Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Explorando a Combinação de Visualização de Software com Clusterização de Dados em um Processo de Reconstrução de Arquitetura

Renato Edésio Rodrigues Paiva

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr.ª Genaína Rodrigues

Coorientador
Prof. Dr. Marcelo Ladeira

Brasília
2015

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

dP149e

de Paiva, Renato Edésio Rodrigues
Explorando a Combinação de Visualização de Software com Clusterização de Dados em um Processo de Reconstrução de Arquitetura / Renato Edésio Rodrigues de Paiva; orientador Genaína Rodrigues; co orientador Marcelo Ladeira. -- Brasília, 2015.
73 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2015.

1. Extração de Arquitetura de Software. 2. Visualização de Software. 3. Clusterização de Dados. I. Rodrigues, Genaína, orient. II. Ladeira, Marcelo, co-orient. III. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Explorando a Combinação de Visualização de Software com Clusterização de Dados em um Processo de Reconstrução de Arquitetura

Renato Edésio Rodrigues Paiva

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr.ª Genaína Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Rodrigo Bonifacio de Almeida
Universidade de Brasília

Prof. Dr. Alessandro Fabricio Garcia
Pontifícia Universidade Católica do Rio de Janeiro

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 05 de Outubro de 2015

Dedicatória

Dedico este trabalho à minha família, em especial aos meus pais pelo amor verdadeiro e incondicional.

Agradecimentos

A Deus por ter me dado saúde e força em minha trajetória acadêmica.
À professora e orientadora Dr.ª Genaína Rodrigues, pela orientação, paciência e confiança que tornaram possível a conclusão desta dissertação.
Ao professor e coorientador Dr. Marcelo Ladeira, pelos conselhos, críticas e direcionamentos que foram essenciais para conclusão deste trabalho.
À Indiara Ceciliano que sempre me apoiou nas horas difíceis.
Aos amigos Andrei Queiroz, Cleison Lucas e Riane Torres pelo apoio e incentivo.
A toda equipe do CPD-UnB.
Agradeço a todos que colaboraram direta ou indiretamente para a conclusão deste trabalho.

Resumo

Modernizar um sistema legado é um processo dispendioso, que requer profunda compreensão da arquitetura do sistema e de seus componentes. Sem um entendimento da arquitetura do *software* que será reescrito, todo o processo de reengenharia pode falhar. Quando há a ausência da documentação arquitetônica, faz-se importante um processo de recuperação de arquitetura que permita a compreensão completa do *software*. Tal processo envolve o mapeamento de entidades do código-fonte em modelos de alto nível. Trabalhos utilizando visualização de *software* e clusterização de dados para recuperação de arquitetura foram propostos e extensivamente utilizados. Entretanto, tem-se ainda um potencial de melhorias importantes que precisam ser abordados com base na referida temática. Assim, este trabalho propõe explorar se a aplicação em conjunto das técnicas de visualização e clusterização pode proporcionar uma maior precisão a um processo de recuperação de arquitetura de *software*. Um estudo experimental foi realizado para avaliar empiricamente a investigação. Os resultados indicaram um incremento estatisticamente significante na exatidão dos modelos produzidos quando utilizado as duas técnicas em conjunto.

Palavras-chave: Extração de Arquitetura de Software; Visualização de Software; Clusterização de Dados

Abstract

Modernizing a legacy system is a costly process that requires deep understanding of the system architecture and its components. Without an understanding of the software architecture that will be rewritten, the entire process of reengineering can fail. When there is absence of architectural documents, it is important to have a recovery process of architecture that allows the complete understanding of the software. Such process involves mapping of source code entities in high-level models. Previous work using visualization and clustering techniques has been proposed and extensively used. However, there is still important improvements that need to be addressed based on this theme. Thus, this work proposes to explore if an approach where visualization and clustering applied together can provide a higher accuracy on the software architecture recovery process. An experimental study was conducted to empirically evaluate our investigation. The results indicated a statistically significant increase in the accuracy of the models produced.

Keywords: Software Architecture Extraction; Software Visualization; Data Clustering

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CA Afferent Couplings. 21

CE Efferent Couplings. 22

MDG Model Dependency Graph. 26–30, 32, 53, 54, 64

MQ Modularization Quality. 28–30

NP Neighboring Partitions. 30

SAE Sistema de Assistência Estudantil. 48

SIEX Sistema de Extensão. 48

SIGRA Sistema de Informação Acadêmica de Graduação. 48

SISRU Sistema do Restaurante Universitário. 48

UML Unified Modeling Language. 9

Capítulo 1

Introdução

Um dos primeiros passos para a construção de um entendimento de um sistema por meio do seu código-fonte é a compreensão de como este se encontra organizado em termos arquiteturais, pois, sem um entendimento da arquitetura, o sistema não pode ser analisado ou compreendido com confiança [18]. O processo de recuperação de arquitetura envolve a extração de informações relevantes ao procedimento de abstração e de descrição dos elementos arquiteturais [1]. O resultado do processo de recuperação é um documento descrevendo as visões arquiteturais implementadas pelo sistema.

Nesse contexto, uma técnica que tem sido amplamente utilizada para recuperação de arquitetura é a técnica de visualização de *software* [13, 45, 21, 11]. A visualização de *software* é uma técnica semiautomática que permite a identificação de elementos arquiteturais por meio da obervação de representações do sistema através de objetos visuais [45]. Tais objetos visuais podem representar módulos, componentes ou um comportamento em tempo de execução. As representações podem ocorrer por meio de diagramas 2D e 3D, grafos de dependência e matrizes de dependências estruturada. Como resultado da abstração de tais artefatos, é possível obter uma percepção sobre como o *software* encontra-se estruturado, facilitando o entendimento da arquitetura do sistema. Contudo, a acurácia dos resultados obtidos por meio uma técnica semiautomática está, geralmente, subjetiva a habilidade do analista responsável pela extração da arquitetura.

Assim, diversos trabalhos de pesquisa surgiram com o intuito de automatizar o processo de recuperação de arquitetura[48]. Nesse sentido, abordagens de recuperação de arquitetura utilizando clusterização de dados têm um notável destaque [41, 14, 49, 28]. A clusterização é uma técnica automática para recuperação de arquitetura cujo o objetivo é identificar grupos similares pertencem a uma instância de um subconjunto similar [9].Tais técnicas são utilizadas em várias áreas de pesquisa para descrever métodos para agrupar dados não classificados. No contexto de recuperação de arquitetura, a clusterização busca agrupar artefatos de *software* em módulos significativos, por meio da análise de dependên-

cias extraídas do código-fonte, como, por exemplo, a chamada de funções, dependências de classes, pacotes, interfaces etc. Contudo, decompor um sistema de *software* em estruturas de alto nível, com um agrupamento lógico coerente, é uma tarefa inerentemente difícil. Uma vez que, devido ao conjunto esparso de dados existentes em um código-fonte, e aos fatores computacionais, muitas soluções encontradas são apenas sub-ótimas, e não representam a real arquitetura do sistema [28].

Em um estudo recente, Lutellier *et al.* [24] apontaram para a baixa acurácia dos modelos obtidos em um processo de recuperação de arquitetura, independente da técnica de recuperação utilizada. Nesse sentido, uma pergunta surge: quando combinado a utilização de diferentes técnicas para extração de arquitetura, é possível obter um processo de recuperação de arquitetura mais preciso? Neste trabalho, propomos investigar essa questão unindo as técnicas de visualização de *software* e clusterização em um único processo de recuperação de arquitetura. Nossa investigação é realizada por meio de um experimento controlado em um ambiente industrial, onde os participantes são profissionais com experiência em análise e desenvolvimento sistemas. Resultados mostraram um aumento estatisticamente significativo na precisão dos modelos produzidos quando utilizado as duas técnicas em conjunto.

1.1 Motivação

Sistemas legados são os sistemas computacionais construídos para atender a uma necessidade funcional dentro de uma organização. São ativos críticos para empresas modernas e incorporam conhecimentos-chave adquiridos ao longo do tempo. Tais sistemas foram continuamente atualizados para refletir a evolução das práticas de negócio [3]. Contudo, as modificações nestes sistemas têm um efeito acumulativo em sua complexidade, e a evolução tecnológica rapidamente os torna obsoletos.

A manutenção de sistemas legados, além de dispendiosa, envolve um alto risco, o que se deve à dificuldade em entender o sistema. Segundo Ganesan e Lindvall [12], grande parte do tempo de manutenção de um sistema é gasto para entender seu funcionamento. Uma das principais razões para o alto custo se dá pela falta de documentação do sistema legado. Sem uma documentação adequada, a única fonte de informação disponível – ou confiável – é seu código-fonte, o que demanda grandes investimentos para análise e compreensão dos códigos concernentes. Porém, é uma tarefa árdua construir um entendimento de todo o sistema tendo como técnica a leitura manual do código-fonte. Assim, técnicas da engenharia reversa devem ser utilizadas para reconstruir a estrutura arquitetural de sistemas de *software*.

Embora a reconstrução da arquitetura possa se concretizar por meio de qualquer recurso disponível, tais como: documentação do sistema, entrevistas com *stakeholders* ou especialistas no domínio, o local mais confiável para se obter informações é o próprio código-fonte do sistema [46]. Este fornece uma enorme quantidade de informações que estão espalhadas em um grande número de arquivos e diretórios. O desafio é encontrar um ponto de partida para a análise e, então, filtrar as informações sem perder nada de importante que explica como o código soluciona uma preocupação específica [12]. Ao extrair tais elementos, têm-se informações suficientes para construir modelos arquiteturais mais precisos.

Contudo, o processo para a recuperação de arquitetura não é algo trivial. Muitas vezes se dá de maneira *ad hoc*, fazendo uso de ferramentas limitadas e de grande quantidade de interpretação por parte de analistas [46]. Assim, o resultado pode ser uma má compreensão do sistema, o que acarreta uma especificação incorreta dos requisitos para o novo sistema e, consequentemente, a criação de um projeto falho, com riscos concretos de fracasso [3]. Por isso, estudos devem ser realizados para efetivamente utilizar técnicas de reengenharia de *software*, como clusterização e visualização de *software*, para tornar o processo de recuperação de arquitetura mais ágil e preciso.

1.2 Objetivo Geral

Este trabalho destina-se a explorar a combinação das técnicas de visualização de *software* e clusterização de dados em um processo de recuperação de arquitetura.

1.3 Objetivos Específicos

Em vistas a atingir o objetivo geral deste trabalho, propõem-se os seguintes objetivos específicos:

- Identificar os conceitos essenciais a um processo de recuperação de arquitetura.
- Descrever o uso de técnicas de visualização de *software* no contexto de recuperação de arquitetura.
- Descrever o uso de técnicas de clusterização de dados para recuperação de conceitos arquiteturais.
- Identificar formas para unir os resultados apresentados pelas técnicas de visualização de *software* e clusterização de dados para produção de um único modelo arquitetural.

- Avaliar empiricamente o uso das técnicas de clusterização de dados e visualização de *software* por meio da condução de um experimento controlado.

1.4 Contribuições

O estudo dos métodos para recuperação de arquitetura é um campo de pesquisa promissor, tendo em vista a dificuldade no entendimento de sistemas computacionais a partir do código-fonte. Enquanto vários métodos e ferramentas de suporte correspondentes para a reconstrução de arquitetura de sistema foram propostas [4, 44, 12, 46, 18], tem-se ainda um potencial de melhorias importantes que precisam ser abordados com base na referida temática [22]. Assim, o presente trabalho espera contribuir com a exploração das técnicas de clusterização de dados e visualização de *software*, aplicadas em conjunto, em um processo de recuperação de arquitetura; apresentamos o *design*, execução e principais conclusões de um estudo empírico que investiga os benefícios de combinar as referidas técnicas. O estudo reporta um acréscimo dos modelos resultantes na razão de 19% a 30% quando utilizado ambas as técnicas em comparação com a utilização de uma única técnica.

1.5 Estrutura do Documento

Esta dissertação foi estruturada da seguinte forma:

- O Capítulo 1 apresenta os fundamentos e motivações que objetivaram o desenvolvimento do trabalho.
- O Capítulo 2 introduz os fundamentos da recuperação arquitetura de sistemas de *software*, bem como apresenta técnicas de visualização de *software* e informações a cerca do uso da clusterização de dados em um processo de recuperação da arquitetura.
- O Capítulo 3 aponta estratégias para unir os resultados das técnicas de visualização de *software* e clusterização de dados em um único modelo arquitetural.
- O Capítulo 4 detalha um estudo experimental conduzido com analistas de sistemas do Centro de Informática da Universidade de Brasília.
- O Capítulo 5 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Reconstrução de arquitetura de software

O processo de reconstrução da arquitetura de um *software* gera uma representação que pode ser utilizada de diversas maneiras. Sua ação principal dá-se na documentação da arquitetura de um sistema existente [19]. Tal documento é a base para o processo de reengenharia de um sistema legado, sendo fundamental para o sucesso do projeto de modernização. É um processo interpretativo e interativo que envolve muitas atividades; não é automático [2].

O objetivo de uma reconstrução de arquitetura é identificar abstrações que representam visões ou elementos arquiteturais de um *software*. Neste contexto, duas fontes de informações são as mais consideradas, quais sejam: a expertise humana e os artefatos de *software* [9].

A expertise humana é primordial para um processo de recuperação de arquitetura. O conhecimento dos objetivos de negócio, dos requisitos, das arquiteturas de referência ou das restrições de projeto é importante para um processo de recuperação de arquitetura [9]. Os engenheiros de *software* devem estar familiarizados com técnicas de compiladores e diferentes ambientes de execução de um sistema para realizar um processo de recuperação de arquitetura [19]. No entanto, Ducasse e Pollet [9] apontam diversos problemas quando se leva em conta fatores humanos e empresarias no processo de recuperação, a saber:

1. Devido à alta taxa de rotatividade entre os especialistas e a falta de documentação atualizada, a arquitetura conceitual com base nos conhecimentos dos especialistas apresenta-se, em geral, obsoleta, inexata, incompleta ou em um nível de abstração inadequada.

2. Ao reconstruir uma arquitetura, os *stakeholders* do sistema apresentam várias preocupações, tais como: desempenho, confiabilidade, portabilidade ou reutilização.

Um processo de recuperação de arquitetura deve oferecer suporte a múltiplas visões e atender todas as preocupações dos *stakeholders*.

3. Muitas vezes, a engenharia reversa se perde na complexidade do *software*. Logo, o processo de recuperação não pode ser realizado em uma única vez, mas sim, de forma interativa e parametrizada.

Outra fonte de informação necessária a um processo de recuperação de arquitetura são os artefatos do *software*, como, por exemplo, os arquivos fontes do sistema. Estas são as poucas fontes confiáveis de informação que contém a real arquitetura de um sistema [9]. Porém, devido à complexidade dos códigos-fontes, algumas ferramentas são utilizadas para a extração de informação do sistema, a fim de colaborar na construção de sucessivos níveis de abstração [19]. Assim, reconstruir a arquitetura do código-fonte levanta vários problemas, devidamente apresentados por Ducasse e Pollet [9] , conforme se segue:

1. A abordagem deve ser escalável para lidar com grandes quantidades de dados.
2. Os sistemas de *software* podem ser grandes e complexos. Logo, o processo de reconstrução de arquitetura deve contemplar diferentes linguagens e tecnologias que são heterogêneas e intercaladas.
3. Uma arquitetura não é explicitamente representada no nível do código-fonte. Logo, é uma tarefa difícil identificar elementos do código-fonte que ajudem a recuperar a arquitetura do sistema.
4. Muitas vezes, tem-se a necessidade de extração de informações dinâmicas para entender como o sistema está funcionando, para, então, identificar como surgem os aspectos comportamentais na arquitetura.

Para mitigar tais problemas no decorrer do processo de recuperação de arquitetura, Kazman, O'Brien e Verhoef [19] destacam várias recomendações, quais sejam:

- Ter metas e objetivo antes de iniciar um processo de reconstrução de arquitetura. Na ausência destes aspectos, grandes esforços poderiam ser gastos para extrair informações arquiteturais que podem ser inúteis para qualquer propósito.
- Inicialmente, obter uma visão de alto nível da arquitetura do sistema para, em seguida, dar início a um processo detalhado de reconstrução. Tal visão irá orientar o processo de extração, ajudando a identificar as informações que necessitarão de refinamento.

- Quando da existência de alguma documentação da arquitetura do sistema, faz-se importante utilizá-la somente para o entendimento de alto nível do sistema. Em muitos casos, a documentação existente pode não refletir com precisão o sistema.
- Envolver os indivíduos familiarizados com o sistema desde o início do projeto. Tal fato permitirá a obtenção de melhor compreensão do sistema que está sendo reconstruído. As ferramentas podem apoiar o esforço de reconstrução e encurtar o processo de reconstrução, mas não podem realizar automaticamente uma reconstrução completa.
- Apontar determinado indivíduo para trabalhar na arquitetura do projeto de reconstrução em tempo integral. A arquitetura de reconstrução envolve uma análise extensiva e detalhada de um sistema e exige um esforço significativo.

Ducasse e Pollet [9] propuseram uma classificação com base no ciclo de vida de um processo de recuperação da arquitetura. Esta aponta para trabalhos relacionados à visualização de *software*, padrões de projetos e extração de funcionalidades, conforme expresso na Figura 2.1.

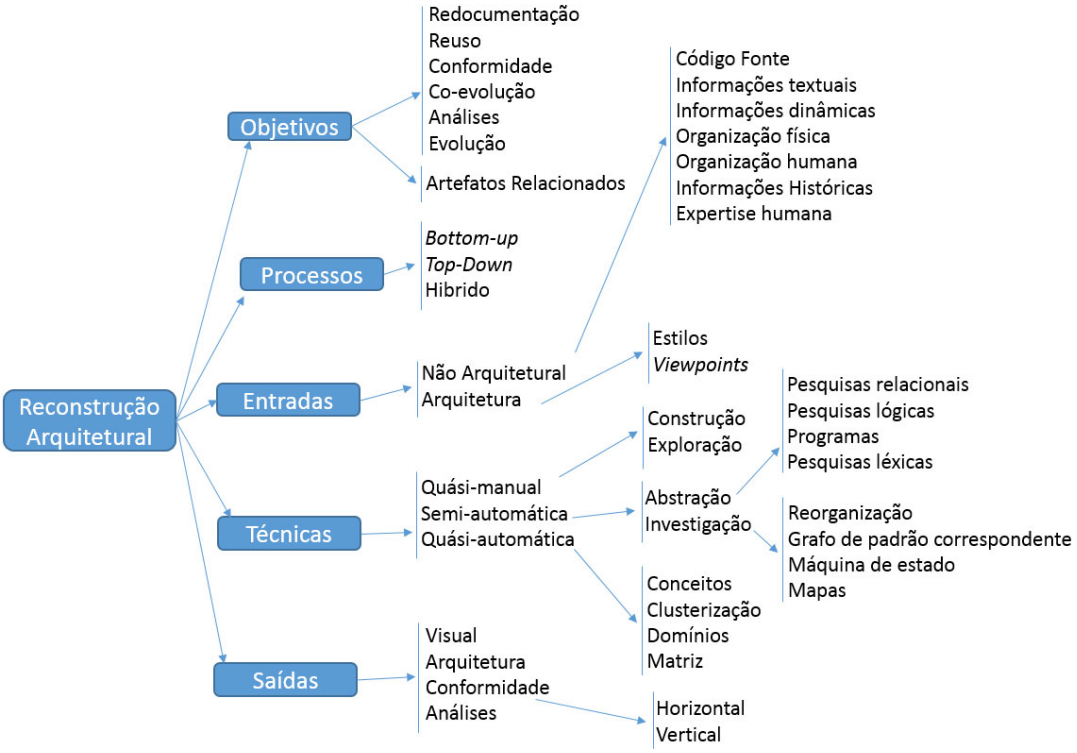

Figura 2.1: Taxonomia para reconstrução de arquitetura de *software*. Fonte: adaptado de Ducasse; Pollet, 2009.

2.1 Processos de reconstrução de arquitetura

Vários processos são utilizados para a criação de abstrações de alto nível de artefatos de *software* já existentes. Neste sentido, Ducasse e Pollet [9] assim classificaram tais processos: *bottom-up*, *top-down* ou híbrido.

2.1.1 Processo bottom-up

No processo de recuperação *bottom-up*, determinadas visões são extraídas dos elementos de baixo nível (código-fonte) para dar suporte à criação de modelos em alto nível. A Figura 2.2, ilustra tal processo.

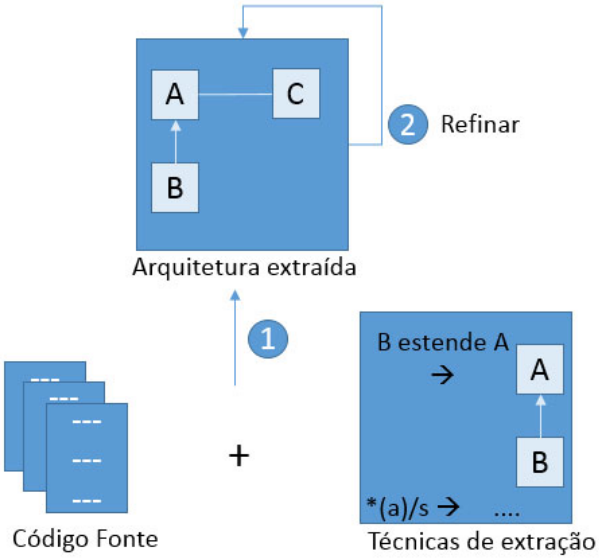

Figura 2.2: Processo de reconstrução *bottom-up*. Fonte: adaptado de Ducasse; Pollet, 2009.

A extração da arquitetura através da análise do código-fonte pode ser realizada através de análise estática do código ou da análise dinâmica do código.

A análise estática do código é um dos métodos utilizados frequentemente pela literatura para recuperar informações sobre um sistema de *software*; resulta em um modelo que contém entidades do sistema e seus relacionamentos [42]. A análise estática faz uso de determinadas técnicas para a extração de informações do código-fonte sem a necessidade de execução do sistema. Tal análise pode ser feita com o auxílio de ferramentas específicas ou com técnicas*ad hoc* [46]. Kazman, Brien e Verhoef [19] apontam um aparato de técnicas para extração de informações estática do código fonte, quais sejam:

- Análise Sintática.

- Árvore Sintática Abstrata.
- Analisadores léxicos.

Já a análise dinâmica faz uso do sistema em tempo de execução para a recuperação de modelos arquiteturais. Também a extração pode fazer uso das ferramentas específicas para a extração de arquiteturas ou via técnicas *ad hoc*. Neste sentido, Riva [35] destaca algumas técnicas para extração dinâmica do código, quais sejam:

- Instrumentação de código.
- Simulação de um conjunto de cenários.
- Coleta de informações de rastreabilidade.

Um exemplo de processo *bottom-up* é o apresentado por Riva [35], onde se tem um processo interativo e incremental que pode ser resumido em quatro fases, a saber:

1. Recuperação de conceitos significativos da arquitetura que constrói o sistema através da análise do código-fonte.
2. Criação dos modelos do sistema cujas entidades são instâncias dos conceitos identificados na fase anterior.
3. O modelo extraído da última fase é uma representação de baixo nível. Neste sentido, é preciso enriquecer o modelo com o conhecimento do domínio específico, resultando em uma visão de alto nível do sistema.
4. Os modelos extraídos são representados em diferentes formas, tais como: grafos hierárquicos, documentos web, grafos em UML e diagramas lógicos de sequência ou de mensagem.

2.1.2 Processo top-down

No processo de recuperação *top-down*, uma hipótese da arquitetura é definida para, em uma fase posterior, ser comparada com o código-fonte, conforme ilustração da Figura 2.3.

Arias *et al.* [4] fazem uso deste conceito para a criação de um processo de recuperação de arquitetura. O processo tem início com um grupo de especialistas do domínio que identificam os conceitos-chave que podem ser resolvidos através da construção de visões de execução, para, então, ocorrer a seleção dos tipos de modelos que podem ser construídos. Com o plano de construção em mãos, faz-se a identificação e interpretação de informações extraídas do sistema através de técnicas de análise dinâmica de código para construção de um modelo de execução com base no tipo de informação extraída.

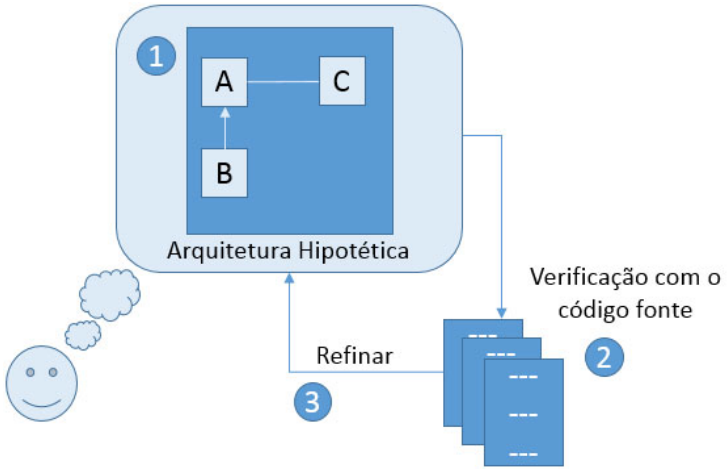

Figura 2.3: Processo de reconstrução *top-down*. Fonte: adaptado de Ducasse; Pollet, 2009.

2.1.3 Processo híbrido

O processo de extração híbrido combina processos de recuperação *bottom-up* com *top-down*. Modelos de baixo nível são extraídos, fazendo uso de várias técnicas de extração, para serem comparadas com modelos de alto nível já construídos, conforme evidencia a Figura 2.4. Em suma, os processos híbridos utilizam modelos arquiteturais conceituais, que são modelos definidos antes ou no decorrer do processo de desenvolvimento do *software*, para serem comparados com modelos de arquitetura concretos extraídos do código fonte do sistema já implementado [9]. Com ambos os modelos, é possível refinar os documentos de arquitetura, garantindo uma maior fidedignidade ao novo modelo de arquitetura.

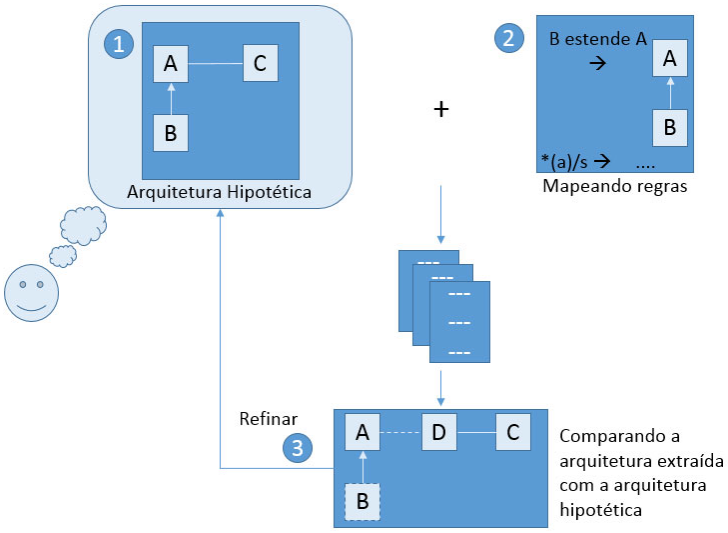

Figura 2.4: Processo de reconstrução híbrido. Fonte: adaptado de Ducasse; Pollet, 2009.

2.2 Recuperando arquitetura utilizando Visualização de Software

Segundo Price, Small e Baecker [34], a visualização de *software* facilita o entendimento através do uso de programas de computadores com base em *design* gráfico, animação, cinematografia e computação gráfica interativa. Assim, a visualização de *software* é o uso de representações visuais com via de melhoria da compreensão de diferentes aspectos do sistema [13].

A necessidade da utilização de técnicas de visualização de *software* torna-se particularmente evidente quando o sistema de *software* é complexo e possui um grande número de módulos e procedimentos relacionados entre si [13]. Neste contexto, a visualização de *software* emprega diferentes técnicas para representar graficamente diversos aspectos do sistema, tais como: arquitetura, evolução, execução, processo de desenvolvimento e código-fonte [40].

O resultado do uso de uma ferramenta de visualização de *software* é uma representação de alto nível do código-fonte do sistema. Tal representação se dá de diversas maneiras, dependendo da ferramenta utilizada. O objetivo da obtenção de modelos de alto nível do sistema é facilitar a visualização da estrutura do código, isto é, a forma como seus componentes, classes e métodos se comunicam, permitindo, assim, o entendimento e a recuperação da arquitetura.

É importante ressaltar que, para a recuperação de arquitetura de um sistema de *software*, uma única representação do código-fonte não é suficiente. Logo, o objetivo da utilização de ferramentas de visualização de *software* é recuperar o maior número de informações possíveis sobre o código-fonte, isto é, vários modelos de alto nível do sistema. Assim, a representação em modelos de alto nível do código deve cobrir toda a extensão do sistema, em diversos níveis de granularidade, como, por exemplo, às relações estruturais (pacotes, módulos, subsistemas) e relações de comportamento (métodos, funções, variáveis).

2.2.1 Visualização de arquitetura de *software*

Compreender a arquitetura de *software* é um passo vital para a construção e manutenção de sistemas de *software*. Porém, a arquitetura de *software* é uma entidade conceitual intangível. Logo, é difícil de compreender uma arquitetura de *software* sem um mapeamento visual que reduz a carga sobre o cérebro humano. A visualização da arquitetura de *software* tem sido um dos temas mais importantes na visualização de *software* [13].

As ferramentas de visualização de arquitetura podem ser classificadas de acordo com a dimensionalidade dos elementos gráficos, como, por exemplo, a visualização 2D e a visualização 3D. A diferença entre as visualizações 3D e 2D é a forma de representação gráfica e, não necessariamente, a dimensionalidade dos dados, uma vez que uma representação 2D de uma arquitetura pode ser capaz de representar mais de duas dimensões de um conjunto de dados [13].

Visualização 2D

A representação do *software* através de imagens e diagramas em duas dimensões é uma maneira simples e didática de analisar grandes quantidades de código. A visualização de *software* em duas dimensões provê perspectivas complementares sobre o sistema analisado. Através desta, é possível explorar a arquitetura do *software* por meio de uma representação com base em diagramas "caixa-linha", podendo, por exemplo, representar interações entre os módulos no sistema. Tal representação pode combinar métricas para incrementar o nível de informação de cada diagrama [23].

E ainda, tais representações podem conceber as visões da arquitetura de diferentes maneiras. Neste contexto, Ghanam *et al.*[13] identificaram duas escolas de pensamentos: a primeira aponta que qualquer ferramenta de visualização de arquitetura deve suportar múltiplas visões em diferentes níveis de detalhes, ou seja, para a visualização ser considerada útil, esta deve fornecer um modo de visualização dos diferentes aspectos de uma arquitetura de *software* através de distintos pontos de vista. Assim, se uma exibição fornece uma visão sobre a estrutura interna das entidades de *software*, outra exibição deve fornecer uma visão das relações e comunicações entre as referidas entidades. A segunda escola de pensamento tem como foco uma visão única e cuidadosamente projetada, assim, mais eficaz e significativa na transmissão dos vários aspectos da arquitetura. Deste modo, cabe ao analista desenhar seus próprios mapas mentais no nível que melhor atenda seus objetivos.

Lanza em [21] destaca uma ferramenta que foca na visualização da arquitetura utilizando a representação gráfica 2D. Tal ferramenta apoia a engenharia reversa através de combinações de métricas e técnicas de visualização de *software*. Aquele autor faz uso dos conceitos de visões polimétricas para facilitar o entendimento do *software*. Neste tipo de visualização, é possível ter noções de como, quando e qual métrica uma classe específica foi representada. A Figura 2.5 , evidência a tela principal da ferramenta CodeCrawler.

Na ferramenta CodeCrawler, a arquitetura é representada através de nós (entidades) e conexões (relacionamentos). Os nós representam artefatos de *software* concretos e abstratos. Os artefatos concretos podem ser localizados no código-fonte e incluem classes, métodos, funções, pacotes etc. Já os artefatos abstratos representam módulos de *soft-*

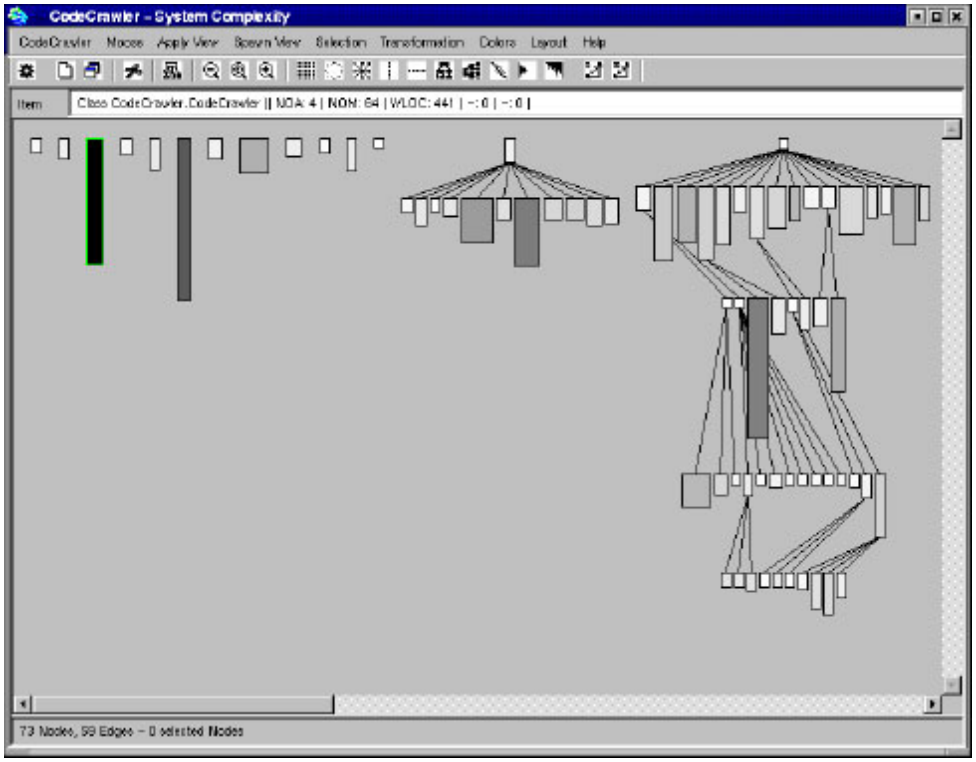

Figura 2.5: *Print Screen* da Ferramenta CodeCrawler. Fonte: LANZA, 2003.

ware idealizados pelos desenvolvedores, e não necessariamente são encontrados no código. As conexões representam os relacionamentos entre os artefatos de *software*, como, por exemplo, a dependência entre dois subsistemas [21].

Visualização 3D

O uso de apenas duas dimensões para representar dados altamente dimensionais pode dificultar o entendimento [13]. As abordagens de visualização 3D tentam criar visualizações que estão mais perto de metáforas do mundo real, melhorando a utilização do espaço ao adicionar uma dimensão extra. O usuário é capaz de girar e mover objetos e navegar dentro de um ambiente 3D [45]. A visualização 3D também oferece novas opções para visualização de uma determinada visão arquitetural, permitindo a ampliação (*zoom-in*) e visualização de vários ângulos dos componentes. As formas clássicas de visualização das visões, como, por exemplo, a manipulação algébrica das relações e a relação entre nós ainda estão disponíveis, mas novos mecanismos podem ajudar a facilitar tal exploração [11].

A visualização em 3D ajuda a identificar aspectos importantes do sistema em análise. A representação de componentes do *software* através do uso de metáforas, como, por exemplo, blocos, prédios e casas, provê novas perspectivas sobre o *software*. Formas bem

Pontos Fortes	Pontos Fracos
Maior densidade de informação.	Computação intensiva.
Integração de visões locais com visões globais.	Implementação mais complexa.
Composição de visões 2D com uma única visão 3D.	Adaptação do usuário a metáforas 3D e dispositivos especiais.
Facilidade da percepção visual do sistema.	Dificuldade do usuário em entender os espaços 3D e executar ações ali exigidas.

Tabela 2.1: Pontos fortes e fracos das ferramentas de visualização de software 3D.

escolhidas podem ajudar os programadores e arquitetos a identificar estruturas complexas. Tal apelo visual pode ser utilizado para fins de apresentação, visando explicar conceitos da arquitetura de um produto a potenciais clientes [11].

Existem diferentes formas de representar o mapeamento entre os elementos do código-fonte em metáforas visuais. Os elementos arquiteturais podem ser representados através de elementos de visualização abstratos, como, por exemplo, polígonos, esfera e outras formas de representação 2D ou 3D. Tem-se ainda a utilização de metáforas visuais reais, como, por exemplo, construções (casas e prédios), cujas características físicas representam os atributos das entidades de *software* [13].

A Tabela 2.1 destaca pontos fortes e fracos das ferramentas de visualização de software 3D. Por um lado, a visualização 3D permite uma maior interação entre usuário e representações do sistema, porém isso exige uma maior adaptação do usuário. A maioria dos usuários só tem experiência com janelas clássicas, ícones, menus etc. Por conseguinte, a interação com objetos 3D podem exigir consideráveis esforços de adaptação àquelas tecnologias [45].

Uma ferramenta de suporte à visualização de arquitetura 3D é apresentada por Feijs e Jong [11]. Denominada como ArchView, faz-se uso da visualização 3D para agrupar módulos de um designer em um número de camadas conforme algumas partições livremente selecionáveis de conjunto de nós. Cada camada imediatamente superior é posicionada de forma equidistante acima da camada anterior. Através da ferramenta é possível visualizar as relações entre os nós, demarcados através de uma seta de uma determinada cor ou através de pequenos tubos com um cone no meio. Assim, é possível visualizar várias relações fazendo uso de cores diferentes para cada relação. Um exemplo das relações na ferramenta entre módulos se dá na Figura 2.6.

2.2.2 Visualização do código-fonte

Algumas abordagens focam na visualização do código-fonte, ao invés de visualização de entidades arquiteturais [9]. Logo, faz-se necessária uma análise dos artefatos produzidos

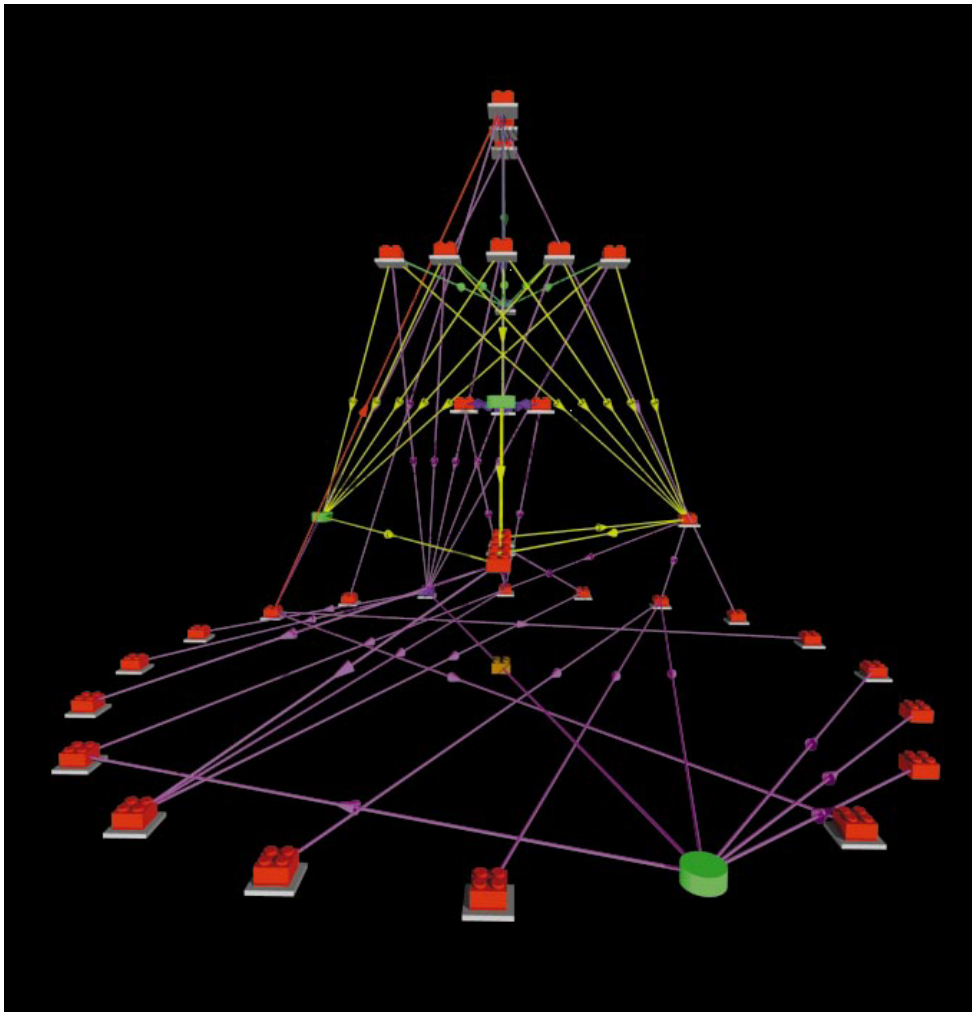

Figura 2.6: *Print Screen* da Ferramenta ArchView. Fonte: FEIJS; JONG, 1998..

por tais visualizações para identificar os componentes da arquitetura do *software*. Estes artefatos podem ser mapas de *software*, grafos, matriz de dependências estruturadas etc [31].

Visualização do código-fonte por meio de grafos

A teoria do grafo – que estuda as propriedades de um grafo – tem sido amplamente aceita como um tema de pesquisa entre os cientistas da computação, uma vez que o estudo das propriedades do grafo pode ser valioso para a compreensão das características de sistemas de *software* [6].

Os grafos são uma maneira conveniente para descrever diferentes tipos de dados, funções, variáveis, estrutura de controles, arquivos e *bugs* no código-fonte [10]. A representação do sistema através de grafos ajuda a entender como as conexões entres as entidades participantes da estrutura do sistema estão retratadas. O ideal é modelar alguns aspectos

do *software* e apresentar o grafo como um modelo de alto nível que facilita o entendimento do sistema.

Existem diversos tipos de grafos. Alguns deles são apresentados na Figura 2.7. Em um grafo, nós podem representar entendidas do sistemas, e as arestas podem representam a estrutura ou dependências entre o sistema. Como exemplo, para sistemas orientado a objetos, os nós podem representar pacotes, entidades, classes, membros de classes ou unidades do código declarados dentro de métodos. Já as arestas podem representar o relacionamento entre os componentes de *software*, quais sejam: pacote contém classes, classes contém métodos, métodos contém variáveis etc [38].

Figura 2.7: Exemplo de diversos tipos de grafos existentes .

Os grafos têm todas as características necessárias para representar componentes do *software*, ao considerar itens no *software* como nós e relacionamento como arestas. No entanto, visualizar todos os relacionamentos em um *software* pode ser equivalente a visualizar um grafo muito complexo, com diversas interconexões, especialmente em grandes aplicações de *software*. O resultado visual desta representação pode ser confuso, com nós e arestas se sobrepondo, o que dificulta a análise do grafo [5].

Visualização do código-fonte por meio de matriz de dependência estruturada

A matriz de dependência estruturada foi criada para aperfeiçoar os processos de desenvolvimento de produtos [37]. Estas são utilizadas para mapear um conjunto de itens em relação a si mesmo (NxN) ou para mapear um conjunto de itens em relação a outro conjunto de itens (NxP) [7]. A matriz de dependência estruturada é útil para analisar as propriedades de aplicações complexas. Em uma análise de arquitetura, os acoplamentos entre os modelos arquiteturais são descritos e várias operações sobre a matriz podem ser realizadas com objetivo de identificar relações arquiteturais [**?**].

O grafo é uma forma intuitiva de mostrar as dependências, mas pode ser totalmente incompreensível quando o número de nós e arestas crescem; a matriz de dependência é menos intuitiva, mas pode ser mais eficiente para representar um grafo grande e complexo.

As informações capturadas em uma análise de uma matriz de dependência estruturada são semelhantes ao de um grafo direcionado. No entanto, a representação matricial torna possível a criação de um modelo mais abrangente do fluxo de informações e facilita a análise de interdependência em projetos complexos [7]. Na Figura 2.8, é possível visualizar um grafo complexo representado através de uma matriz de dependência estruturada.

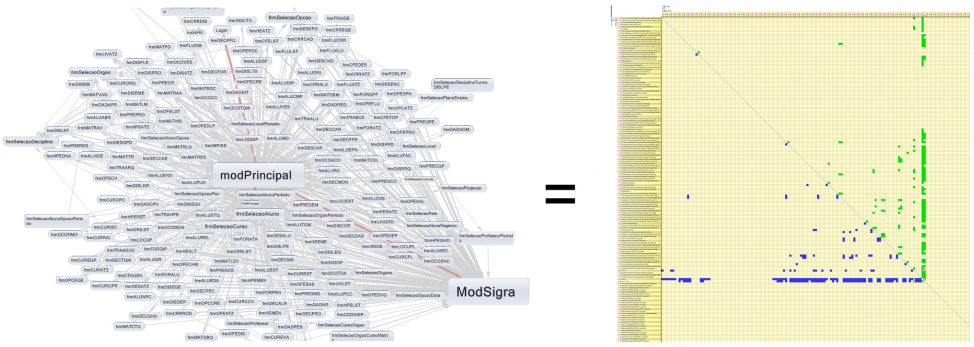

Figura 2.8: Grafo complexo representado através de uma matriz de dependência estruturada.

Diante do exposto, a matriz de dependência estruturada é uma forma compacta de representar as dependências entre os componentes. Esta representa a mesma informação de um grafo. Cada nó em um grafo é um cabeçalho na matriz; as setas de ligação no grafo são representadas por células não vazias na matriz. A Figura 2.9, evidencia detalhadamente como o grafo de fluxo de informações pode ser representado em um uma matriz de dependências.

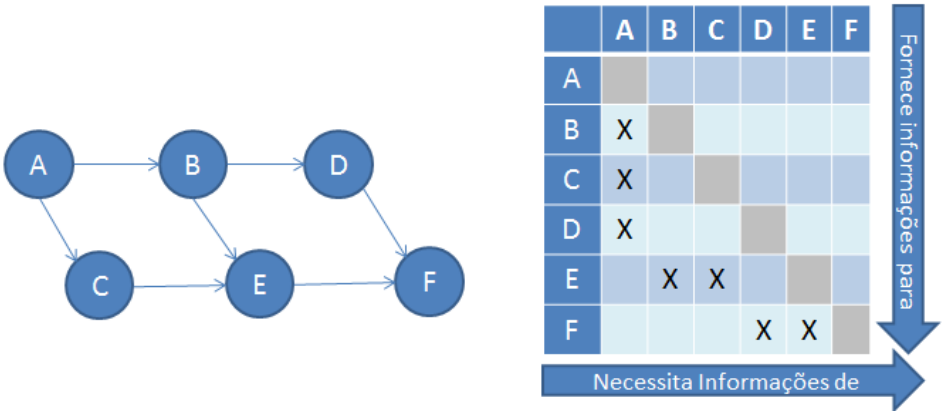

Figura 2.9: Grafo representado através de uma matriz de dependência estruturada. Fonte: adaptado de Danilovic; Sandkull (2005).

Para Jordan, Sinha e Jackson [37], o uso de matriz de dependências estruturais no contexto de Engenharia de *Software* traz diversas vantagens, quais sejam:

- Uma representação em matriz escala melhor a representação do *software* do que um grafo de dependências ou outros diagramas linha-caixa.
- Facilita a identificação de camadas no *software* e destaca dependências cíclicas.
- Algoritmos de agrupamento fornecem um mecanismo automático para descobrir a arquitetura em uma grande base de código, uma vez que o agrupamento elimina ciclos através da formação de subsistemas.

Várias operações sobre uma matriz de dependências podem ser utilizadas para facilitar a leitura e o entendimento de matrizes complexas. Uma destas operações é a de agrupamento, que consiste em agrupar tarefas similares com o intuído de reduzir a complexidade de uma matriz. Tendo como base a matriz da Figura 2.10(a), se A e C são consideradas como um agrupamento de tarefas, tal ciclo pode ser eliminado. O resultado mostra-se evidente na Figura 2.10(b), com as tarefas compostos indicados pelo sombreado [37].

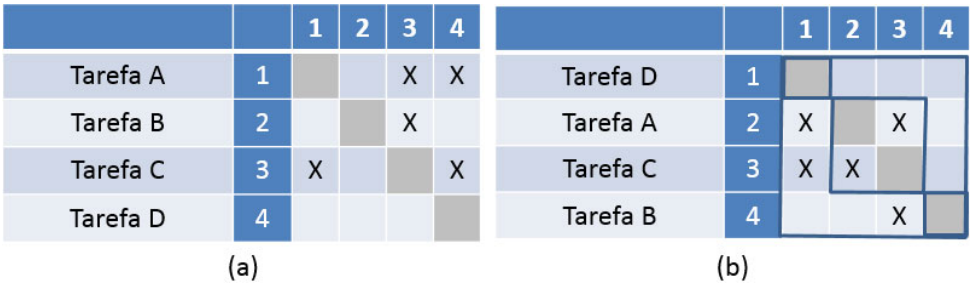

Figura 2.10: Operação de agrupamento em uma matriz de dependência estruturada. Fonte: adaptado de Jordan; Sinha; Jackson (2005).

A matriz de dependência pode facilmente revelar um padrão arquitetural de um sistema existente [37]. Tem-se na Figura 2.11, um exemplo de um sistema em camadas. Neste sentido, é possível notar na referida figura que cada camada depende somente de uma camada abaixo.

As violações arquiteturais também podem ser encontradas a partir da observação da matriz. Conforme evidenciado na Figura 2.12, a violação arquitetural se dá na coluna 5, uma vez que uma camada inferior utiliza uma camada superior: no caso, a camada "Util" faz uso da camada de "Aplicação". [37].

A matriz de dependência também permite a visualização e avaliação do *software* em termos de acoplamento e coesão. Módulos com alta coesão são representados através da agregação dos quadrados ao redor da diagonal. Este tipo de agrupamento indica que tais módulos são fortemente dependentes uns dos outros. Por outro lado, o fato de que a maioria das células ao redor da matriz encontra-se vazia indica um baixo acoplamento

		1	2	3	4	5
Aplicação	1					
Modelo	2	37				
Domínio	3	17	29			
Framework	4	75	53	42		
Util	5	10	13	16	13	

Figura 2.11: Exemplo de camada em uma matriz de dependência estruturada. Fonte: adaptado de Jordan; Sinha; Jackson (2005).

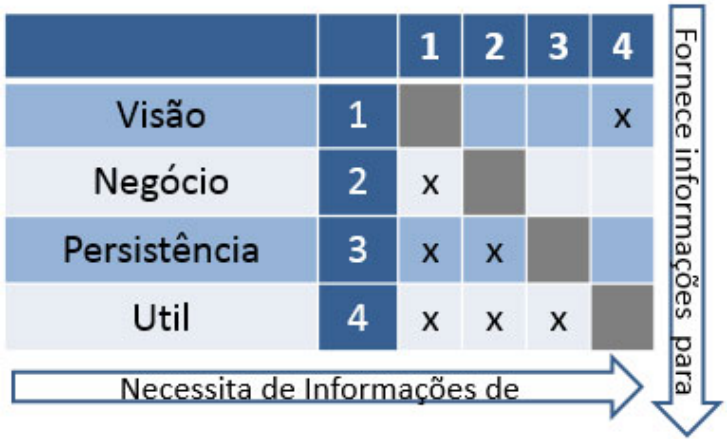

Figura 2.12: Exemplo de violação em uma matriz de dependência estruturada. Fonte: adaptado de Jordan; Sinha; Jackson (2005).

entre os elementos. A Figura 2.13, aponta um sistema com um alto agrupamento entre os eixos x e y.

Através do uso de matriz de dependência estruturada é possível recuperar conceitos importantes de uma arquitetura de *software*. E ainda, tal abordagem possibilita destacar aspectos da estrutura do sistema que se afastam da arquitetura idealizada. Logo, através da análise é possível lograr uma caracterização da arquitetura de um sistema e encontrar violações arquiteturais. Estas violações podem ser apenas falhas no projeto arquitetônico que podem ser corrigidos modificando o código [37].

2.2.3 Métricas de Software

As métricas de *software* são um ponto chave para a análise e o entendimento em abordagens de visualização de *software*. Por meio destas é possível identificar características importantes sobre a estrutura de um sistema [32]. O intuito da utilização das métricas de

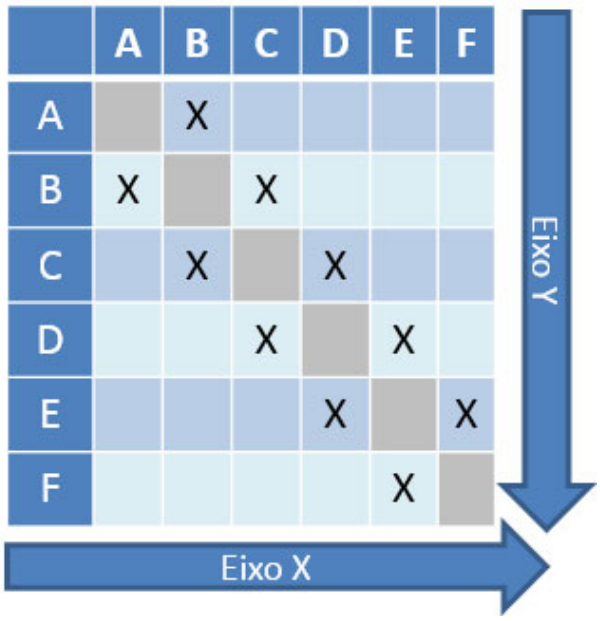

Figura 2.13: Matriz de dependência estruturada com alto agrupamento entre o eixo x e y.

software por ferramentas de visualização é condensar diversas informações em um único modelo, a fim de facilitar o entendimento completo do sistema. A Figura 2.14, apresenta um exemplo de métricas utilizadas em uma ferramenta de visualização de *software*, sendo possível observar várias métricas em um mesmo diagrama – métricas que representam diferentes aspectos da entidade do *software*.

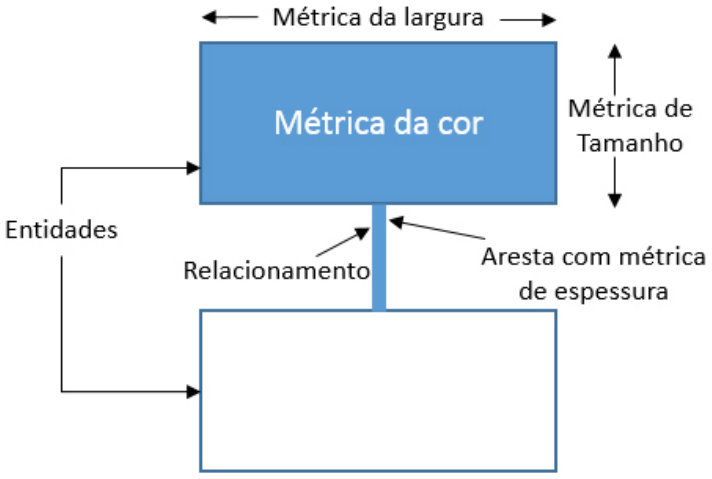

Figura 2.14: Exemplo de métricas utilizadas por uma ferramenta de visualização de *software*. Fonte: adaptado de Lanza (2003).

No contexto de recuperação de arquitetura, as métricas mais importantes estão relacionadas com aspectos estruturais do código. Através destas é possível identificar como as unidades do *software* estão acopladas umas às outras, e quão forte é o acoplamento de dependências.

Neste sentido, a métrica de instabilidade é uma forma de identificar componentes arquiteturais em uma representação de alto nível do sistema; consiste em classificar módulos do sistema em estáveis e instáveis. A estabilidade é definida através da quantidade de módulos que dependem um do outro.

Os módulos que encapsulam a arquitetura de alto nível do sistema são classificados como componentes estáveis, pois, uma alteração em tal módulo afeta todos os módulos dele dependentes [27]. A Figura 2.15 mostra um componente estável. Ali é possível observar vários componentes dependendo do "Componente 1" e, assim, uma alteração pode afetar sua totalidade; logo, é um componente estável.

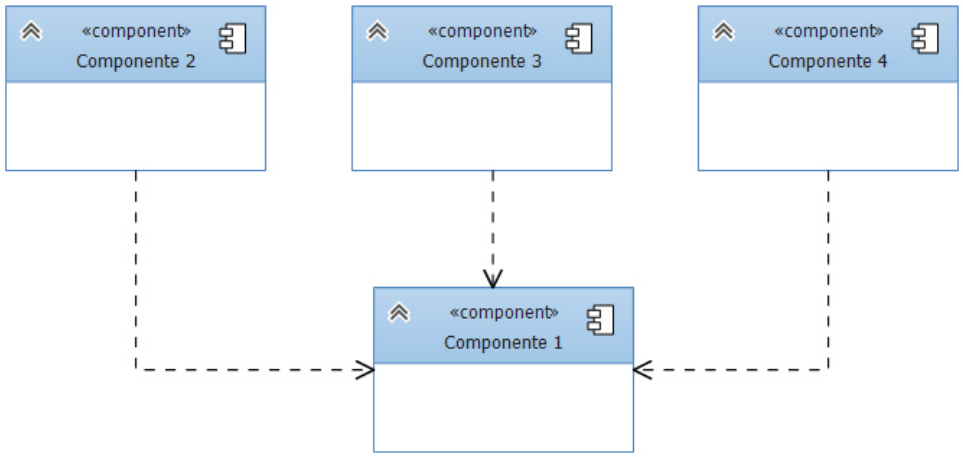

Figura 2.15: Exemplo de um componente estável. Fonte: adaptado de R. Martin, M. Martin (2006).

A Figura 2.16, destaca um exemplo de componente instável, uma vez que o "Componente 1" não tem nenhuma dependência; logo, nenhuma responsabilidade sobre outros módulos. Assim, uma alteração neste componente não irá afetar nenhum outro. Tal componente é tido com instável e, provavelmente, não contém nenhuma decisão arquitetural [27].

Diante do exposto, faz-se importante salientar sobre a forma de cálculo da métrica de instabilidade, conforme se segue [27]:

- Afferent Couplings (CA): o número de entidades externas que acessam um determinado componente. .

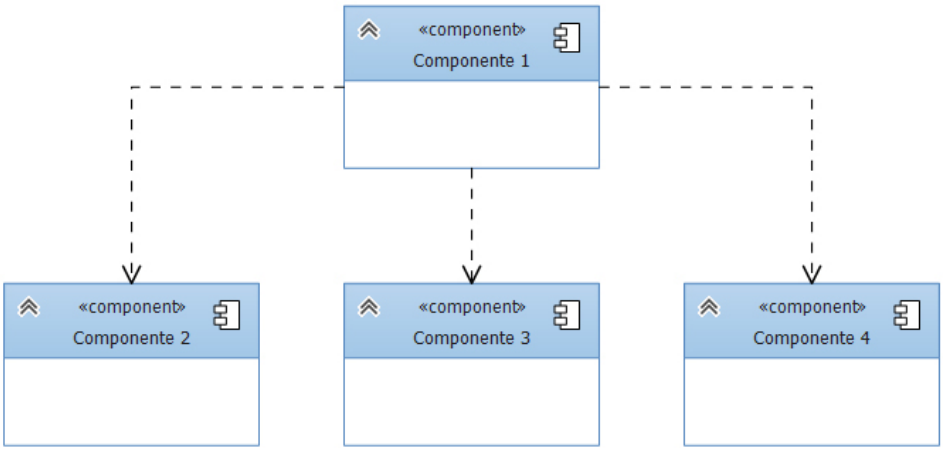

Figura 2.16: Exemplo de um componente instável. Fonte: adaptado de R. Martin, M. Martin (2006).

- Efferent Couplings (CE): o número de entidades internas de um determinado componente que acessam entidades fora deste componente.
- I(instabilidade): $I = \frac{C_e}{C_a+C_e}$

A referida métrica tem o intervalo de 0 até 1, sendo que I=0 indica um componente completamente estável, e I=1 indica um componente completamente instável. Considerando o exemplo na Figura 2.17(a), o grafo direcionado aponta as relações entre diferentes módulos de um sistema. A Figura 2.17(b) apresenta em formato de matriz de dependência as mesmas relações. É possível identificar que quatro módulos dependem do módulo F; logo, Ca=4. Por outro lado, o módulo F não depende de nenhum módulo; logo, Ce=0. Assim, a instabilidade deste componente é I= 0, ou seja, ele é tido como um componente estável e é um candidato a módulo arquitetural no sistema.

2.3 Recuperando arquitetura utilizando Clusterização

A análise de *clusters* é o estudo formal de algoritmos e métodos de agrupamento ou classificação de objetos. Tal agrupamento se dá através da classificação do conjunto de dados em grupos ou hierarquia de grupos [16]. Neste sentido, os grupos similares pertencem a uma instância de um subconjunto similar, enquanto as instâncias não similares pertencem a grupos diferentes. Como resultado, as instâncias são organizadas em uma representação eficiente que caracteriza a população da amostra [25].

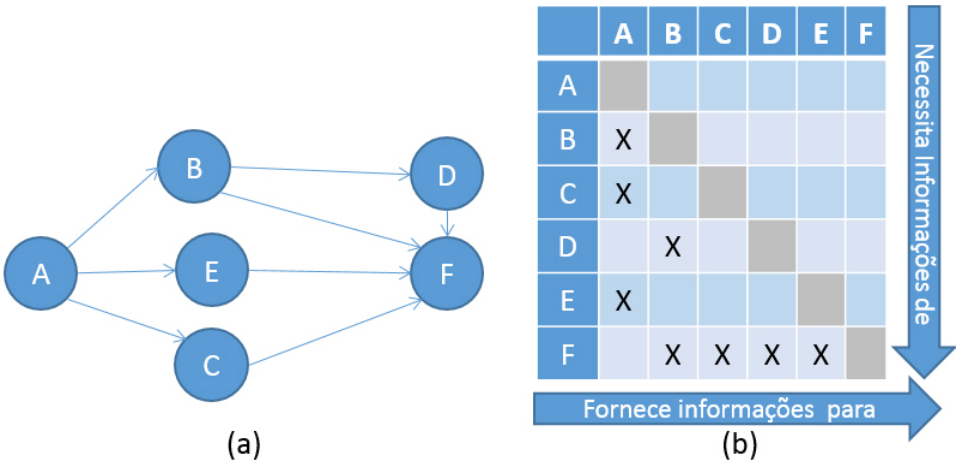

Figura 2.17: Exemplo de grafo direcionado e sua matriz de dependência correspondente.

Um processo de clusterização é a forma de explorar a estrutura de um conjunto de dados sem exigir premissas, ação denominada "aprendizagem não supervisionada" [16]. A aprendizagem não supervisionada tem como objetivo agrupar uma determinada coleção sem uma pré-classificação definida. O objetivo é encontrar objetos semelhantes dentro de um grupo e diferentes de objetos de outros grupos. Quanto maior for a semelhança dentro de um grupo, e quanto maior for a diferença entre os outros grupos, melhor será o resultado da clusterização [43].

O problema de clusterização foi abordado em muitos contextos e por pesquisadores de diversas disciplinas, o que demonstra sua importância, sendo fundamental para a análise exploratória de dados [14]. Neste sentido, a clusterização de dados tem sido usada principalmente nos seguintes propósitos [15]:

1. Obter conhecimentos sobre os dados, gerar hipóteses, detectar anomalias e identificar características relevantes em estruturas de dados subjacentes.
2. Identificar o grau de semelhança entre as formas ou organismos (relação filogenética).
3. Caracterizar-se como um método para organizar os dados e resumi-los através de protótipos de *cluster*.

A clusterização também é eficaz no que tange às análises de padrões, determinação de agrupamentos, oferta de suporte na tomada de decisões e situações de aprendizado de máquina – incluindo a mineração de dados, recuperação de documentos, segmentação de imagens e classificação de padrões. Em muitas das ações supramencionadas, tem-se pouca informação prévia disponível sobre os dados. É sobre tais restrições que a metodologia de clusterização é particularmente apropriada [14].

Uma vez que a clusterização é o agrupamento de objetos similares, faz-se importante algum tipo de medida para determinar se dois objetos são semelhantes ou não. Embora as definições de similaridade possam variar de um modelo para outro, o conceito de semelhança é obtido com base no cálculo das distâncias entre os objetos em um conjunto. Existem diversas fórmulas para cálculo de distância entre objetos, sendo a distância euclidiana e a distância cosseno as mais usuais [47].

Além disso, algumas técnicas de clusterização caracteriza cada *cluster* em termos de um protótipo de *cluster*, ou seja, um objeto de dados que é representante dos outros objetos no *cluster*. Os protótipos de *cluster* podem ser utilizados como a base para uma análise de dados ou técnicas de processamento de dados [43].

2.3.1 Clusterização de *Software*

O termo "clusterização" é utilizado em várias áreas de pesquisa para descrever métodos que agrupem dados não classificados. Na Engenharia de *Software*, a clusterização agrupa artefatos de *software*, como, por exemplo, código-fonte em subsistemas, a fim de auxiliar no processo de compreensão da estrutura de alto nível de um sistema de *software* complexo [41]. Além da compreensão, tais subsistemas podem ser utilizados para visualização de *software*, re-modularização e recuperação de arquitetura [36].

Shtern e Tzerpos [41] descrevem três situações onde as técnicas de clusterização podem ser uteis para resolver problemas na disciplina de Engenharia de *Software*, quais sejam:

- Análise reflexiva: o objetivo da análise reflexiva é mapear os componentes encontrados no código-fonte em componentes conceituais definidos por uma arquitetura hipotética.
- Evolução de *software*: os sistemas de *software* evoluem através de esforços para adicionar novas funcionalidades, para corrigir as falhas existentes e para melhorar a sustentabilidade. Normalmente, as ferramentas de clusterização de *software* tentam melhorar a estrutura de *software* ou reduzir a complexidade de grandes módulos: tal ação também se mostra útil para a identificação de código duplicado ou previsão da predisposição a falhas de módulos de *software*
- Recuperação de informação: o objetivo da engenharia reversa é recuperar componentes ou extrair abstrações de sistemas. A recuperação de módulos de *software* através do uso de clusterização foca na descoberta de módulos através da análise de dependências extraídas de sistemas de *software*, como, por exemplo, a chamada de funções.

A grande quantidade de esforço de pesquisa direcionada ao problema de clusterização de *software* é uma boa indicação da sua importância para a área da Engenharia de *Software*. Porém, várias abordagens descritas na literatura fazem uso de diferentes critérios para a determinação dos *clusters*. Assim, não é possível afirmar que tais abordagens recuperam a estrutura real do sistema [28]. Logo, para a clusterização de *software* ser útil, deve haver um estudo prévio sobre as vantagens e desvantagens associadas ao uso de diferentes algoritmos de clusterização para cada tipo de sistema. Também é preciso avaliar os resultados de clusterização individuais em relação a outros padrões de referências previamente definidos

2.3.2 Abordagem de clusterização para recuperação de arquitetura

A recuperação de arquitetura de *software* através do uso de clusterização foca na descoberta de módulos por meio da análise de dependências extraídas de sistemas de *software*, como, por exemplo, a chamada de funções, dependências de classes, pacotes, interfaces etc.

Para o processo de classificação das entidades deve ser utilizado algoritmos para clusterização. Wiggerts [49] classifica tais algoritmos da seguinte forma:

- Algoritmos com base em grafos: neste tipo de algoritmo, nós representam entidades e arestas representam relacionamentos. Nós com relacionamentos semelhantes são agrupados de forma que caracterizem um subgrafo significativo.
- Algoritmos de agregação: reduz o número de nós em um grafo por meio da fusão de entidades similares em nós agregados.
- Algoritmos de construção: atribui entidades a um *cluster* de uma única vez. Os *clusters* podem ser predefinidos (supervisionados) ou construídos como parte do processo de atribuição (não supervisionados).
- Algoritmos hierárquicos: constrói uma hierarquia de agrupamentos, utilizando recursividade para encontrar *clusters* aninhados.

O resultado de um processo de clusterização é o agrupamento dos dados em uma representação que caracteriza a população da amostra [25]. Contudo, a abrangência dos dados obtidos em um processo de clusterização pode não caracterizar *clusters* significativos. Neste sentido, os seguintes aspectos precisam ser abordados quando se considera os processos de clusterização de *software* [36]:

- As entidades que devem ser agrupadas: ao aplicar uma técnica de clusterização, faz-se importante observar a natureza dos dados e a utilização dos parâmetros adequados para estes tipos de dados.
- As medidas de similaridade que podem ser utilizadas: as métricas utilizadas para calcular a similaridade entre dois objetos são os coeficientes de associação, as medidas de correlação e as métricas de distância. A semelhança é o inverso da distância, o que significa que quanto maior for a distância entre dois objetos, menor será a similaridade entre eles. A correlação também oferta uma estimativa de similaridade. Neste sentido, quanto maior a correlação, maior será a semelhança entre duas entidades.
- O algoritmo de clusterização a ser aplicado: as técnicas de clusterização se comportam diferentemente quando aplicadas em distintos tipos de sistema de *software*.

Mitchell em [28], apresenta uma abordagem para clusterização cujo o objetivo é criar automaticamente decomposições da estrutura do sistema de *software* em subsistemas significativos. Tal abordagem é automatizada pela ferramenta denominada Bunch. A ferramenta disponibiliza um ambiente completo para o processo de clusterização, possui uma interface gráfica intuitiva. A ferramenta é disponibilizada de forma gratuita, com manual de operação e códigos de exemplo.

A Figura 2.18, ilustra o processo necessário para a clusterização de um sistema utilizando a ferramenta Bunch. O processo tem início com a extração do código-fonte para a criação de um Model Dependency Graph (MDG). Em seguida, diversos algoritmos podem ser utilizados para a localização de diferentes partições do MDG. Posteriormente, o resultado é apresentado como um MDG particionado em subsistemas.

As seções apresentadas a seguir descrevem detalhadamente conceitos importantes utilizados pela ferramenta em questão.

Module Dependency Graph

A ferramenta Bunch faz uso de um MDG como entrada para o processo de clusterização. A reprodução de um sistema em grafo se dá através do uso de nós para representar instâncias do *software* – como, por exemplo, classes, métodos, variáveis etc. – e de arestas, representando as conexões entre eles.

Formalmente, um $MDG = (M, R)$ é um grafo onde M é o conjunto de módulos de um sistema de *software*, e $R \subseteq M \times M$ é o conjunto de pares (u, v) que representam as dependências no código-fonte, como, por exemplo, o acesso às variáveis, a chamada à função etc [26]. A Figura 2.19, é uma representação gráfica de um MDG, obtido através

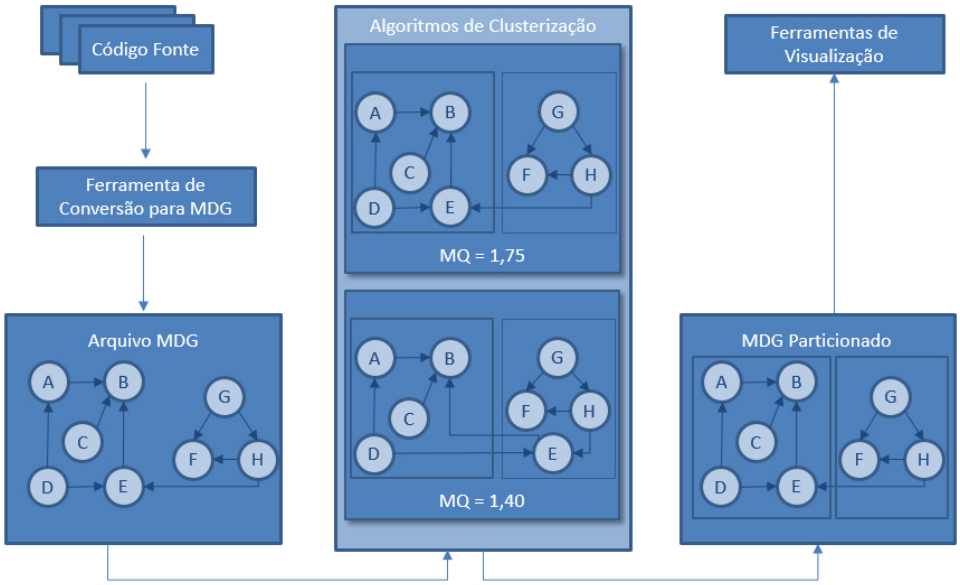

Figura 2.18: Processo de Clusterização da Ferramenta Bunch. Fonte: adaptado de Mitchell (2002).

da análise estática de um sistema em Visual Basic. Neste caso, os nós podem representar métodos ou variáveis, e as arestas representam as conexões entre eles.

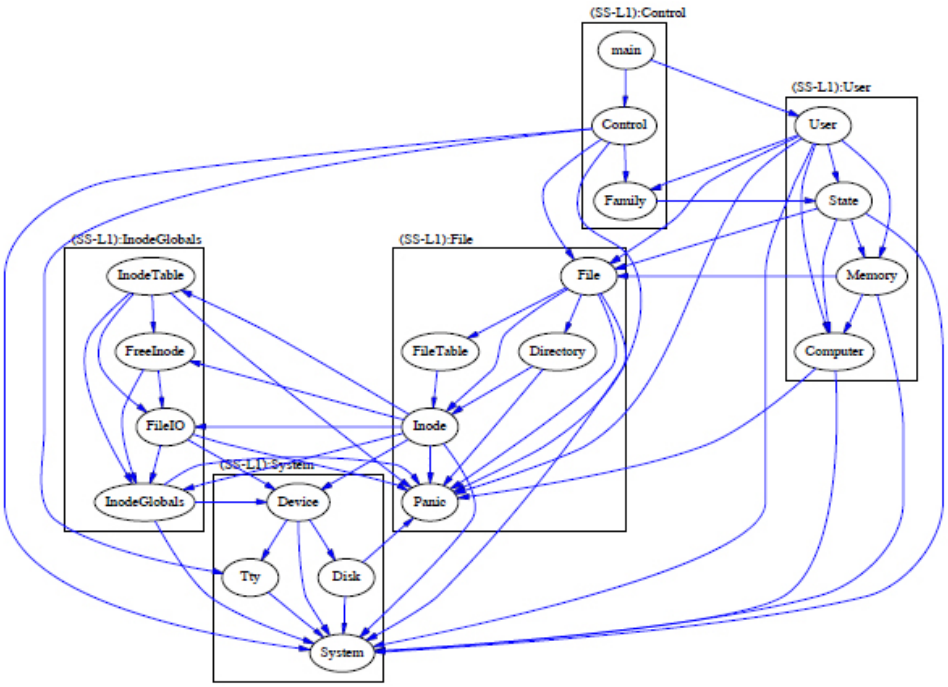

Figura 2.19: Representação gráfica de um MDG. Fonte: adaptado de Mitchell (2002).

Em um arquivo MDG, cada linha especifica uma relação entre duas entidades do

código-fonte. A direção da dependência é no sentido de Entidade_1 para Entidade_2, o que indica que a Entidade_1 faz uso de um ou mais recursos da Entidade_2. Opcionalmente, o MDG também pode especificar o peso das relações entre as duas entidades e o tipo da relação (como, por exemplo, herança, associação etc). Caso nenhum peso ou tipo seja atribuído à relação, a ferramenta atribui automaticamente o valor 1 (um) [28].

Avaliando partições em um MDG

O principal objetivo dos algoritmos de clusterização utilizados pela ferramenta é encontrar uma "boa partição" em um grafo MDG. Uma partição é a decomposição de um conjunto de elementos (todos os nós do grafo) em *clusters* mutualmente disjuntos. Encontrar boas partições envolve a navegação por todas as partições possíveis de uma forma sistemática. Para a realização desta tarefa de forma eficiente, faz-se uso de algoritmos de clusterização como um problema de pesquisa. O objetivo da pesquisa é de maximizar o valor de uma função objetivo, denominada Modularization Quality (MQ). [28].

O MQ determina a qualidade de uma partição em um MDG quantitativamente como um *trade-off* entre dependências de módulos dentro de subsistemas distintos (intraconectividade) e dependências entre módulos de um mesmo subsistema (interconectividade). Quanto maior o valor do MQ, mais próximo é a partição do MDG, que está da estrutura real do subsistema. O valor da medição MQ é delimitado entre -1, sem coesão dentro dos subsistemas, e 1, sem acoplamento entre os subsistemas [26].

Formalmente, o MQ de uma partição MDG, particionado em K *cluster*, é a diferença entre a média da intra e interconectividade de k *clusters*, como exposto na fórmula que se segue [28]:

$$BasicMQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} & \text{if } k > 1 \\ A_i & \text{if } k = 1 \end{cases}$$

A intraconectividade (A_i) de um *cluster* i consistindo de N_i nós e μ_i intra-arestas é a fração de μ_i sob o número máximo de intra-arestas de i, (isto é N_i^2). Assim, A_i é definido por:

$$A_i = \frac{\mu i}{\eta_i^i}$$

A interconectividade $E(i,j)$ entre dois *cluster* distintos i e j consistindo de N_i e N_j nós, respectivamente, e com $E(i,j)$ inter-arestas é a fração de $E(i,j)$ sobre o máximo número de interarestas entre i e j (isto é $2N_iN_j$); logo, $E(i,j)$ é definido por:

$$E_{i,j} = \begin{cases} 0 & i = j \\ \frac{E_{i,j}}{2N_1N_2} & i \neq j \end{cases}$$

2.3.3 Algoritmo de busca

Uma vez que um sistema de *software* pode apresentar várias decomposições válidas, o resultado de algoritmo de clusterização de *software* pode, em alguns casos, não ser significativo para um engenheiro de *software* [41]. Para resolver tal questão, a abordagem Bunch faz uso de 03 (três) algoritmos para a seleção de partições em um Grafo MDG, quais sejam: (1) algoritmos de busca exaustiva; (2) algoritmo *hill-climbing*; e, (3) algoritmo genético.

Algoritmo de busca exaustiva

A busca exaustiva é uma abordagem direta para solucionar problemas combinatórios; gera cada e qualquer elemento do problema do domínio, seleciona aqueles que satisfazem todas as restrições e, em seguida, encontra um elemento desejado, como, por exemplo, o que otimiza alguma função objetivo. Embora a ideia de busca exaustiva seja bastante simples, a sua implementação tipicamente requer um algoritmo que gere determinados objetos combinatórios [20].

No caso da abordagem Bunch, o algoritmo de busca exaustiva funciona medindo o valor MQ para todas as partições. A partição com o maior MQ é a selecionada. Porém, a busca exaustiva é impraticável na maioria dos casos, com exceção de problemas com poucas instâncias [20]. Assim, tais algoritmos são úteis somente em pequenos sistemas, com menos de 15 módulos. Para melhorar o desempenho do algoritmo, a ferramenta Bunch gera um conjunto de partições fazendo uso de um algoritmo combinatório que minimiza a distância entre as partições adjacentes. Assim, é possível transformar uma partição em outra utilizando um número ínfimo de operações [26]. Contudo, esta solução ainda continua sendo inviável para a maioria dos sistemas de *software*.

Algoritmo *hill-climbing*

O *hill-climbing* é um algoritmo de busca que tem por objetivo resolver tarefas computacionais e cognitivas em um problema com muitas soluções possíveis; tem uma longa tradição na área de otimização contínua [39]. Inicialmente, o algoritmo de busca *hill-climbing* seleciona um valor de forma randômica. Em seguida, é selecionado qualquer outro valor local que melhore o valor atual tendo como base uma função objetivo. Assim, outros valores são continuamente selecionados, com o objetivo de melhorar esta função. Tradicionalmente, o algoritmo deve parar quando nenhum movimento local possa ser realizado para melhorar a função. Após o termino, a busca pode encontrar o melhor valor local, mas não necessariamente o melhor global da função objetivo [39].

Na abordagem Bunch, o algoritmo *hill-climbing* se dá com uma partição aleatória de um MDG. Em seguida, os módulos desta partição são sistematicamente reorganizados

em uma tentativa de encontrar uma melhor partição, com um valor MQ maior. Se uma partição melhor for encontrada, tem-se a repetição do processo com o uso da partição encontrada como base para a nova busca. O algoritmo termina sua execução quando nenhuma partição com um valor MQ maior que atual for encontrada [26].

Como em algoritmos tradicionais, cada partição inicialmente gerada de forma randômica de um MDG eventualmente converte para um máximo local. Porém, nem todas as partições iniciais de um MDG produzem soluções subótimos aceitáveis. Tal problema tem sua solução na criação de uma população inicial de partições aleatórias. Assim, o algoritmo *hill-climbing* agrupa cada partição aleatório dentro da população e seleciona o melhor resultado, com o melhor MQ, como a solução subótimo. Na medida em que o tamanho da população aumenta, a probabilidade de encontrar uma boa solução subótima também aumenta [26].

Em uma tentativa de melhorar o valor MQ de uma partição, é realizada uma operação de movimento entre os módulos. Esta tarefa se dá através da geração de um conjunto de partições vizinhas – Neighboring Partitions (NP). Uma partição NP é uma partição vizinha de uma partição P se e somente se NP é exatamente o mesmo que P, exceto por um único elemento dentro da partição P diferente da partição NP. Se a partição P contém n nós e k *clusters*, o número total de vizinhos é $O(n \times k)$. Para várias partições de um MDG, o número de vizinhos é exatamente $n \times k$. Entretanto, se uma partição contém *cluster* com um ou dois nós, o número total de vizinhos distintos é ligeiramente menor. A Figura 2.20, ilustra um exemplo de partição e todas as partições vizinhas dela [28].

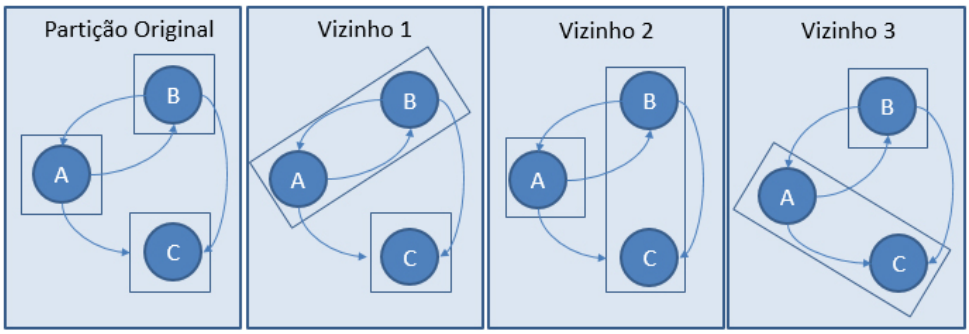

Figura 2.20: Partições vizinhas. Fonte: adaptado de Mitchell (2002).

O benefício-chave para uma pesquisa *hill-climbing* é o fato de esta fazer uso de uma quantidade limitada de memória (somente a do estado atual é armazenado), é fácil de implementar eficientemente e, se um ou mais soluções existirem no espaço, surpreendentemente pode encontrar tal solução [39].

Algoritmos genéticos

Os algoritmos genéticos são programas de computadores que simulam os processos de evolução biológica, a fim de resolver problemas e modelar sistemas evolutivos. É um método eficiente de busca dentro de um grande conjunto de possíveis soluções [30]. Estes surgiram como alternativa para superar alguns dos problemas de métodos de pesquisa tradicionais, como, por exemplo, o *hill-climbing*. Assim, tais algoritmos buscam solucionar os desafios de tais algoritmos, como, por exemplo, o problema de ficar preso em soluções subótimas e, portanto, não encontrar a melhor solução. Possuem ainda uma excelente aplicabilidade em problemas altamente restritos, onde o número de "boas" soluções é pequeno em relação ao tamanho do espaço de busca [8].

Tradicionalmente, a execução dos algoritmos genéticos envolve as seguintes operações [30]:

- Seleção: este operador seleciona cromossomos na população para reprodução. Quanto melhor o cromossomo, maior é a chance de ele ser utilizado para reprodução.
- Recombinação: uma operação que combina os cromossomos selecionados, gerando os descendentes para uma nova população. Assim, os descentes recebem códigos genéticos de ambos os cromossomos.
- Mutação: este operador randomicamente altera algumas partes em um cromossomo, a fim de manter uma variabilidade genética na população, evitando que a população fique estagnada em uma solução sub-ótima.

Os algoritmos genéticos fazem uso das operações supramencionadas para operar em uma população através do seguinte processo interativo [28]:

1. Gerar a população inicial, criando cromossomos aleatórios de tamanho fixo.
2. Criar uma nova população, aplicando o operador de seleção e reprodução para selecionar pares de cromossomos. O número de pares é o tamanho da população dividido por 02 (dois); assim, o tamanho da população continua constante entre as gerações.
3. Aplicar o operador de recombinação para os pares de cromossomos da nova população.
4. Aplicar o operador de mutação em cada cromossomo da nova população.
5. Substituir a antiga população pela a recentemente criada.
6. Se o número de interações é menor que o número máximo definido, ir para o passo 2. Senão, parar o processo e exibir o melhor resultado encontrado.

Na ferramenta Bunch, o grafo MDG é tratado como uma sequência de caracteres. Por exemplo, o grafo da Figura 2.21, é codificado com a sequência S= 2 2 4 4 1.

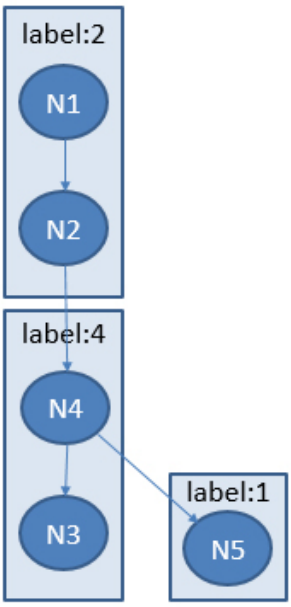

Figura 2.21: Exemplo de partição. Fonte: adaptado de Mitchell (2002).

Cada nó no grafo possui um número identificador atribuído a ele. Por exemplo, o nó N1 é atribuído ao identificador 1; o nó N2 é atribuído ao identificador 2; e, assim, sucessivamente. Os identificadores únicos definem qual posição na sequência é utilizada para o nó no *cluster*. Assim, na sequência S = 2 2 4 4 1, o primeiro caractere em S, indica que o primeiro nó (N1) é contido no *cluster* 2; do mesmo modo, o segundo nó (N2) também está contido no *cluster* 2 [28].

Os parâmetros utilizados nas operações genéticas têm uma taxa fixa que variam de acordo com o problema. As taxas foram derivadas empiricamente depois de experimentos com sistemas de diferentes tamanhos. Porém, os parâmetros utilizados no algoritmo podem ser alterados pelos usuários da ferramenta. As taxas utilizadas nos algoritmos genéticos são as seguintes, assumindo que N é o número de nós do MDG [28]:

1. Taxa de Recombinação: 80% para populações de 100 ou menos indivíduos; 100% para populações de mil indivíduos ou mais, e que varia linearmente entre os referidos valores populacionais.

2. Taxa de mutação: $0.004 \log_2(n)$. Em geral, a taxa de mutação é 4/1000. Entretanto, devido à forma de codificação decimal utilizada pela ferramenta, faz-se necessário multiplicar pelo número de *bits* que seria utilizado se a codificação fosse binária, para obtenção de uma taxa de mutação equivalente.

3. Tamanho da população: $10N$.

4. Número de gerações: $200N$.

Capítulo 3

Combinando visualização de software e clusterização de dados

A Figura 3.1, ilustra a visão macro para recuperação de arquitetura utilizando visualização de *software* e clusterização. Inicialmente, é feito uma seleção dos artefatos do código-fonte do sistema para análise. Em seguida, dá-se a recuperação de visões arquiteturais com o auxílio de técnicas de visualização de *software* e clusterização. Os modelos recuperados são unidos com o objetivo da criação de um único modelo. A cada ciclo de execução é recuperado uma nova visão arquitetural do sistema.

A extração de dados do código-fonte recolhe as informações necessárias para a descrição da arquitetura do sistema de *software* por meio da análise do código. Como acontece em *softwares* legados, onde uma documentação detalhada do sistema e sua arquitetura estão defasadas, quando presentes, o código-fonte é a única fonte de informação confiável para a criação de uma descrição da arquitetura do sistema. Portanto, é preciso analisá-lo para a coleta das informações arquitetônicas relevantes. A análise do código-fonte pode ser realizada de maneira manual, porém, devido à granularidade existente no código, faz-se importante a utilização de ferramentas que facilitem a análise dos arquivos do código-fonte [9].

Na visualização de *software* é utilizado técnicas e ferramentas de visualização para obtenção de uma representação da estrutura do sistema e as interações entre diferentes partes do código. A escolha da ferramenta dependerá da linguagem de programação utilizada pelo sistema que será analisado. Algumas ferramentas trabalham com um formato de arquivo próprio. Se a ferramenta não possuir um *parser* automático, então, far-se-á necessário a utilização de técnicas de representação interna do código para conversão do código-fonte em um arquivo no formato específico da ferramenta.

A clusterização tem início com a seleção das entidades do código-fonte que serão classificadas. A seleção da entidade deve estar no mesmo nível de granularidade do código-fonte

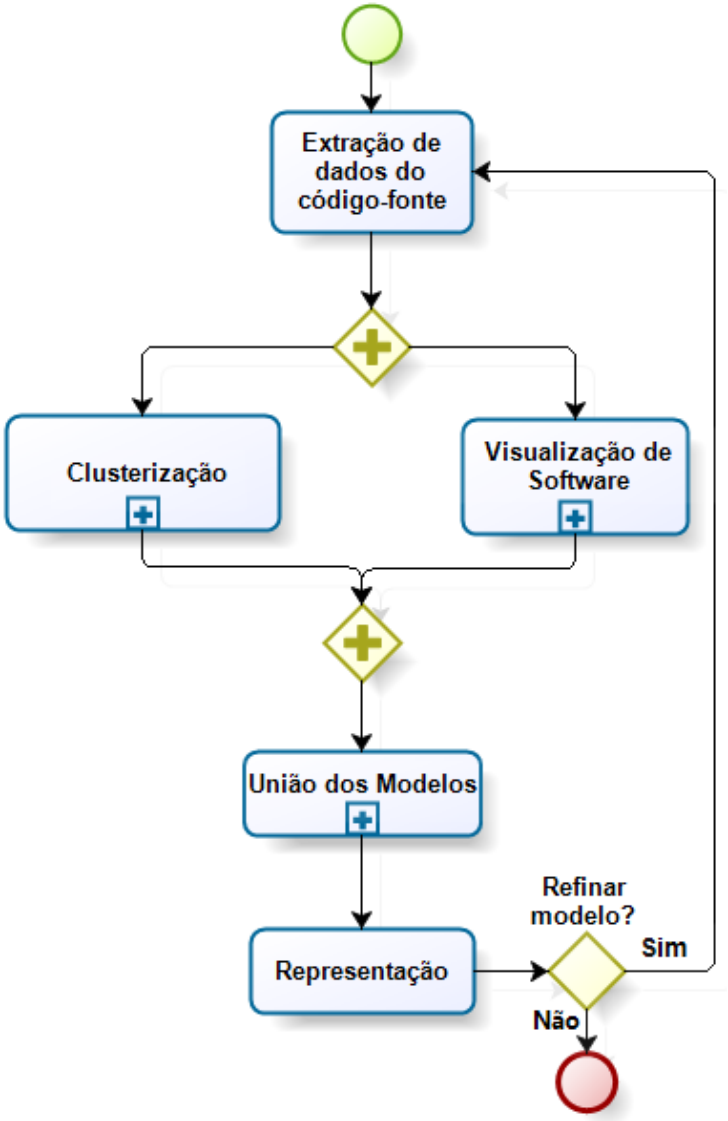

Figura 3.1: Abordagem para recuperação de arquitetura.

analisado pelo processo de visualização de *software*. Isto é, caso o processo de visualização tenha sido realizado buscando o entendimento das relações estruturais do código – como, por exemplo, pacotes, módulos, subsistema – então, o processo de clusterização deve ser realizado nestas entidades. Por outro lado, se o processo de visualização teve foco na recuperação das relações comportamentais do sistema, através da análise de métodos, variáveis e objetos, então, a clusterização também deverá ser feita em tais entidades.

Em muitos casos, o analista, com certo conhecimento do sistema, pode realizar uma análise nos resultados produzidos pelas técnicas de visualização de *software* e clusterização para criar os modelos necessários para representação da arquitetura de um sistema de *software*. Entretanto, especialmente para sistemas complexos, faz-se necessário o uso de estratégias para interpretação dos resultados apresentados. Tais estratégias envolvem a

Relação	Ocorrência
{1,2,3}	100%
{1,2,3,4}	25%
{4,5}	50%
{4,5,6}	25%
{5}	25%
{6}	25%
{6,7,8,9}	50%
{7,8,9}	100%

Tabela 3.1: Ocorrências das entidades entre os resultados.

observação de padrões repetidos e a identificação de violações arquiteturais no código-fonte.

Em geral, métodos de reconstrução de arquitetura automáticos, como os obtidos em um processo de clusterização, tem a vantagem de produzir diferentes modelos para um mesmo sistema de *software* em um curto espaço de tempo. Tais modelos podem ser construídos de forma diferente por meio de ajustes de configurações nos algoritmos de clusterização. Através da análise de diferentes modelos é possível identificar padrões que se repetem com frequência entre os resultados.

Porém, quando não se tem uma ideia clara de como a estrutura do sistema foi composta, os vários modelos produzidos pelo processo de clusterização podem não favorecer o entendimento do sistema [29], uma vez que apresentam diferentes visões de alto nível da arquitetura. Neste sentido, a utilização da técnica de visualização de *software* pode permitir a observação de diferentes resultados em um menor nível de abstração. Por meio de operações interativas nos modelos produzidos pela visualização de *software* é possível a decomposição de componentes do sistema em representações mais detalhadas. Dessa forma, é possível a observação de conceitos que compõe a arquitetura do sistema com uma maior profundidade.

Nesse contexto, unindo os modelos produzidos pelo processo de clusterização com o modelo produzido pelo processo de visualização, é possível obter diferentes representações do sistema para formar um modelo final com uma maior precisão. Para exemplificação, será levado em conta os resultados apresentados na Figura 3.2. Supondo que no processo de obtenção de modelos da arquitetura de um sistema, foram encontrados quatro resultados diferentes, provenientes tanto da técnica de visualização de *software* quanto da execução de diferentes algoritmos de clusterização. Cada modelo apresenta 9 entidades, sendo elas: {1,2,3,4,5,6,7,8,9}. Ao realizar uma contagem das entidades agrupadas de forma similar é possível obter a relação de quantas vezes uma entidade foi classificada de maneira similar a outra. O resultado dessa contagem pode ser observado na Tabela 3.1.

Por meio da análise dos resultados, é possível observar que as entidades {1,2,3} podem ser classificadas em um único módulo, uma vez que aparecem 100% das vezes na mesma relação; o mesmo ocorre com as entidades {7,8,9}. Já as entidades {4,5,6} podem tanto serem classificadas como um módulo único, ou cada entidade ser agregada a algum outro módulo adjacente, uma vez que não existe um consenso entre os resultados. Nestes casos, uma análise adicional deve ser realizada para cada entidade. Esse simples exemplo apenas ilustra o uso da agregação de resultados para ajudar a composição de um modelo único final, em muitos casos esse tipo de técnica ajuda ao analista obter confiança sobre resultados obtidos por meio de diferentes representações de um mesmo modelo [29].

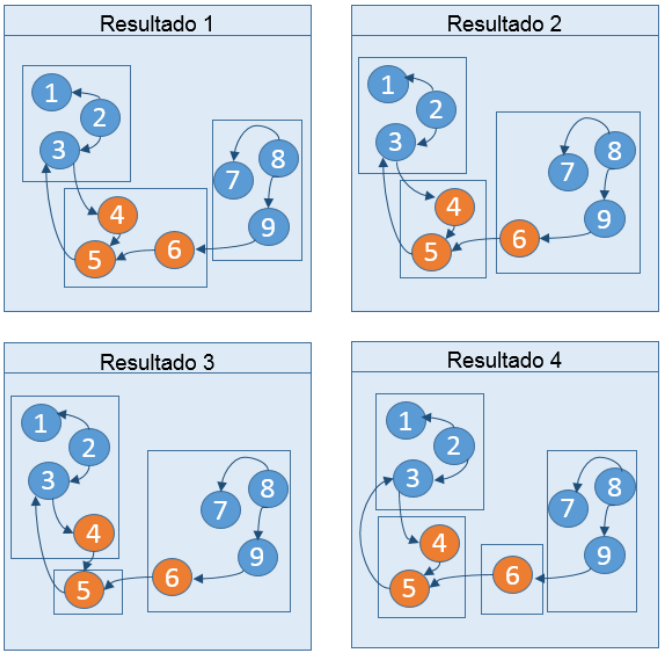

Figura 3.2: Diferentes resultados para comparação.

Porém, em uma análise de resultados, outros fatores podem influenciar a obtenção de um modelo final. Um sistema de *software*, durante todo o seu ciclo de vida, está suscetível a diversas alterações em sua arquitetura. Contudo, tais operações podem introduzir violações arquiteturais no código, como, por exemplo, a violação de camadas, a quebra de abstrações, a duplicação de funcionalidades etc [18]. Métodos de engenharia reversa são fortemente afetados por tais deficiências na base de código do sistema [33].

Para exemplificar o efeito de uma violação arquitetural em um código utilizaremos os mesmos resultados da Figura 3.2. Supondo que a entidade 4 fosse uma entidade mapeada de forma errônea, devido a uma violação arquitetural, e sua ligação com a entidade 5 não existisse. Todos os resultados seriam classificados de forma diferente se essa violação arquitetural fosse removida, conforme mostra a Figura 3.3. O resultado da nova análise, levando em conta a agregação de entidades similares, pode ser observado na Tabela 3.2.

Relação	Ocorrência
{1,2,3,4}	100%
{5}	50%
{5,6}	50%
{6}	25%
{6,7,8,9}	25%
{7,8,9}	100%

Tabela 3.2: Ocorrências das entidades entre os resultados após eliminação da violação arquitetural

Perante análise nos resultados é possível verificar o impacto da violação arquitetural identificada no mapeamento das entidades. O módulo inicialmente mapeando contendo as relações {1,2,3}, agora agrega a entidade 4, uma vez que esse conjunto foi classificado de forma similar em todos os resultados. Já as relações {5,6} também apresentam uma maior chance de serem classificadas em uma mesmo módulo, uma vez que ocorre com uma maior frequência.

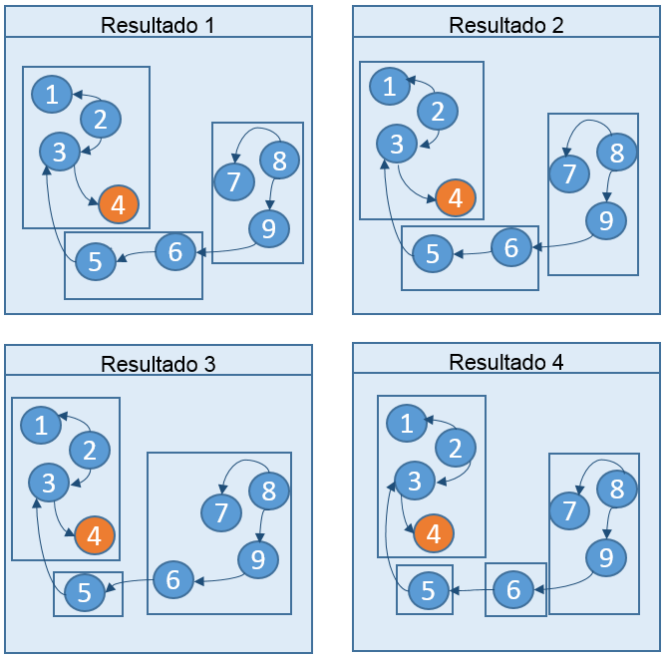

Figura 3.3: Eliminando a violação arquitetural dos resultados.

As violações arquiteturais podem ser encontradas por meio de uma análise nos artefatos produzidos pela visualização de *software*. Como exemplo, levemos em conta a matriz de dependência estruturada apresentada na Figura 3.4. Através da observação da matriz é possível notar que existe uma relação entre camadas, e que uma camada utiliza recursos das camadas superiores. No entanto, essa relação é violada pela camada (5), uma vez que ela utiliza recursos da camada (4), configurando uma violação arquitetural. Uma

vez identificada a violação arquitetural, é necessário tratá-las como uma exceção, e, se necessário, modificar o conjunto de dados para o processo de clusterização. De forma que tal relação não seja levada em conta durante o processo. Assim, evitando a produção de modelos errôneos que possam afetar a interpretação dos resultados.

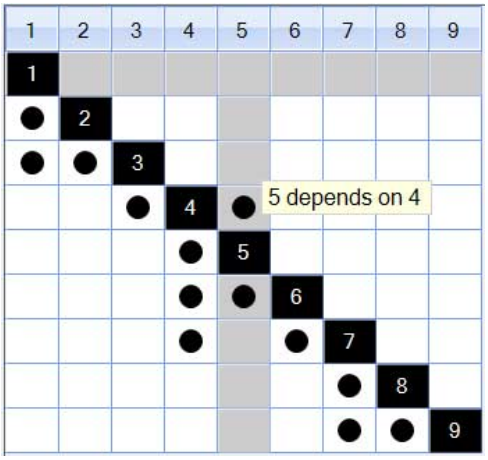

Figura 3.4: Exemplo de violação em uma matriz de dependência estruturada..

3.0.4 Exemplo de recuperação de arquitetura

Nesta seção iremos utilizar um exemplo para recuperação de arquitetura utilizando as técnicas de visualização de *software* e clusterização. O sistema apresentado neste exemplo é o Sistema de Informação Acadêmica de Graduação (SIGRA). Este sistema foi desenvolvido em 1988 e reformulado em 2000, escrito na linguagem de programação Microsoft Visual Basic, é responsável por controlar a execução de várias rotinas acadêmicas da Universidade de Brasília, como matriculas, certificados, expedição de documentos, habilitação de cursos etc.

A análise do sistema SIGRA por meio de visualização de *software* envolveu o uso da ferramenta VBDepend. Esta ferramenta oferece diversos mecanismos que facilitam a exploração da arquitetura de implementação de um sistema na linguagem Visual Basic. Através da análise estática é possível obter diversos tipos de artefatos do código-fonte, tais como: grafos de chamadas, grafos de acoplamento e matrizes de dependências. Esta ferramenta de análise foca na visualização do código-fonte, ao invés de visualização de entidades arquiteturais. Logo, faz-se necessária uma análise dos artefatos produzidos por tais visualizações para identificar os componentes da arquitetura do *software*.

Através da referida ferramenta, é possível a obtenção de grafos de chamadas, conforme exposto na Figura 3.5. Através de um grafo de chamadas é possível obter uma visão geral

sobre os acoplamentos entre os componentes do sistema. Cada nó no grafo representa um componente do sistema, já as arestas representam a conexão entre eles.

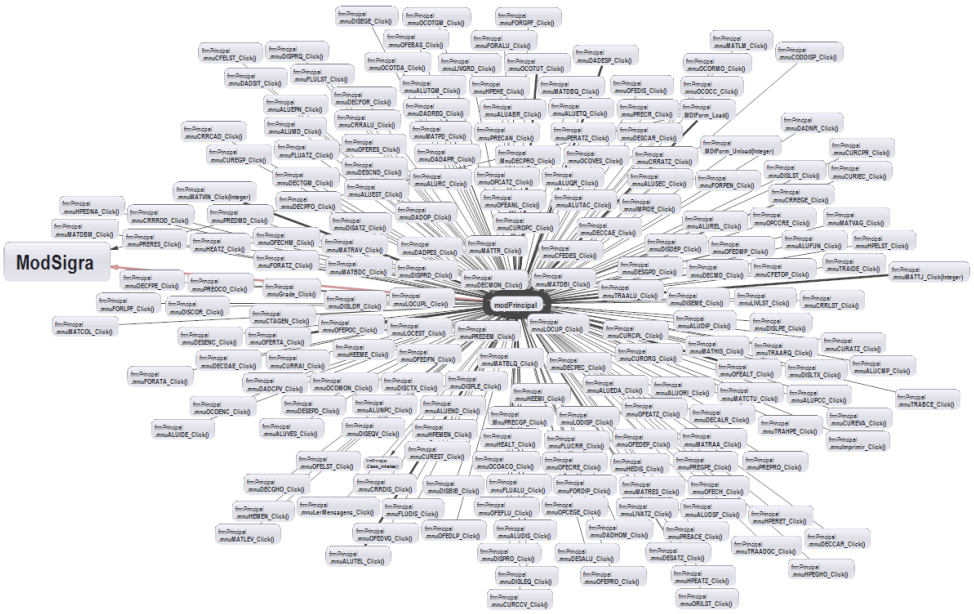

Figura 3.5: Grafo de chamadas obtido através pela ferramenta de visualização de software.

Os nós do grafo podem representar diferentes aspectos do *software* de acordo com uma métrica especifica, como, por exemplo, o número de linhas de código, grau de complexidade, acoplamento aferente, acoplamento inferente, etc. Isto ajuda a ter uma compreensão de diversos pontos de vista do sistema, uma vez que em uma só representação é possível ter várias informações sobre o componente em análise. A Figura 3.6, ilustra um exemplo de métrica utilizada em um grafo para destacar os módulos com maior acoplamento eferente. Através dela é possível notar que, se destacam os nós "modPrincipal" e "ModSigra" uma vez que, estes possuem um alto grau de acoplamento aferente

Através de operações no grafo é possível refinar os resultados apresentados, para recuperação em mais detalhes de determinados aspectos do grafo. Por exemplo, é possível expandir o nó "modPrincipal" para uma análise mais detalhada sobre este componente. O resultado é mostrado na Figura 3.7, onde é possível ter uma maior compreensão deste módulo. A métrica utilizada para determinar o tamanho dos nós nesta imagem foi o número de linhas de código de cada entidade.

A representação do sistema através de grafos ajuda a entender como as conexões entres as entidades participantes da estrutura do sistema estão retratadas. No entanto, em sistemas complexos, o resultado visual desta representação pode ser confuso, com nós e arestas se sobrepondo, o que dificulta a análise do grafo. Nestes casos, o uso de matriz de dependências estruturada pode favorecer a análise do sistema. A Figura 3.8, mostra a

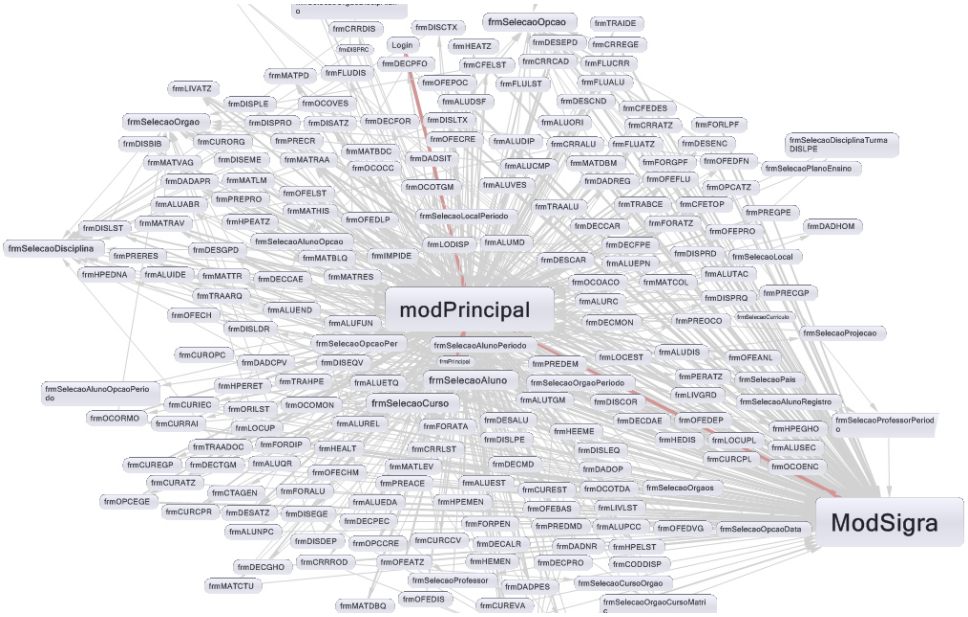

Figura 3.6: Grafo de chamadas utilizando métrica de acoplamento aferente..

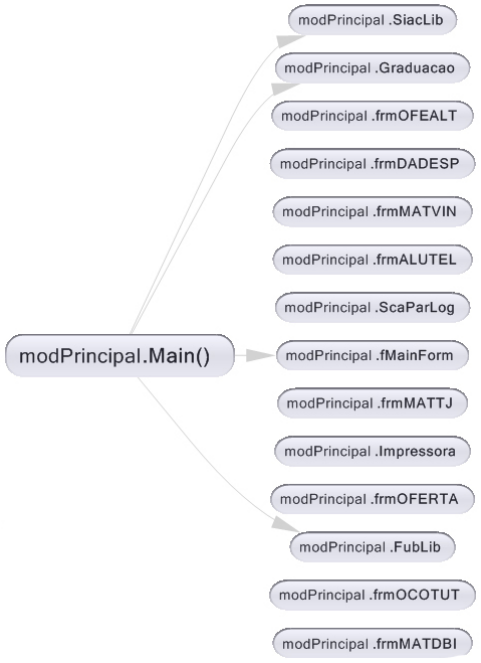

Figura 3.7: Decomposição de um grafo..

matriz extraída do sistema SIGRA. A matriz representa as mesmas informações que um grafo, os elementos no topo da coluna representam os nós do grafo, as células não vazias representam as conexões.

Em uma matriz de dependências, as células não vazias podem conter diferentes tipos

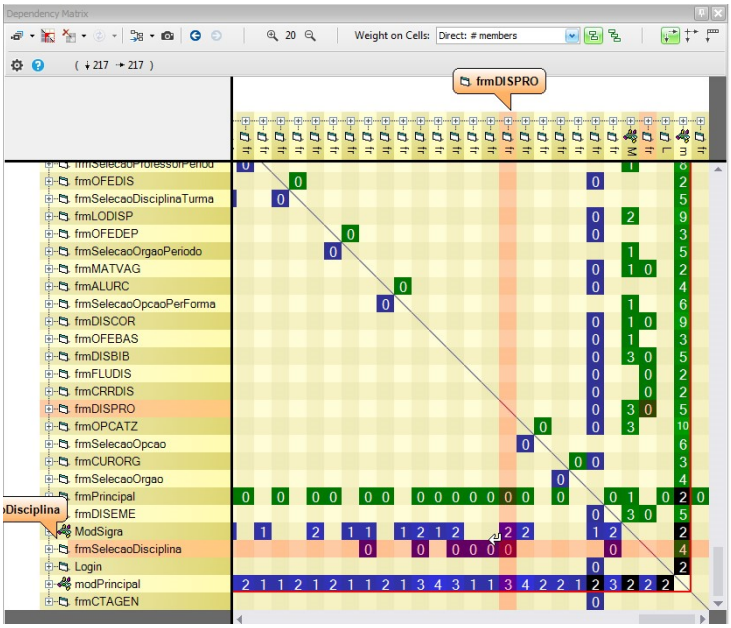

Figura 3.8: Detalhes de uma matriz de dependência estruturada..

de cores. A cor azul indica que o elemento da coluna usa o elemento da linha. Já a cor verde indica que o elemento da coluna é usado pelo elemento da linha. Por fim, a cor preta indica que o elemento da coluna e o elemento da linha estão diretamente usando um ao outro. Uma coluna não vazia também pode conter números. Esses números representam a grau do acoplamento entre os elementos no nó.

Foi utilizada a ferramenta Bunch para recuperação da arquitetura por meio da técnica de clusterização. O primeiro passo para o processo de clusterização do SIGRA, foi à extração do Module Dependency Graph (MDG) do sistema. A extração foi feita através da análise estática do código, buscando as relações de dependências entre formulários e bibliotecas. A Figura 3.9, mostra uma parte do MDG extraído.

```
1  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
2  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarFormando
3  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.GerarAlunoProvavelFormando
4  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarMonitoria
5  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarDominioConexo
6  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarCurriculo
7  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarCurriculoCadeia
8  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarFluxo
9  Biblioteca.AcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
10 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
11 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarFormando
12 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.GerarAlunoProvavelFormando
13 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarMonitoria
14 Biblioteca.CriarCurriculo Biblioteca.AtualizarDominioConexo
15 Biblioteca.CriarCurriculo Biblioteca.AtualizarCurriculo
16 Biblioteca.CriarCurriculo Biblioteca.AtualizarCurriculoCadeia
17 Biblioteca.CriarCurriculo Biblioteca.AtualizarFluxo
18 Biblioteca.CursoCadastrado Biblioteca.AtualizarApostila
```

Figura 3.9: DGM extraído do sistema SIGRA..

Tabela 3.3: Resultados obtidos por meio da clusterização com o algoritmo hill-climbing.

População	Número de Clusters	Tempo(Segundos)
1	6	0.172
10	6	0.983
100	7	1.912

Tabela 3.4: Resultados obtidos por meio da clusterização com o algoritmo genético.

Nr. Geração	População	Crossover	Mutação	Nr. Clusters	Tempo (segundos)
382	382	0.8	0.04	32	16.153
7650	765	0.8	0.04	30	713.315
15300	1530	0.8	0.04	27	2722.77

Após a extração do MDG, tem início o processo de clusterização. A ferramenta disponibiliza três algoritmos que percorre o MDG buscando por classificações que representem módulos do sistema, sendo eles: (1) algoritmo de busca exaustiva; (2) algoritmo hill-climbing; (3) algoritmo genético. Devido ao tamanho do sistema, não foi possível obter resultados com o algoritmo de busca exaustiva.

Em uma análise inicial, utilizando o algoritmo hill-climbing, foram obtidos os resultados apresentados na Tabela 3.3. Neste caso, a coluna População indica quantas vezes o algoritmo é reiniciado em uma posição aleatória diferente; quanto maior a população, maior será o tempo gasto, como observado na coluna Tempo(Segundos).

Algoritmos genéticos apresentaram resultados inferiores aos algoritmos *hill-climbing*. Além de um maior tempo de execução, o número de *clusters* produzidos não correspondem com o número aproximado de módulos do sistema. Apesar de alterações nas configurações do algoritmo, nenhum resultado produzido foi satisfatório.

Por meio da análise de padrões repetidos foi possível observar a ocorrência de módulos classificados de forma similar em todos os modelos produzidos por algoritmos *hill-climbing*. A Figura 3.10 detalha os resultados da execução do algoritmo *hill-climbing* com diferentes configurações. Neste caso, representamos os resultados da clusterização por meio de um grafo onde os nós de tamanho maior constituem o centroide do *cluster*, representando os módulos arquiteturais definidos pela clusterização. Também, é possível observar as setas direcionadas para os diversos nós, assim, é possível observar visualmente o quão forte são as dependências entre os centroides e os demais *clusters*.

Quando observado os resultados produzidos pela abordagem de visualização de *software* é possível confirmar a existência de módulos arquiteturais comuns no processo de clusterização. No caso da visualização de *software*, foi utilizado a métrica de instabilidade para detecção de tais conceitos arquiteturais. Através do grafo, representado na Figura 3.11, é possível notar que várias entidades dependem da entidade “Biblio-

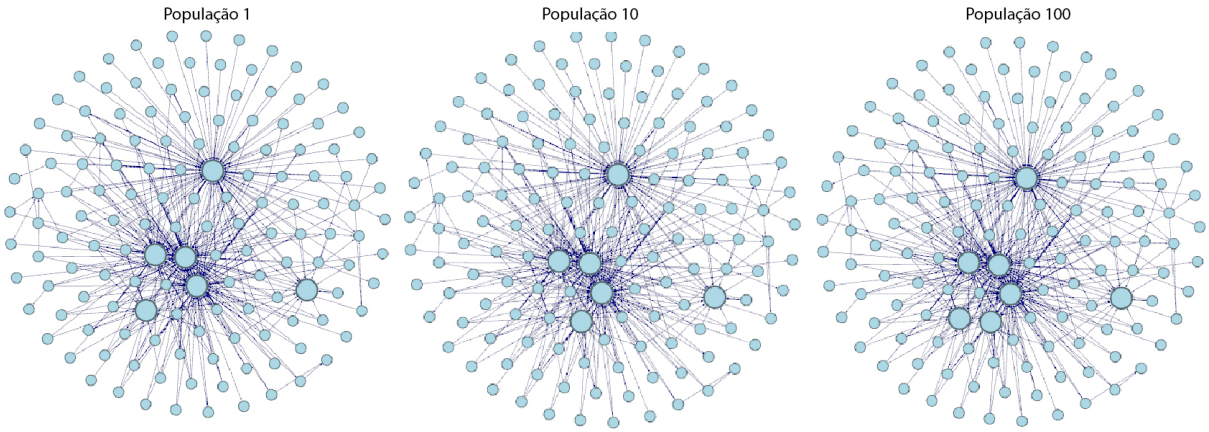

Figura 3.10: Resultados clusterização..

teca.ScaLib", porém, esta entidade não depende de ninguém. Logo ela é uma entidade estável e configura um componente arquitetural. Esta mesma entidade foi classificada como um elemento arquitetural em todos os resultados da clusterização.

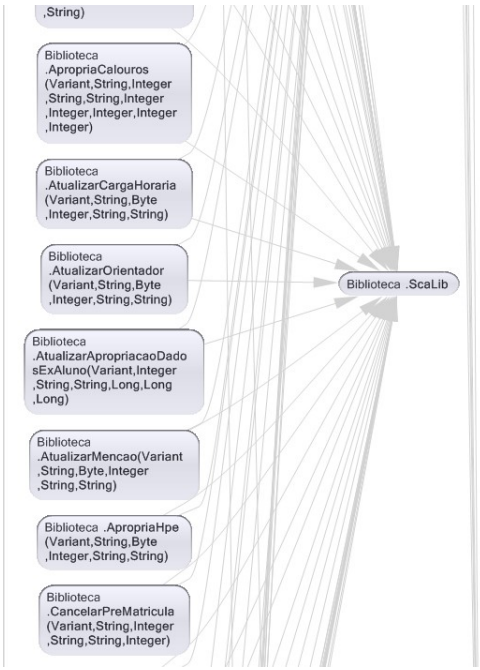

Figura 3.11: Exemplo de um componente estável através da representação de um grafo..

Na matriz de dependências, a métrica de instabilidade pode ser observada através da ocorrência várias células azuis em uma mesma linha, como mostra a figura Figura 3.12. A linha em destaque na cor vermelha, possui várias células na cor azul, indicando que outras entidades do sistema utilizam esta entidade. Sendo assim, esse elemento possui várias dependências e uma alteração nele pode afetar seus vários elementos no sistema,

logo é um candidato a módulo arquitetural. Esse mesmo módulo observado na matriz de dependência, também foi classificado pelos algoritmos de clusterização, como um módulo da arquitetura, e isso ajuda a ter confiança que este é um modulo arquitetural.

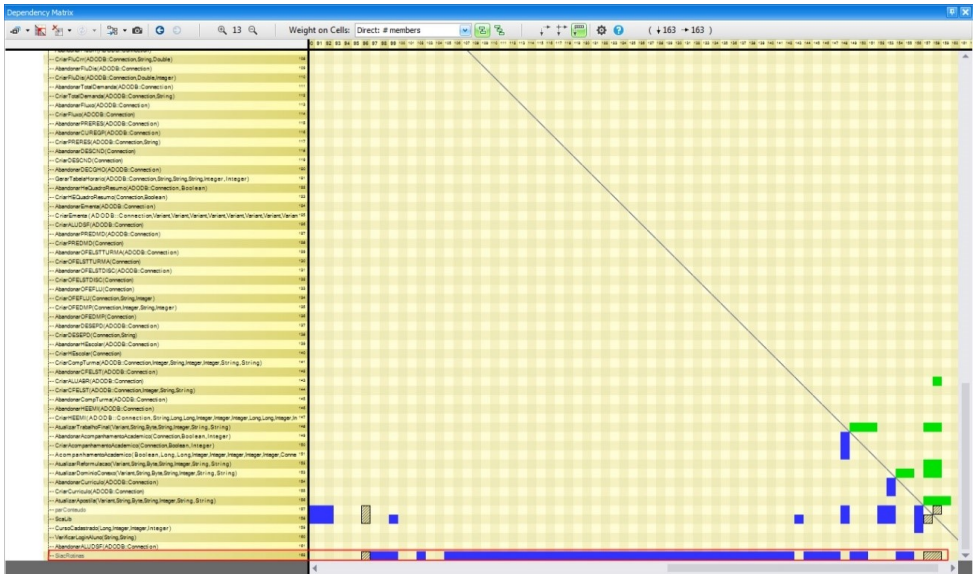

Figura 3.12: Exemplo de um componente estável em destaque através da cor vermelha..

Apesar dos resultados semelhantes entre as duas abordagens, elementos arquiteturais observados pela técnica de visualização não foram apontados de forma consistente pela técnica de clusterização. Tais elementos representam módulos arquiteturais, contudo, o grau de acoplamento entre eles não é tão forte como os outros módulos mapeados pela clusterização, e isso pode ter afetado os resultados obtidos por meio desta técnica. Porém, é possível observar que, devido ao aumento da configuração "População", nos algoritmos *hill-climbing*, o resultado obtido apresentou uma maior semelhança com o resultado da clusterização. Desta forma, acredita-se que, com um maior tempo de execução, seja mais provável que os resultados sejam similares.

Capítulo 4

Estudo Experimental

A investigação empírica apresentada neste estudo experimental utiliza a abordagem quantitativa, na qual o objetivo da pesquisa é obter uma relação numérica entre diversas variáveis ou alternativas em análise. O objetivo do experimento é avaliar o uso das técnicas de visualização de *software* e de clusterização de dados no contexto de uma recuperação de arquitetura de *software*.

4.1 Planejamento

Esta seção descreve o plano de estudo que visa demonstrar como o experimento foi projetado. Isso permite a execução de outros estudos com base no mesmo plano [17].

A Figura 4.1 ilustra o *design* do experimento. Os participantes foram divididos em dois grupos. Participantes do Grupo 1 iniciam a extração da arquitetura nos sistemas em Visual Basic utilizando a técnica de clusterização. Em seguida, é apresentado a segunda técnica, neste caso visualização de *software*, e solicitado a produção de um segundo modelo, aplicando ambas as técnicas. O mesmo procedimento é realizado nos sistemas em Java, porém na ordem invertida, isto é, começando com visualização de *software* e, em seguida, clusterização. Para os participantes do grupo 2, o mesmo procedimento é aplicado, porém começando com a técnica de visualização de *software*.

A análise foi elaborada tendo como base um plano *Goal Question Metric* (GQM), definido na Tabela 4.1.

A questão de pesquisa subjacente ao experimento é a seguinte:
RQ1: O uso da técnica de visualização de *software*, juntamente com a técnica de clusterização, aumenta a exatidão do modelo arquitetural produzido em comparação com o uso de somente uma das técnicas?

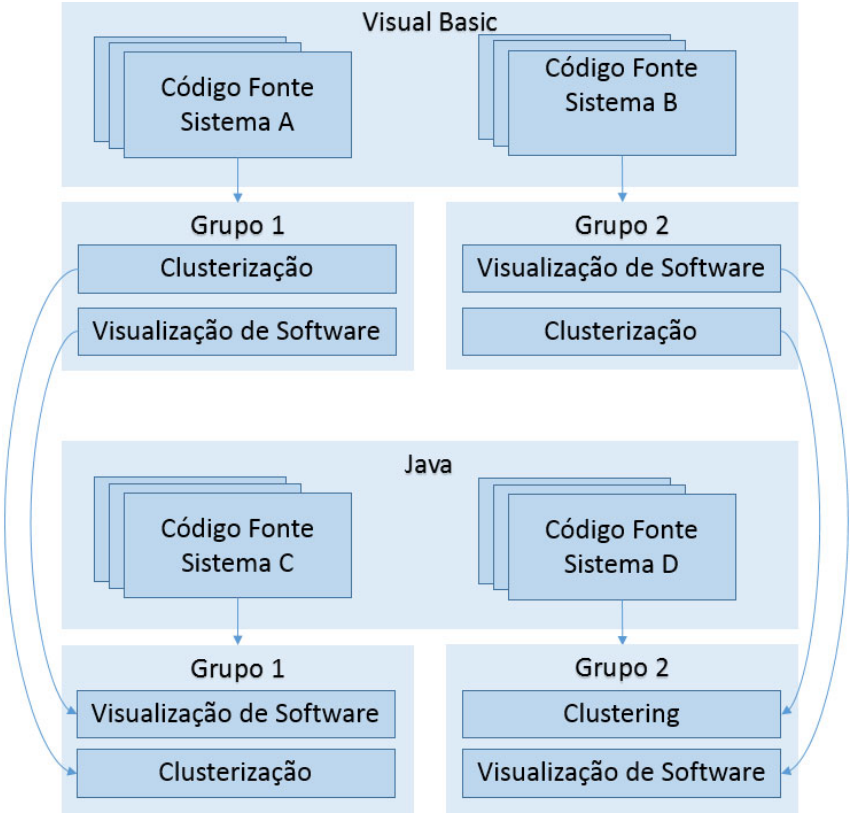

Figura 4.1: Design do experimento.

Propósito	Avaliar
Enfoque	Exatidão dos modelos produzidos
Objeto	Clusterização e visualização para recuperação de arquitetura
Ponto de vista	Arquiteto de software/responsável pela recuperação de arquitetura
Contexto	Sistemas da Universidade de Brasília

Tabela 4.1: Definição da avaliação do processo de recuperação de arquitetura.

4.1.1 Objeto de experimento

O estudo utiliza quatro sistemas de *software*, escritos em duas linguagens de programação, sendo elas: Visual Basic e Java. A Tabela 4.2 apresenta informações relevantes sobre as características de cada sistema objeto do experimento. Os sistemas A e B são sistemas legados ainda em funcionamento na Universidade de Brasília. Tais sistemas são utilizados para gerenciar a vida acadêmica dos estudantes na Universidade. Os sistemas C e D, desenvolvidos na plataforma Java, são utilizados em rotinas administrativas da instituição.

Nome do Sistema	Linguagem	LOC	Nº Métodos	Nº Arquivos
Sistema A-SAE	Visual Basic	25425	1551	133
Sistema B-SIGRA	Visual Basic	36169	2828	218
Sistema C-SIEX	Java	19609	2699	195
Sistema D-SISRU	Java	14238	1734	178

Tabela 4.2: Sistemas objetos utilizados no experimento

4.1.2 Definição das hipóteses

Hipóteses nula: H_0= Para recuperação de uma visão arquitetural de um *software*, utilizar somente uma técnica (somente clusterização ou somente visualização de *software*) produz o mesmo resultado que utilizar as duas técnicas em conjunto. Ou seja:

$$H_0 : \mu_{\text{UmaTecnica}} = \mu_{\text{DuasTecnicas}}$$

Hipótese alternativa:

H_1= Para recuperação de uma visão arquitetural de um *software*, utilizar somente uma técnica (somente clusterização ou somente visualização de *software*) não produz o mesmo resultado que utilizar as duas técnicas em conjunto. Dessa forma:

$$H_1 \neq H_0$$

4.1.3 Seleção dos indivíduos

Os participantes do experimento são profissionais da área de desenvolvimento de sistema da Universidade de Brasília, com diferentes níveis de experiência no campo de análise e desenvolvimento de sistemas. No total, são dez participantes, sendo oito ocupantes do cargo de Analista de Tecnologia da Informação e dois ocupantes do cargo de Técnico de Tecnologia da Informação.

No início de cada sessão, os participantes foram questionados sobre o nível de conhecimento a cerca de aspectos relevantes ao processo de recuperação de arquitetura dos sistemas em análise. A Figura 4.2 detalha a expertise dos participantes do Grupo 1. O eixo y representa o percentual de profissionais com um determinado nível de expertise.

A Figura 4.3 detalha informações sobre o nível de conhecimento dos participantes do Grupo 2.

4.1.4 Variável dependente

O resultado de um experimento, o qual deve ser quantitativo, é referido como variável dependente [17]. Nesse sentido, a variável dependente do experimento é a exatidão dos

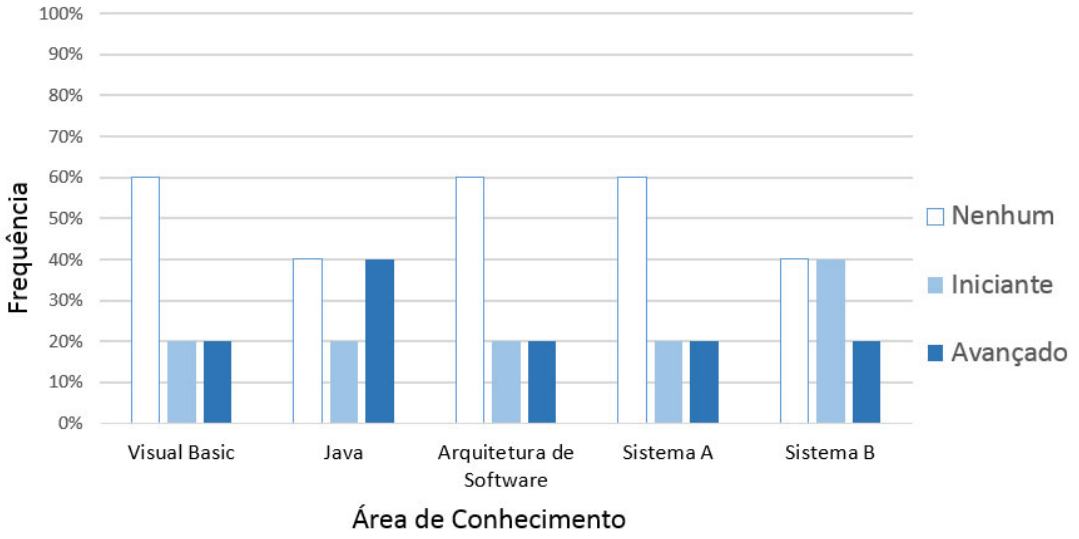

Figura 4.2: Conhecimento dos participantes do Grupo 1.

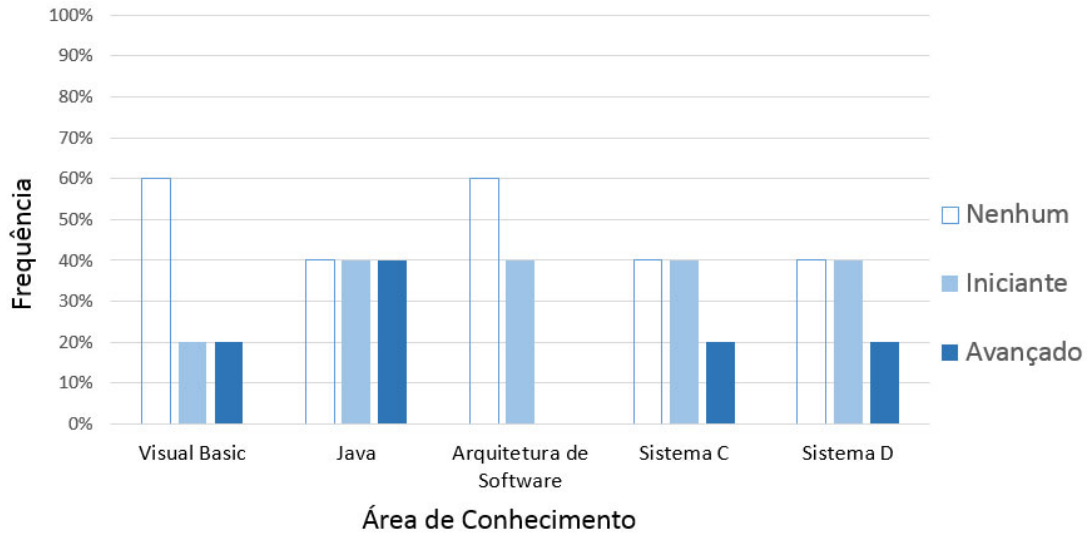

Figura 4.3: Conhecimento dos participantes do Grupo 2.

modelos produzidos por cada participante em comparação com o modelo produzido por um especialista no domínio.

4.1.5 Variáveis independentes

Variável independente é o fator que afeta a variável dependente do experimento. A experimentação tem por objetivo analisar as influências desses fatores sobre o valor das variáveis dependentes [17]. Neste contexto, a variável independente do experimento é definida como: a abordagem de recuperação de arquitetura utilizando somente uma técnica, seguida da recuperação de arquitetura utilizando as duas técnicas em conjunto.

Tabela 4.3: Distribuição dos objetos do experimento entre os tratamentos.

Ordem		Visual Basic		Java	
1º Técnica	2º Técnica	Sistema A	Sistema B	Sistema C	Sistema D
Visualização	Clusterização	T1	-	T3	-
Clusterização	Visualização	-	T2	-	T4

4.1.6 Tratamento

Os possíveis valores dos fatores durante cada experimento são chamados de tratamento. O termo "tratamento" tem origem nos primórdios dos projetos experimentais, concebido inicialmente com a experimentação agrícola [17].

O tratamento aplicado a este estudo experimental é o uso da abordagem de recuperação de arquitetura, utilizando as técnicas de visualização de *software* e de clusterização em diferentes contextos. Neste sentido, são utilizados sistemas escritos em diferentes linguagens de programação, e é alterada a ordem de execução das técnicas a cada seção. A Tabela 4.3 ilustra a distribuição do tratamento utilizado no experimento.

Para os tratamentos T1 e T3, inicia-se a recuperação utilizando a técnica de visualização de *software*, seguida pela técnica de clusterização. Já nos tratamentos T2 e T4, o processo de recuperação se inicia com a técnica de clusterização, para, em seguida, ser utilizada à técnica de visualização de *software*.

4.2 Execução do Experimento

A recuperação da arquitetura é um processo interpretativo e interativo, que envolve muitas atividades, sendo assim, demanda um longo período de execução. Devido a isso, o experimento busca recuperar somente a estrutura geral da arquitetura do sistema. Para isso, foram apresentados a cada participante os conceitos necessários, bem como os elementos essenciais para a execução dessa atividade. O resultado é a recuperação da arquitetura real implementada no código-fonte. Antes do início de cada sessão, os participantes realizaram um treinamento em cada ferramenta utilizada para recuperação dos sistemas objetos. Durante o treinamento, os participantes eram instruídos sobre formas de identificar a arquitetura por meio dos artefatos apresentados por tais ferramentas. A duração do treinamento tinha a duração de no mínimo 10 e no máximo 20 minutos.

A análise dos sistemas objetos na linguagem Visual Basic por meio de visualização de *software*, envolveu o uso da ferramenta VBDepend. Esta ferramenta oferece diversos mecanismos que facilitam a exploração da arquitetura de implementação de um sistema na linguagem Visual Basic com base na visualização de grafos de dependências e matrizes de

dependência estruturadas. A Figura 4.4 destaca uma matriz de dependência estruturada extraída por meio do uso da ferramenta VBDepend.

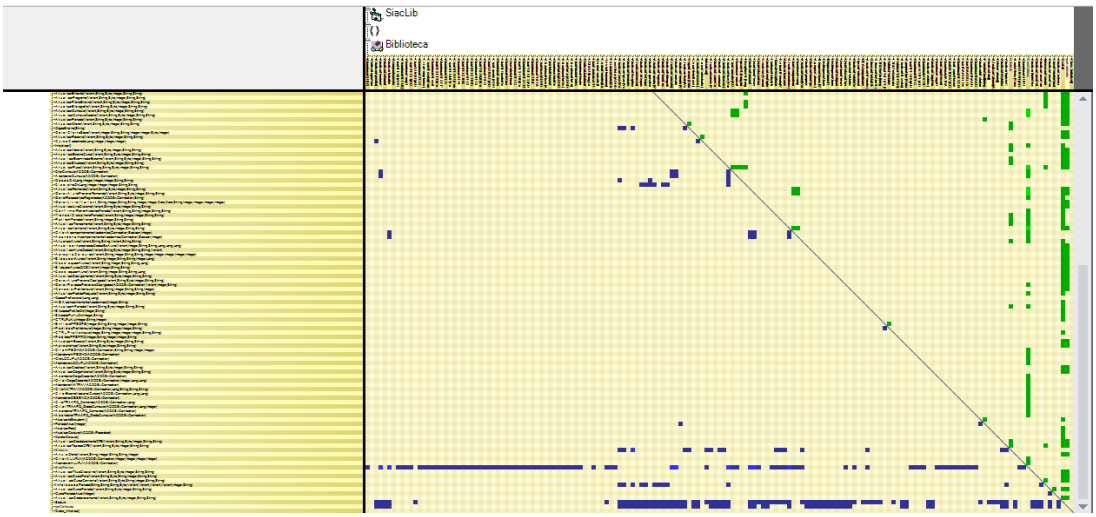

Figura 4.4: Matriz de dependências obtida por meio da ferramenta VBDepend.

Para análise dos sistemas em Java, foram utilizadas as ferramentas de análise estáticas X-Ray e Architexa.

A ferramenta de visualização de *software* X-Ray é um *software* de código aberto, disponível por meio de um *plug-in* para a IDE Eclipse. Por meio desta ferramenta é possível obter visões de dependências de classe-pacote e visões de complexidade sistêmica em projetos Java.

Um exemplo de uma visão de complexidade sistêmica pode ser visto na Figura 4.5. Neste tipo de visão, classes são representadas por retângulos; a largura dos retângulos representa o número de métodos implementados em uma classe específica, enquanto o tamanho do retângulo representa o número de linhas de código da classe. A aresta de ligação representa o relacionamento entre as classes. Os nós podem ser ajustados como em uma árvore vertical, destacando a hierarquia das classes.

A visão de dependência de classe e pacote ajuda a obter um entendimento da estrutura geral do sistema. Nesta visão, ilustrada na Figura 4.6, classes e pacotes são dispostos em um círculo bidimensional, ligados uns aos outros por uma aresta. As arestas representam as dependências entre os elementos do *software*. Tais arestas podem possuir pesos, os quais destacam quão forte é a dependência entre as entidades. Quanto mais escuro e espesso é a ligação entre dois nós, mais forte é a dependência entre eles.

Outra ferramenta utilizada para a visualização do código-fonte, e disponível para os participantes do experimento, é a Architexa. Esta ferramenta permite a criação de dife-

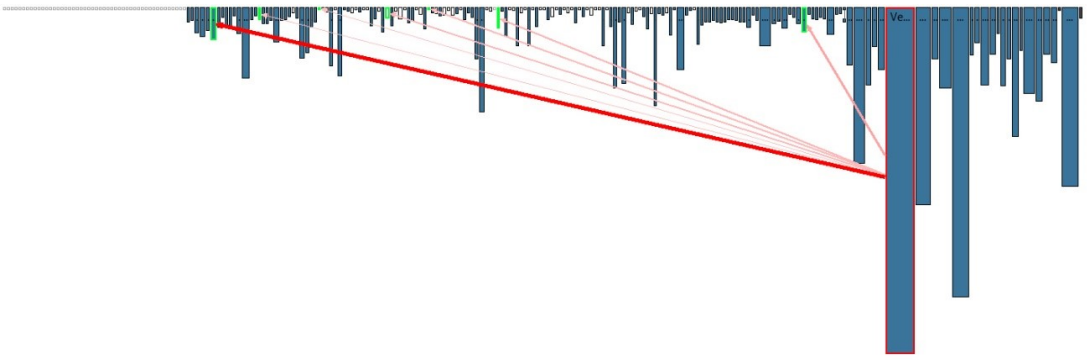

Figura 4.5: Visão de complexidade sistêmica extraída de um sistema em Java.

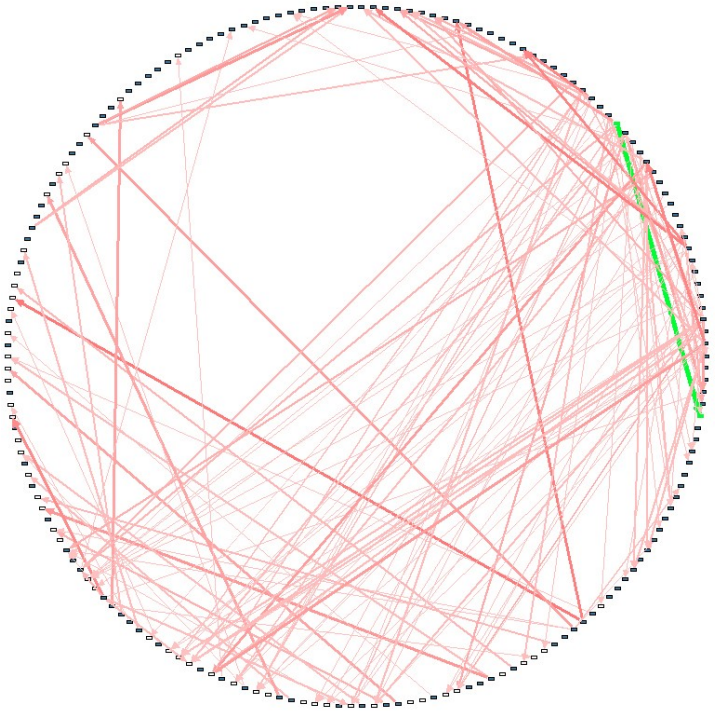

Figura 4.6: Visão de dependência de classe e pacote de um sistema em Java.

rentes diagramas a partir da análise estática do código. Ela também está disponível na forma de um *plug-in* para a IDE Eclipse.

Com o uso desta ferramenta, os participantes do experimento podem explorar diferentes aspectos do código-fonte, como em um diagrama de camadas, visto na Figura 4.7. O diagrama de camadas agrupa classes baseado em seus respectivos diretórios ou pacotes, e ilustra as relações de dependência entre eles. Também, diferentes métricas são utilizadas para destacar aspectos importantes do sistema, como, por exemplo, o tamanho do retângulo que representa as entidades do *software*. Entidades do *software* que contêm

uma grande quantidade de código são representadas por retângulos proporcionais ao seu tamanho, o que facilita a identificação de aspectos importantes da estrutura do código.

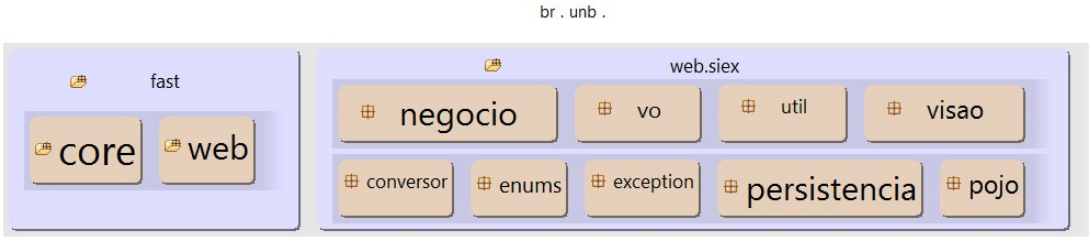

Figura 4.7: Diagrama de camadas extraído do código-fonte.

Por meio da análise do diagrama de camadas, é possível identificar as relações de dependência entre as entidades do *software*. Quando selecionada uma entidade do *software*, a ferramenta de visualização revela as suas respectivas dependências, por meio de uma seta, que indica a origem e o destino da relação, como ilustrado na Figura 4.8. Quanto mais espessa a seta, maior a correlação entre os elementos. Além disso, as cores de cada retângulo são alteradas para enfatizar as dependências.

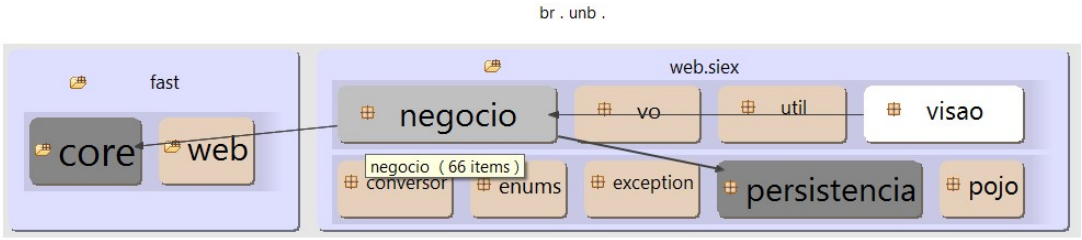

Figura 4.8: Relações entre entidades do sistema em um diagrama de camadas.

Além da análise de alto nível dos pacotes e diretórios, o usuário pode explorar detalhes do relacionamento entre classes do sistema. Ao expandir o diagrama, é possível examinar classes pertencentes a cada pacote e identificar as relações entre eles, como demonstrado na Figura 4.9. Esse tipo de decomposição ajuda a ter um entendimento das relações entre classes e módulos, auxiliando na compreensão tanto das funcionalidades quanto das dependências existentes no código-fonte.

A ferramenta Bunch [28] foi utilizada para a abordagem de clusterização, uma vez que ela é disponibilizada de forma gratuita e possui uma interface gráfica intuitiva. Esta ferramenta agrupa entidades e dependências do código-fonte em módulos significativos.

Como entrada para o processo de clusterização é necessária a extração de um Model Dependency Graph (MDG). Para os sistemas-objetos do experimento, essa extração foi feita por meio da análise estática do código. A Figura Figura 4.10 ilustra um MDG extraído de um dos sistemas-objetos.

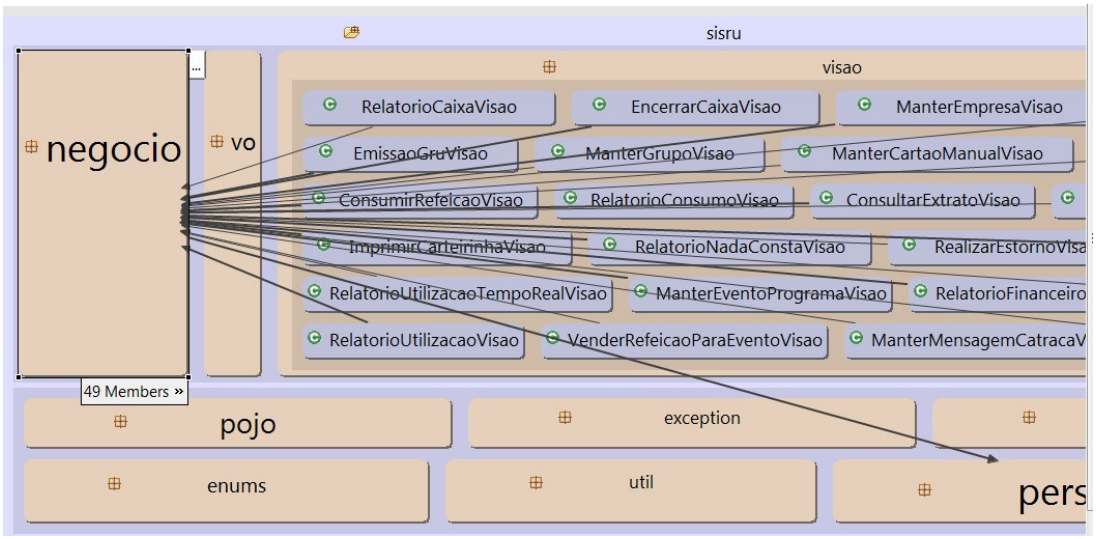

Figura 4.9: Decomposição do sistema por meio do diagrama de camadas.

```
1  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
2  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarFormando
3  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.GerarAlunoProvavelFormando
4  Biblioteca.AbandonarAcompanhamentoAcademico Biblioteca.AtualizarMonitoria
5  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarDominioConexo
6  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarCurriculo
7  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarCurriculoCadeia
8  Biblioteca.AbandonarCurriculo Biblioteca.AtualizarFluxo
9  Biblioteca.AcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
10 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarTrabalhoFinal
11 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarFormando
12 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.GerarAlunoProvavelFormando
13 Biblioteca.CriarAcompanhamentoAcademico Biblioteca.AtualizarMonitoria
14 Biblioteca.CriarCurriculo Biblioteca.AtualizarDominioConexo
15 Biblioteca.CriarCurriculo Biblioteca.AtualizarCurriculo
16 Biblioteca.CriarCurriculo Biblioteca.AtualizarCurriculoCadeia
17 Biblioteca.CriarCurriculo Biblioteca.AtualizarFluxo
18 Biblioteca.CursoCadastrado Biblioteca.AtualizarApostila
```

Figura 4.10: DGM extraído do sistema Sigra.

A ferramenta Bunch possui uma interface gráfica intuitiva, como ilustra Figura 4.11. Os participantes do experimento são instruídos a selecionar o MDG do sistema em análise, escolher entre o algoritmo *hill-climbing* para executar o processo de clusterização, uma vez que os algoritmos genéticos e busca exaustiva, poderiam consumir tempo em demasia.

Os participantes podem alterar os parâmetros de entrada dos algoritmos e executar o processo de clusterização da ferramenta quantas vezes achar necessário. A saída do processo é um arquivo contendo os módulos determinados automaticamente pelo algoritmo selecionado. Por intermédio da análise da saída, os participantes compõem um modelo da visão de módulo do sistema.

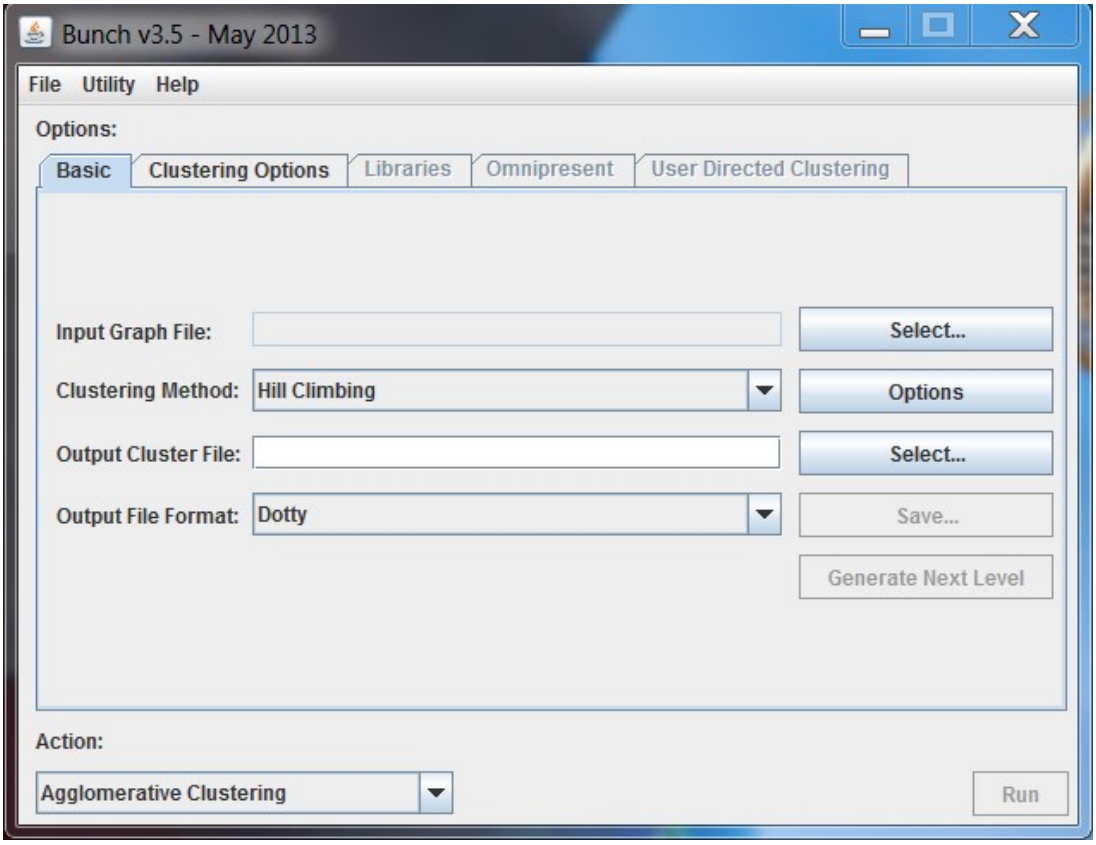

Figura 4.11: Screenshot da ferramenta Bunch.

4.3 Análise

Para calcular a exatidão dos modelos arquiteturais obtidos, e responder as questões do experimento, foi utilizado o coeficiente de similaridade de Jaccard, definido pela fórmula:

$$Sj = \frac{a}{a+b+c}$$

Onde: Sj é o coeficiente de Jaccard; a= número de módulos comuns em ambos os modelos; b= número de módulos no modelo B, mas não em A; c=número de módulos no modelo A, mas não em B. No experimento, o modelo B foi produzido por especialistas do domínio, tais modelos podem ser observados na Figura 4.12.

Cada participante elaborou dois modelos por sessão. No primeiro modelo, o participante utiliza somente uma técnica (ou clusterização ou visualização de *software*). Após o término do primeiro modelo, é apresentado a outra técnica e solicitada a produção de um modelo final, com base nas duas técnicas. Assim, será possível calcular a similaridade do modelo produzido utilizando somente uma técnica e comparar com a similaridade do modelo produzido utilizando as duas técnicas. A figura 4.13 demostra um exemplo

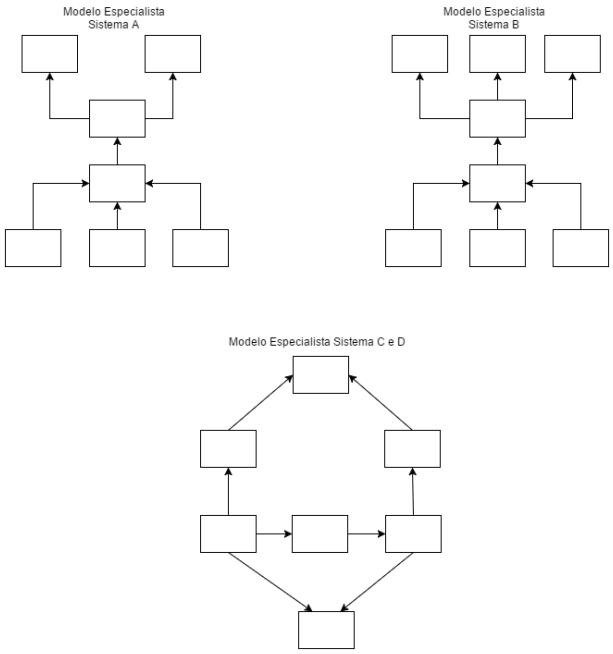

Figura 4.12: Modelos produzidos por especialistas no domínio.

de modelos produzidos por um participante comparados aos modelos produzidos pros especialista no domínio.

4.4 Resultados

Por meio dos resultados obtidos, calculados a partir da comparação dos modelos produzidos pelos participantes com modelos produzidos por um especialista no domínio, foi possível investigar o uso da abordagem de recuperação de arquitetura utilizando as técnicas de clusterização e visualização de *software*.

A Figura 4.14 apresenta a pontuação obtida pelos participantes do Grupo 1 e Grupo 2, nos sistemas em Visual Basic e Java. Por meio da análise dos resultados é possível observar que a pontuação obtida utilizando uma técnica foi inferior à pontuação obtida utilizando duas técnicas.

Primeiramente, foram analisados os resultados dos sistemas escritos em Visual Basic. A Tabela 4.4 descreve os dados da comparação entre as pontuações obtidas utilizando uma e duas técnicas para recuperação da arquitetura. Por meio dela é possível observar que a exatidão média dos modelos utilizando uma técnica foi de aproximadamente 51%, a exatidão média dos modelos obtidos utilizando duas técnicas foi de aproximadamente 77%. Uma diferença de 26%.

Foi utilizado o teste-t pareado para analisar a significância dos resultados obtidos. Este teste é utilizado quando a observação da primeira amostra é pareada com a observação

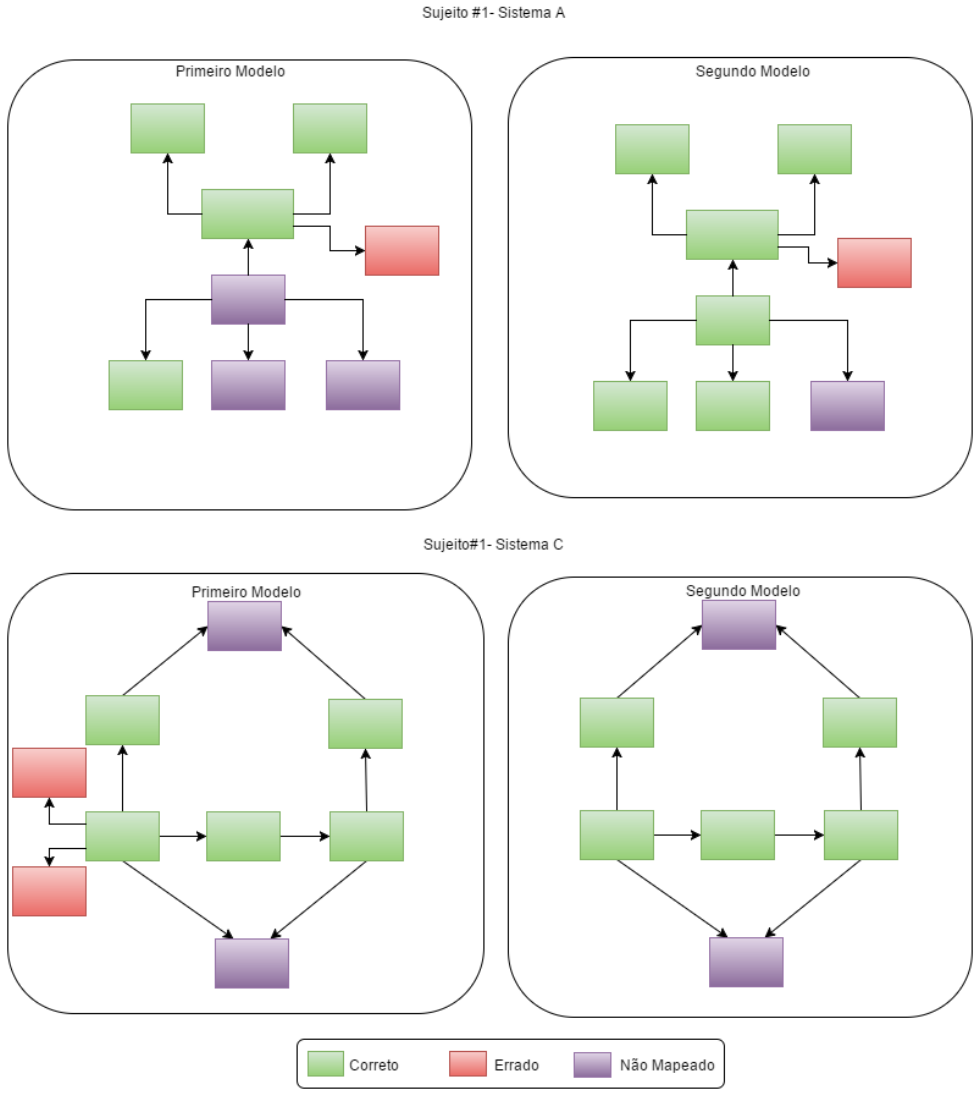

Figura 4.13: Exemplo de modelos produzidos por um participante.

da segunda amostra. Uma vez definido o uso do teste-t pareado para análise das médias, é necessária a observação das seguintes suposições: a observação deve consistir de dois grupos relacionados; não deve existir valores extremos significativos; as diferenças devem seguir uma distribuição aproximadamente normal.

Todas essas exigências foram observadas ao analisar os resultados obtidos. A Tabela 4.5 detalha o teste-t pareado para os sistemas em Visual Basic. Para interpretar o resultado é necessário observar o valor da coluna *Sig.(2-tailed)*, também conhecido como p-valor. Esse valor determina se as médias são estatisticamente diferentes. Caso o valor do *Sig.(2-tailed)* seja maior que 0.05 então não existe significância estatística entre as duas condições. Por outro lado, se o valor for inferior ou igual a 0.05 então existe uma significância estatística entre as duas condições. O nível de significância de 0.05 reflete o intervalo de confiança de 95% usado como parâmetro para os cálculos.

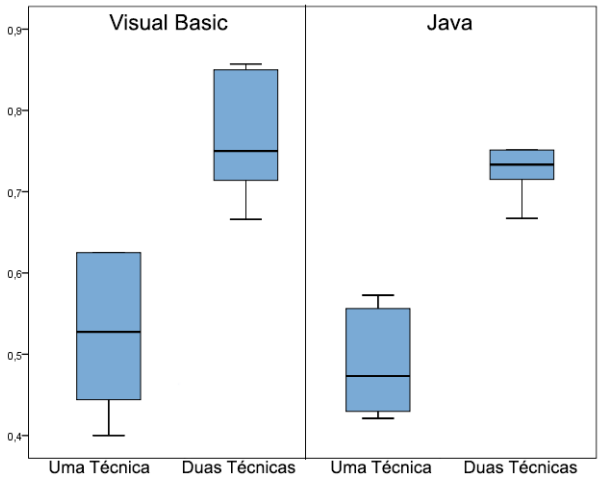

Figura 4.14: Pontuação dos participantes do experimento nos sistemas objeto.

	Técnica	N	Média	Desvio Padrão	Erro Padrão
Pontuação	Uma Técnica	10	0.517	0.091	0.028
	Duas Técnicas	10	0.771	0.077	0.024

Tabela 4.4: Tabela descritiva dos dados dos sistemas em Visual Basic.

Através da análise dos dados é possível concluir que existem diferenças significativas entre as pontuações obtidas utilizando uma técnica (Média=0.517; Desvio Padrão=0.0915) e duas técnicas (Média=0.771; Desvio Padrão=0.771); t(9)=-10.046, p=0.000. Assim, com 95% de confiança, é possível concluir que existe um acréscimo no exatidão dos resultados de 0.196 a 0.310 quando utilizado duas técnicas para recuperação da arquitetura.

	Diferença Pareada					t	df	Sig. (2-tailed)
	Média	Desvio Padrão	Erro Padrão Médio	95% Intervalo de Confiança				
				Inferior	Superior			
UmaTécnica-DuasTécnicas	-0.253	0.079	0.025	-0.310	-0.196	-10.046	9	0.000

Tabela 4.5: Teste-t pareado para sistemas em Visual Basic.

Em seguida, foi feita uma análise dos sistemas em Java. A Tabela 4.6 apresenta estatísticas descritivas para a comparação das abordagens. Por meio da análise dos resultados é possível observar que a média dos modelos obtidos utilizando uma técnica foi de aproximadamente 48%; quando utilizadas as duas técnicas, a média obtida foi de aproximadamente 72%. Uma diferença de 24%.

Para analisar se os resultados obtidos apresentam significância estatística, foi realizado um teste-t pareado. O resultado pode ser o observado na Tabela 4.7. Diante do exposto, é possível concluir que existe uma diferença significativa entre as médias obtidas utilizando

	Técnica	N	Média	Desvio Padrão	Erro Padrão
Pontuação	Uma Técnica	10	0.483	0.060	0.019
	Duas Técnicas	10	0.728	0.059	0.018

Tabela 4.6: Tabela descritiva dos dados dos sistemas em Java.

uma técnica (Média=0.483, Desvio Padrão= 0.060) e duas técnicas (Média=0.72, Desvio Padrão=0.59); t(9)=-14.507, p= 0.000. Assim, com 95% de confiança, a pontuação dos modelos obtidos tiveram um incremento que varia de 0.207 a 0.283 quando utilizado duas técnicas para recuperação da arquitetura.

	Diferença Pareada					t	df	Sig. (2-tailed)
				95% Intervalo de Confiança				
	Média	Desvio Padrão	Erro Padrão Médio	Inferior	Superior			
UmaTécnica-DuasTécnicas	-0.245	0.053	0.016	-0.283	-0.207	-14.507	9	0.000

Tabela 4.7: Teste t pareado para comparação dos resultados nos sistemas em Java.

Após a analise, é possível concluir que utilizar duas técnicas produziu um melhor resultado do que utilizar somente uma técnica em um processo de recuperação de arquitetura de um sistema de *software*. Assim, é possível responder a RQ1, onde é questionado se a utilização das duas técnicas aumenta a exatidão dos modelos arquiteturais obtidos. Diante do exposto, é possível afirmar que a utilização das duas técnicas afetou positivamente na exatidão dos modelos produzidos, uma vez que todos os resultados apontaram uma melhoria na exatidão dos modelos quando utilizadas as duas técnicas em conjunto, assim podemos refutar a H_0.

Para reforçar os resultados obtidos, foi realizado uma análise sobre o efeito da linguagem de programação nos resultados. A Figura 4.15 detalha a pontuação dos grupos quando levado em conta à média dos resultados obtidos utilizando tanto uma técnica quanto duas técnicas. Por meio de uma análise visual da figura, é possível observar que as duas linguagens apresentaram resultados similares.

A Tabela 4.8 detalha informações relacionadas a linguagem de programação utilizada pelos sistemas objetos e a sua influência nos resultados. É possível observar que nos em sistemas Visual Basic a média de exatidão dos modelos produzidos foi de aproximadamente 60%. Em sistemas escritos em Java, a exatidão obtida foi de aproximadamente 64%. Esses valores são relativos à média dos resultados obtidos utilizando tanto uma técnica quanto duas técnicas. Uma vez que cada linguagem é analisada duas vezes no mesmo grupo, então, para este caso, N é igual a 20.

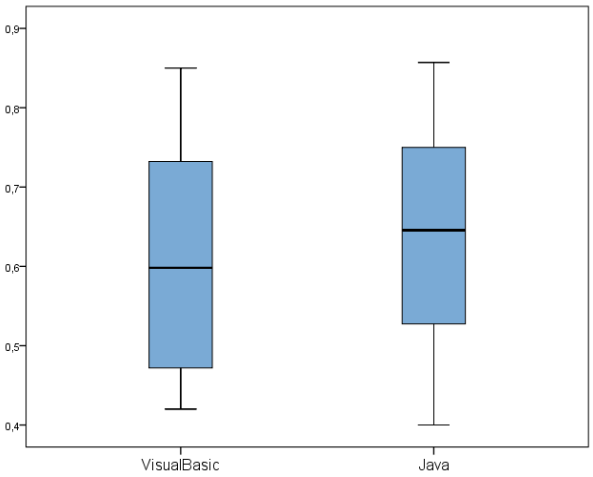

Figura 4.15: Resultados relativos à média dos resultados obtidos utilizando tanto uma técnica quanto duas técnicas.

	Linguagem	N	Média	Desvio Padrão	Erro Padrão
Par1	Visual Basic	20	0.605	0.138	0.031
	Java	20	0.644	0.153	0.034

Tabela 4.8: Tabela descritiva comparando as linguagens de programação.

Para calcular a significância das médias e determinar se a linguagem de programação afeta o processo de recuperação, foi conduzido um teste-t pareado. Os resultados podem ser observados na Tabela 4.9. Por meio da análise dos dados, é possível concluir que não existe diferença significativa entre as médias obtidas nos sistemas em Visual Basic (Média= 0.605, Desvio Padrão = 0.138) e sistemas em Java (Média= 0.644, Desvio Padrão= 0.153); t(19)= -1.867, p = 0.007. Dessa forma é possível responder a Questão 2: diante do apresentado, é possível afirmar que a linguagem de programação não teve efeito estatisticamente significante nos resultados do experimento, uma vez que o valor do *Sig(2-tailed)* obtido é maior que 0.05. Logo, a média das diferenças está no intervalo (-0.082;0.004), o qual incluí o 0.

	Diferença Pareada					t	df	Sig. (2-tailed)
	Média	Desvio Padrão	Erro Padrão Médio	95% Intervalo de Confiança				
				Inferior	Superior			
Visual Basic-Java	-0.0387	0.0928	0.020	-0.082	0.004	-1.867	19	0.077

Tabela 4.9: Teste-t pareado para comparação das duas linguagens de programação.

Por fim, é possível analisar se a ordem de execução das técnicas tiveram efeito nos resultados obtidos. A Figura 4.16 ilustra a comparação entre a ordem de execução. Em

uma primeira observação é possível notar que a ordem não afetou os resultados, uma vez que visualmente os resultados são similares.

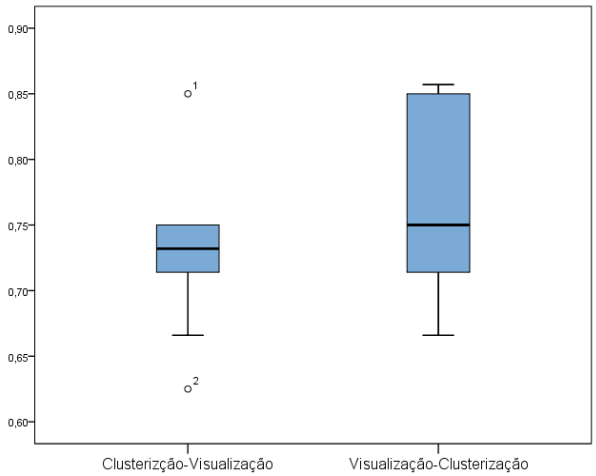

Figura 4.16: Comparação da ordem utilizada no processo de reconstrução nos resultados obtidos.

A Tabela 4.10 detalha os resultados descritivos dos dados relativos a ordem de execução da abordagem. Quanto utilizado a técnica de visualização de *software* seguida da técnica de clusterização, a exatidão dos modelos obtidos foi de aproximadamente 77%. Por outro lado, utilizando primeiramente a técnica de clusterização seguida da técnica de visualização de *software*, a exatidão dos modelos obtidos foi de aproximadamente 72%.

	Ordem	N	Média	Desvio Padrão	Erro Padrão
Par1	Visualização-Clusterização	10	0.771	0.077	0.024
	Clusterização-Visualização	10	0.728	0.059	0.018

Tabela 4.10: Tabela descritiva dos dados relacionados a ordem de execução das técnicas.

Para verificar se existe significância entre as médias, foi conduzido um teste-t pareado. Os resultados podem ser observados na Tabela 4.11. Por meio da análise dos dados é possível concluir que não existe significância entre as médias obtidas utilizando visualização antes de clusterização (Média= 0.771, Desvio Padrão= 0.077) e clusterização antes de visualização (Média=0.728, Desvio Padrão=0.018); t(9)= 1.456, p=0.179.

Diante do exposto, é possível afirmar que não existem evidências estatísticas que possam afirmar que a ordem de execução da abordagem teve influência nos resultados obtidos.Uma vez que valor do *Sig(2-tailed)* é maior que 0.05, resultando nos intervalos (-0.023,0.109) no qual o 0 está incluído.

	Diferença Pareada					t	df	Sig. (2-tailed)
	Média	Desvio Padrão	Erro Padrão Médio	95% Intervalo de Confiança				
				Inferior	Superior			
VisualizaçãoClusterização-ClusterizaçãoVisualização	0.0427	0.0927	0.029	-0.023	0.109	1.456	9	0.179

Tabela 4.11: Teste-t pareado para comparação da ordem das técnicas.

4.5 Validade Interna

Todos os participantes do experimento têm experiência em análise e desenvolvimento de sistemas, o que contribui para a representatividade do conjunto de analistas e desenvolvedores de sistemas que possam necessitar de um processo de reconstrução arquitetural de sistema de *software*. Os participantes dos dois grupos do experimento possuem características semelhantes, tanto no conhecimento dos sistemas objetos quanto na linguagem de programação utilizada por tais sistemas, de forma que tal equivalência entre os grupos diminui aspectos que podem interferir nos resultados.

O uso de duas linguagens de programação, Visual Basic e Java, contribui para a representatividade de um ambiente organizacional, no qual não existe uma homogeneidade em relação à linguagem de programação utilizada pelos sistemas na empresa. Apesar de o fato dos participantes serem separados em dois grupos e executarem o experimento em sistemas objetos diferentes, a abordagem de recuperação foi realizada em sistemas com arquiteturas semelhantes. Devido a esta semelhança, é esperado que os resultados obtidos não sejam influenciados pela linguagem de programação ou diferença entre os sistemas objetos.

Também, os modelos produzidos pelos especialistas no domínio podem não representar a arquitetura real implementada no código-fonte. Contudo, os responsáveis pela criação dos modelos foram os profissionais encarregados pela concepção inicial da arquitetura e evolução da mesma. Assim, tais profissionais possuem um entendimento empírico de todo ciclo de vida relacionado a concepção e evolução da estrutura arquitetural dos sistemas em análise.

4.6 Validade Externa

O tamanho em linha de código e a complexidade dos sistemas objetos podem influenciar a generalização dos resultados obtidos. Porém, tais sistemas apresentam complexidade e tamanho semelhantes à grande parte dos sistemas desenvolvidos para atender necessidades setoriais de uma organização.

Também, o número limitado de participantes do experimento pode não permitir uma generalização do experimento, porém, por meio do método de separação dos participantes entre dois grupos de forma aleatória, é esperada uma diminuição no fator de confusão.

4.7 Discussão

O estudo experimental realizado permitiu observar o impacto do uso das técnicas de visualização de *software* e clusterização de dados no entendimento da arquitetura de sistemas de *software*.

A curva de aprendizagem da abordagem foi relativamente rápida, com um treinamento de aproximadamente 20 minutos, todos os participantes já estavam familiarizados com os diagramas, matrizes e diferentes representações do código-fonte. Também, foi comum aos participantes, após o início da execução da segunda técnica, voltarem aos resultados obtidos com a primeira técnica. Isso demostrar que, sob uma outra perspectiva do sistema em análise, novo conceitos podem ser observados. Dessa forma, favorece o esclarecimento de dúvidas remanescente, assim, permitindo a produção de um modelo final com maior confiança.

Primeiramente, foi analisado se a utilização das técnicas de visualização de *software* e clusterização, em conjunto, fornecem benefícios a um processo de recuperação de arquitetura. Os resultados apontaram que, quando utilizada uma técnica seguida da outra, a exatidão dos modelos produzidos teve um incremento significativo. Isso se deve ao fato das técnicas agirem de maneira complementar, fornecendo informações adicionais. Também, o uso das duas técnicas em conjunto forneceu aos participantes um ponto de vista diferente do *software*, até então não percebido ou não representado, quando comparado ao uso de somente uma técnica. Dessa forma, permitindo uma recuperação da arquitetura com mais precisão. Cabe ressaltar que era esperado melhores resultados para os sistemas na linguagem programação Java. Isso se deve ao fato que, sistemas nessa linguagem, são estruturados utilizando conceitos de pacote bem definidos, sendo assim, mais intuitivo à análise do código. Contudo, o fato dos sistemas em Visual Basic possuírem uma arquitetura bem definida, permitiu aos participantes um melhor entendimento da estrutura do sistema nessa linguagem, resultado em resultados similares para as duas linguagens.

Em segundo lugar, foi realizada uma análise para verificar se a linguagem de programação pode afetar a recuperação da arquitetura. Os resultados apontaram que a linguagem de programação não teve efeito significativo nos resultados. Isso demonstra a flexibilidade das técnicas de clusterização e visualização de *software*. Essa flexibilidade se deve ao fato de que existem diversas ferramentas de visualização de *software* disponíveis, assim, possibilitando a análise de diferentes linguagens de programação. Também, a abordagem de

clusterização, implementada pela ferramenta Bunch foi criada para ser independente da linguagem, uma vez que é necessária apenas a criação de um MDG contendo as relações de dependência entre as entidades do código-fonte, e isso é obtido por meio de técnicas de análise estática do código-fonte.

Por fim, foi possível observar que a ordem de execução das técnicas não afetou significativamente os resultados obtidos. Isto indica que não existe uma ordem específica para a aplicação das técnicas. Dessa forma, a execução das técnicas pode acontecer de forma subjetiva, da maneira que melhor atenda às necessidades do analista responsável pela recuperação da arquitetura. Porém, um ponto que merece ser citado, foi o fato da confiança que os participantes depositaram no processo de clusterização quando esta técnica foi realizada em primeiro lugar. Como grande parte dos participantes não tinham nenhum conhecimento dos sistemas estudados, somente os resultados da clusterização não permitiam um conhecimento profundo da estrutura do *software*. Porém, quando apresentado a técnica de visualização de *software*, os participantes tiveram uma melhor noção dos resultados apresentados pela clusterização e, dessa forma, tiveram mais segurança para produção do modelo final.

Para obter um *feedback* de cada participante sobre as impressões em relação a abordagem de recuperação de arquitetura apresentada, foi solicitado a resposta a três questões:

1. A utilização das técnicas de clusterização e visualização de *software*, em conjunto, foram úteis para a recuperação da arquitetura?
2. É preferível utilizar as técnicas apresentadas, ou somente a análise manual do código-fonte já é suficiente para o entendimento da arquitetura de um sistema?
3. É preferível iniciar o processo recuperação utilizando a clusterização ou a técnica de visualização de *software*?

Em relação a questão 1, todos afirmaram que o uso das duas abordagens foi útil para um melhor entendimento do *software*, pois as técnicas agiram de maneira complementar, fornecendo informações as vezes despercebida quando da utilização de somente uma técnica. Também o uso das duas técnicas, em conjunto, fornecem um ponto de vista diferente do *software*, até então não percebido, ou não representado, quando comparado ou uso de somente uma técnica. Foi observado que os grafos gerados pela abordagem de visualização de *software* são formas intuitivas de mostrar as dependências, porém pode ser de difícil entendimento, quando o número de nós e arestas crescem; a matriz de dependência é menos intuitiva, mas pode ser mais eficiente para representar um grafo grande e complexo. Juntando tais representações, com modelos gerados pela clusterização, favorece a redução do fator de confusão nos resultados obtidos. Assim, várias dúvidas dos participantes foram solucionadas quando feita a união das abordagens.

Na questão 2, todos os participantes concordaram que é preferível a utilização das duas abordagens ao invés da análise manual do código-fonte. Os participantes afirmaram que as técnicas apresentadas forneciam informações mais claras e de fácil entendimento que a leitura manual do código. Também, a forma que o sistema era representado, facilitava tanto o entendimento global da estrutura do *software*, quanto o entendimento das funcionalidades ali implementadas.

Em relação a questão 3, não houve consenso entre os participantes. Enquanto quatro participantes preferiram iniciar com a técnica de clusterização, os outros seis preferiram iniciar com a técnica de visualização de *software*. Os resultados do experimento podem reforçar esse posicionamento, uma vez que, quando verificado a ordem de execução, este fator não afetou significativamente os resultados obtidos.

Capítulo 5

Conclusão

Um dos fatores que podem influenciar o sucesso ou fracasso de um processo de modernização do sistema legado é o entendimento da sua arquitetura. Neste sentido, o tempo gasto para recuperar tais conceitos é tão importante quanto o tempo gasto no planejamento do novo sistema. Isso se deve ao fato de que, para um entendimento completo de um sistema, é necessário primeiro entender a sua arquitetura, pois está é a base que sustenta todas as funcionalidades do sistema. Dessa forma, para que o processo de recuperação de arquitetura seja o mais completo e preciso possível, é essencial o uso de diferentes técnicas da Engenharia de *Software*.

Neste contexto, o uso de técnicas de visualização de *software* para análise e recuperação da arquitetura de um sistema é essencial, uma vez que ela fornece uma maior flexibilidade ao processo de modernização. Por meio desta técnica, é possível obter uma representação compacta de toda a estrutura do código-fonte. Os diversos aspectos do *software*, implementados em inúmeras linhas de código, podem ser representados por um único diagrama, que resume toda essa complexidade. Dessa forma, é possível ao arquiteto de *software* identificar conceitos importantes da estrutura do sistema apenas com a análise visual dos artefatos produzidos pelas ferramentas de visualização. A abstração da complexidade do sistema por meio da visualização de *software* foi observada durante a condução do estudo experimental. Participantes que não tinham o menor conhecimento, tanto com linguagem de programação quanto da estrutura geral do sistemas, em um curto espaço de tempo, desenvolveram um conhecimento significativo do funcionamento do sistema apenas através da observação e interação com os resultados produzidos pelas ferramentas de visualização de *software*.

Em adição, um processo de recuperação de arquitetura utilizando técnicas de clusterização também é eficaz. Uma representação criada automaticamente por um processo de clusterização é uma forma rápida e prática de explorar sistemas complexos, muitas vezes totalmente desconhecido para o analista. O que é um excelente passo inicial para o

entendimento de aspectos importantes do *software*. Em outros casos, o processo de clusterização pode fornecer diferentes perspectivas para o entendimento de funcionalidades do *software*. Durante a condução do estudo experimental, foi observado que os resultados apresentados pela clusterização muitas vezes serviam de guia para o entendimento do código, sendo um passo inicial para o processo de recuperação.

Assim, o uso de ambas as técnicas, visualização de *software* e clusterização, fornecem uma grande gama de representações do sistema em análise. Tais representações irão permitir diferentes visões sobre vários aspectos do *software*, o que contribui para o entendimento de toda estrutura do sistema. Dessa forma, a utilização das referidas técnicas permite uma recuperação da arquitetura de um sistema, de forma ágil e precisa.

A exatidão dos resultados obtidos por meio das duas técnicas foi explorada em um estudo experimental, no qual a recuperação de uma visão arquitetural do *software*, utilizando as duas técnicas em conjunto, apresentou uma melhora significativa sobre a exatidão do modelo arquitetural produzido. Resultado o qual mostra a importância de utilizar diferentes técnicas e representações do *software* para uma completa recuperação da arquitetura do sistema de *software* em análise.

5.1 Trabalhos futuros

Devido à complexidade de uma recuperação de aspectos arquiteturais do *software*, pesquisas complementares devem ser realizadas para torna o processo de recuperação de arquitetura mais ágil e preciso. Como, por exemplo, a integração das abordagens de visualização de *software* e clusterização em uma única ferramenta. Dessa forma, o processo de recuperação de arquitetura poderia obter uma maior agilidade. Uma vez que a integração das duas tecnologias iria permitir uma representação única do *software* e, consequentemente, uma melhor visualização dos resultados, para posterior produção dos artefatos que compõem a arquitetura do sistema.

Também a pesquisa pode ser estendida para levar em consideração aspectos evolucionais do código-fonte do sistema de *software*. Essa atividade requer a análise de grandes quantidades de dados que descrevem o estado atual do *software*, bem como o seu histórico prévio. Assim, é possível obter informações sobre todo o ciclo de desenvolvimento do *software* e entender aspectos arquiteturais que foram base para a evolução do sistema.

De forma complementar, para uma representação completa do sistema, aspectos dinâmicos da arquitetura também podem ser integrados à abordagem de recuperação utilizando técnicas de clusterização e visualização de software. Assim, levando em conta aspectos dinâmicos do *software*, seria possível obter um melhor entendimento da comunicação entre os elementos arquiteturais do *software* em tempo de execução. Dessa forma,

o analista responsável pela recuperação da arquitetura teria em mãos informações essenciais para uma profunda análise das entidades envolvidas na composição da arquitetura implementada pelo sistema.

Referências

[1] Marwan Abi-Antoun e Jonathan Aldrich. A field study in static extraction of runtime architectures. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 22–28. ACM, 2008. 1

[2] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, e Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, second edition, 2010. 5

[3] Jesus Bisbal, Deirdre Lawless, Bing Wu, Jane Grimson, Vincent Wade, Ray Richardson, e Donie O'Sullivan. An overview of legacy system migration. In *4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97 / ICSC '97), 2-5 December 1997, Clear Water Bay, Hong Kong*, page 529, 1997. 2, 3

[4] Trosky B. Callo Arias, Pierre America, e Paris Avgeriou. A top-down approach to construct execution views of a large software-intensive system. *Journal of Software: Evolution and Process*, 25(3):233–260, 2013. 4, 9

[5] P. Caserta e O. Zendra. Visualization of the static aspects of software: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):913–933, July 2011. 16

[6] Alexander Chatzigeorgiou, Nikolaos Tsantalis, e George Stephanides. Application of graph theory to oo software engineering. In *Proceedings of the 2006 International Workshop on Workshop on Interdisciplinary Software Engineering Research*, WISER '06, pages 29–36, New York, NY, USA, 2006. ACM. 15

[7] Mike Danilovic e Bengt Sandkull. The use of dependence structure matrix and domain mapping matrix in managing uncertainty in multiple project. *International Journal of Project Management*, 23(3):193 – 203, 2005. 16, 17

[8] D. Doval, S. Mancoridis, e B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *In Proceedings of Software Technology and Engineering Practice*, pages 73–91, 1998. 31

[9] Stéphane Ducasse e Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009. 1, 5, 6, 7, 8, 10, 14, 34

[10] John Ellson, EmdenR. Gansner, Eleftherios Koutsofios, StephenC. North, e Gordon Woodhull. Graphviz and dynagraph - static and dynamic graph drawing tools. In Michael Junger e Petra Mutzel, editors, *Graph Drawing Software*, Mathematics and Visualization, pages 127–148. Springer Berlin Heidelberg, 2004. 15

[11] Loe Feijs e Roel De Jong. 3d visualization of software architectures. *Commun. ACM*, 41(12):73–78, December 1998. 1, 13, 14

[12] Dharmalingam Ganesan e Mikael Lindvall. ADAM: external dependency-driven architecture discovery and analysis of quality attributes. *ACM Trans. Softw. Eng. Methodol.*, 23(2):17:1–17:51, 2014. 2, 3, 4

[13] Yaser Ghanam e Sheelagh Carpendale. A survey paper on software architecture visualization. 2008. 1, 11, 12, 13, 14

[14] A. K. Jain, M. N. Murty, e P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, September 1999. 1, 23

[15] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recogn. Lett.*, 31(8):651–666, June 2010. 23

[16] Anil K. Jain e Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. 22, 23

[17] Natalia Juristo e Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition, 2010. 46, 48, 49, 50

[18] Rick Kazman e S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 290–299. IEEE, 1998. 1, 4, 37

[19] Rick Kazman, Liam O'Brien, e Chris Verhoef. Architecture reconstruction guidelines. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2003. 5, 6, 8

[20] Dexter Campbell Kozen. *Design and Analysis of Algorithms*. Texts and Monographs in Computer Science. Springer, 1992. 29

[21] Michele Lanza. Codecrawler-lessons learned in building a software visualization tool. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 409–418. IEEE, 2003. 1, 12, 13

[22] Claudia Lopez, Victor Codocedo, Hernan Astudillo, e Luiz Marcio Cysneiros. Bridging the gap between software architecture rationale formalisms and actual architecture documents: An ontology-driven approach. 77(1):66–80, 2012. 4

[23] M. Lungu e M. Lanza. Exploring inter-module relationships in evolving software systems. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 91–102, March 2007. 12

[24] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidović, e Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 69–78, Piscataway, NJ, USA, 2015. IEEE Press. 2

[25] Oded Maimon e Lior Rokach. *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 22, 25

[26] S. Mancoridis, B. S. Mitchell, Y. Chen, e E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 50–, Washington, DC, USA, 1999. IEEE Computer Society. 26, 28, 29, 30

[27] Robert C. Martin e Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. 21

[28] Brian S. Mitchell. A heuristic approach to solving the software clustering problem. In *19th International Conference on Software Maintenance (ICSM*, pages 285–288, 2003. 1, 2, 25, 26, 28, 30, 31, 32, 53

[29] B.S. Mitchell e S. Mancoridis. Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions [clustering results analysis framework and tools]. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 93–102, 2001. 36, 37

[30] Melanie Mitchell. Genetic algorithms: An overview. *Complexity*, 1(1):31–39, 1995. 31

[31] Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, e Ronald J. Koontz. Architectural dependency analysis to understand rework costs for safety-critical systems. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 185–194, New York, NY, USA, 2014. ACM. 15

[32] Martin Pinzger, Harald Gall, Jean-Francois Girard, Jens Knodel, Claudio Riva, Wim Pasman, Chris Broerse, e Jan Gerben Wijnstra. Architecture recovery for product families. In *Software Product-Family Engineering*, pages 332–351. Springer, 2004. 19

[33] M.C. Platenius, M. von Detten, e S. Becker. Archimetrix: Improved software architecture recovery in the presence of design deficiencies. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 255–264, March 2012. 37

[34] Blaine A. Price, Ronald Baecker, e Ian S. Small. A principled taxonomy of software visualization. *J. Vis. Lang. Comput.*, 4(3):211–266, 1993. 11

[35] Claudio Riva. Architecture reconstruction in practice. In *Software Architecture: System Design, Development and Maintenance, IFIP 17th World Computer Congress*

- *TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA3), August 25-30, 2002, Montréal, Québec, Canada*, pages 159–173, 2002. 9

[36] M. Saeed, O. Maqbool, H. A. Babri, S. Z. Hassan, e S. M. Sarwar. Software clustering techniques and the use of combined algorithm. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, pages 301–, Washington, DC, USA, 2003. IEEE Computer Society. 24, 25

[37] Neeraj Sangal, Ev Jordan, Vineet Sinha, e Daniel Jackson. Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10):167–176, October 2005. 16, 17, 18, 19

[38] Lajos Schrettner, Lajos Jenő Fülöp, Rudolf Ferenc, e Tibor Gyimóthy. Visualization of software architecture graphs of java systems: Managing propagated low level dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 148–157, New York, NY, USA, 2010. ACM. 16

[39] Bart Selman e Carla P Gomes. *Hill-climbing Search*. John Wiley and Sons, Ltd, 2006. 29, 30

[40] Z. Sharafi. A systematic analysis of software architecture visualization techniques. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 254–257, June 2011. 11

[41] Mark Shtern e Vassilios Tzerpos. Clustering methodologies for software engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, January 2012. 1, 24, 29

[42] Niels Streekmann, Wilhelm Hasselbring, e Andreas Winter. *Clustering-Based Support for Software Architecture Restructuring*. Springer, 2011. 8

[43] Pang-Ning Tan, Michael Steinbach, e Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. 23, 24

[44] Bedir Tekinerdogan, Frank Scholten, Christian Hofmann, e Mehmet Aksit. Concern-oriented analysis and refactoring of software architectures using dependency structure matrices. In *Proceedings of the 15th Workshop on Early Aspects*, EA '09, pages 13–18, New York, NY, USA, 2009. ACM. 4

[45] A.R. Teyseyre e M.R. Campo. An overview of 3d software visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 15(1):87–105, Jan 2009. 1, 13, 14

[46] Arie Van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, e Claudio Riva. Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 122–132. IEEE, 2004. 3, 4, 8

[47] Haixun Wang, Wei Wang, Jiong Yang, e Philip S. Yu. Clustering by pattern similarity in large data sets. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 394–405, New York, NY, USA, 2002. ACM. 24

[48] Richard Wettel, Michele Lanza, e Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560. ACM, 2011. 1

[49] T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 33–43, Oct 1997. 1, 25