* não foram identificados pela extração de informações estáticas. Adicionalmente, a visão estática mostra que a classe *PrimitiveOp* possui um método com o nome de *Compute*. No entanto, a visão dinâmica não mostra a presença desta classe, mas mostra classes como *ArithmeticOp*, *AttachOp* e *StringOp*; todas possuidoras do método *Compute*. Por meio da comparação das visões chega-se a conclusão que as classes presentes na visão dinâmica são especializações da classe *PrimitiveOp* e que as chamadas aos seus métodos são efetuadas por meio do polimorfismo que ocorre em tempo de execução.

Este exemplo mostra que as ferramentas de extração não são perfeitas, fazendo-se necessário validar e confrontar informações extraídas de diferentes formas. As visões estáticas e dinâmicas devem ser fundidas a fim de se obter uma visão mais completa e acurada do software em estudo (KAZMAN; O'BRIEN; VERHOEF, 2003).

2.3.4 Composição de Visões Arquiteturais

A fase de Composição de Visões Arquiteturais consiste na interpretação e abstração das informações obtidas sobre o software e na formulação de hipóteses sobre sua arquitetura através da manipulação das visões existentes. Seu objetivo é encontrar e validar as visões arquiteturais do software em estudo.

A composição de visões arquiteturais consiste em duas atividades principais. A primeira é a visualização e interação, que consiste na visualização e manipulação das visões obtidas por meio de ferramentas de apoio. A segunda é a definição de *scripts* e interpretação, que visa interpretar o software e abstrair as informações de baixo nível criando visões de mais alto nível até que essas visões alcancem o nível desejado de abstração (KAZMAN; O'BRIEN; VERHOEF, 2003).

A Figura 4 mostra um *script* para agregar variáveis locais às funções nas quais são definidas.

```
#collapse the function's local_variables
$c = desc(system.types.function);
$c.merge(/ext="+");
collapse($c,/graph="FUNCTION+",/type=system.types.function);
show();
```

Figura 4 - *Script* para agregação de variáveis locais às funções (KAZMAN; O'BRIEN; VERHOEF, 2003)

O *script* tem como objetivo identificar as variáveis declaradas pelas funções e agregar essas variáveis às funções que as definem. Assim, as variáveis locais, que proporcionam pouca informação a respeito da arquitetura do software, são omitidas dos níveis mais altos da abstração. A mesma agregação poderia ser feita com classes e suas respectivas variáveis, no caso de um software orientado a objeto.

A Figura 5 ilustra a aplicação do *script* mostrado acima.

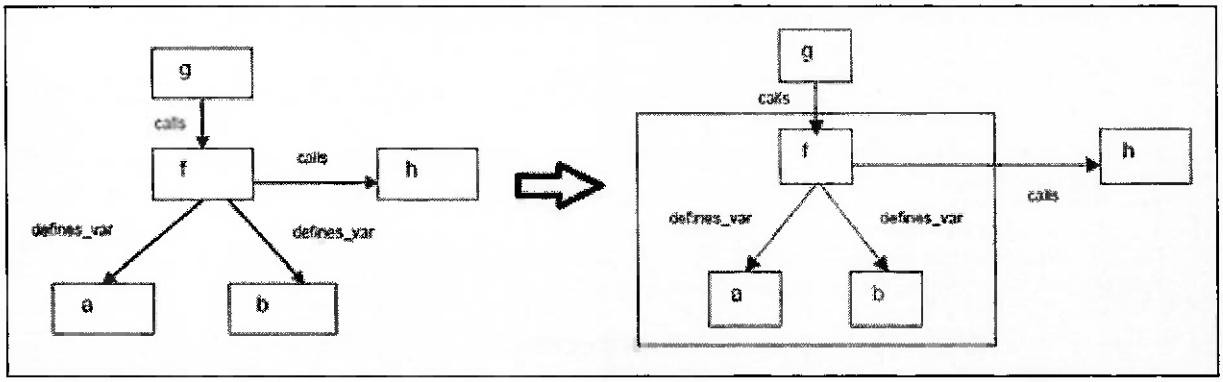

Figura 5 - Aplicação do *script* para agregação de variáveis locais às funções Adaptado de: Kazman; O'Brien e Verhoef (2003).

As estruturas arquiteturais não estão representadas explicitamente no código, tornando difícil a identificação das mesmas. Essas estruturas são implementadas por diversos mecanismos como funções, classes, arquivos, objetos, entre outros; que foram mapeados do projeto do software para os elementos de implementação. É necessário, portanto, que o responsável pela recuperação da arquitetura aplique um mapeamento inverso nesses elementos a fim de identificar as possíveis estruturas arquiteturais que eles representam (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003; SEACORD; PLAKOSH; LEWIS, 2003).

Esse mapeamento inverso envolve várias atividades para abstrair as informações obtidas dos artefatos até que se alcance o nível desejado de abstração. Entre essas atividades estão as agregações, a identificação de padrões, a análise de documentos e o contato com desenvolvedores a fim de identificar abstrações documentadas ou conhecidas pelos desenvolvedores (O'BRIEN; STOERMER, 2003).

A recuperação da arquitetura é um processo interpretativo, interativo e iterativo; não um processo automático. Este fato fica explícito nesta fase do processo onde o trabalho do responsável é analisar e interpretar as visões obtidas, construir as agregações necessárias e produzir as visões arquiteturais do software em estudo. Essas visões devem então ser validadas ou rejeitadas, podendo ainda sofrer modificações ou refinamentos (KAZMAN; O'BRIEN; VERHOEF, 2003).

A Figura 6 mostra um exemplo de visão arquitetural obtida através do processo de recuperação da arquitetura.

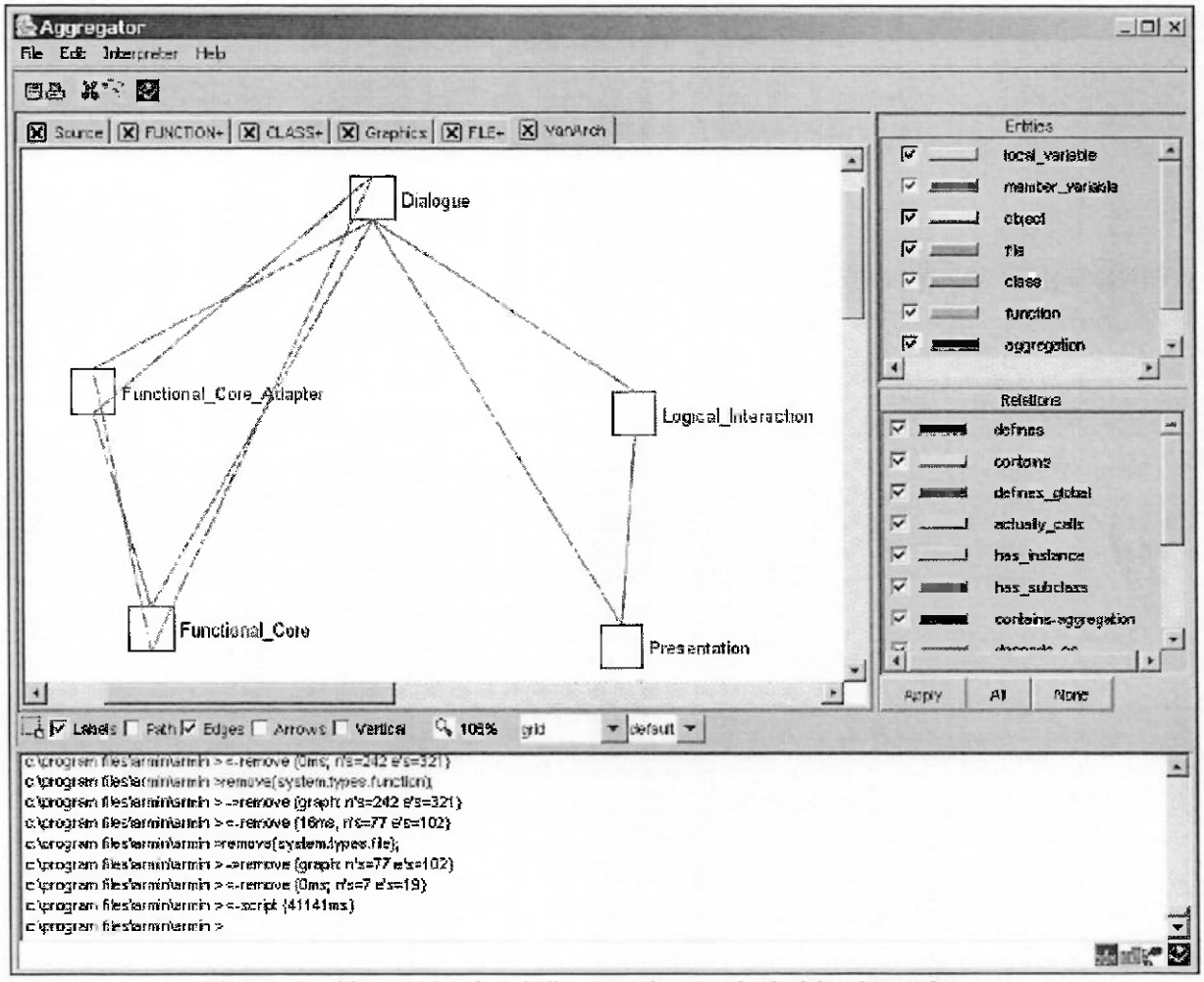

Figura 6 - Exemplo de visão arquitetural obtida do software (KAZMAN; O'BRIEN; VERHOEF, 2003)

Não há um critério para o fim do processo de recuperação da arquitetura. Este deve ser encerrado quando as representações arquiteturais forem suficientes para o suporte às necessidades de seus usuários (KAZMAN; O'BRIEN; VERHOEF, 2003).

2.4 Considerações do Capítulo

Uma vez em operação, o software precisa mudar ou evoluir para que continue a atender a seus objetivos com eficácia, eficiência e qualidade. A manutenção de

software, portanto, é essencial para que o software continue a ser útil durante seu ciclo de vida.

A manutenção de software consome a maior parte dos recursos financeiros utilizados no ciclo de vida do software e grande parte destes recursos é utilizada na atividade de compreensão do software. Esta atividade é extremamente importante para que as modificações feitas durante o processo sejam feitas da forma correta e sem comprometer a arquitetura ou a qualidade do software.

A arquitetura de software constitui um modelo compreensível de como o software está estruturado e de como seus elementos interagem a fim de alcançar seus objetivos. A arquitetura de um software é uma importante ferramenta de comunicação entre os indivíduos interessados no software sendo freqüentemente utilizada nas atividades de compreensão do software que receberá a manutenção.

A recuperação da arquitetura de software utiliza técnicas de engenharia reversa nos artefatos do software disponíveis para analisá-los e extrair deles as informações necessárias para a construção das visões arquiteturais do software. A recuperação da arquitetura se caracteriza por ser um processo interpretativo, interativo e iterativo. É realizado com o auxílio de ferramentas de apoio e deve contar com o suporte de alguém familiarizado com o software a fim de auxiliar na formulação e validação das hipóteses sobre a arquitetura a ser recuperada.

A recuperação da arquitetura de software é uma atividade essencial para que softwares sem documentação ou cuja documentação não ofereça informações confiáveis a respeito do mesmo possam ser mantidos e continuem operando conforme especificado. A recuperação da arquitetura também está intimamente relacionada com a compreensão dos softwares durante o processo de manutenção.

O processo de recuperação da arquitetura deve ser aberto, ou seja, permitir a utilização de novas ferramentas quando necessário. Dentre os diversos métodos de recuperação da arquitetura existentes, o ARMIN foi escolhido para servir como base para o método a ser proposto.

A escolha do ARMIN como base para o método a ser proposto se deve ao fato deste atender aos requisitos de adaptação a novas análises e ferramentas de extração, solucionando assim problemas comuns encontrados na recuperação da arquitetura de software como a análise de arquivos binários e a análise de código escrito em múltiplas linguagens de programação. Outro ponto positivo do ARMIN é o fato deste proporcionar grande flexibilidade para a criação das representações do software, podendo ser adaptado para a criação de um conjunto de visões que são eficazes para a compreensão do software. O fato do ARMIN e de seu antecessor, o Dali, serem adotados com sucesso pelo SEI, também contribuiu para a nossa escolha.

3. AVALIAÇÃO E SELEÇÃO DAS ANÁLISES E REPRESENTAÇÕES DA ARQUITETURA DO SOFTWARE

Como visto anteriormente, a recuperação da arquitetura é um importante processo para a compreensão dos softwares legados. Visto que os softwares atuais são muito grandes e complexos para terem sua arquitetura recuperada manualmente, é necessária a utilização de análises e ferramentas para auxiliar nas atividades da recuperação da arquitetura.

Devido a esta necessidade, a combinação das análises do software e da representação da arquitetura utilizadas na recuperação da arquitetura se torna crucial para o sucesso da compreensão dos softwares legados (LÖWE et al., 2002).

As várias análises do software e formas de representação de arquitetura existentes na literatura são estudadas neste capítulo a fim de identificar as mais efetivas para a atividade de compreensão dos softwares legados.

3.1 Representações da Arquitetura de Software

A arquitetura representa uma abstração intelectualmente compreensível do software, mesmo para softwares grandes e complexos. Por este motivo, os interessados no software a utilizam para a compreensão do software, comunicação, negociação e consenso (BASS; CLEMENTS; KAZMAN, 2003).

Existem diversas formas de se representar a arquitetura de um software, cada uma utilizando um conjunto distinto de visões. Tal fato acarreta em uma falta de consenso entre arquitetos e engenheiros de software sobre qual conjunto de visões deve ser utilizado para representar o software tanto no processo de desenvolvimento quanto no processo de recuperação da arquitetura.

Uma das práticas recomendadas para o processo de recuperação da arquitetura é a determinação dos objetivos do processo antes de iniciá-lo a fim de evitar a

realização de atividades de extração de informações e de geração de visões desnecessárias (KAZMAN; O'BRIEN; VERHOEF, 2003).

Portanto, são os objetivos do processo de recuperação da arquitetura que determinam as visões que devem ser obtidas e as informações que devem ser extraídas do software em estudo. No caso deste trabalho, o método proposto visa facilitar a compreensão dos softwares legados pelos responsáveis pela manutenção dos mesmos.

Segundo Bass; Clements e Kazman (2003), a documentação da arquitetura é a principal maneira para apresentar e prover uma introdução sobre o software para novos desenvolvedores, patrocinadores e outros que necessitem ter uma visão geral sobre o software.

Como o objetivo do processo recuperação da arquitetura proposto é facilitar a compreensão dos softwares legados pelos responsáveis pela sua manutenção, necessita-se, portanto, documentar a arquitetura do software em estudo para os responsáveis pelas atividades de manutenção desse software.

Essa documentação é direcionada aos principais participantes do processo de manutenção dos softwares legados: os desenvolvedores responsáveis pelas atividades de manutenção dos softwares legados, os responsáveis pelos testes do software, além dos gerentes e patrocinadores do projeto. Como, muitas vezes, a manutenção dos softwares legados é feita por grupos ou organizações que não têm conhecimento prévio sobre o software, é importante também direcionar a documentação para pessoas com esta característica.

Apesar das muitas inovações na área da arquitetura de software, ainda não há um consenso sobre a melhor maneira de descrever ou documentar a arquitetura de software (IEEE, 2000).

De acordo com Clements et al. (2010), documentar uma arquitetura é uma questão de documentar as visões relevantes e então adicionar a documentação que se aplica a mais de uma visão.

Uma das regras fundamentais da documentação técnica em geral, assim como da documentação de arquiteturas de software, é documentar do ponto de vista do leitor ou usuário desta documentação (BASS; CLEMENTS; KAZMAN, 2003). O documento deve servir ao seu usuário e às suas intenções, portanto é necessário identificar quais serão seus usuários, o que eles sabem e o que esperam do documento (CLEMENTS, 2010).

Outra regra importante para a documentação das arquiteturas de software é evitar a repetição de informações. Cada tipo de informação deve ser registrado apenas uma vez no documento. Visões com informações repetidas ou semelhantes podem ser combinadas ou substituídas por outras. Isto facilita o entendimento da arquitetura e evita confusões sobre as informações, pois informações repetidas tendem a ser ligeiramente diferentes.

A documentação da arquitetura deve representar o que é essencial sobre software de maneira clara e concisa. Ambiguidades devem ser evitadas a fim de que o usuário não interprete o documento de maneira inadequada. A utilização de uma notação bem definida e com uma semântica precisa é a melhor maneira de eliminar as ambiguidades da descrição da arquitetura.

Embora útil, o uso de uma notação não é obrigatório. A adoção de um conjunto de convenções e seu uso de maneira consistente e rigorosa eliminará muitas fontes de ambiguidades, desde que tenham sua simbologia explicada e detalhada. No caso da adoção de uma notação, é importante que o leitor consiga determinar o significado da notação utilizada. Informações como o nome da notação, sua versão e referências para a sua documentação são de grande importância.

Uma descrição arquitetural deve selecionar um ou mais pontos de vista para uso. A seleção desses pontos de vista deve ser baseada em quem serão os futuros

usuários da descrição arquitetural (IEEE, 2000). Portanto, as visões que serão criadas pelo método proposto neste trabalho serão selecionadas de acordo com os futuros usuários delas e seus objetivos.

O número de visões criadas para representar o software deve ser limitado para que seu conjunto possa ser compreensível. Ao mesmo tempo, essas visões devem representar todos os aspectos do software em estudo para que esse possa ser compreendido no seu todo.

O nível de granularidade das visões do software também deve ser adequado aos futuros usuários desta. Gerentes e patrocinadores do projeto necessitam apenas de uma visão geral, não sendo importante para eles a representação dos detalhes do software. Já os desenvolvedores necessitam de visões mais detalhadas, que proporcionem um maior entendimento do software, de seus componentes e de seus comportamentos.

Segundo Bass; Clements e Kazman (2003) e Clements et al. (2010), a maneira ideal para selecionar um conjunto de visões a serem documentadas é criar a Tabela de Necessidades de Informação dos *Stakeholders* do processo de desenvolvimento ou manutenção dos softwares. Essa tabela é utilizada para mostrar o nível de informação que cada *stakeholder* necessita das visões candidatas a serem documentadas. Nas colunas da tabela devem estar as visões candidatas, enquanto nas linhas, os participantes do processo, ou seja, os futuros usuários da arquitetura. A tabela deve ser preenchida com o nível de informação que os usuários necessitam de cada visão.

Ainda segundo Bass; Clements e Kazman (2003) e Clements et al. (2010), após a criação da tabela, a escolha das visões deve ser feita a partir da combinação e priorização das visões. Visões que necessitem serem criadas apenas superficialmente são candidatas a serem combinadas ou substituídas por visões que contenham essas mesmas informações e que sejam mais consistentes. Visões com informações repetidas ou semelhantes também são candidatas à combinação assim como visões com informações que sejam úteis a um ou poucos usuários.

Independentemente de quais forem as visões escolhidas, o mapeamento entre elas deve ser claro, identificando as entidades de uma visão que estão presentes em outras visões.

Nas seções seguintes são apresentadas as visões arquiteturais mais comuns com o objetivo de identificar as informações do software fornecidas por cada uma das visões e seus usos mais comuns nos processos de desenvolvimento e manutenção dos softwares. A seguir, são apresentadas as abordagens para a representação da arquitetura dos softwares adotadas pelos autores estudados na elaboração deste trabalho com a finalidade de identificar um conjunto de visões e de informações comum a essas abordagens. Também são apresentadas as necessidades dos principais participantes do processo de manutenção dos softwares que devem ser supridas pelas visões criadas pelo método proposto.

Por fim, a Tabela de Necessidades de Informação dos *Stakeholders* do processo de manutenção é preenchida levando-se em conta quais informações são necessárias para cada tipo de usuário e quais informações são fornecidas pelas visões candidatas a serem criadas pelo método. O nível de detalhamento dessas visões também é levado em conta no preenchimento da tabela.

Por fim, é feita a priorização e a combinação das visões candidatas baseadas nas informações da tabela e em um conjunto de visões considerado essencial para o entendimento dos softwares. Esse conjunto é obtido a partir do estudo das abordagens de representação da arquitetura estudadas neste capítulo.

3.1.1 Visões Arquiteturais

As visões arquiteturais mais comuns são apresentadas nesta seção de acordo com a classificação adotada por Clements et al. (2010).

3.1.1.1 Visões Modulares

As visões modulares são compostas pelos módulos do software, que são unidades de implementação que detém um conjunto definido de responsabilidades. Os principais relacionamentos entre os módulos de um software são os relacionamentos de agregação, dependência e generalização e especialização.

As visões modulares fornecem um modelo para a construção do software, auxiliam a análise de impacto e descrevem as funções do software além de fornecer suporte para o rastreamento dos requisitos, a designação de responsabilidades, a definição do cronograma e a estimativa dos custos do projeto.

3.1.1.1.1 Visão de Decomposição

A visão de decomposição é utilizada para decompor o software em unidades de implementação. A visão de decomposição descreve a organização do código em módulos e submódulos e mostra como as responsabilidades do software estão divididas entre eles.

Os elementos desta visão são os módulos e submódulos do software que se relacionam entre si por meio de relações de agregação.

Suas principais funções são a apresentação da estrutura do software para novos membros da equipe em partes, a fim de facilitar o entendimento do software; o fornecimento das bases para a designação das responsabilidades e a facilitação da localização das alterações no software.

Na UML, os pacotes podem ser utilizados para representar os módulos, que podem conter outros módulos ou classes, que são representados como caixas. Propriedades dos módulos como suas responsabilidades podem ser apresentados textualmente. Os estereótipos podem fornecer informações adicionais sobre os tipos dos módulos.

A Figura 7 mostra um exemplo de visão de decomposição.

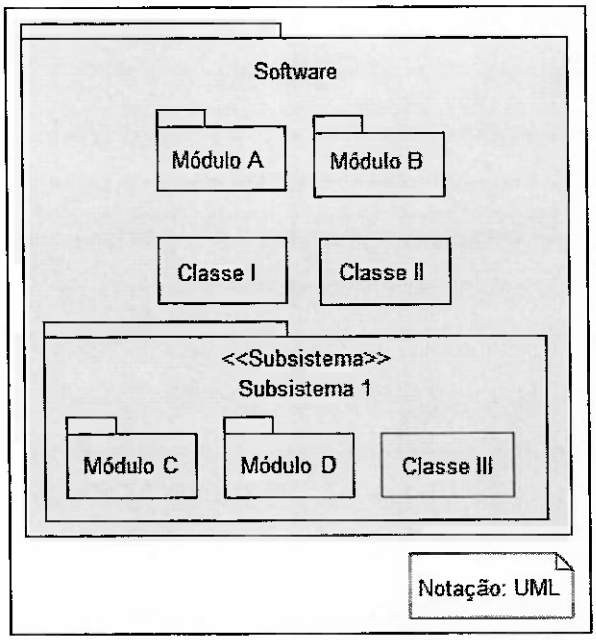

Figura 7 - Visão de decomposição
Adaptado de: Clements et al. (2010).

3.1.1.1.2 Visão de Utilização

A visão de utilização mostra como os módulos são utilizados por outros módulos, ou seja, as dependências entre os módulos.

Os elementos desta visão são os módulos do software e as relações entre eles são do tipo de dependência, ou seja, um módulo A utiliza o módulo B se A depende do correto funcionamento de B para alcançar seus objetivos.

Este tipo de visão é útil para o planejamento do desenvolvimento, pois define subconjuntos e incrementos do software. Também é utilizada para testes e análise de impacto de alterações.

Na UML, os módulos podem ser representados por pacotes enquanto as relações de dependência podem ser representadas pelo estereótipo <<usa>>.

A Figura 8 mostra um exemplo de visão de utilização.

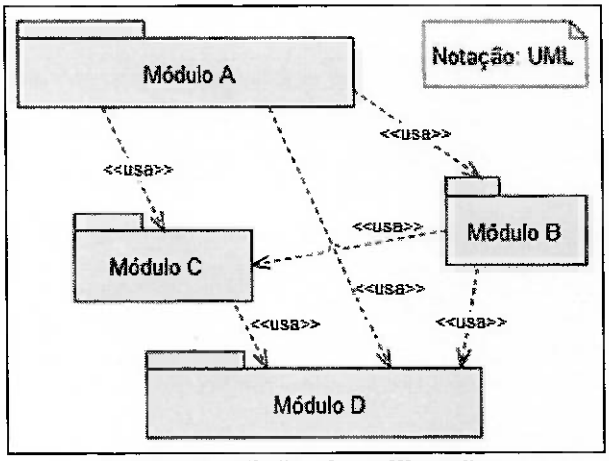

Figura 8 - Visão de utilização
Adaptado de: Clements et al. (2010).

3.1.1.1.3 Visão de Generalização

A visão de generalização emprega relacionamentos de generalização e especialização para oferecer suporte à extensão e à evolução da arquitetura e de seus elementos. Os módulos são descritos mostrando seus aspectos comuns e suas variações.

Os elementos deste tipo de visão são os módulos, que podem ser abstratos de modo a indicar que esses não são completamente implementados. Os relacionamentos de generalização e especialização podem indicar herança de classes e interfaces ou realizações de interface.

Este tipo de visão é utilizada para descrever herança em projetos orientados a objeto, descrever evolução incremental, capturar aspectos comuns e variações e oferecer suporte ao reuso.

Na UML, módulos são comumente mostrados como classes ou interfaces.

A Figura 9 mostra um exemplo de visão de generalização.

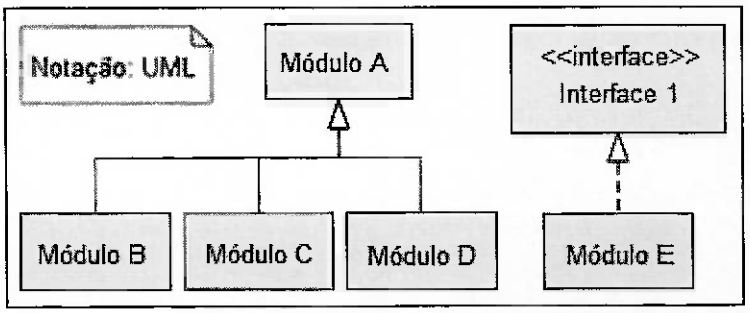

Figura 9 - Visão de generalização
Adaptado de: Clements et al. (2010).

3.1.1.1.4 Visão de Camadas

A visão de camadas agrupa módulos que oferecem um conjunto coerente de serviços e os dispõe de forma que eles tenham um relacionamento unidirecional de permissão de uso.

Os elementos deste tipo de visão são as camadas. Os módulos que fazem parte de uma camada devem estar contidos na descrição da mesma. Os relacionamentos entre as camadas são do tipo de dependência e devem estar definidas na visão. Os relacionamentos de permissão de uso não são transitivas, ou seja, se um módulo A tem permissão para usar o módulo B e este tem permissão para usar o módulo C, não significa que o módulo A tenha permissão para usar o módulo C.

Os principais usos da visão de camadas são para promover modificabilidade e portabilidade, gerenciar complexidade, promover o reuso e promover a separação de conceitos.

Na UML não há uma construção primitiva para camadas, no entanto, estas podem ser representadas por pacotes estereotipados. Os relacionamentos podem ser descritas como dependências estereotipadas entre os pacotes.

A Figura 10 mostra um exemplo de visão de camadas.

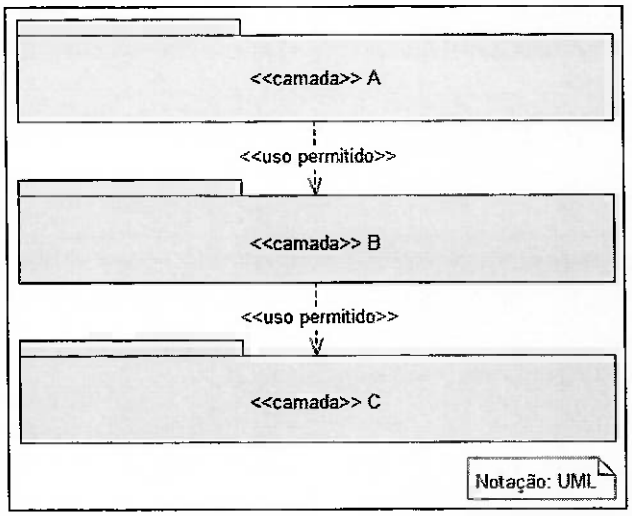

Figura 10 - Visão de camadas
Adaptado de: Clements et al. (2010).

3.1.1.1.5 Visão de aspectos

A visão de aspectos mostra módulos que implementam as responsabilidades transversais do software e descrevem como elas agem sobre o software.

Os elementos deste tipo de visão são os aspectos, que são módulos especializados que implementam as responsabilidades transversais. Os relacionamentos são as ligações entre os aspectos e os módulos afetados por estes.

Este tipo de visão é utilizada para modelar as responsabilidades transversais e aperfeiçoar a modificabilidade dos softwares.

Na UML, os aspectos são representados por classes estereotipadas e as ligações são representadas por dependências estereotipadas que ligam o aspecto a cada módulo afetado. No entanto, não é prático desenhar uma linha entre o aspecto e cada módulo afetado por esse em grandes softwares. Essa ligação pode ser omitida adicionando um comentário ao aspecto categorizando os módulos afetados por ele.

A Figura 11 mostra um exemplo de visão de aspectos.

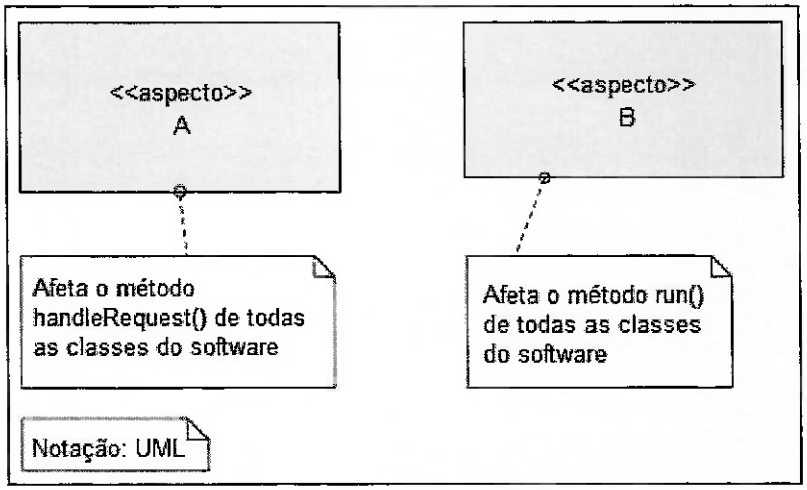

Figura 11 - Visão de aspectos
Adaptado de: Clements et al. (2010).

Na UML, os aspectos são representados por classes estereotipadas e as ligações são representadas por dependências estereotipadas que ligam o aspecto a cada módulo afetado. No entanto, não é prático desenhar uma linha entre o aspecto e cada módulo afetado por esse em grandes softwares. Essa ligação pode ser omitida adicionando um comentário ao aspecto categorizando os módulos afetados por ele.

3.1.1.1.6 Visão de Modelo de Dados

A visão de modelo de dados descreve a estrutura das entidades de dados e seus relacionamentos.

Os elementos da visão de modelo de dados são as entidades de dados que são objetos que retém as informações que necessitam ser armazenadas. Suas propriedades incluem nome, atributos de dados, chaves primárias e regras de acesso. Os relacionamentos podem ser do tipo de associação, generalização e especialização e agregação entre as entidades.

Este tipo de visão é utilizado descrever a estrutura de dados utilizada pelo software, fornecer suporte à análise de impacto nas alterações do modelo de dados, assegurar a qualidade dos dados evitando redundâncias e inconsistências e guiar a implementação dos módulos que acessam os dados.

A visão de modelo de dados pode ser representada na UML através de um diagrama de classes, onde as classes correspondem às entidades de dados. A área dos atributos é preenchida com os atributos das entidades de dados enquanto a área das operações é deixada em branco. Os relacionamentos entre as entidades de dados podem ser representadas por associações e a multiplicidade delas deve ser mostrada dos dois lados da associação.

A Figura 12 mostra um exemplo simplificado de visão de modelo de dados baseado no diagrama de classes da UML.

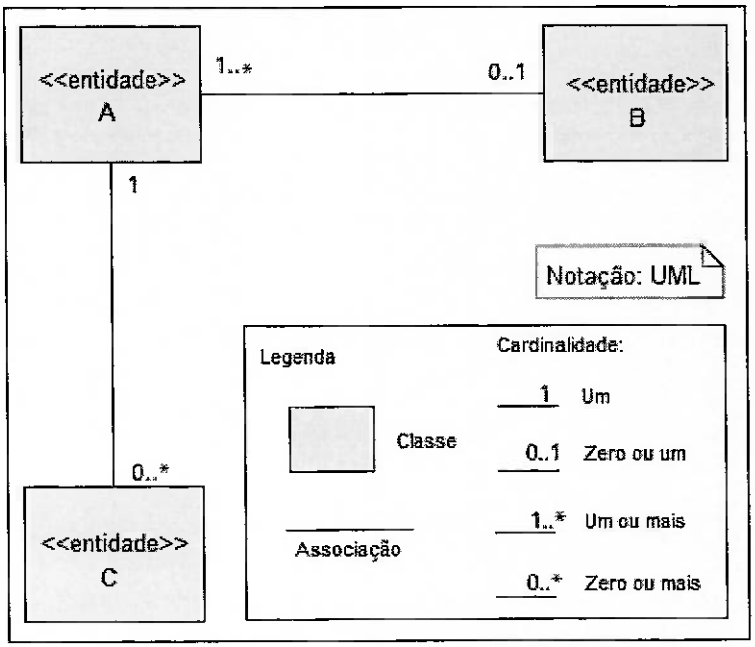

Figura 12 - Visão de modelo de dados
Adaptado de: Clements et al. (2010).

Existem outras notações bastante difundidas para a representação dos modelos de dados que podem ser utilizadas na representação deste.

3.1.1.2 Visões de Componentes e Conectores

Neste tipo de visão os componentes são as principais unidades de processamento e armazenamento de dados do software enquanto os conectores são os caminhos pelos quais os componentes interagem. Os componentes possuem um conjunto de

portas por onde ocorrem as interações com outros componentes. Os conectores, por sua vez, possuem papéis definidos, indicando o tipo de interação que pode ser realizada através deles.

Os relacionamentos neste tipo de visão são conexões. Os componentes são associados aos conectores através de suas portas para interagirem com outros componentes.

As visões de componentes e conectores têm a função de mostrar como o software é executado, guiar o desenvolvimento especificando a estrutura e o comportamento dos elementos em tempo de execução e auxiliar na compreensão de atributos de qualidade como desempenho, confiabilidade e disponibilidade.

Na UML, os componentes são o mecanismo adequado para representar os elementos do tipo componente deste tipo de visão, pois permitem a documentação intuitiva de informações como interfaces, propriedades e descrições comportamentais. Os componentes devem ser representados pelas instancias dos componentes da UML. A convenção dos nomes faz a distinção entre tipos de componentes e suas instancias: nomes que não incluem o sinal de dois pontos (:) são tipos de componentes; nomes que incluem o sinal de dois pontos são instancias de componentes, com o nome da instancia na parte da esquerda do sinal. Instancias anônimas podem ser representadas iniciando-se com o nome após o sinal de dois pontos.

As portas da UML são utilizadas para representar as portas dos componentes deste tipo de visão, podendo ainda ser adicionada a multiplicidade da conexão à porta. A UML fornece ainda a notação de encaixe (*socket*) a fim de mostrar interfaces fornecidas ou requeridas pelos componentes.

Os conectores da UML são utilizados para representar os conectores deste tipo de visão. O tipo dos conectores pode ser representado por estereótipos e eles estão sempre conectados às portas dos componentes. Informações como atributos e descrições comportamentais dos conectores não possuem um mecanismo de representação na UML, tornando-se difícil a representação dessas propriedades por meio da UML. Uma forma de representação é a colocação de etiquetas (*labels*) nas extremidades dos conectores para identificar as propriedades que são descritas em outros pontos da documentação.

A Figura 13 mostra um exemplo de visão de componentes e conectores.

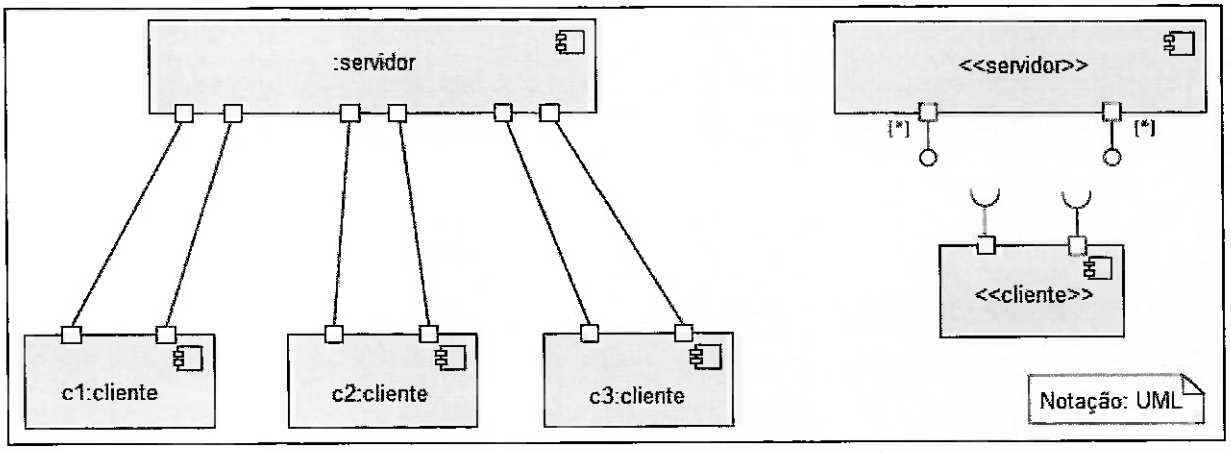

Figura 13 - Visão de componentes e conectores
Adaptado de: Clements et al. (2010).

Os principais tipos de visão de componentes e conectores são apresentados nas seções seguintes.

3.1.1.2.1 Visão de Fluxo de Dados

A visão de fluxo de dados representa um modelo computacional no qual os componentes agem como transformadores de dados e os conectores transmitem os dados da saída de um componente à entrada de outro. Os componentes possuem um número definido de entradas e saídas e sua função é consumir dados em suas portas de entradas e escrever dados transformados em suas portas de saída.

Os componentes podem agir como filtros de dados e suas propriedades podem especificar taxas de processamento, formatos de entrada e saída e transformações realizadas. Os conectores agem como transmissores dos dados de um componente a outro sem fazer qualquer transformação nesses dados e podem especificar protocolos de interação e o formato de dados transmitidos.

As conexões associam as portas de saídas dos componentes que escrevem dados com a entrada dos conectores e a saída desses com as portas de entrada dos componentes que consomem dados.

Este tipo de visão é utilizada principalmente para descrever o processamento dos dados levando-se em conta propriedades como paralelização e taxas de entrada e saídas dos dados.

3.1.1.2.2 Visão de Chamadas

Este tipo de visão representa um modelo computacional onde os componentes fornecem um conjunto de serviços que podem ser invocados por outros componentes. Um componente que invoca um serviço se mantém em estado de espera até que esse serviço seja completado. Os conectores são responsáveis por transferir a requisição de serviço do componente que o invoca ao componente que o fornece e por retornar seus resultados.

Este tipo de visão é utilizado para descrever estruturas do tipo cliente-servidor, na qual os clientes são os componentes que invocam serviços e os servidores são os componentes que fornecem serviços. Também é utilizado para descrever estruturas do tipo ponto-a-ponto (*peer-to-peer*), onde os componentes se conectam uns aos outros invocando e fornecendo serviços ao mesmo tempo de forma cooperativa. Ainda pode ser utilizada para descrever arquiteturas orientadas a serviços, onde os componentes agem de forma cooperativa fornecendo e invocando serviços em uma rede com interação coordenada.

Este tipo de visão mostra aspectos do software como segurança, escalabilidade e disponibilidade, além de ser útil para a análise de dependência.

3.1.1.2.3 Visão de Eventos

Este tipo de visão descreve um modelo computacional onde a comunicação entre os componentes é feita por meio de mensagens assíncronas. Este tipo de sistema é organizado como um conjunto de componentes com baixo acoplamento que disparam comportamentos em outros componentes através de eventos.

Um tipo comum de modelo descrito por este tipo de visão é o modelo de envio de eventos a destinatários desconhecidos, como listas de e-mail na qual os usuários se registram para receber mensagens de tópicos específicos. Este modelo é utilizado para isolar produtores de eventos de seus receptores.

3.1.1.2.4 Visão de Repositório de Dados

As visões de repositório de dados contêm um ou mais componentes que retém grandes quantidades de informação e interagem com outros componentes que lêem e escrevem dados nos repositórios.

Em muitos casos os acessos ao repositório são mediados por softwares gerenciadores de banco de dados que fornecem interfaces para a manipulação dos dados. Estas interfaces podem conter diversos tipos de serviços como suporte a transações atômicas, serviços de segurança, controle de concorrência e serviços de integridade de dados. Os componentes normalmente representam o conjunto de repositório e gerenciador de banco de dados nas visões deste tipo.

As visões de repositório de dados descrevem aspectos relacionados à modificabilidade pela decomposição do software em consumidores e fornecedores de dados.

3.1.1.3 Visões de Alocação

As visões de alocação descrevem o mapeamento da arquitetura do software com o ambiente onde esse está alocado.

Os elementos deste tipo de visão são os elementos da arquitetura do software e do ambiente. Elementos do software possuem certas propriedades que são satisfeitas pelas propriedades dos elementos do ambiente.

Os relacionamentos entre esses elementos são do tipo de alocação, ou seja, elementos do software estão alocados a elementos do ambiente.

3.1.1.3.1 Visão de Disposição

A visão de disposição descreve o mapeamento dos componentes e conectores da arquitetura do software no hardware da plataforma computacional.

Os elementos do software são os mesmos elementos das visões de componentes e conectores, ou seja, as unidades de processamento e armazenamento de dados. As propriedades desses elementos incluem características de processamento e memória. Os elementos do ambiente são os elementos do hardware, como os processadores, memórias e elementos da rede.

Os relacionamentos entre os elementos desta visão são do tipo de alocação. Os elementos do software estão alocados a uma unidade física quando residem nestas durante a execução do mesmo.

Na UML, um diagrama de disposição contém elementos conectados por associações. Os elementos correspondem às unidades de processamento e armazenamento de dados e podem conter componentes instanciados, indicando que esses componentes residem nos elementos. Os componentes podem estar conectados a outros componentes por associações de dependência e podem conter objetos, indicando que esses objetos são parte do componente. Um elemento é

representado por uma caixa tridimensional com um nome, que pode ser opcional. Os componentes são representados pelos componentes da UML. A natureza das relações pode ser descrita por estereótipos das associações.

A Figura 14 mostra um exemplo de visão de disposição.

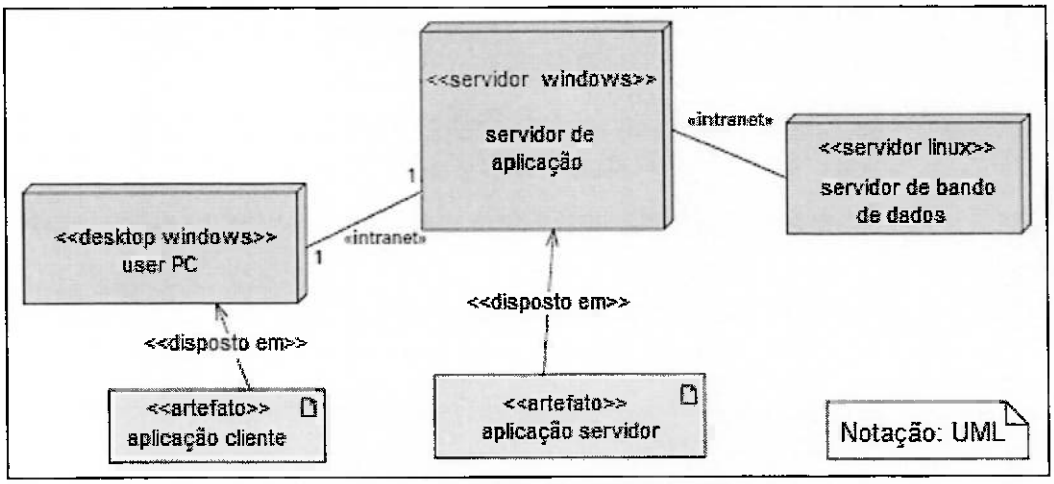

Figura 14 - Visão de disposição
Adaptado de: Clements et al. (2010).

3.1.1.3.2 Visão de Instalação

A visão de instalação descreve o mapeamento dos componentes da arquitetura do software em um sistema de arquivos no ambiente de produção.

Os elementos do software são os mesmos elementos das visões de componentes e conectores. As propriedades desses elementos normalmente são requisitos do ambiente de produção tais como suporte à Java ou permissões de acesso ao sistema de arquivos. Os elementos do ambiente normalmente são itens de configuração, como arquivos ou pastas, que fornecem propriedades requisitadas pelos elementos do software.

Os relacionamentos entre elementos desta visão são do tipo de alocação. Elementos do software são alocados em itens de configuração que satisfazem suas necessidades.

A notação utilizada para este tipo de visão precisa mostrar os componentes, os itens de configuração e o mapeamento entre eles. Estruturas do tipo árvore podem ser utilizadas para organizar os elementos. Na UML, itens de configuração e os relacionamentos de alocação podem ser representados por estereótipos enquanto os componentes são representados pelos próprios componentes da UML.

A Figura 15 mostra um exemplo de visão de instalação.

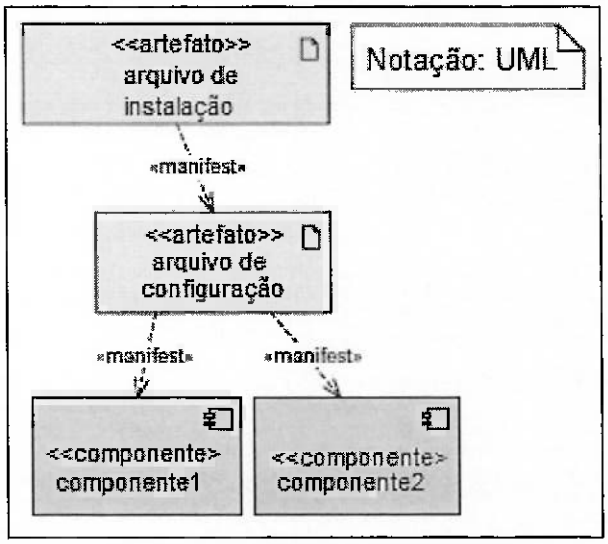

Figura 15 - Visão de instalação
Adaptado de: Clements et al. (2010).

3.1.1.3.3 Visão de Implementação

A visão de implementação mapeia os elementos da arquitetura do software nos responsáveis pelo seu desenvolvimento.

Os elementos do software são módulos que podem conter propriedades como as habilidades necessárias para o desenvolvimento do mesmo. Os elementos do ambiente são os elementos organizacionais; como pessoas, equipes ou departamentos; que são responsáveis pelo desenvolvimento do software. Suas propriedades podem incluir um conjunto de habilidades, a capacidade de trabalho e a disponibilidade de horas.

Os relacionamentos entre os elementos desta visão são do tipo de alocação. Os elementos do software estão alocados aos elementos do ambiente responsáveis pelo seu desenvolvimento.

Não existem notações para este tipo de visão. A representação tabular é uma forma simples e prática de descrever a designação de responsabilidades.

3.1.1.4 Visões Comportamentais

As visões comportamentais documentam aspectos comportamentais do software, descrevendo como os elementos da arquitetura interagem. Este tipo de visão fornece muitos benefícios tanto para o desenvolvimento quanto para a manutenção do software, pois as informações presentes nela podem ser utilizadas para obtenção de conhecimento sobre o software.

3.1.1.4.1 Diagrama de Casos de Uso

O diagrama de casos de uso mostra como os atores do software o utilizam para alcançar seus objetivos. Os casos de uso são normalmente utilizados para capturar os requisitos funcionais do software.

A UML fornece uma notação gráfica para os casos de uso, mas não indica como os textos que descrevem os casos de uso devem ser escritos. O diagrama fornecido pela UML é útil para mostrar uma visão geral do sistema, de seu comportamento e dos atores, no entanto o maior desafio neste tipo de visão é construção da representação textual.

A representação textual deve conter pelo menos o nome do caso de uso, uma pequena descrição desse, os atores que iniciam o caso de uso, outros atores que participam do caso de uso, fluxo de eventos, fluxos de eventos alternativos e casos de terminação anormal do caso de uso. Podem ainda ser aperfeiçoados com pré e pós condições, prioridades e outras informações.

A Figura 16 mostra um exemplo de diagrama de casos de uso

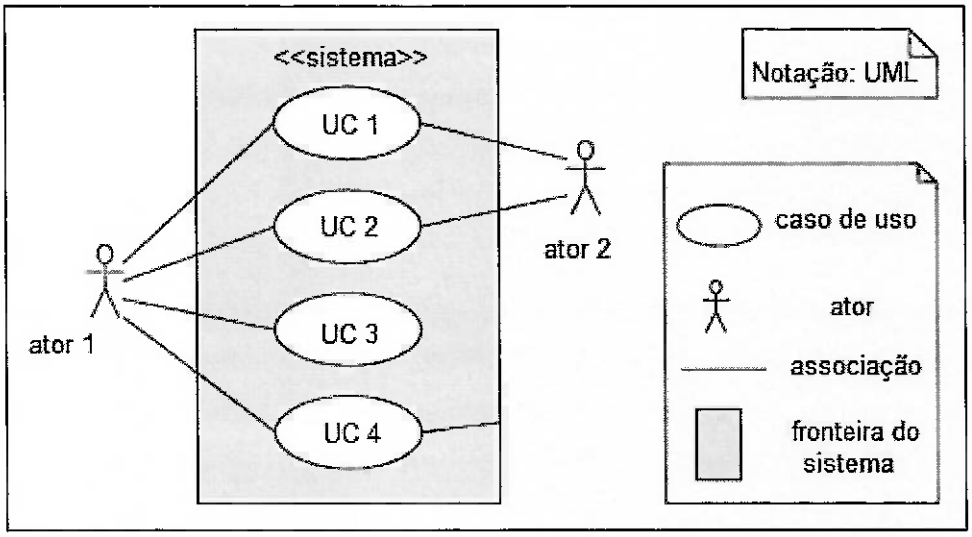

Figura 16 - Diagrama de casos de uso
Adaptado de: Clements et al. (2010).

3.1.1.4.2 Diagrama de Sequência

Um diagrama de sequência mostra uma sequência de interações entre as instancias dos elementos do software a fim de alcançarem um objetivo.

O diagrama tem duas dimensões: vertical, que representa o tempo, e horizontal, que representa as instancias que participam da interação. Apenas as instancias que participam da interação são documentadas neste tipo de diagrama. As interações são organizadas de cima para baixo pela sequência na qual ocorrem.

A notação da UML para os diagramas de sequência é, na prática, simples. As mensagens de retorno podem ser excluídas do diagrama, assim como as barras de ocorrência de execução. Um tipo único de seta também é normalmente utilizado para todos os tipos de mensagens.

A Figura 17 mostra um exemplo de diagrama de sequência.

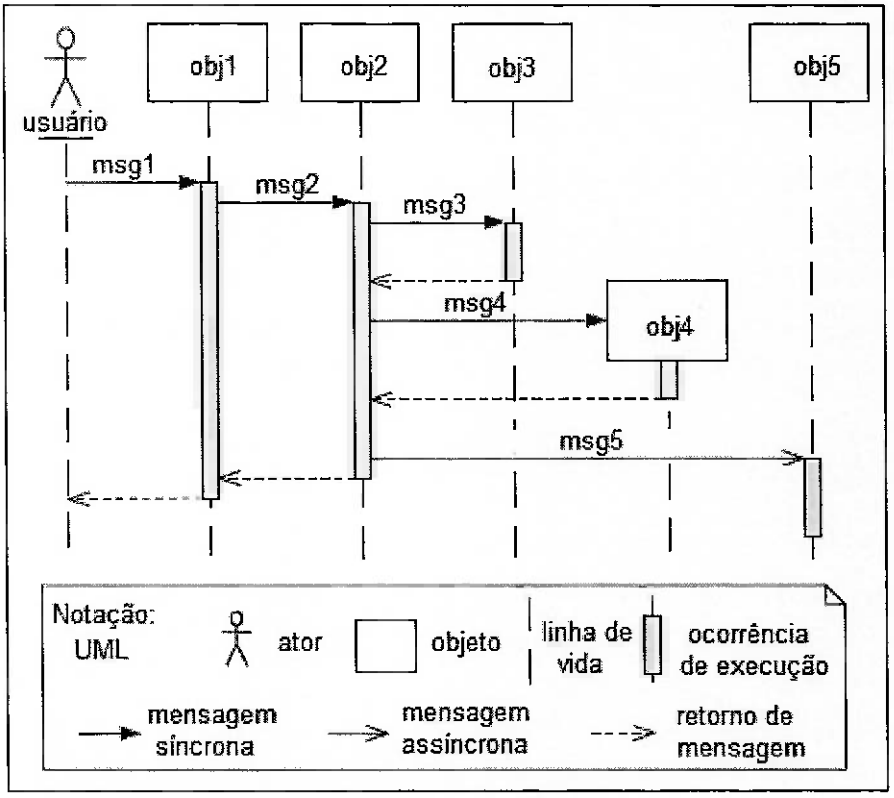

Figura 17 - Diagrama de sequência
Adaptado de: Clements et al. (2010).

3.1.1.4.3 Diagrama de Comunicação

O diagrama de comunicação mostra uma interação ordenada entre os elementos do software de modo a cumprir determinada tarefa. Enquanto os diagramas de sequência mostram a ordem das interações com base no tempo, o diagrama de comunicação mostra um gráfico dos elementos da interação com um número denotando a ordem das mesmas. Assim como nos diagramas de sequência, os diagramas de comunicação mostram apenas as instancias dos elementos que participam da interação.

São úteis para verificar se uma determinada arquitetura satisfaz os requisitos funcionais. No entanto, não são úteis para o entendimento de questões de concorrência e desempenho.

Os diagramas de comunicação e sequência expressam essencialmente o mesmo tipo de informação. Enquanto os diagramas de sequência mostram as sequências

de tempo explicitamente, tornando fácil a visualização da ordem das interações; os diagramas de comunicação indicam a ordem por números, no entanto, permitem avaliar como os elementos da interação estão estaticamente conectados, ao contrário dos diagramas de sequência.

A Figura 18 mostra um exemplo de diagrama de comunicação.

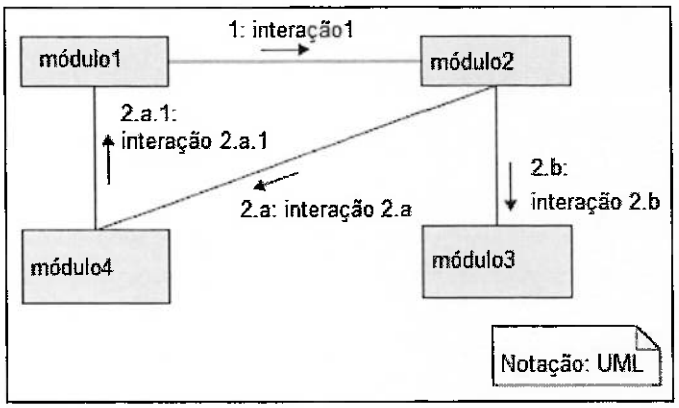

Figura 18 - Diagrama de comunicação
Adaptado de: Clements et al. (2010).

3.1.1.4.4 Diagramas de Atividade

Os diagramas de atividade mostram um processo de negócios como um sequência de passos ou ações além de possuir estruturas para expressar desvios condicionais, concorrência e envio e recebimento de eventos.

As setas entre os elementos indicam o fluxo de controle. Os elementos da arquitetura ou atores que desempenham as ações podem ser indicados pela divisão das atividades em blocos, sendo que cada bloco corresponde às atividades de um elemento ou ator.

Ao contrário dos diagramas de sequência e de comunicação, os diagramas de atividade não mostram as operações sendo executadas nos objetos específicos. A maior utilidade deste tipo de diagrama é mostrar os passos ou ações necessárias para se alcançar determinado objetivo.

A Figura 19 mostra um exemplo de diagrama de atividade.

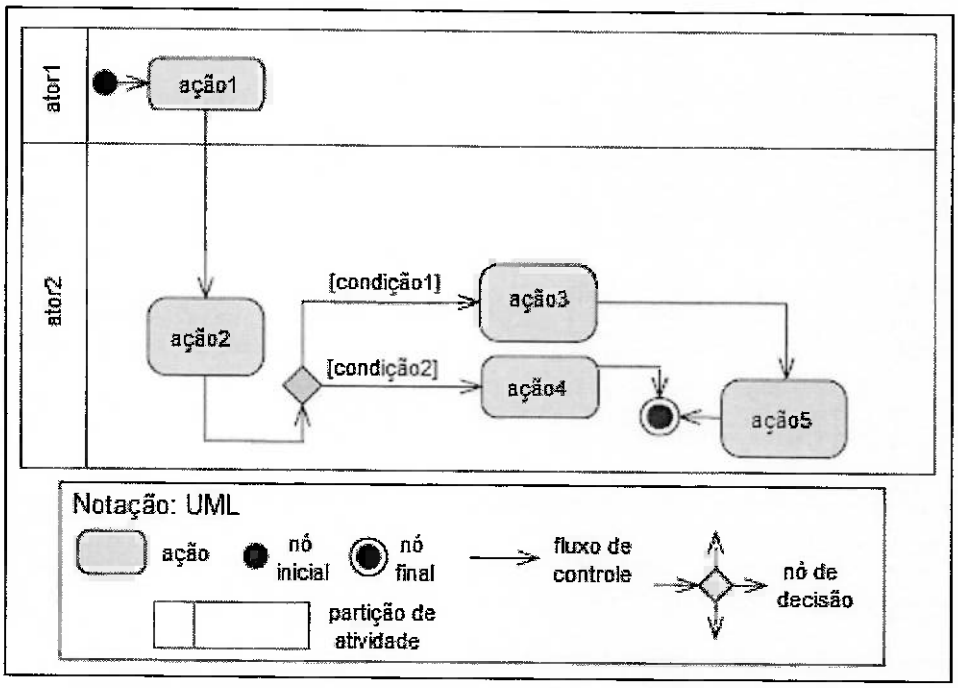

Figura 19 - Diagrama de atividade
Adaptado de: Clements et al. (2010).

3.1.1.4.5 Diagrama de Estados

O diagrama de estados representa o comportamento dos elementos da arquitetura com relação aos possíveis passos ou ações necessários para ir de um estado inicial a um estado final.

Os diagramas de estados permitem que se trace o comportamento do software de acordo com entradas específicas. Os estados são representados por caixas e as transições por setas que ligam os estados. Cada transição possui uma etiqueta que representa o evento causador da mesma.

Os diagramas de estado são utilizados para modelar elementos da arquitetura que possuem estados complexos, facilitando o projeto e o entendimento desses elementos.

A Figura 20 mostra um exemplo de diagrama de estados.

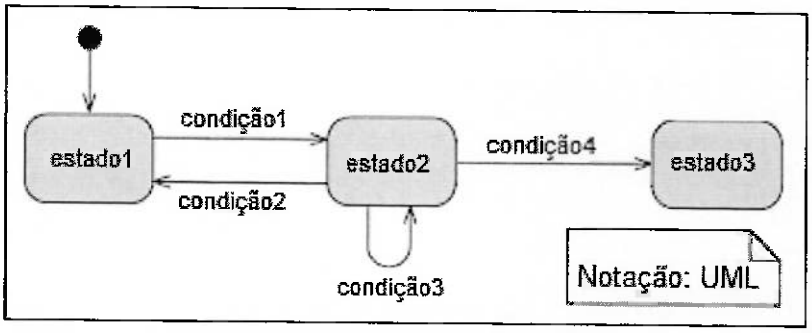

Figura 20 - Diagrama de estados
Adaptado de: Clements et al. (2010).

3.1.1.5 Diagrama de Contexto

O diagrama de contexto é utilizado para ilustrar o escopo do software no ambiente onde este está situado. Outra utilização comum deste tipo de diagrama é a ilustração do escopo de um subsistema ou mesmo um elemento da arquitetura dentro do contexto do software.

As entidades do ambiente podem ser seres humanos, outros sistemas computacionais ou objetos físicos, como sensores e controladores. No caso da representação de subsistemas, as entidades podem ser outros subsistemas que interagem com o subsistema em questão.

Diagramas de contexto são úteis para esclarecer quais partes de um sistema devem ser desenvolvidas por certa equipe ou departamento. É comum uma organização ser responsável pelo desenvolvimento de um software que fará parte de um grande sistema, e o diagrama de contexto esclarece isso.

Um diagrama de contexto normalmente apresenta uma ilustração do sistema ou subsistema, delimitando seu escopo, além de apresentar as entidades externas que interagem com esse sistema e as relações entre o sistema e essas entidades.

Na UML não existe um mecanismo explicito para a representação dos diagramas de contexto. Os diagramas de contexto podem ser construídos utilizando-se as

notações dos diferentes tipos de visão para representar os elementos adequados. Um diagrama de contexto de uma visão de componentes e conectores pode ser construído utilizando-se a notação de componentes e conectores.

Uma maneira mais genérica de construir os diagramas de contexto na UML consiste na utilização de uma combinação de diagramas de casos de uso com diagramas de classes.

A Figura 21 mostra um exemplo dessa construção.

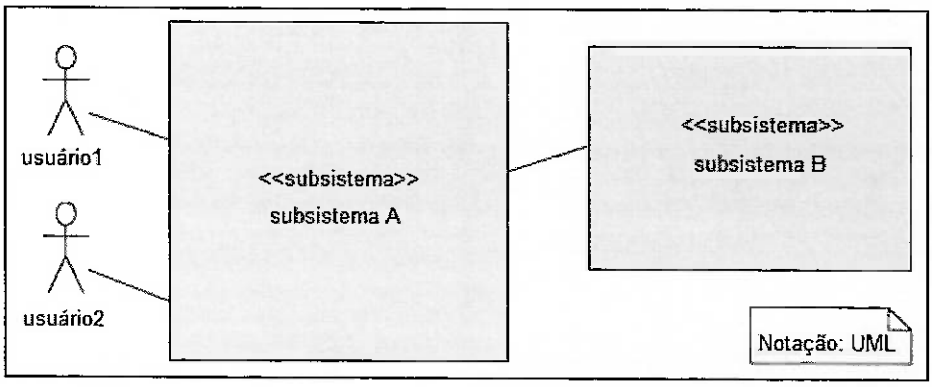

Figura 21 - Diagrama de contexto
Adaptado de: Clements et al. (2010).

3.1.2 Principais Abordagens Propostas na Literatura para a Representação da Arquitetura de Software

Existem diversas abordagens na literatura para a representação da arquitetura de software por meio de um conjunto de visões. A seguir, tem-se uma introdução sobre algumas das principais abordagens propostas na literatura.

3.1.2.1 Bass; Clements e Kazman

Segundo Bass; Clements e Kazman (2003) e Clements (2005), os arquitetos devem pensar sobre o software de três maneiras diferentes, relacionadas às unidades de implementação, às unidades de execução e à alocação de recursos de desenvolvimento e execução.

Assim, as visões se classificam como parte de três grupos básicos:

1. Estruturas modulares: os elementos desta estruturas são módulos ou unidades de implementação do software. Este tipo de estrutura destaca os módulos e sua utilização por outros módulos, as camadas do software e as classes do software com suas generalizações.
2. Estruturas de componentes e conectores: os elementos desta estrutura são componentes de tempo de execução e conectores, que são veículos de comunicação entre os componentes. Este tipo de estrutura destaca os processos de comunicação, concorrência, compartilhamento de dados, repositórios e estruturas do tipo cliente-servidor.
3. Estruturas de alocação: este tipo de estrutura destaca a relação entre os elementos do software e os elementos do ambiente onde este é executado. Este tipo de estrutura também mostra a implementação, a divisão do trabalho e a disposição dos elementos do software no ambiente.

O conjunto de visões que representará a arquitetura do software deve, portanto, representar pelo menos estes três grupos de estruturas para ser completo.

3.1.2.2 Clements et al.

A abordagem de Clements et al. (2010) é muito semelhante à abordagem proposta por Bass; Clements e Kazman (2003). Esta última utiliza as mesmas estruturas utilizadas pela abordagem anterior, no entanto propõe a documentação de uma visão geral sobre o software e de aspectos adicionais, como os aspectos comportamentais e as interfaces dos elementos do software. As visões adicionais propostas são:

- Visão geral sobre o software e o ambiente onde esse está inserido: este tipo de visão tem o objetivo de fornecer uma introdução sobre o software e o ambiente com o qual o mesmo interage, com baixo nível de detalhamento. O diagrama de contexto é normalmente utilizado para oferecer este tipo de informação.

- Visões comportamentais: tem o objetivo de fornecer informações sobre o comportamento do software durante sua execução, com destaque para as interações entre os elementos do software. Os diagramas comportamentais oferecidos pela UML podem ser utilizados para oferecer estas informações.

3.1.2.3 Hofmeister; Nord e Soni

Hofmeister; Nord e Soni (2000) definiram como base para sua abordagem para a representação da arquitetura de um software um conjunto composto por quatro visões arquiteturais, que foi definido a partir do estudo de vários softwares industriais. Este modelo ficou conhecido como o Modelo de 4 Visões da Siemens (*Siemens Four View Model*).

As quatro visões propostas são:

1. Visão Conceitual: é uma visão próxima do domínio de aplicação, pouco restrita pelas plataformas do software e do hardware. Nesta visão o software é modelado em um conjunto de componentes e conectores que pode ser dividido em várias visões com maior nível de detalhe. Aspectos como desempenho e dependências devem ser descritos nesta visão.
2. Visão Modular: nesta visão, o software é modelado utilizando pacotes e classes e suas relações.
3. Visão de Execução: esta visão descreve o sistema em função de seus aspectos de execução como processos, por exemplo. Esta visão é útil para a descrição de aspectos como desempenho, tempo de resposta, entre outros.
4. Visão de Código: esta visão descreve a organização do código fonte e dos arquivos binários e suas relações. Este tipo de visão é útil para identificar os diferentes tipos de bibliotecas utilizados na organização do software.

3.1.2.4 Kruchten

Kruchten (1995) propôs uma das mais influentes abordagens para a representação da arquitetura por meio de múltiplas visões. Sua abordagem foi chamada de Modelo de Visão 4+1 (*Four plus One View Model*) e é utilizada com base conceitual pelo Rational Unified Process (KRUCHTEN, 2003).

Segundo o autor, a representação da arquitetura de um software deve se concentrar em quatro visões: visão lógica, visão de processo, visão física e visão de desenvolvimento. A descrição arquitetural é ilustrada utilizando-se alguns casos de uso selecionados que descrevem a utilização do software. Estes casos de uso compõem a quinta visão da abordagem.

As cinco visões utilizadas por Kruchten (1995) são:

1. Lógica: a visão lógica trata dos requisitos funcionais do software. É uma abstração do modelo de projeto do software e identifica os pacotes, subsistemas e classes do software.
2. Processo: a visão de processo trata dos requisitos não funcionais do software como aspectos relacionados à desempenho, disponibilidade, concorrência, integridade e tolerância a falhas.
3. Desenvolvimento: a visão de desenvolvimento descreve a organização dos módulos estáticos do software no ambiente de desenvolvimento indicando quais desenvolvedores são responsáveis por quais módulos ou grupos de módulos.
4. Física: a visão física trata do mapeamento dos componentes do software nas plataformas onde este será executado.
5. Casos de Uso: a visão de casos de uso contém alguns cenários de utilização, considerados os mais importantes e abrangentes, com a finalidade de guiar e validar o projeto da arquitetura e descrever como as outras visões da arquitetura funcionam.

3.1.2.5 Löwe et al.

De acordo com Löwe et at. (2002), é crucial para a compreensão do software que o conjunto de visões utilizado para representá-lo possua pelo menos as visões estática e dinâmica do software, ou seja, a representação de suas estruturas e de seus comportamentos.

3.1.2.6 Rozanski e Woods

Rosanski e Woods (2005) propõem um conjunto de seis visões para ser utilizado na documentação das arquiteturas de software. Estas seis visões foram definidas como uma extensão do conjunto de visões proposto por Kruchten (1995). As visões propostas são as seguintes:

1. Visão funcional: documenta os elementos funcionais, suas responsabilidades, interfaces e interações primárias. A visão funcional é a pedra fundamental da descrição arquitetural do software e tem importante papel na avaliação dos atributos de qualidade do software.
2. Visão de informação: esta visão documenta a maneira pela qual o software manipula, armazena, gerencia e distribui as informações. A visão de informação descreve as estruturas de dados e o fluxo de informações no software.
3. Visão de concorrência: descreve aspectos relacionados à concorrência, mapeando os elementos e as partes do software que se relacionam com a execução, coordenação e controle da concorrência.
4. Visão de desenvolvimento: descreve a arquitetura que fornece suporte às atividades de desenvolvimento.
5. Visão de disposição: descreve o software e o ambiente onde esse está disposto, capturando aspectos do hardware necessários a execução do software e mapeando os elementos do software aos elementos do hardware em que estes estarão alocados durante sua execução.

6. Visão operacional: descreve como o software é executado no ambiente de produção; capturando aspectos da instalação, gerenciamento e operação deste.

3.1.2.7 Seacord; Plakosh e Lewis

Segundo Seacord; Plakosh e Lewis (2003), existem três visões básicas para a representação do software que compreendem um conjunto de subvisões que tratam de propriedades específicas do software.

De acordo com os autores, pelo menos uma visão, baseada nos módulos, deve representar a estrutura estática do software. Outra visão, baseada em componentes e conectores, deve representar o software durante sua execução. E deve haver pelo menos uma visão de disposição que mostre como os elementos do software são mapeados no hardware. Além destas três visões básicas, um diagrama de contexto pode ser utilizado para representar o software e seu ambiente.

3.1.3 Necessidades dos Interessados na Recuperação da Arquitetura

Para auxiliar na escolha do conjunto de visões a ser recuperado pelo método proposto neste trabalho, são apresentadas as necessidades dos principais interessados na arquitetura dos softwares baseadas na abordagem de Clements et al. (2010).

3.1.3.1 Desenvolvedores

Os desenvolvedores que farão a manutenção do software utilizam a arquitetura como ponto de partida para as atividades de manutenção.

Uma visão importante para os desenvolvedores é a visão de decomposição, que permite a eles identificar os locais onde serão realizadas as alterações. Outra visão importante é a visão de utilização, que permite aos desenvolvedores fazer a análise

de impacto, identificado quais elementos do software podem ser afetados pelas alterações.

Deve ser dada ênfase no detalhamento das informações das visões modulares e de componentes e conectores da arquitetura, bem como nos diagramas comportamentais.

Portanto, as visões nas quais os desenvolvedores podem estar interessados são:

- Modulares: decomposição, utilização ou camadas e generalização.
- Componentes e conectores: várias, mostrando os componentes sob responsabilidade do desenvolvedor e os componentes que interagem com esses.
- Alocação: disposição, instalação e implementação.
- Outras: diagrama de contexto e visões comportamentais dos componentes sob sua responsabilidade.

3.1.3.2 Testadores

A arquitetura especifica o comportamento das unidades e dos módulos que serão testados nos testes. Para testes do tipo caixa-preta, a arquitetura define quais são as peças a serem testadas e quais são seus relacionamentos. Para testes unitários, a arquitetura fornece as informações comportamentais dos elementos a serem testados.

A ênfase no detalhamento das informações deve estar nas visões modulares e de alocação.

As visões nas quais os testadores podem estar interessados são:

- Modulares: decomposição, utilização e de modelo de dados.
- Componentes e conectores: todas.
- Alocação: disposição, instalação e implementação.

- Outras: diagrama de contexto e visões comportamentais.

3.1.3.3 Gerentes de Projeto

Os gerentes se preocupam com questões diferentes dos desenvolvedores e testadores, como cronograma e recursos destinados. Informações sobre o comportamento geral do software bem como dos módulos que sofrerão manutenção, sua complexidade e suas dependências são úteis a este tipo de interessado. Informações específicas do projeto ou das interfaces dos elementos não são necessárias.

A ênfase no detalhamento das informações deve estar nas visões de alocação, com pouca ênfase nas demais.

De modo geral, os gerentes estão interessados nas seguintes visões:

- Modulares: decomposição e utilização ou camadas.
- Componentes e conectores: nenhuma.
- Alocação: disposição e implementação.
- Outras: diagrama de contexto.

3.1.3.4 Patrocinadores

Os patrocinadores estão interessados nos custos do projeto e em saber se o software atenderá aos requisitos funcionais e aos atributos de qualidade desejados. Além disso, os patrocinadores, muitas vezes, dão o suporte necessário aos ambientes nos quais os software são executados, portanto, desejam saber se este será interoperável com outros softwares no ambiente de execução.

A ênfase no detalhamento das informações deve estar nas visões de alocação, com pouca ênfase nas demais.

- Modulares: nenhuma.

- Componentes e conectores: uma das visões, dependendo do interesse do patrocinador.
- Alocação: disposição e implementação.
- Outras: diagrama de contexto e uma das visões comportamentais, dependendo do interesse do patrocinador.

3.1.3.5 Novos Membros do Projeto

Novos membros do projeto estão interessados em ver informações introdutórias, mostrando o software como um todo, como diagrama de contexto ou outras visões com baixo nível de detalhamento.

Normalmente, os novos membros do projeto estão interessados nas mesmas informações que as pessoas que desempenham a mesma função que a sua, no entanto, com menor detalhamento.

3.1.4 A Escolha do Conjunto de Visões

Após a apresentação das visões arquiteturais mais comuns, das abordagens propostas pelos autores estudados para a representação da arquitetura dos softwares e das necessidades dos principais participantes do processo de manutenção; é feita a escolha das visões baseada em três critérios:

1. Necessidades dos participantes do processo de manutenção: os participantes no projeto de manutenção dos softwares legados necessitam de diferentes tipos de informações a respeito do software e sua arquitetura para desempenhar suas atividades. Suas necessidades são avaliadas a fim de classificar quais as visões fornecem as informações necessárias para eles.
2. Utilidade das visões apresentadas: cada visão fornece um conjunto distinto de informações sobre o software e sua arquitetura. As informações oferecidas pelas diversas visões são analisadas a fim de determinar quais são mais úteis para as atividades de compreensão e manutenção dos softwares legados.

3. Abordagens para representação da arquitetura existentes: as abordagens apresentadas neste trabalho são analisadas a fim de identificar os aspectos e informações sobre o software cuja representação é considerada essencial para a compreensão e manutenção dos softwares legados. Também é avaliada a utilidade do conjunto de visões proposto pelos autores apresentados.

Estes critérios foram escolhidos para que as visões criadas pelo método proposto sejam direcionadas ao atendimento das necessidades dos participantes do processo de manutenção. A partir da identificação dessas necessidades, é possível identificar quais visões podem supri-las a partir das informações fornecidas por cada uma delas. As abordagens estudadas auxiliam na identificação de visões cuja representação é considerada essencial pelo fato de conter importantes informações sobre os softwares, auxiliando na priorização e combinação das visões que serão criadas pelo método proposto.

A Tabela de Necessidades de Informação dos *Stakeholders* do processo de manutenção é construída com base na tabela adotada por Clements et al. (2010). Seu preenchimento é baseado nas necessidades de informações dos participantes do processo de manutenção e nas informações oferecidas pelas visões apresentas.

Por se tratar de um método para ser aplicado em qualquer tipo de software; independente de variações relacionadas à decisões de projeto, como padrões arquiteturais e linguagens de programação; a tabela é composta por todas as visões apresentadas neste trabalho, de acordo com a classificação adotada por Clements et al. (2010). Outros tipos de documentação arquitetural não são incluídos na tabela visto que o foco do trabalho é a seleção das visões a serem criadas pelo método proposto.

Os participantes identificados como essenciais ao processo de manutenção integram a tabela, visto que as visões a serem criadas são direcionadas a eles. Os novos membros do projeto não foram adicionados à tabela, visto que seu interesse é

o mesmo que o de outros integrantes que desempenham funções semelhantes no processo.

A Tabela 2 mostra as necessidades de informação dos *stakeholders* do processo de manutenção em relação às visões da arquitetura apresentadas.

Construída a tabela, é necessário selecionar quais visões oferecem as informações necessárias, priorizá-las e combiná-las a fim de obtermos um conjunto de visões conciso e adequado para representar o software em estudo. A priorização e a combinação das visões são feitas baseadas nos dados da tabela e nas abordagens propostas pelos autores estudados neste capítulo. Para isso, é feita uma comparação entre as abordagens a fim de identificar um conjunto de visões comum utilizado pelos autores.

Segundo as abordagens de Bass; Clements e Kazman (2003), Clements (2005) e Seacord; Plakosh e Lewis (2003), a representação do software deverá ter uma visão que represente sua estrutura modular, sua estrutura de componentes e conectores e sua estrutura de alocação. Essa abordagem é muito semelhante à proposta por Clements et al. (2010) que sugere a adição de uma visão comportamental e uma visão geral sobre o software. Partindo dessas três estruturas em comum entre as quatro abordagens; modular, de componentes e conectores e de alocação; é feita uma comparação com outras abordagens estudadas.

A abordagem proposta por Kruchten (1995) também possui aspectos em comum com as citadas anteriormente. A visão lógica é equivalente a uma representação modular; identificando os pacotes, subsistemas e classes do software. A visão de processo; que trata de questões como desempenho, disponibilidade e concorrência; pode ser representada por uma das visões de comportamento, como o diagrama de sequência. A visão de casos de uso também representa o comportamento do sistema. As visões de desenvolvimento e física são equivalentes as visões de disposição e de implementação propostas por Clements et al. (2010), que são utilizadas para a representação da estrutura de alocação do software.

Tabela 2 - Necessidades de Informação dos *Stakeholders*
Adaptado de: Clements et al. (2010)

	Visões de Módulos						Visões de C&C				Visões de Alocação			Diagramas de Comportamento					Outros
Usuários	Visão de Decomposição	Visão de Utilização	Visão de Generalização	Visão de Camadas	Visão de Aspectos	Visão de Modelo de Dados	Visão de Fluxo de Dados	Visão de Chamadas	Visão de Eventos	Visão de Repositório de Dados	Visão de Disposição	Visão de Instalação	Visão de Implementação	Diagrama de Casos de Uso	Diagrama de Sequência	Diagrama de Comunicação	Diagrama de Atividade	Diagrama de Estados	Diagrama de Contexto
Desenvolvedores	D	D	D	D	D	D	D	D	D	D	G		P	P	D	D	P	P	G
Testadores	D	D	D	D	D	D	D	D	D	D	G		P	P	D	D	P	P	G
Gerentes de Projeto	P	P	P	P							G		D						G
Patrocinadores	G			G							G	G		P					G
D: Informações detalhadas / P: Poucas informações / G: Informações gerais																			

Já a abordagem de Löwe et al. (2002) apenas trata de aspectos relacionados à estrutura do software e à sua execução. Uma visão modular e uma comportamental são suficientes para representar o software de acordo com esta abordagem.

A abordagem proposta por Rosanski e Woods (2005) sugere a utilização de uma visão funcional, que é equivalente a visão de decomposição proposta por Clements et al. (2010), ou seja, uma representação modular. A visão de informação proposta pode ser representada pela visão de fluxo de dados, que representa a estrutura de componentes e conectores do software, enquanto a visão de concorrência pode ser representada por uma das visões de comportamento que tratam do aspecto concorrência. As visões de desenvolvimento, equivalente a visão de implementação, e disposição fazem parte da representação de alocação do software. Por fim, tem-se a visão operacional, que consiste na maior diferença entre essa abordagem e as outras propostas.

Nota-se que na maioria das abordagens propostas, os autores utilizam pelo menos uma visão para a representação das estruturas modulares, uma para componentes e conectores e uma para alocação, utilizadas como base para a comparação feita. Além dessas três estruturas, nota-se que a representação do comportamento do software também é uma preocupação da maioria dos autores. Baseado nas abordagens propostas, chega-se a conclusão que a representação destes quatro tipos de estruturas é fundamental para se construir uma boa representação arquitetural. Portanto, a representação criada pelo método proposto deve conter pelo menos uma visão que represente a estrutura modular, uma que represente a estrutura de componentes e conectores, uma que represente a estrutura de alocação e, por fim, uma que represente os comportamentos do software.

Esse conjunto de visões representando as estruturas do software pode ser considerado como um conjunto essencial para a representação da arquitetura do software visto que sua representação é recomendada pela maioria dos autores estudados.

Com a finalidade de diminuir o número de visões visando facilitar a compreensão do software são selecionadas as visões mais abrangentes para a representação de cada uma das estruturas da arquitetura dos softwares. Essas serão as visões prioritárias para a representação da arquitetura. As visões restantes são combinadas sempre que possível com as visões prioritárias a fim de diminuir o número de visões a serem criadas facilitando a compreensão dos softwares pelos responsáveis pela manutenção.

No conjunto das visões que representam as estruturas modulares do software, existem visões que são dependentes de padrões arquiteturais adotados e que são recuperadas apenas para os softwares construídos utilizando-se esses padrões. A visão de generalização, por exemplo, pode ser recuperada em projetos orientados a objetos, mas pode ser substituída pela visão de decomposição para outros tipos de software. O mesmo acontece com a visão de camadas que é usualmente substituída pela visão de decomposição nos software onde a estrutura de camadas não está presente.

As visões de aspectos e de modelo de dados podem, da mesma forma, ser recuperadas para softwares específicos, mas devido à sua dependência a decisões de projeto não são representativas para os softwares de maneira geral. Portanto, não são prioritárias para o método proposto.

Já a visão de utilização, cuja principal característica é a representação das dependências entre os módulos do software, pode ser substituída pela visão de chamadas já que esta também apresenta informações sobre as dependências presentes no software.

A visão de decomposição pode, portanto, ser considerada prioritária para a representação da estrutura de módulos do software, sendo eventualmente substituída pela visão de camadas ou de generalização, quando adequado. As visões de aspectos e de modelo de dados podem ser criadas quando possível, de acordo com as decisões de projeto adotadas no desenvolvimento do software em estudo.

No conjunto das visões que representam as estruturas de componentes e conectores, a visão de chamadas pode ser utilizada para descrever aspectos que possui em comum a visão de fluxo de dados não sendo necessária a criação desta última. A visão de chamadas também será utilizada para a representação das dependências entre os componentes do software, eliminando-se a necessidade da criação da visão de utilização, como já discutido anteriormente. Além disso, a visão de chamadas é uma escolha natural para representar softwares que sofrerão manutenção, pois a partir dela é possível fazer a análise de impacto das alterações do software. Portanto, a visão de chamadas é escolhida como uma das visões que serão criadas pelo método.

As visões de repositório de dados e de eventos são dependentes de decisões de projeto e são recuperadas apenas para os softwares que tenham sido desenvolvidos utilizando padrões que resultam em características descritas por essas visões. Portanto, não são prioritárias para o método proposto.

No conjunto das visões de alocação, a visão de disposição pode ser descrita por meio de uma visão de componentes e conectores, bastando para isso acrescentar na visão escolhida os elementos do hardware onde os componentes estarão dispostos. Esta será, portanto, combinada com a representação de componentes e conectores do software. Já a visão de instalação fornece poucas informações para os participantes do processo de manutenção, sendo importante apenas para fornecer informações gerais sobre o software aos patrocinadores do projeto. Esta visão não será, portanto, criada pelo método proposto. A visão de implementação, por sua vez, pode ser facilmente obtida da visão de decomposição, não sendo necessários esforços para a criação da mesma, bastando atribuir os módulos do software às equipes responsáveis pela manutenção dos mesmos.

No conjunto das visões que representam os comportamentos dos softwares, os diagramas de sequência e de comunicação são os que apresentam as informações que desenvolvedores e testadores mais necessitam saber. Ambos, entretanto, apresentam os mesmo tipos de informação, com a diferença que aspectos como concorrência e desempenho são representados apenas pelo diagrama de

sequência. Os diagramas de casos de uso, atividade e de estados, apenas são necessários para fornecer informações gerais ou quando o software possui um alto grau de complexidade. Portanto, o diagrama de sequência deve ser utilizado para representar os aspectos do comportamento do software quando for necessário representar informações de desempenho e concorrência, podendo ser substituído pelo diagrama de comunicação em outros casos. Em casos específicos, os diagramas de casos de uso, atividade e de estados podem ser recuperados.

Como resultado do processo de seleção das visões arquiteturais, é definido o conjunto de visões a ser recuperado visando representar as estruturas do software de maneira concisa e completa. Tal conjunto é apresentado na Tabela 3.

Tabela 3 - Visões selecionadas

Necessidade de Criação	Visões de Módulos	Visões de C&C	Visões de Alocação	Diagramas de Comportamento
Sempre	Visão de Decomposição ou de Camadas ou de Generalização	Visão de Chamadas	Visão de Implementação	Diagrama de Sequência ou de Comunicação
Sempre que possível	Visão de Aspectos e de Modelo de Dados	Visão de Eventos e de Repositório de Dados		
Sempre que necessário				Diagrama de Casos de Uso, de Atividade e de Estados

As visões selecionadas formam um conjunto representativo para a compreensão dos softwares legados e fornecem as bases para as atividades de manutenção dos mesmos. Este conjunto de visões, entretanto, não consiste em um conjunto definitivo, podendo ser alterado em razão de tipos específicos de software e projetos de desenvolvimento ou de necessidades específicas do processo de manutenção pelo qual o software passará.

3.2 Análises do Software

Como exposto anteriormente, os objetivos do processo de recuperação da arquitetura determinam as visões que devem ser obtidas do software em estudo. Para que essas visões sejam construídas, é necessário analisar os softwares para extrair as informações necessárias e identificar os padrões arquiteturais presentes neles.

A fim de proporcionar um conjunto de informações completo, acurado e adequado para a construção das visões propostas, são estudadas nesta seção as principais análises realizadas sobre os softwares legados.

A principal dificuldade da extração de informações sobre os softwares legados é a falta de fontes confiáveis de informações. Na maioria das vezes, a documentação dos softwares está desatualizada ou até mesmo não existe e o contato com os desenvolvedores do software é restrito ou impossível.

Outro fator de risco é a diversidade dos artefatos disponíveis para extração de informações. Esses artefatos podem ser de diversos tipos e níveis de abstração, bem como modelos de projeto, códigos fonte, bibliotecas, arquivos de *build*, arquivos executáveis, arquivos de *log*, entre outros.

Em grande parte dos casos, no entanto, a única fonte de informações confiável a respeito do software é o código fonte, ou em alguns casos, os arquivos binários (CANFORA; DI PENTA, 2008). É necessário extrair o máximo de informações possíveis desses artefatos de forma a construir as representações desejadas para as atividades de manutenção desses softwares.

Outro problema normalmente encontrado é a diversidade de linguagens de programação utilizadas no desenvolvimento dos softwares. Em muitos casos, os softwares são desenvolvidos em várias linguagens de programação ou dialetos destas.

É necessário, portanto, integrar as várias ferramentas utilizadas para analisar o código escrito em diferentes linguagens e para analisar os diferentes tipos de artefatos do software.

A extração de informações é feita com o auxílio de ferramentas de apoio, pois devido ao tamanho e complexidade dos softwares atuais é praticamente impossível realizar tal atividade manualmente. Existem diversas ferramentas que implementam diferentes modos de análise do software e operam sobre diferentes linguagens de programação. Entretanto, o foco deste trabalho não é o estudo das ferramentas disponíveis e sim das técnicas implementadas por elas.

Diferentes tipos de análise podem ser realizadas nos artefatos do software que terá sua arquitetura reconstruída. Essas formas de análise podem ser classificadas como de baixo nível ou de alto nível. Dentro do conjunto das análises de baixo nível destacam-se as análises estática, dinâmica e binária. Na análise de alto nível destacam-se as a identificação de padrões arquiteturais e as análises de métricas e de dominância. Estes tipos de análise são apresentados a seguir.

3.2.1 Análise de Baixo Nível

A análise de baixo nível é uma análise do software feita utilizando-se de técnicas da área de construção de compiladores. As informações são extraídas utilizando-se de analisadores sintáticos e léxicos, e representações fundamentais do software podem ser construídas a partir delas.

A análise de baixo nível contém a mesma quantidade de informações que o artefato analisado e, normalmente, é reversível. No entanto, esta fornece poucas informações a respeito da arquitetura para grandes softwares sendo necessário construir representações em níveis mais altos de abstração para tal fim.

Segundo Seacord; Plakosh e Lewis (2003), na engenharia reversa, o conhecimento de cada nível de abstração tem como base o conhecimento obtido em níveis inferiores de abstração.

Seguindo essa linha de pensamento, os diferentes tipos de análise de baixo nível podem ser utilizados para fornecer as bases para a análise de alto nível dos softwares. Podem ser utilizados também para fornecer informações detalhadas sobre módulos, componentes ou comportamentos do software quando necessário.

Um aspecto importante da análise de baixo nível é a precisão das informações obtidas. A extração de informações erradas pode dificultar o trabalho das fases posteriores de análise ou até mesmo comprometer o resultado final do processo de recuperação da arquitetura. Mais de um tipo de análise ou ferramenta pode ser utilizado de modo a obter um conjunto de informações mais confiável a respeito do software.

Outro fator que deve ser levado em conta quando este tipo de análise é feita, em especial quando se analisa softwares de grande porte, é o desempenho das ferramentas utilizadas. Softwares com grande volume de dados podem consumir muitos recursos para serem analisados.

Os diferentes tipos de análise de baixo nível, como a análise estática, a análise dinâmica e a análise binária, são apresentados a seguir.

3.2.1.1 Análise Estática

A análise estática é a análise feita sobre os artefatos disponíveis antes de execução do software. O objetivo deste tipo de análise é a extração de informações e construção de representações fundamentais do software em estudo.

Os tipos mais comuns de análise estática são apresentados a seguir.

3.2.1.1.1 Análise Sintática

Analisadores sintáticos são aplicados sobre o código fonte a fim de criar representações internas desse código para geração de código de máquina. Essas

representações podem ser utilizadas para a atividade de compreensão do software em estudo ou como fonte de informações para a análise de alto nível.

A análise sintática do código pode resultar em uma grande variedade de representações tais como gráficos de chamada de função, gráficos de fluxo de controle, gráficos estruturais e análise da definição e uso dos tipos de dados e variáveis.

A análise sintática é uma técnica de extração de informações bastante difundida e conhecida e a maioria das ferramentas de apoio à engenharia reversa fornece alguma variedade de analisador sintático.

São exemplos ferramentas que implementam analisadores sintáticos: *Understand for C/C++/Java*, *Imagix*, *SNiFF+*, *CIA*, *C/C++ Information Abstractor* e *rigiparse*.

Os analisadores de árvores sintáticas abstratas fazem um trabalho semelhante aos analisadores sintáticos, no entanto, criam uma representação em forma de árvore das informações analisadas.

Alguns exemplos de analisadores de árvores sintáticas abstratas são *Gen++*, *Refine*, *Design Maintenance System (DMS)*, *TXL* e *Stratego*.

3.2.1.1.2 Análise Léxica

Analisadores léxicos examinam os elementos léxicos ou palavras dos artefatos do software. Analisadores léxicos podem ser configurados para buscar palavras ou grupos de palavras e produzir uma determinada saída quando o resultado da busca for positivo.

Um exemplo de utilização dos analisadores léxicos é a busca por diretivas de inclusão. O analisador reconhece as instâncias de *#include <nome de arquivo>* e produz como saída o nome do arquivo analisado e o nome do arquivo incluído, denotando a relação existente entre eles. Este tipo de análise é útil para construir

gráficos de dependência do software. Outro exemplo de utilização é a busca por repetições de trechos de códigos no código fonte.

Um exemplo de ferramenta de análise léxica é o *Lightweight Source Model Extractor (LSME)*.

3.2.1.2 Análise Dinâmica

A análise dinâmica é a análise que é feita sobre o software durante uma ou mais execuções controladas deste em seu ambiente operacional. Sua finalidade é gerar representações baseadas em informações dinâmicas do software, não disponíveis a partir da análise estática, a fim de ajudar no entendimento de sistemas que utilizam ligação dinâmica. Softwares orientados a objetos que utilizam polimorfismo são um exemplo de utilização de ligação dinâmica.

A análise dinâmica pode ser baseada no uso de ferramentas de apoio que produzem informações sobre o código quando este é executado. Analisadores dinâmicos coletam informações em tempo de execução como sequências de chamadas de função e fluxo de dados. Um exemplo de ferramenta para análise dinâmica é o *gproof*.

Analisadores de tráfego de dados em ambiente de redes também podem ser úteis para analisar o fluxo de informações entre componentes de um sistema de software quando este é executado.

A instrumentação de código também pode ser utilizada para adicionar código ao software enquanto esse é executado a fim de produzir a saída de informações específicas.

3.2.1.3 Análise Binária

Bibliotecas e sistemas executáveis que fazem parte de um sistema de software, normalmente só estão disponíveis em arquivos binários que, em muitos casos, são a única fonte de informações sobre o software.

O objetivo deste tipo de análise é recuperar o máximo de informações sobre o software ou seus componentes possível, com auxílio de ferramentas que implementam o processo inverso dos compiladores e montadores.

Os desmontadores traduzem um programa executável para uma representação em linguagem Assembly. Os descompiladores traduzem o programa para uma representação em linguagem de alto nível.

A descompilação é feita em passos. O primeiro passo é a análise sintática do arquivo binário com o objetivo de identificar os códigos e parâmetros armazenados no arquivo que são dependentes da máquina na qual o software é executado. Os resultados dessa análise são submetidos a uma análise semântica a fim de identificar os tipos de informações contidas no arquivo e a finalidade das instruções ou de grupos de instruções, ou seja, identificar o que as instruções contidas no arquivo realmente fazem.

Este processo é relativamente bem compreendido, entretanto a recuperação de funções e chamadas pode ser difícil, bem como a recuperação de tipos compostos de dados e de estruturas de controle.

3.2.2 Análise de Alto Nível

A análise de alto nível é a análise que se destina a recuperar uma visão arquitetural do software em altos níveis de abstração, normalmente em níveis mais altos que o nível de código.

Normalmente, este tipo de análise não é feita diretamente sobre o código fonte do software; uma análise de baixo nível é aplicada e representações de baixo nível são criadas a fim de ser utilizadas como artefato de entrada da análise de alto nível. A análise de alto nível consiste em diminuir a quantidade de informação em detrimento da precisão para facilitar a compreensão do software analisado. As representações de baixo nível podem ser compostas tanto por informações estáticas quanto dinâmicas do software.

Este tipo de análise é feita a partir do mapeamento inverso dos elementos extraídos do software a fim de identificar as possíveis estruturas arquiteturais que eles representam. A identificação de componentes, módulos e padrões arquiteturais é uma forma de se obter estruturas arquiteturais.

Uma das formas mais utilizadas de fazer este tipo de identificação é a de filtros e agregações nas informações.

A qualidade deste tipo de análise é dependente tanto da precisão das informações obtidas na análise de baixo nível quanto da qualidade do mapeamento inverso realizado e da definição de quais informações são necessárias e quais podem ser descartadas na construção de visões de alto nível.

Este é o tipo de análise que será utilizada para a criação das visões arquiteturais dos softwares legados que serão estudados e o produto deste tipo de análise pode incluir árvores de dominância, diagramas da UML, gráficos de componentes e conectores, gráficos de módulos e outros.

A seguir, são apresentadas algumas formas comuns de análise de alto nível.

3.2.2.1 Identificação de Padrões Arquiteturais

A identificação de padrões arquiteturais consiste na análise do software em busca de evidências de padrões de projetos utilizados na construção do software. Estes

padrões são soluções eficazes para determinados tipos de tarefas que os softwares precisam executar.

A identificação de padrões é útil para identificar se o software em estudo foi desenvolvido segundo algum padrão arquitetural, facilitando a identificação de relações entre os componentes do software que são normalmente encontradas nos padrões identificados.

Bibliotecas que implementam padrões arquiteturais conhecidos podem ser utilizadas para identificar os padrões dentro de um conjunto de dados obtidos das análises de baixo nível.

3.2.2.2 Análise de Métricas

A análise de métricas consiste identificação de componentes a partir da procura por módulos e classes com base em métricas de complexidade, acoplamento e coesão.

Uma abordagem comum neste tipo de análise é a busca por classes ou módulos centrais, caracterizados pela alta complexidade interna e pelo alto acoplamento externo. Elementos do software conectados com estas classes ou módulos e que possuem baixo nível de acoplamento externo, possivelmente farão parte desses módulos e poderão ser agregados.

3.2.2.3 Análise de Dominância

Este tipo de análise é feito graficamente e tem como objetivo a identificação de nós de dominância em gráficos de chamada de função.

Estes nós são caracterizados por serem os únicos pontos de entrada para uma subseção do gráfico analisado. Os nós dominantes e suas subseções são candidatos a módulos do software pelo fato de representarem um conjunto de funções com baixo acoplamento externo.

3.2.3 A Escolha das Análises

A escolha das análises leva em conta a utilidade de cada uma delas em relação à obtenção de um conjunto completo e preciso de informações. Essas informações servirão como ponto de partida par a criação das visões selecionadas, portanto, devem ser adequadas às visões, fornecendo as bases para a criação das mesmas.

Segundo Löwe et al. (2002), o resultado do processo de recuperação da arquitetura não depende somente da análise de alto nível e das representações construídas por esta. A qualidade das informações obtidas pela análise de baixo nível e das representações básicas do software criadas por esta é fundamental para o sucesso do processo.

A qualidade destes dois tipos de analises é, portanto, fundamental para que o processo alcance seus objetivos. Como o objetivo da recuperação de arquitetura é criar representações arquiteturais com alto nível de abstração, devemos incluir tanto a análise de alto nível quanto a de baixo nível no método proposto.

É importante também, de acordo com Löwe et al. (2002) que o processo de recuperação da arquitetura seja baseado tanto em informações estáticas quanto em informações dinâmicas, pois as informações estáticas e dinâmicas são complementares.

Portanto, as análises estáticas e dinâmicas são importantes a fim de obter o máximo de informações possível do sistema. A utilização desses dois tipos de análises complementares é especialmente importante nos casos em que há ligação dinâmica no software, ou seja, alguns fluxos de controle e de dados são determinados dinamicamente.

A análise dinâmica pode ser feita por meio dos analisadores dinâmicos que coletam as informações do software, de analisadores de tráfego de dados ou da instrumentação de código. A escolha das ferramentas para a realização da análise dinâmica dependerá do tipo do software a ser analisado.

Da mesma forma, a análise estática pode ser feita por meio de analisadores léxicos, sintáticos ou de árvores sintáticas abstratas. A escolha de uma delas, ou de um conjunto, pode ser feita de acordo com as características do software a ser analisado e das ferramentas disponíveis.

O custo da extração deve ser levado em conta na escolha das análises e ferramentas. A utilização de análises bem conhecidas, como as análises sintáticas e léxicas, pode significar economia de esforço e tempo necessários na análise de alto nível. O custo da infraestrutura necessária para a realização de algumas análises e para a execução de algumas ferramentas em grandes softwares também deve ser levado em conta.

A análise binária pode ser feita adicionalmente para obtenção de informações nos casos onde o código fonte do software está disponível para análise, entretanto, pode ser descartada visto que é relativamente mais complexa que outras análises.

Entretanto, nos casos onde o software utiliza bibliotecas armazenadas em arquivos binários ou quando nem todo o código fonte está disponível para análise, este tipo de análise é fundamental para coletar informações a respeito dos componentes armazenados nos arquivos binários.

Portanto, a realização deste tipo de análise dependerá dos artefatos disponíveis do software que será analisado e de suas características.

As análises estudadas neste trabalho podem ser feitas em sua totalidade ou não. Muitas vezes com uma das análises apresentadas é possível descobrir muito sobre a arquitetura do sistema e criar as representações arquiteturais necessárias. No entanto, nem sempre isso é possível e, normalmente, é necessário fazer várias análises, dependendo do tamanho e da complexidade do software analisado.

O responsável pela análise de alto nível pode iniciar a análise escolhendo uma técnica que ele acredite se adaptar melhor ao sistema em estudo e que ele esteja

mais familiarizado. A utilização de mais de várias técnicas, apesar de não ser sempre necessária, pode ser útil para confrontar os resultados obtidos.

Portanto, chega-se a uma seleção de análises que serão feitas nos softwares cuja arquitetura será recuperada pelo método proposto. A Tabela 4 mostra as análises selecionadas.

Tabela 4 - Análises selecionadas

Análises		Utilização
Análise de alto nível		Sempre
Análise de baixo nível	Análise estática	Sempre
	Análise dinâmica	Sempre
	Análise binária	Sempre que necessário

A realização de outras análises não está descartada, pois o método proposto é um método aberto, bastando fazer as adaptações necessárias para a adição.

3.3 Considerações do Capítulo

A utilização de análises e ferramentas de apoio para auxiliar nas atividades da recuperação da arquitetura é essencial devido ao tamanho e a complexidade dos softwares atuais. Portanto, a seleção e combinação dessas análises se tornam fundamentais para o sucesso do processo de recuperação da arquitetura.

Uma das práticas recomendadas para o processo de recuperação da arquitetura é a determinação dos objetivos do processo antes de iniciá-lo a fim de evitar a realização de atividades de extração de informações e de geração de visões desnecessárias (KAZMAN; O'BRIEN; VERHOEF, 2003).

Portanto, tanto as visões selecionadas quanto as análises realizadas sobre o software devem ser adequadas aos objetivos do processo, ou seja, a criação de documentação adequada à compreensão dos softwares legados que sofrerão manutenção.

Devem também, ser selecionadas de acordo o software em estudo, pois os diferentes tipos de projeto e os diferentes tipos de artefatos disponíveis para análise têm influência sobre o método de recuperação de arquitetura e suas atividades.

A documentação deverá ser voltada aos participantes do processo de manutenção que necessitam compreender os softwares legados. Portanto, as visões selecionadas são voltadas à compreensão dos softwares e as análises são voltadas à identificação dos padrões arquiteturais e à produção dessas visões com precisão e completude de informações.

Ao contrário do que possa aparentar, a criação de muitas visões sobre o software seria inadequada para a compreensão do mesmo, pois os usuários da documentação da arquitetura poderiam se perder em meio a muitas informações. As visões selecionadas foram estudadas de modo representar o que é essencial sobre o software, de maneira clara, concisa e evitando-se ambiguidades e repetições de informações. O nível de detalhamento destas pode ser ajustado de acordo com as necessidades de cada usuário e visões com níveis diferentes de detalhamento podem ser criadas para usuários diferentes.

A utilização de uma notação bem definida e com uma semântica precisa é a melhor maneira de eliminar as ambiguidades da descrição da arquitetura, embora seu uso não seja obrigatório.

A UML se tornou uma notação padrão para documentar as arquiteturas de software proporcionando resultados satisfatórios (BASS; CLEMENTS; KAZMAN, 2003). Visto que a UML tem sido amplamente utilizada na descrição arquitetural e que o conhecimento dessa linguagem já está bastante difundido entre os engenheiros de software, arquitetos e desenvolvedores, seu uso para apresentação das visões arquiteturais é adotado pelo método proposto neste trabalho como forma a facilitar a compreensão da arquitetura a ser recuperada. No entanto, o uso de outros tipos de notações pode ser apropriado e não está descartado.

Para a escolha das visões a serem criadas pelo método proposto neste trabalho, é utilizado método descrito por Bass; Clements e Kazman (2003) e Clements et al. (2010). O conjunto de visões foi selecionado com base em três aspectos considerados essenciais: a utilidade das visões arquiteturais mais comuns na literatura, as necessidades dos participantes do processo de manutenção dos softwares legados e as informações e visões consideradas essenciais para a compreensão dos softwares obtidas a partir do estudo das diversas abordagens propostas para tal fim.

A Tabela 3 mostra as visões que serão criadas para representar a arquitetura dos softwares analisados pelo método proposto. Algumas dessas visões, consideradas essenciais para a compreensão dos softwares, devem ser sempre criadas durante a execução do método; outras visões, dependentes de padrões arquiteturais, são criadas sempre que os softwares forem desenvolvidos utilizando-se desses padrões; há ainda um conjunto de visões que são criadas somente em casos específicos, quando os gerentes e patrocinadores necessitam de informações não existentes nas outras visões ou quando o software possuir alto grau de complexidade e esta necessitar ser compreendida pelos responsáveis pela manutenção.

As análises, por sua vez, foram selecionadas com a finalidade de produzir um conjunto de informações adequado, preciso e completo para a criação das visões arquiteturais desejadas e identificar os padrões arquiteturais presentes nos softwares em estudo.

O conjunto de análises do software selecionado procura ser amplo em relação aos principais artefatos disponíveis, e acurado e completo em relação à qualidade das informações para a construção das visões arquiteturais. No entanto, é essencial para a obtenção de informações de qualidade que a seleção de ferramentas seja feita de forma adequada e considerando os objetivos da recuperação da arquitetura.

A Tabela 4 mostra as análises pelas quais os softwares passarão na execução do método proposto. Algumas dessas análises são consideradas essenciais e devem ser realizadas sempre, outras devem ser realizadas apenas quando necessário. O

conjunto de análises e ferramentas utilizado nas análises deve ser escolhido pelos responsáveis pela aplicação do método de acordo com o software em estudo e com a experiência deste.

Outros tipos de análises do software podem ser aplicados em diversos artefatos do software como os modelos de projetos, históricos de alterações e outros documentos, a fim de aumentar o conjunto de informações extraídas do software em estudo.

Por fim, vale ressaltar que as análises e representações arquiteturais discutidas e avaliadas neste capítulo não são específicas ao método proposto neste trabalho e podem ser aplicadas a outros métodos abertos de recuperação da arquitetura no seu todo ou em partes.

No capítulo seguinte, é apresentado o método proposto contendo as atividades de criação das visões selecionadas e as atividades de análise selecionadas neste capítulo.

4. MÉTODO DE RECUPERAÇÃO DA ARQUITETURA PARA A COMPREENSÃO DE SOFTWARES LEGADOS

Neste capítulo é apresentado o método proposto por este trabalho para a recuperação da arquitetura dos softwares legados que necessitam de manutenção. Este método foi elaborado com o objetivo de facilitar a compreensão dos softwares legados e fornecer as bases para a realização das atividades do processo de manutenção.

O diagrama de atividades do método proposto é apresentado na Figura 22.

Este método foi construído com base no método ARMIN (KAZMAN; O'BRIEN; VERHOEF, 2003; BASS; CLEMENTS; KAZMAN, 2003) que fornece a estrutura do método proposto. Essa estrutura compreende a divisão do método em quatro fases, a conversão das informações extraídas em um formato padrão, o armazenamento das informações em um banco de dados relacional e a fusão das visões fundamentais obtidas. Do método ARMIM, são derivadas também algumas das características do método proposto como a interatividade, a interpretatividade e a iteratividade.

As análises realizadas no software, tanto na fase de Extração de Informações quanto na fase de Composição das Visões Arquiteturias são características do método proposto. A seleção das visões arquiteturais que serão construídas para a representação do software em estudo e sua construção também são provenientes do método proposto neste trabalho.

Como pode ser notado pelo diagrama de atividades, tanto o método em si como a fase de Composição das Visões Arquiteturais podem ser feitos de maneira iterativa.

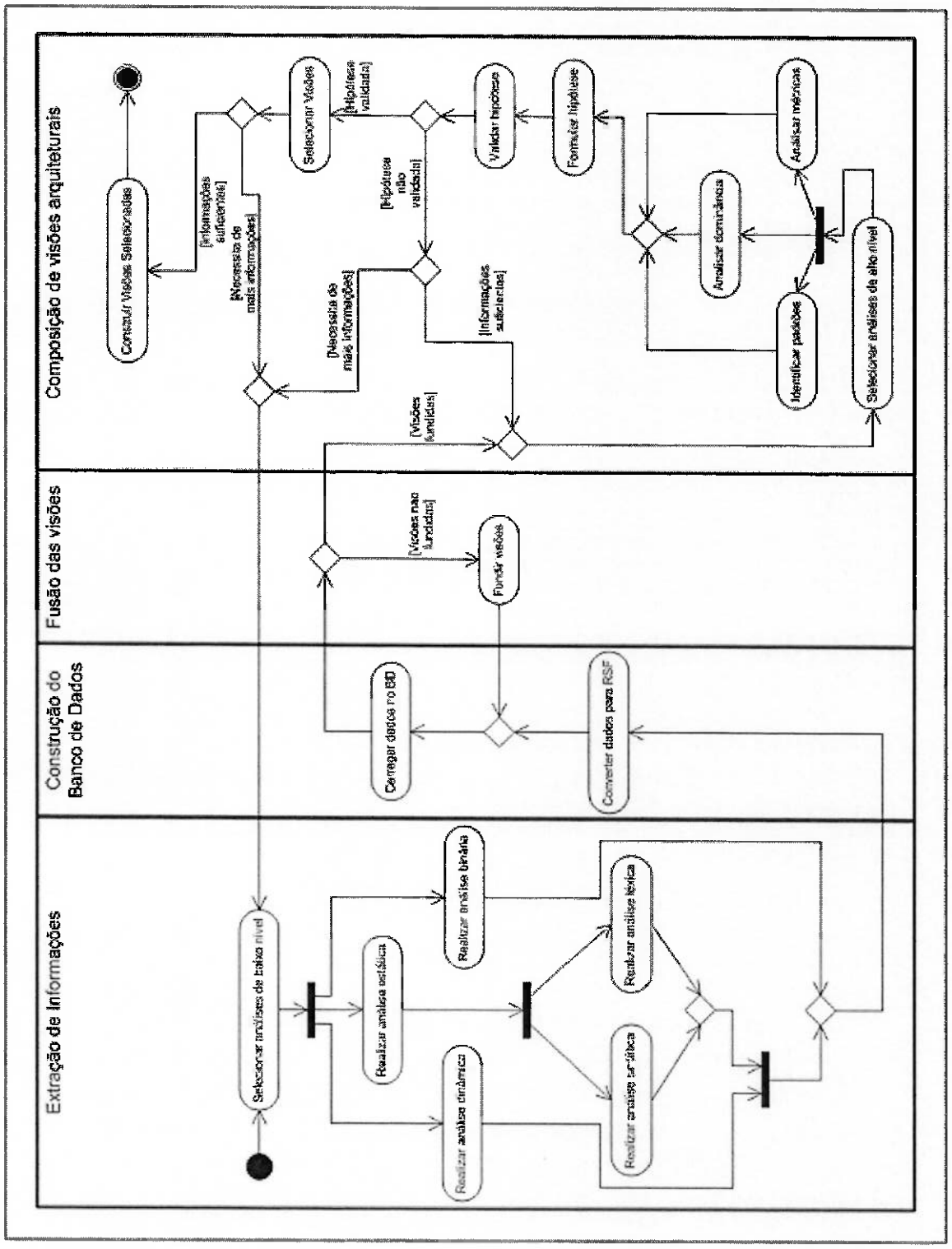

Figura 22 - Diagrama de atividades do método proposto

Não há um número definido de iterações que devem ser realizadas. Esse número depende de alguns fatores como a complexidade do software em estudo, o conhecimento prévio obtido do software pelos responsáveis pela aplicação do método e a própria experiência destes com métodos de engenharia reversa.

De acordo com Kazman; O'Brien e Verhoef (2003), não há um critério para o fim do processo de recuperação da arquitetura. Este deve ser encerrado quando as representações arquiteturais forem suficientes para o suporte às necessidades de seus usuários.

Portanto, a definição do número de iterações bem como do encerramento do processo cabe aos responsáveis pela aplicação do método e pode ser feita durante a aplicação do método. Esta definição dependerá das necessidades de novas análises e extrações de informações para a proposição e validação de uma arquitetura e para a construção das visões arquiteturais com o nível de detalhamento desejado.

A seguir são apresentadas as fases do método proposto e suas atividades.

4.1 Extração de Informações

A fase de Extração de Informações compreende a análise do software e a extração de informações dos artefatos disponíveis. As informações devem ser extraídas dos artefatos na forma de um conjunto de elementos e suas relações, característica herdada do método utilizado como base para a proposição deste.

O método proposto neste trabalho propõe a utilização de diferentes tipos de análises de baixo nível com o objetivo de extrair as informações do software necessárias à compreensão da arquitetura do mesmo e a construção das visões arquiteturais.

Como pode ser visto na Figura 22, a primeira atividade desta fase é a seleção das análises de baixo nível que serão realizadas. Essa seleção dependerá dos tipos de artefatos disponíveis para análise.

De acordo com a Tabela 4, apresentada no capítulo 3, a análise estática e a análise dinâmica são atividades obrigatórias para todos os softwares, desde que se tenham condições para executar o software e acesso ao seu código fonte, ou parte dele.

Dentro da análise estática, tem-se duas vertentes que se destacam: a análise sintática e a análise léxica. A escolha de uma delas ou de ambas dependerá de fatores como a disponibilidade de ferramentas que analisem a linguagem na qual o software foi construído, o domínio da técnica e das ferramentas disponíveis pelo responsável pela atividade, os recursos exigidos pelas ferramentas disponíveis, entre outros. No caso da existência de facilidades para a realização tanto da análise sintática quanto da análise léxica, recomenda-se a realização de ambas a fim de obter um conjunto mais confiável e completo de informações.

A análise dinâmica deverá ser realizada preferencialmente no ambiente operacional do software e por meio de analisadores passivos, ou seja, que não alterem as condições do software e de seu ambiente durante a execução do mesmo. As ferramentas devem ser selecionadas de acordo com o tipo do software em estudo e as facilidades oferecidas por elas.

A análise binária é obrigatória para os casos em que o software utilize bibliotecas ou componentes armazenados em arquivos binários, e não se tenha acesso ao código fonte destes. Para os casos que se tenha acesso ao código fonte, essa análise é opcional e pode utilizada para comparação e validação de resultados obtidos de outras análises. A escolha das ferramentas adequadas fica a critério dos responsáveis por essa atividade.

Cada conjunto de informações extraídas consiste em uma visão fundamental do software. As visões extraídas devem ser armazenadas no banco de dados para posterior fusão dos diferentes tipos de informações.

Esta fase pode ser realizada novamente sempre que for necessário obter informações complementares sobre o software, seja para a formulação e validação das hipóteses sobre a arquitetura do software ou para a obtenção de informações para a criação ou detalhamento das visões arquiteturais que serão criadas.

4.2 Construção do Banco de Dados

A fase de Construção do Banco de Dados consiste em duas atividades: conversão dos dados para um formato padrão e carregamento dos dados no banco de dados. Esta fase do método proposto é derivada do método ARMIN.

A primeira atividade consiste na conversão dos conjuntos de elementos e suas relações para um formato padrão. A maioria das ferramentas tem a capacidade de armazenar os resultados das extrações de informações em arquivos de texto. Essas informações são então convertidas por meio de *scripts* para um formato padrão. Neste método é proposto o uso do padrão RSF (O'BRIEN; STOERMER, 2003).

Os arquivos RSF têm o seguinte formato: *relação <elemento 1> <elemento 2>*.

A segunda atividade consiste no carregamento dos dados convertidos no banco de dados utilizado e não possui nenhuma restrição ou recomendação a ser destacada.

Os dados presentes no banco de dados podem ser posteriormente convertidos para outros padrões para serem utilizados como entrada em ferramentas de análise gráficas utilizadas na fase de Composição de Visões Arquiteturais.

4.3 Fusão das Visões

A fase de Fusão das Visões consiste na fusão das visões fundamentais do software obtidas na fase de Extração de Informações. Normalmente a fusão é feita entre visões com o mesmo tipo de informação, obtidas das análises estática, dinâmica e binária.

Esta fusão permite que se obtenham visões fundamentais mais completas, acuradas e confiáveis do software em estudo que serão utilizadas na fase seguinte para a elaboração das hipóteses sobre a arquitetura do mesmo.

O método proposto neste trabalho não acrescenta nenhuma atividade ou característica a esta fase do método, sendo ela derivada do método utilizado como base para a proposição do presente método.

4.4 Composição de Visões Arquiteturais

A fase de Composição de Visões Arquiteturais consiste na visualização, interpretação e abstração das informações obtidas sobre o software e na formulação de hipóteses sobre sua arquitetura através da manipulação das visões existentes. Estas atividades são derivadas do método ARMIN. Seu objetivo é encontrar e validar as hipóteses sobre a arquitetura do software em estudo e construir as visões arquiteturais desejadas.

Os diferentes tipos de análise de alto nível que são realizadas nesta fase bem como as atividades de seleção e construção das visões arquiteturais são atividades propostas pelo presente método a fim de facilitar a compreensão da arquitetura do software em estudo pelos responsáveis pela recuperação da arquitetura e pelos participantes do processo de manutenção que utilizarão essa arquitetura em suas atividades.

Esta fase tem início com a seleção das análises de alto nível que serão utilizadas. Esta seleção é feita pelo responsável pela atividade levando-se em consideração alguns fatores como o tipo do software em estudo, as informações disponíveis, a experiência do responsável pela atividade com as análises, a disponibilidade de ferramentas que suportem as análises, entre outros.

Nenhuma das análises de alto nível é considerada fundamental para o sucesso do processo. Podem existir casos que utilização de apenas uma delas seja suficiente para a obtenção e validação de uma arquitetura hipotética para o software em estudo. Entretanto, para softwares grandes e complexos, pode ser necessária a utilização de várias análises diferentes simultaneamente.

Independente da escolha das análises que serão utilizadas, as atividades da análise de alto nível podem ser realizadas com o auxílio de ferramentas de manipulação de visões gráficas, obtidas das informações armazenadas no banco de dados. Essas ferramentas são úteis para a visualização e manipulação das visões obtidas do software com o objetivo de facilitar o entendimento deste. Filtros para abstração das informações podem ser utilizados a fim de obter visões de alto nível. Normalmente, a filtragem é feita com a utilização de *scripts* ou de pesquisas no banco de dados.

Durante a análise de alto nível deve ser formulada uma hipótese sobre a arquitetura do software em estudo. Esta hipótese deve ser validada ao final da análise utilizando-se como parâmetro as informações obtidas do software. Caso a arquitetura proposta não seja validada, uma nova análise pode ser realizada ou mais informações podem ser extraídas de modo a fundamentar a hipótese atual ou fornecer dados para a geração de uma nova hipótese.

Caso a hipótese seja validada, a próxima atividade é a construção das visões arquiteturais do software. Para essa atividade, também pode ser necessário a extração de informações adicionais, tanto para a geração das visões quanto para o detalhamento das mesmas.

4.5 Considerações do Capítulo

O método proposto para recuperação de arquiteturas dos softwares legados destina-se a orientar os responsáveis por esta atividade a construir um conjunto de visões arquiteturais do software que facilite a compreensão deste pelos participantes do processo de manutenção, sendo voltado principalmente aos desenvolvedores.

Pode-se notar grande semelhança do método proposto com o método ARMIN. Ambos os métodos são métodos interpretativos, interativos e iterativos. Os métodos utilizam um repositório de dados central construído sob um formato padrão de dados de forma a reunir os resultados das extrações de dados feitas por diferentes ferramentas. Por fim, as fases de Construção do Banco de Dados e de Fusão das Visões são idênticas nos dois métodos.

As principais diferenças entre o método ARMIN e o método proposto consistem nas atividades das fases de extração de informações e análise do software e na construção das visões arquiteturais.

Na fase de Extração de Informações, é proposto um conjunto de análises de baixo nível do software a fim de extrair as informações do mesmo. Algumas dessas análises são consideradas essenciais para a obtenção de um conjunto de informações completo e acurado sobre o software. Outras análises têm sua aplicação considerada opcional em alguns casos e obrigatória em outros, dependendo dos artefatos disponíveis para análise. Outras análises não citadas também podem ser incluídas ao método quando o responsável por esta atividade julgar conveniente.

Na fase de Composição das Visões Arquiteturais, é proposto um conjunto de análises de alto nível a fim de elaborar e validar as hipóteses sobre a arquitetura do sistema. A utilização de cada uma das análises propostas ou de um conjunto delas fica a critério do responsável pela execução desta fase. Novas extrações de informações, a carga destas no banco de dados e a fusão das visões fundamentais podem ser feitas quando necessário. Após a validação das hipóteses sobre a arquitetura do sistema, novas extrações podem ser feitas a fim de fornecer as bases para a construção das visões arquiteturais desejadas.

A conversão dos dados extraídos para um formato padrão e seu armazenamento em um banco de dados favorecem a independência das análises de baixo nível em relação às análises de alto nível. No entanto, as análises de alto nível e a construção das visões arquiteturais do software irão utilizar as informações fornecidas pelas análises de baixo nível. Portanto, as extrações de informações feitas pela análise de baixo nível devem ser direcionadas à obtenção das informações necessárias às atividades posteriores do método proposto. Além desta relação, não há qualquer tipo de restrição quanto à utilização de um tipo de análise em relação às demais nem é necessária qualquer adaptação adicional delas.

5. APLICAÇÃO PRÁTICA E ANÁLISE

Este capítulo apresenta a aplicação do método em um software e os resultados obtidos.

O MemNotes 4.55 é uma interface de linha comando para manter pequenas notas no console do Linux. Possui sistema de etiquetas, busca e filtros. Qualquer editor de texto ou o próprio console podem ser utilizados para criar e editar notas. O software possui 2126 linhas de código divididas entre 34 arquivos.

Este software é um software livre e pode ser utilizado sob a *GNU General Public License*. Seu código é aberto e pode ser obtido a partir da página do projeto no SourceForge.net (SOURCEFORGE, 2011).

5.1 Extração de Informações

A fase de Extração de Informações tem início com a seleção das análises de baixo nível a serem realizadas sobre o software em estudo. Esta seleção foi feita de acordo com a Tabela 4, apresentada no capítulo 3.

A análise estática, composta pelas análises sintática e léxica, e a análise dinâmica foram selecionadas para a extração de informações sobre o software em estudo. A análise binária não foi selecionada pelo fato do código fonte completo do software estar disponível para análise.

As análises sintática e léxica, que compõem a análise estática do software, foram realizadas em ambiente Windows 7 pela ferramenta *Understand for C/C++*. Os artefatos gerados em arquivo de texto pela análise são:

- FileAverageMetrics.txt: métricas de complexidade dos arquivos do software.
- FileMetrics.txt: métricas dos arquivos do software.

- Files.txt: arquivos do software com as respectivas declarações de funções, tipos e variáveis.
- IncludeFiles.txt: inclusões de arquivos.
- InvocationTrees.txt: árvore de chamadas de função.
- ProgramUnitComplexity.txt: métricas de complexidade das unidades do software.
- ProgramUnitMetrics.txt: métricas das unidades do software.
- ProgramUnits.txt: unidades do software com as respectivas declarações, chamadas e tipos.
- ProjectMetrics.txt: métricas do projeto.
- SimpleInvocationTrees.txt: chamadas de função simples.

A Figura 23 mostra um trecho do arquivo IncludeFiles.txt.

```
core.h
    Include [ioshell.c, 15]          ioshell.c
    Include [ioshvbm.c, 14]          ioshvbm.c
    Include [main.c, 8]              main.c
    Include [noteff.c, 14]           noteff.c
    Include [sfi.c, 15]              sfi.c
```

Figura 23 - Trecho do arquivo IncludeFiles.txt

A análise dinâmica foi realizada em ambiente Ubuntu Linux pela ferramenta *gprof*. Foram selecionados três casos de uso para a realização da análise: exibição de notas armazenadas, adição de nota e exclusão de nota. Os artefatos gerados em arquivo de texto pela análise são:

- append_gprof.out: adição de nota.
- delete_gprof.out: exclusão de nota.
- openbase_gprof.out: exibição de notas armazenadas.

A Figura 24 mostra um trecho do arquivo append_gprof.out.

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    0.00      13/13          baseread [23]
[4]      0.0    0.00    0.00      13           m455fgetszs [4]
                0.00    0.00      13/51         m455fgetus [1]
                0.00    0.00      13/13         m455fgetns [3]
```

Figura 24 - Trecho do arquivo append_gprof.out

5.2 Construção do Banco de Dados

A fase de Construção do Banco de Dados tem início com a conversão dos dados extraídos pelas análises de baixo nível para um formato padrão. Essa conversão é feita por meio de *scripts* construídos em linguagem BOO que manipulam os arquivos de texto gerados pelas ferramentas e convertem os dados para o formato RSF.

Os *scripts* construídos nesta fase juntamente com suas entradas e saídas estão mostrados a seguir no formato "*entrada* > *script* > *saída*":

- append_gprof.out > call_gprof.boo > call1_gprof.rsf
- delete_gprof.out > call_gprof.boo > call2_gprof.rsf
- Files.txt > file.boo > file.rsf
- Files.txt > function.boo > function.rsf
- IncludeFiles.txt > include.boo > include.rsf
- openbase_gprof.out > call_gprof.boo > call3_gprof.rsf
- SimpleInvocationTrees.txt > call.boo > call.rsf

A Figura 25 mostra um trecho do arquivo include.rsf.

```
include defs.h addict.h
include ioshell.c core.h
include ioshvbm.c core.h
include main.c core.h
include noteff.c core.h
```

Figura 25 - Trecho do arquivo include.rsf

Cada um dos arquivos gerados no formato RSF é então carregado em uma tabela no banco de dados construído no Microsoft Access. Os demais arquivos no formato RSF gerados em fases posteriores da execução do método também são carregados nesse banco de dados.

5.3 Fusão das Visões

A fase de Fusão das Visões consiste na fusão de visões que contém o mesmo tipo de informação em visões mais acuradas e completas por meio de *scripts* construídos em linguagem BOO.

Como o software em estudo é um software simples e foram utilizadas apenas duas ferramentas de extração temos apenas uma fusão de visões relevante, que consiste na fusão das visões das chamadas de função obtidas pelas ferramentas utilizadas.

O *script* construído nesta fase juntamente com suas entradas e saídas estão mostrados a seguir no formato "*entrada* > *script* > *saída*":

- append_gprof.rsf, call.rsf, delete_gprof.rsf, openbase_gprof.rsf > fuse_call.boo > callfusedfinal.rsf

5.4 Composição de Visões Arquiteturais

A primeira atividade da fase de Composição de Visões Arquiteturais é a seleção das análises de alto nível que serão realizadas sobre o software em estudo com o objetivo de formular e validar uma hipótese sobre sua arquitetura. A análise de métricas e a identificação de padrões arquiteturais foram escolhidas e realizadas com auxílio de *scripts* construídos em linguagem BOO e de manipulação gráfica das visões obtidas por meio da ferramenta Rigiedit com o objetivo de abstrair as informações presentes nas visões fundamentais.

A Figura 26 mostra um exemplo de uma visão fundamental obtida do arquivo call.rsf no Rigiedit.

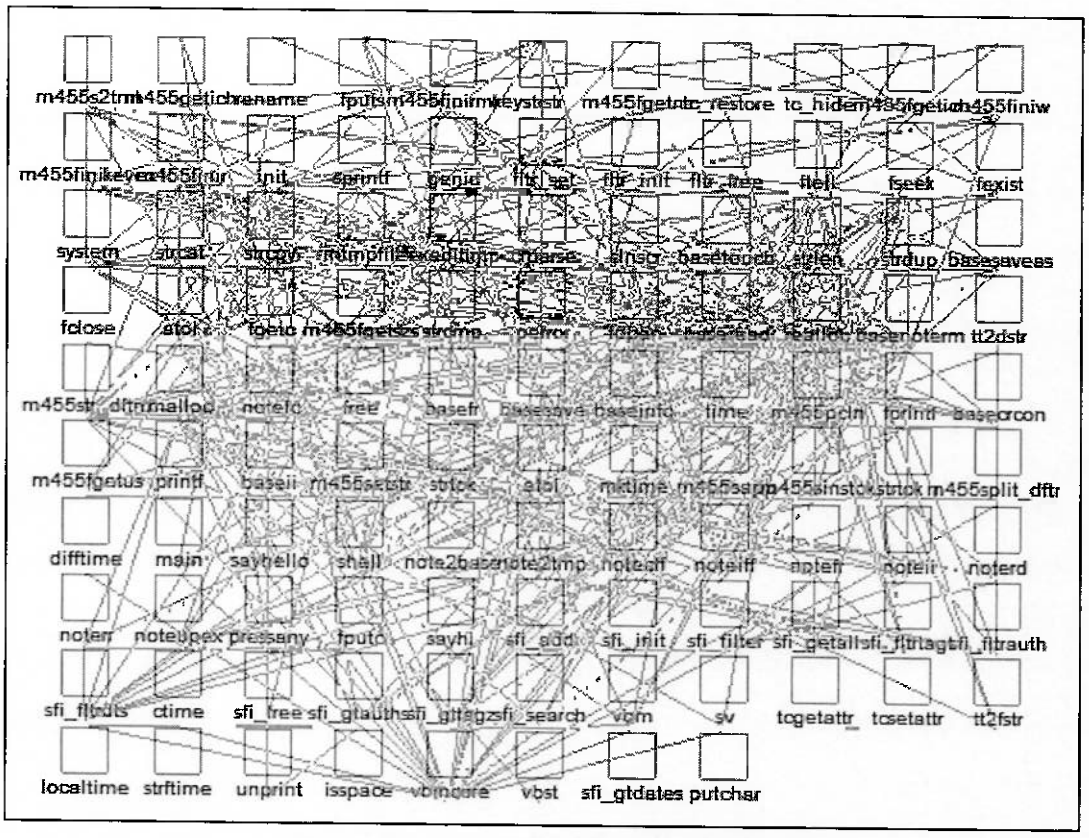

Figura 26 - Visão de call.rsf no Rigiedt

A visão mostrada acima representa as chamdas de funções do software em estudo e foi utilizada para a formulação de uma hipótese baseada nos possíveis módulos do software. Esta hipótese sobre a arquitetura do software foi formulada por meio da análise das métricas dos arquivos do software, da manipulação gráfica das visões obtidas e da utilização de *scripts* para abstração das informações.

A partir da visão de chamadas mostrada acima, as chamadas das funções foram agrupadas em chamadas dos arquivos correspondentes por meio de *scripts*. O agrupamento dos arquivos em módulos foi feito com base em métricas de complexidade e acoplamento. Esse agrupamento foi feito pela agregação de módulos que possuem baixo acoplamento externo aos módulos centrais, caracterizados pela alta complexidade interna e pelo alto acoplamento externo.

A Figura 27 mostra a agregação dos módulos *addict* e *m455fini* ao módulo *defs* e dos módulos *sayhi* e *version* ao módulo *main*. Os módulo *defs* e *main* são exemplo de módulos centrais do software.

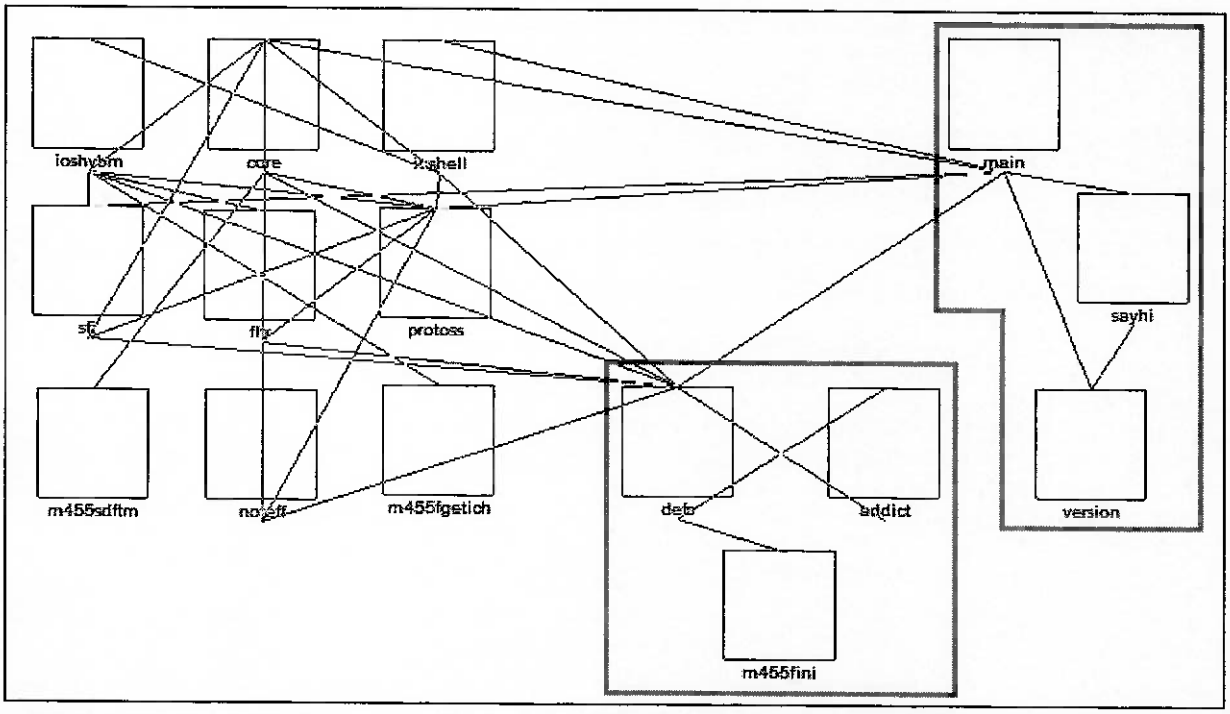

Figura 27 - Agregação de arquivos em módulos com base em análise de métricas de acoplamento e complexidade

Os *scripts* construídos nesta fase juntamente com suas entradas e saídas estão mostrados a seguir no formato "*entrada* > *script* > *saída*":

- include.rsf > modules.boo > modules.rsf, modules2.rsf
- include.rsf > modules_dep.boo > modules_dep.rsf, modules_dep2.rsf

A visão das chamadas de função mostrada acima, agora agrupadas por módulos, é mostrada na Figura 28, obtida do arquivo modules_dep2.rsf.

A validação dessa hipótese foi feita pelo confronto das informações sobre as chamadas de módulos e das informações de chamadas de função obtidas pelas ferramentas de extração. De acordo com a visão das chamadas dos módulos do software obtida, esse software tem uma arquitetura semelhante ao padrão de

arquitetura por camadas, com cada módulo representando uma camada, já que cada módulo possui apenas chamadas a módulos inferiores.

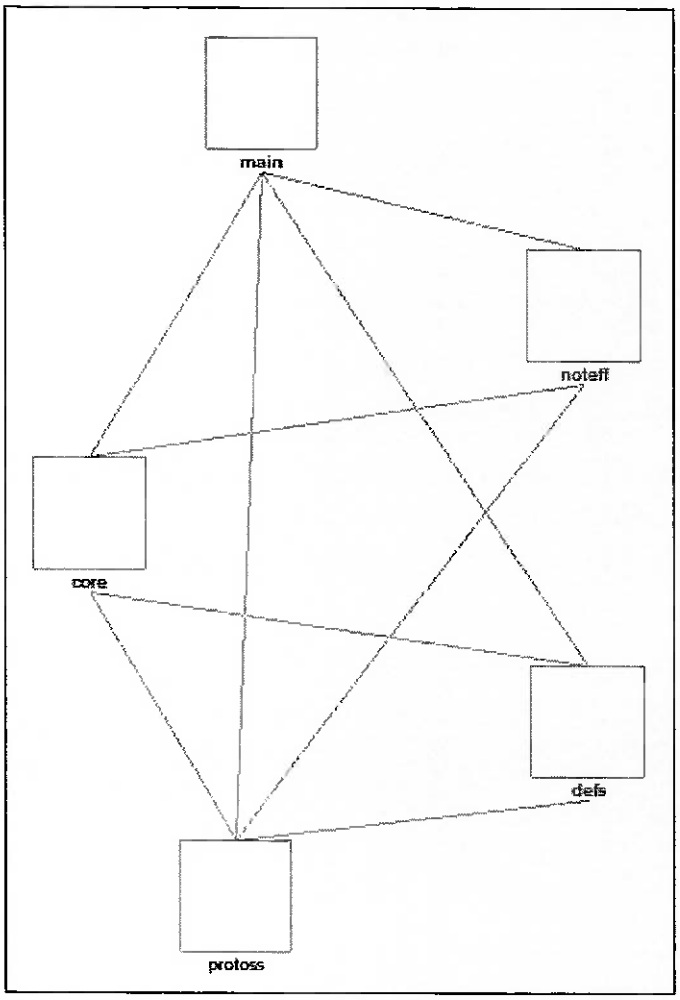

Figura 28 - Visão de modules_dep2.rsf no Rigiedit

A partir da validação da hipótese sobre a arquitetura do software em estudo foi feita a seleção das visões arquiteturais a serem construídas de acordo com a Tabela 3, apresentada no capítulo 3. As visões selecionadas são: visão de decomposição, visão de chamadas, visão de implementação e diagrama de comunicação.

As visões são construídas com alto nível de abstração, visto que o objetivo deste exemplo é ilustrar a aplicação do método proposto, não sendo necessária a representação de detalhes do software em estudo.

Para a construção das visões selecionadas foram utilizados alguns scripts construídos em linguagem BOO para abstrair as informações utilizadas. Os scripts construídos nesta fase juntamente com suas entradas e saídas estão mostrados a seguir no formato "*entrada* > *script* > *saída*":

- file-function.rsf, call.rsf > modules_call.boo > modules_call.rsf
- file-function.rsf > modules_call2.boo > modules_call2.rsf

A visão de decomposição foi construída a partir do arquivo modules_dep2.rsf assim como a visão de chamadas. A visão de implementação foi construída de forma tabular a partir da visão de decomposição. Já o diagrama de comunicação foi construído com a utilização dos *scripts* citados acima e resulta dos arquivos modules_call2.rsf, append_gprof.out e InvocationTrees.txt além da análise do código fonte.

As visões arquiteturais recuperadas do software em estudo, criadas no Microsoft Visio com base nas representações obtidas do Rigiedit, são mostradas a seguir.

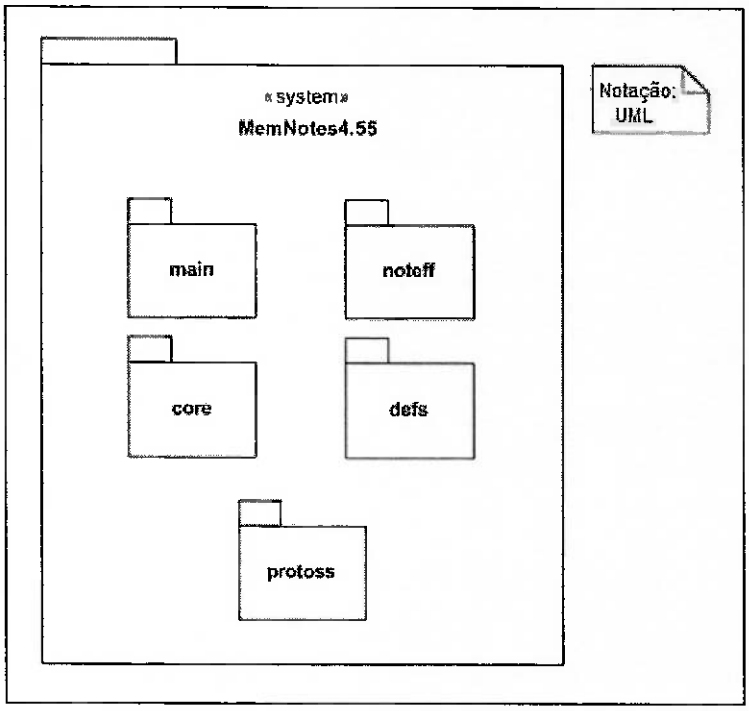

Figura 29 - Visão de decomposição

Tabela 5 - Visão de implementação

Módulo	Equipe
main	equipe1
noteff	equipe2
core	equipe3
defs	equipe4
protoss	equipe5

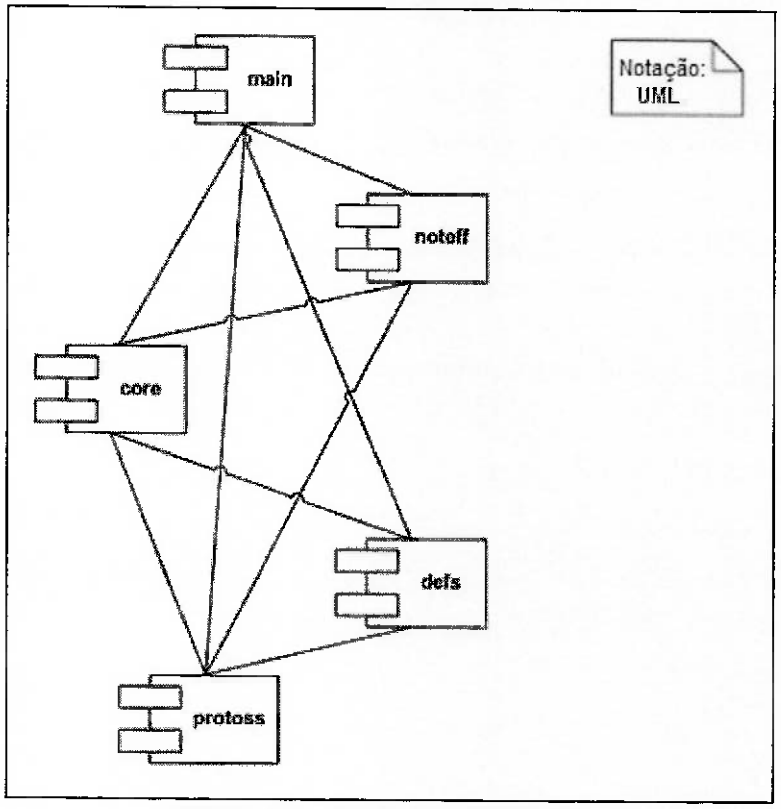

Figura 30 - Visão de chamadas

5.5 Considerações do Capítulo

Neste capítulo, é apresentada uma aplicação do método proposto neste trabalho em um software. Os resultados obtidos da aplicação do método mostram que o método é útil para a recuperação da arquitetura dos softwares legados.

As visões arquiteturais obtidas da aplicação do método condizem com a estrutura e as funções do software analisado. Este fato pode ser observado no diagrama de

comunicação construído, que mostra as responsabilidades dos módulos identificados bem definidas.

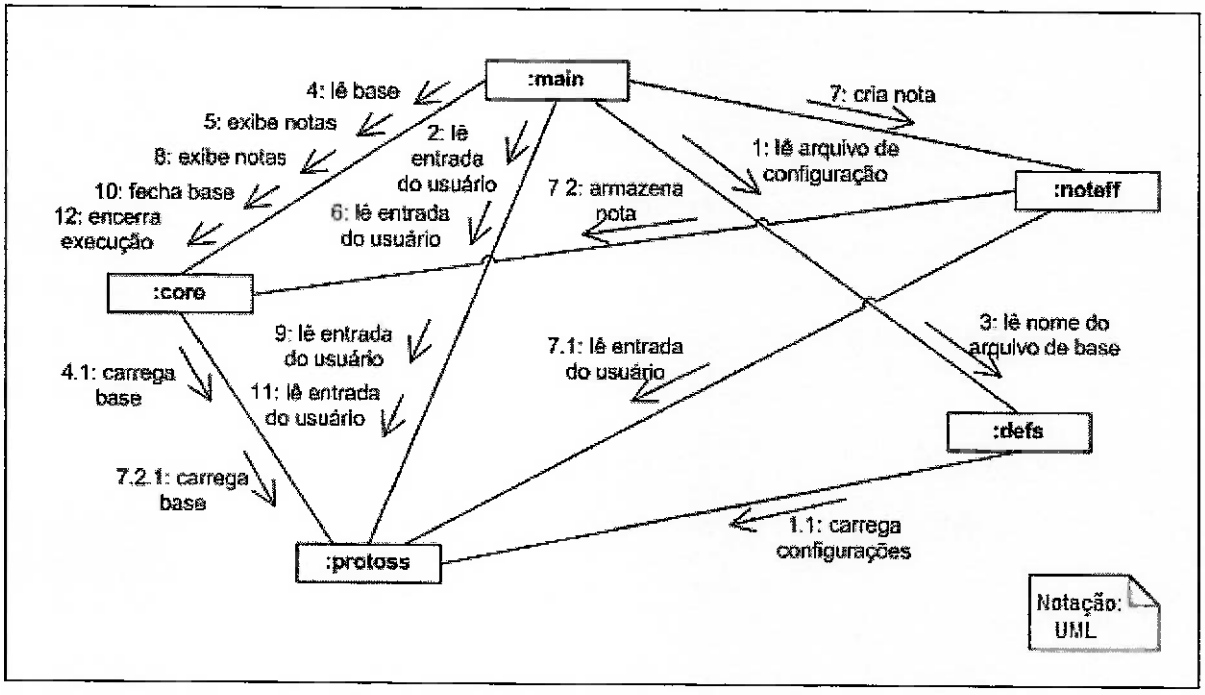

Figura 31 - Diagrama de comunicação

A aplicação do método proposto neste trabalho depende de fatores relacionados ao responsável pela aplicação do mesmo. Atividades como a seleção das análises de baixo nível, seleção e aplicação das análises de alto nível e formulação e validação das hipóteses sobre a arquitetura do software dependem da experiência com a aplicação de técnicas de engenharia reversa, da experiência com o uso das ferramentas selecionadas e do conhecimento de arquiteturas de software do responsável pela aplicação do método.

Entretanto, para indivíduos com níveis semelhantes de experiência e conhecimento, os resultados obtidos da aplicação do método para um mesmo software tendem a serem muitos semelhantes.

6. CONSIDERAÇÕES FINAIS

6.1 Contribuições do Trabalho

A manutenção de softwares legados é essencial para que o mesmo consiga continuar atendendo aos seus objetivos com eficácia, eficiência e qualidade durante seu ciclo de vida. Atualmente, muitas empresas têm optado por criar grupos especializados em manutenção de softwares legados ou terceirizar tais atividades.

Este fato resulta na necessidade da compreensão do software que receberá manutenção pelos responsáveis pelas atividades deste processo, atividade que emprega grande parte do esforço e do custo do processo de manutenção.

A contribuição deste trabalho é a proposta de um método de recuperação da arquitetura de software que resulta em um conjunto de visões arquiteturais do software em estudo adequado à compreensão deste pelos responsáveis pelas atividades do processo de manutenção.

Este conjunto de visões é resultado do estudo das diferentes formas de representação das arquiteturas dos softwares, suas utilidades e das necessidades dos responsáveis pelo processo de manutenção que usarão as visões obtidas. A arquitetura do software em estudo é obtida a partir de análises selecionadas para facilitar a extração das informações, a identificação dos padrões arquiteturais e a construção das visões desejadas.

A aplicação do método em um software legado é outra contribuição deste trabalho. A partir dela foi possível recuperar uma arquitetura do software em análise de modo que esta seja utilizada para a compreensão do mesmo. A arquitetura recuperada para o software em estudo é consistente e coerente com as funcionalidades do mesmo, mostrando que o método cumpre seus objetivos com eficácia.

Por se tratar de um método aberto, o método proposto pode ser utilizado para a recuperação da arquitetura de softwares desenvolvidos utilizando-se de diferentes padrões arquiteturais e diferentes linguagens de programação bastando para isso a obtenção de ferramentas adequadas e a criação dos *scripts* necessários para a conversão dos dados obtidos por essas ferramentas.

Vale ressaltar que o método proposto neste trabalho é dependente de fatores externos como os relacionados aos indivíduos que o aplicam e ao software em estudo. No entanto, para a análise de um mesmo software e indivíduos com graus semelhantes de experiência em aplicação de técnicas de engenharia reversa e de conhecimentos de arquitetura de software, os resultados obtidos devem ser muito semelhantes.

6.2 Trabalhos Futuros

Uma das propostas de continuidade deste trabalho trata da aplicação do método para softwares com padrões arquiteturais diversos e a recuperação de diferentes tipos de visões arquiteturais, já que as visões recuperadas, muitas vezes, são dependentes de decisões de projeto às quais a implementação do software respeita.

Uma experiência de comparação entre o método proposto e os diversos métodos existentes na literatura também pode ser realizada a fim de mostrar a maior eficácia deste para a compreensão dos softwares legados que sofrerão manutenção. A mesma experiência pode ser realizada com softwares cuja documentação existe e está atualizada a fim de comparar a arquitetura recuperada com a real arquitetura do software e obter uma métrica de acurácia das visões recuperadas.

Desta forma, pode-se aperfeiçoar o método proposto para que este atenda com precisão os softwares desenvolvidos utilizando diferentes padrões arquiteturais e diferentes linguagens de programação. Por fim, uma ferramenta de recuperação de arquiteturas de softwares pode ser proposta com base no método proposto.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. Second Edition. Boston, MA, USA: Pearson Education Inc, 2003, 560p. ISBN: 0-321-15495-9.

BUSS, E.; DE MORI, R.; GENTLEMAN, M.; HENSHAW, J.; JOHNSON, H.; KONTOGIANNIS, K.; MERLO, E.; MULLER, H.; MYLOPOULOS, J.; PAUL, S.; PRAKASH, A.; STANLEY, M.; TILLEY, S.; TROSTER, J.; WONG, K. **Investigating Reverse Engineering Technologies: The CAS Program Understanding Project**. In: IBM Systems Journal. Volume 33, Issue 3. North York, ON, Canada. 1994. p. 477-500. ISSN: 0018-8670

CANFORA, G.; DI PENTA, M. **Frontiers of Reverse Engineering: a Conceptual Model**. Proceedings of the 2008 IEEE International Conference on Software Maintenance. Beijing, China. 2008. p. 38-47. ISBN: 978-1-4244-2614-0.

CLEMENTS, P. **Comparing the SEI's Views and Beyond Approach for Document Software Architectures with ANSI-IEEE 1471-2000**. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2005, 28 p. CMU/SEI-2005-TN-017.

CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R.; MERSON, P.; NORD, R.; STAFFORD, J. **Documenting Software Architectures: Views and Beyond**. Second Edition. Boston, MA, USA: Pearson Education Inc, 2010, 537p. ISBN: 0-321-55268-6.

GUO, G. Y.; ATLEE, J. M.; KAZMAN, R. **A Software Architecture Reconstruction Method**. Proceedings of the TC2 First Working IFIP Conference on Software Architecture. Deventer, The Netherlands: Kluwer, B.V., 1999, p. 15-34. ISBN: 0-7923-8453-9.

HOFMEISTER, C.; NORD, R.; SONI, D. **Applied Software Architecture**. Reading, MA, USA: Addison-Wesley Longman, Inc.2000. 432 p. ISBN: 0201325713.

IEEE. **IEEE Std 1219-1998, IEEE Standard for Software Maintenance**. Piscataway, NJ, USA: The Institute of Electrical and Electronics Engineers, 1998, 47p. ISBN: 0-7381-0336-5.

IEEE. **IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems**. Piscataway, NJ, USA: The Institute of Electrical and Electronics Engineers, 2000, 23p. ISBN: 0-7381-2518-0.

IEEE. **SWEBOK, Guide to the Software Engineering Body of Knowledge**. 2004 Version. Los Alamitos, CA, USA: The Institute of Electrical and Electronics Engineers, 2004, Chapter 6, p. 6-1–6-11. ISBN: 0-7695-2330-7.

IEEE. **ISO/IEC 14764 – IEEE Std 14764-2006, Standard for Software Engineering – Software Life Cycle Processes – Maintenance**. Second edition. Piscataway, NJ, USA: The Institute of Electrical and Electronics Engineers, 2006, 56p. ISBN: 0-7381-4960-8.

IEEE. **ISO/IEC 12207 – IEEE Std 12207-2008, Systems and Software Engineering – Software Life Cycle Processess**. Second edition. Piscataway, NJ, USA: The Institute of Electrical and Electronics Engineers, 2008, 124p. ISBN: 0-7381-5663-9.

KAZMAN, R.; CARRIÈRE, S. J. **Playing Detective: Reconstructing Software Architecture From Available Evidence**. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 1997, 51p. CMU/SEI-97-TR-010.

KAZMAN, R.; O'BRIEN, L.; VERHOEF, C. **Architecture Reconstruction Guidelines**. Third Edition. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2003, 42p. CMU/SEI-2002-TR-034.

KRUCHTEN, P. **The 4+1 View Model of Architecture**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, 9p. ISSN: 0740-7459.

KRUCHTEN, P. **The Rational Unified Process: An Introduction**. Third Edition. Boston, MA, USA: Pearson Education Inc, 2003, Chapter 5, p. 81–96. ISBN: 0-321-19770-4.

LI, Z. **Identifying High-Level Dependence Structures Using Slice-Based Dependence Analysis**. Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM). Edmonton, AB, Canada. 2009. P. 457-460. ISBN: 978-1-4244-4897-5.

LÖWE, W. et al. **Software Comprehension – Integrating Program Analysis and Software Visualization**. Växjö, Sweden: [s.n.], 2002, 11p. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.7624&rep=rep1&type=pdf> Acesso em: 17 jul. 2010.

O'BRIEN, L.; STOERMER, C. **Architecture Reconstruction Case Study**. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2003, 24p. CMU/SEI-2003-TN-008.

O'BRIEN, L.; STOERMER, C.; VERHOEF, C. **Software Architecture Reconstruction: Practice Needs and Current Approaches**. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2002, 38p. CMU/SEI-2002-TR-024.

ROZANSKI, N.; WOODS, E. **Software Systems Architecture**. Upper Saddle River, NJ, USA: Pearson Education Inc, 2005. 546 p. ISBN: 0-321-11229-6.

SEACORD, R. C.; PLAKOSH, D.; LEWIS, G. A. **Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices**. Boston, MA, USA: Pearson Education Inc, 2003. 352 p. ISBN: 0-321-11884-7

SOURCEFORGE **MemNotes 4.55 Project Page**. Disponível em: <http://sourceforge.net/projects/memnotes455>. Acesso em: 19 jan. 2011.

ANEXO A – Trecho do Código Fonte do Software MemNotes 4.55

Este anexo apresenta o arquivo *main.c* do código fonte do software MemNotes 4.55 (SOURCEFORGE, 2011).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
        ///additional includes
#include "version.h"          ///autoversion
#include "sayhi.c"                    ///std greentings
#include "defs.h"                     ///global defs
#include "core.h"                     ///basic operations
#include "ioshell.h"
#include "protoss.h"
#include "sfi.h"

///super global
char*ED=NULL;                 ///editor
short sw_ed=0;                ///editor defined
short op_autosave=0;          ///auto save base on change

/** /000////////////////////////////////////////////////////////////////////////////////
////      purpose:      command line parser
////      input:
////              int             argc    count of args
////              char** argv     array of args
////      result:
////              void
//// ////////////////////////*/
void clparse(int argc, char **argv)
{
        int i;

        printf("params parser:\n");
        for(i=0;i<argc;i++)
        {
                printf("%5d:\t[%s]\n",i,argv[i]);
        }
        printf("===end params\n\n");

        return;
}

/***2011-01-17*************************************************************
```

```
purpose:        read ini file, set options
input:
        void
result:
        void
***000**************************************************************************/
static void init(void)
{
        char*  s=NULL;
        char*  ss=NULL;
        char*  EDKEY="editor=";
        char*  ASKEY="autosave=";


        printf("...init...\n");
        if (fexist(IFN)>0)
        {
                printf("ini file exists\n");

                ss=m455finir(IFN,EDKEY);
                if(ss!=NULL)
                {
                        ED=strdup(ss);
                        printf("external editor [%s]\n\n",ED);
                        sw_ed=1;
                }
                ss=m455pcln(ss);

                if(m455finikeyex(IFN,ASKEY))
                {
                        ss=m455finir(IFN, ASKEY);
                        if(!strcmp(ss,"1"))
                        {
                                op_autosave=1;
                        }
                        ss=m455pcln(ss);
                }
                else
                {
                        m455finiw(IFN,ASKEY,"0");
                }

        }
        else
        {
                printf("ini file NOT exists\n");
                printf("please, define shell command to run external plaintext editor"
                                "\nor press enter to avoid...");
```

```
                s=m455fgetus(stdin,'\n');
                if(s!=NULL&&strlen(s)>0)
                {
                        printf("your external editor[%s]\n",s);
                        m455finiw(IFN,EDKEY,s);
                        ED=strdup(s);
                        sw_ed=1;
                }
                m455finiw(IFN,ASKEY,"0");
        }

        s=m455pcln(s);

        return;
}

int main(int argc, char **argv)
{
        ///greetings
        sayhello(PROGNAME);
        ///command line mess
        ///disabled due no command line keys present
//      clparse(argc, argv);
        ///ini reads
        init();
        ///running shell mode
        shell();
        ///free global
        ED=m455pcln(ED);

   return 0;
}
```