Pós-Graduação em Ciência da Computação

ALESSANDRO BORGES RODRIGUES

Uma Abordagem Gradativa de Modernização de Software Monolítico e em Camadas para SOA

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

Alessandro Borges Rodrigues

Uma Abordagem Gradativa de Modernização de Software Monolítico e em Camadas para SOA

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR: **Prof. Dr. Vinicius Cardoso Garcia**

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R696a Rodrigues, Alessandro Borges
Uma abordagem gradativa de modernização de software monolítico e em camadas para SOA / Alessandro Borges Rodrigues – 2017.
84 f.: il., fig., tab.

Orientador: Vinícius Cardoso Garcia.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.

1. Engenharia de software. 2. Arquitetura de software. I. Garcia, Vinícius Cardoso (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2017-113

Alessandro Borges Rodrigues

Uma Abordagem Gradativa de Modernização de Software Monolítico e em Camadas para SOA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre Profissional em 27 de março de 2017.

Aprovado em: ___/___/_____.

BANCA EXAMINADORA

_____

Prof. Leopoldo Motta Teixeira
Centro de Informática / UFPE

_____

Prof. Leandro Marques do Nascimento
Universidade Federal Rural de Pernambuco

_____

Prof. Vinícius Cardoso Garcia
Centro de Informática / UFPE
(Orientador)

Dedico este trabalho à minha esposa, Rejane, pelo apoio e auxílio durante toda nossa vida juntos, e por me proporcionar o privilégio de me tornar pai.

AGRADECIMENTOS

Agradeço aos meus pais, Dulce e Adail, por terem me ensinado que a educação é um patrimônio para toda a vida e por ter me dado o suporte necessário para seguir neste caminho.

Aos companheiros de mestrado, em especial aos amigos do alojamento do IFPE.

Aos meus colegas de trabalho do IFTO e a esta instituição que, junto com a UFPE e demais instituições, possibilitaram a oportunidade desse mestrado à vários servidores de todo o país.

Aos professores do Centro de Informática da UFPE pelos seus ensinamentos, aos funcionários do CIn que contribuíram para que este curso fosse realizado e especialmente ao meu orientador Vinicius Garcia, pela acompanhamento deste trabalho.

"Se você encontrar um caminho sem obstáculos, ele provavelmente não leva a lugar nenhum."

Frank Clark

RESUMO

A constante evolução tecnológica, tanto de hardware quanto de software, faz com que muitos sistemas tornem-se obsoletos, apesar de ainda atenderem seus requisitos e serem estáveis. Outrora foi a época dos sistemas procedurais, hoje vemos que a própria evolução deles, os orientados a objetos, em muitos casos, se tornaram obsoletos, grandes e complexos, com tecnologias ultrapassadas e contendo centenas ou milhares de classes, sendo esses problemas agravados naqueles que foram construídos de forma monolítica, possuindo assim apenas um único arquivo como resultado. A arquitetura orientada a serviços permite a criação de sistemas com menor complexidade, já que seus serviços possuem baixo acoplamento, permitindo atualizações individuais sem afetar os demais serviços. Porém, a reconstrução dos sistemas já existentes nessa nova arquitetura é inviável, devido ao custo necessário (tempo, mão de obra etc.), sendo a reengenharia deles uma possível solução, que permite a reformulação desses sistemas de uma maneira menos onerosa. Apesar da arquitetura em camadas ser bastante utilizada nos sistemas orientados a objetos, faltam soluções de reengenharia que leve esse fato em consideração, não sendo tão efetivas quando executadas em sistemas com essa arquitetura. Este trabalho busca definir uma abordagem para modernização de sistemas monolíticos, orientados a objetos e que tenham sido desenvolvidos com a arquitetura em camadas, para a arquitetura orientada a serviços, de uma forma semi-automatizada, sem a necessidade de o engenheiro de software possuir um profundo conhecimento do sistema a ser modernizado. No sistema reconstruído, as classes das camadas de negócio e persistência serão agrupadas de acordo com seus relacionamentos, e os métodos das classes de negócio serão disponibilizados como serviços. As etapas da abordagem proposta são constituídas de técnicas, cujas fórmulas e algoritmos podem ser adicionados/transformados em ferramentas que automatizarão o processo. Esta metodologia de modernização permite que os *web services* criados possuam uma quantidade menor de classes, além de menor complexidade em cada serviço, mantendo a funcionalidade original. Isso é conseguido tanto através de refatorações no código original que diminui a quantidade de dependência entre as classes, quanto através da separação de agrupamentos de classes em pedaços menores. Foram obtidos resultados satisfatórios no estudo de caso, como redução de 24% da dependência média entre as classes, diminuição de 80% e 6,33% do tamanho e da complexidade estática do componente (CSC), respectivamente e 100% de sucesso nos testes de regressão.

Palavras-chave: Reengenharia de Software. Migração. SOA. Refatoração. Modernização. Arquitetura de Software.

ABSTRACT

The constant technological evolution, both hardware and software, makes many systems become obsolete, although they still attend their requirements and are stable. Once was the time of procedural systems, today we see that the very evolution of them, the object-oriented, in many cases, have become obsolete, large and complex, with outdated technologies and containing hundreds or thousands of classes, these problems being aggravated in those that were built in a monolithic way, thus possessing only a single file as a result. The service-oriented architecture allows the creation of systems with less complexity, as their services have low coupling, allowing individual updates without affecting other services. However, reconstruction of existing systems in this new architecture is not feasible due to the cost needed (time, labor etc), reengineering them being a possible solution, which allows the reformulation of these systems in a less costly way. Although the layered architecture is the most used in object oriented systems, it lacks reengineering solutions that take this fact into account, not being so effective when executed in systems with this architecture. This work aims to define an approach to the modernization of monolithic and layered systems for service-oriented architecture, in a semi-automated manner, without the need for the sotware engineer has a deep knowledge of the system to be modernized. In the rebuilt system, the business and persistences layer classes will be grouped according to their relationships, and methods of business classes will be made available as services. The steps of the proposed approach are techniques, whose formulas and algorithms can be added/transformed into tools that will semi-automate the process. This modernization methodology allows the created web services to have a smaller number of classes, in addition to less complexity in each service, maintaining the original functionality. This is accomplished both by refactoring in the original code that decreases the amount of dependency between classes, and by separating class clusters into smaller pieces. Satisfactory results were obtained in the case study, such as a reduction of 24% in average dependence between classes, a decrease of 80% and 6.33% in component size and static complexity (CSC), respectively, and a 100% success rate in the tests regression analysis.

Keywords: Software Reengineering. Migration. SOA. Refactoring. Modernizing. Software Architecture.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IDE: Integrated Development Enviroment

JSON: JavaScript Object Notation

OO: Orientação a Objetos

REST: Representational State Transfer

SOA: Service-Oriented Architecture

SOAP: Simple Object Access Protocol

WSDL: Web Services Description Language

XML: eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

Atualmente existe um grande número de empresas que trabalham com sistemas implementados com linguagens de programação antigas, além de arquiteturas muito restritivas. A defasagem das linguagens e arquiteturas é um problema recorrente e pode acontecer com qualquer sistema em seu ciclo de vida, pois sempre surgem novas tecnologias e paradigmas que melhoram a qualidade dos sistemas desenvolvidos, além de diminuir o tempo de desenvolvimento necessário, tornando as tecnologias existentes obsoletas.

Apesar das grandes vantagens na utilização das novas tecnologias, reconstruir os sistemas legados é geralmente um trabalho complexo, demanda tempo, possui um custo alto e muitas vezes chega a ser inviável. Porém, devido a quantidade de informações armazenadas e a confiabilidade que suas funcionalidades possuem após vários anos de utilização e aprimoramento, esses sistemas continuam sendo essenciais para as empresas (CONNALL; BURNS, 1993; ULRICH, 1994), de forma que uma opção mais viável é a atualização do sistema legado existente para que usufrua das novas tecnologias.

A Reengenharia de Software é uma forma de conseguir evoluir esses sistemas legados, mantendo o conhecimento existente neles (Dos Santos Brito et al., 2008). No início dos anos 90, com a popularização das linguagens orientadas a objetos (OO), foram iniciadas várias pesquisas relacionadas à modernização dos sistemas procedurais para OO através da Reengenharia, devido ao ganho na reutilização e manutenibilidade que a orientação a objetos possibilita.

Em pouco tempo, linguagens orientadas a objetos se tornaram as mais utilizadas no desenvolvimento de sistemas (TIOBE.COM, 2017), o que significa que uma grande quantidade de sistemas foram e vêm sendo desenvolvidos nessas linguagens OO. Porém, com o tempo, o aprimoramento gradual desses sistemas OO aumentou sua complexidade na manutenção e testes. Essa complexidade é agravada naqueles que são monolíticos (STOJANOVI, 2005), ou seja, geram apenas um executável ou componente como resultado, dificultando também na escalabilidade e introdução de novas tecnologias em funcionalidades específicas, (DRAGONI et al., 2016), sendo essas novas tecnologias, em alguns casos, incompatíveis com as utilizadas.

A arquitetura orientada a serviço (SOA) resolve os problemas dos sistemas monolíticos, melhorando a escalabilidade, que pode ser feita para cada serviço, além de diminuir a complexidade da manutenção e testes, já que estes podem ser feitos em componentes menores (VILLAMIZAR et al., 2015).

Existem na literatura várias técnicas de Reengenharia para auxiliar na modernização de sistemas OO para SOA, como Erdemir e Buzluca (2014), Yousef, Adwan e Abushariah (2014), Liang Bao et al. (2010), mas muitas delas ignoram o fato de que a maioria dos sistemas OO possuem uma arquitetura em camadas, de modo que a utilização dessas técnicas pode agrupar as classes de forma incorreta, podendo, por exemplo, separar uma classe de negócio de uma classe de persistência ou entidade que ela utiliza, impossibilitando, ou aumentando as refatorações necessárias para sua utilização, conforme observado por Wang et al. (2008).

Motivado pelos problemas existentes nos sistemas legados, em especial aqueles monolíticos, orientados a objetos e que possuem uma arquitetura em camadas, esta pesquisa pretende apresentar uma proposta de modernização, quebrando essa arquitetura monolítica e modernizando-a para SOA. As limitações impostas pelos requisitos se deve ao fato, como citado anteriormente, dos sistemas orientados a objetos englobar grande parte das aplicações existentes e as pesquisas relacionadas a modernização de sistemas OO não serem tão efetivas quando executadas sobre aqueles com arquiteturas em camadas.

1.2 Objetivos

Esta dissertação tem o objetivo de definir um catálogo de técnicas para auxiliar na modernização de sistemas orientados a objetos, monolíticos e com arquitetura em camadas para SOA, independente da linguagem de programação utilizada. Por causa da aplicabilidade dessa abordagem em várias linguagens de programação, não são definidas ferramentas específicas a serem utilizadas, e sim técnicas, fórmulas e algorítimos que podem ser adicionados/transformados em ferramentas de apoio. Esta abordagem é composta de três etapas, diminuição de dependências, clusterização [1] e criação dos serviços, sendo que cada uma delas possui técnicas semi-automatizadas que agilizará o processo. O foco principal dessa abordagem é na identificação dos possíveis serviços, através do agrupamento de classes que se relacionam entre si.

Um dos objetivos da primeira etapa, diminuição de dependências, é a redução do acoplamento entre as classes, diminuindo a quantidade de relacionamentos entre elas e permitindo a criação de serviços menores, ou seja com menos classes. Essa etapa também padroniza a comunicação que deve acontecer somente entre as classes da camada de negócio.

Na segunda etapa, "clusterização", é definida uma técnica que permite a divisão de um serviço em serviços menores, de acordo com a modularidade que essa divisão irá gerar. O objetivo dessa etapa é, também, permitir a criação de serviços menores, que facilitará

[1] Clusterização é a palavra equivalente em português para o termo *clustering*, em inglês, que significa agrupamento, referenciando a agrupamento de classes no contexto desta pesquisa.

futuras manutenções e testes (VILLAMIZAR et al., 2015).

O objetivo da terceira etapa é definir regras que devem ser observadas no momento da criação dos serviços através de *web services*, de maneira a permitir que a funcionalidade continue operando da mesma forma que a original. Nessa etapa não são definidas técnicas para criação de *web services*, pois podem variar de acordo com a linguagem de programação e IDE utilizada, e a idéia da proposta de modernização é que ela possa ser executada em sistemas escritos em qualquer linguagem de programação, desde que sejam observados os requisitos iniciais, que é ser orientado a objetos, monolítico, e com arquitetura em camadas.

Este trabalho não abordará a alteração da camada de apresentação para a utilização dos serviços criados, pois implica em análise tando da parte de *back-end* da camada de apresentação quanto do próprio *front-end* da aplicação, que pode variar de acordo com o framework utilizado. Também não será abordado a separação do banco de dados para cada serviço ou conjunto de serviços, que implica, possivelmente na criação/alteração de campos/tabelas, remoção de restrições, sincronização de dados entre serviços etc, afetando vários scripts SQL (*Structured Query Language*) que possam existir. Além disso, as alterações no banco de dados irá refletir em mudanças nas classes de entidades e scripts HQL (*Hibernate Query Language*) ou similares, já que estas são a representação OO do banco de dados. Essas exclusões foram feitas devido a complexidade e tempo necessário para fazê-los, ficando essas tarefas para um trabalho futuro.

1.3 Organização da Dissertação

Esta dissertação está organizada em seis capítulos mais uma seção relacionada às referências bibliográficas, sendo o primeiro capítulo relacionado à introdução.

No Capítulo 2 são apresentados os principais conceitos sobre Reengenharia de Software e são exibidos vários tipos uma categorização dos trabalhos relacionados a modernização de sistemas, através da reengenharia, para a arquitetura orientada a serviços (SOA).

No Capítulo 3 é apresentada uma visão geral das arquiteturas envolvidas no processo de modernização, desde as arquiteturas de origem (monolítica e em camadas) até a de destino (SOA).

O Capítulo 4 detalha a abordagem proposta para a realização da modernização dos sistemas, descrevendo as etapas, técnicas e ferramentas utilizadas.

O Capítulo 5 apresenta um estudo de caso feito sobre um sistema real, com o objetivo de validar a eficácia da abordagem.

Por último, o Capítulo 6 apresenta as considerações finais, abordando os trabalhos

correlatos, as principais contribuições e os trabalhos futuros.

2 REENGENHARIA DE SOFTWARE

Nos dias atuais é fácil perceber a dependência das organizações em relação aos softwares por ela utilizados, não sendo mais possível a existência de uma organização sem eles (CONNALL; BURNS, 1993; ULRICH, 1994). Tal importância faz com que essas organizações procurem utilizar as mais modernas tecnologias, seja para melhorar o desempenho interno, seja para ganhar alguma vantagem em relação a seus concorrentes. Porém, grande parte dos sistemas críticos utilizados foram desenvolvidos há muitos anos e apenas sua manutenção não é o bastante para mantê-los atualizados.

Lehman e Belady (1985) demonstram que pior que a desatualização, quando não se faz alguma melhoria, é a degradação da qualidade através da manutenção e, consequentemente, a manutenibilidade do software. Visaggio (2001) chama essa degradação de "sintomas de envelhecimento"(*aging symptoms*) e apresenta evidências de que o processo de Reengenharia pode diminui-los.

Neste capítulo são apresentados conceitos de Reengenharia de Software que serão relevantes na abordagem proposta.

2.1 Conceitos

Segundo Seacord, Plakosh e Lewis (2003), reengenharia é uma forma de modernização que melhora as capacidades ou manutenibilidade de um sistema legado através da introdução de tecnologias e práticas modernas. Os objetivos principais da Reengenharia são (SNEED, 1995):

- **Melhorar manutenibilidade:** pode-se, por exemplo, reduzir a manutenção com a Reengenharia de módulos menores com interfaces mais explícitas. É difícil de medir seu ganho porque também pode ocorrer por outros fatores como equipe mais treinada, ou utilização de métodos mais eficientes;
- **Migração/Modernização:** a Reengenharia pode ser utilizada para realizar a migração de um sistema entre ambientes operacionais diferentes. Também pode mudar sistemas desenvolvidos em linguagens ou arquiteturas obsoletas para outras mais modernas e flexíveis;
- **Obter maior confiabilidade:** os testes extensivos necessários para garantir a equivalência das funcionalidades podem evidenciar erros antigos e a reestruturação revela potenciais defeitos. Esse objetivo pode ser medido através da análise de erros;

- **Preparação para melhorias funcionais:** decompor um sistema em módulos menores melhora sua estrutura além de isolá-los uns dos outros. Isso facilita a adição ou alteração de funções sem afetar outros módulos.

2.1.1 Terminologia

Chikofsky e Cross (1990) apresenta a terminologia empregada na engenharia de software relacionadas as tecnologias de análise e entendimento de sistemas legados, sendo os principais termos:

- **Engenharia avante:** é o processo tradicional de desenvolvimento, indo de abstrações e lógicas de alto nível com projetos independentes de implementação para a implementação física de um sistema;
- **Engenharia reversa:** é o processo de análise de um sistema para identificar seus componentes e inter-relacionamentos, e criar representações do sistema em outra forma ou em um nível maior de abstração. Há várias subáreas na engenharia reversa, sendo que as que são mais amplamente referenciadas são redocumentação e recuperação de projeto;
  - **Redocumentação:** é a criação ou revisão de representação semanticamente equivalente a outra, mantendo o mesmo nível de abstração. As representações resultantes são geralmente consideradas visões alternativas (por exemplo, fluxo de dados, estrutura de dados e fluxo de controle) com o objetivo de serem analisadas por pessoas;
  - **Recuperação de projeto:** é um subconjunto da Engenharia Reversa em que o conhecimento do domínio, informações externas e dedução ou raciocínio difuso são adicionados às observações do sistema alvo para identificar abstrações de alto nível significativas além daquelas obtidas diretamente através da examinação direta do sistema;
- **Reestruturação:** é a transformação de uma representação para outra no mesmo nível de abstração, preservando o comportamento externo do sistema (funcionalidades e semântica). Um exemplo é a alteração de um código para melhorar sua estrutura, no sentido tradicional de projeto estruturado; e
- **Reengenharia:** é a análise e alteração de um sistema para reconstituí-lo e implementá-lo em uma nova forma.

A Figura 2.1 mostra o relacionamento entre esses termos, sendo que foi considerado apenas três estágios do ciclo de vida (Requisitos, Projeto e Implementação), com claras

diferenças no nível de abstração. Os Requisitos especificam o problema a ser resolvido, incluindo objetivos, restrições e regras de negócio. O Projeto trata de especificação da solução e a Implementação refere-se a codificação, teste e entrega de um sistema em funcionamento. Nessa figura nota-se que a Engenharia Avante vai do nível de abstração mais alto (Requisitos) para o mais baixo (Implementação), enquanto a Engenharia Reversa faz o caminho inverso. A Reengenharia pode ser feita tanto somente na parte de implementação ou em conjunto com a parte de Projeto.

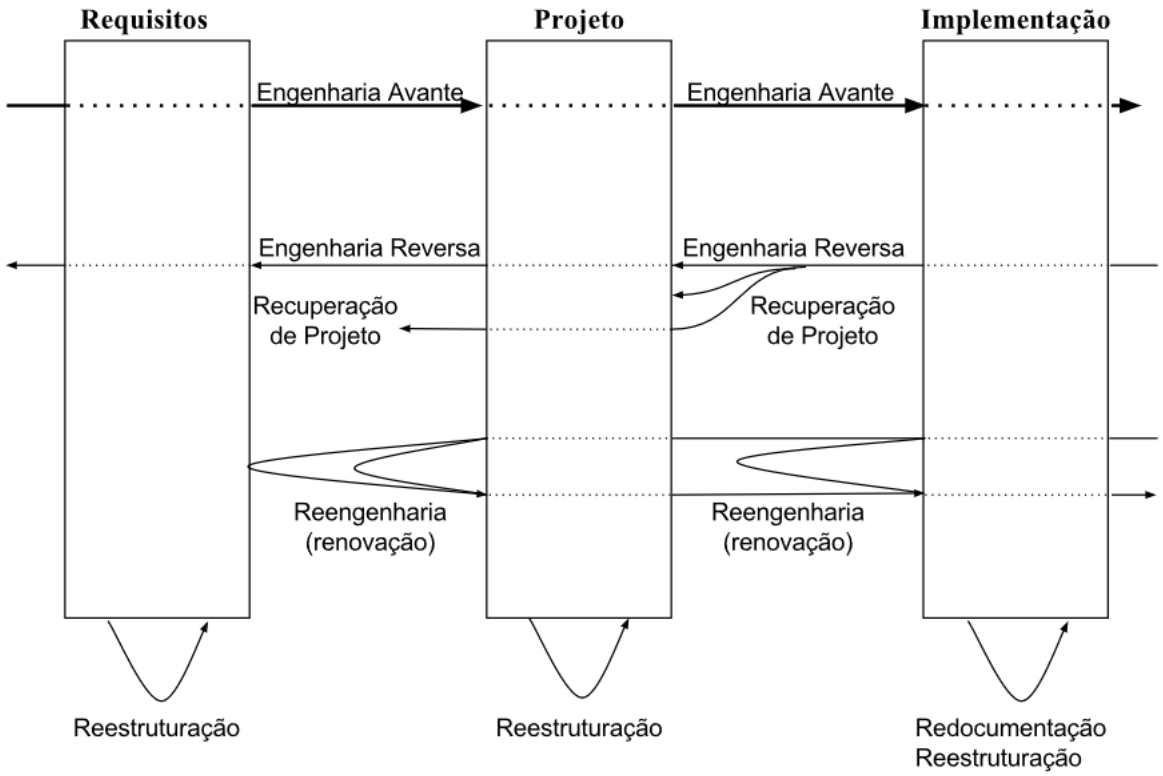


Figura 2.1 – Ciclo de Vida do Software (Tradução minha)(CHIKOFSKY; CROSS, 1990)

2.2 Abordagens de Reengenharia de Sofware para SOA

Na literatura existem várias abordagens de reengenharia que tratam sobre a modernização de sistemas legados para SOA, indo desde a descrição de passos a serem seguidos a técnicas semi-automatizadas. Razavian e Lago (2015) analisou 75 abordagens existentes, verificando suas semelhanças e diferenças, agrupando-as em 8 diferentes "famílias", de acordo com as atividades executadas.

Para fazer esse agrupamento foi utilizado o framework SOA-MF criado por Razavian e Lago (2010), que é um esqueleto de atividades genéricas representando as necessidades a serem executadas em um projeto de modernização. Esse framework é composto de três

processos: Engenharia Reversa, Transformação (Reestruturação) e Engenharia Avante. As famílias definidas são:

- **Família da transformação de código (*code transformation family*):** se limita a transformações no nível de sistema, convertendo o código legado para baseado em serviços. Nesta família a modernização implica em mover o sistema legado como um todo para SOA, sem decompor o sistema existente. Se uma transformação desse tipo for executado em um sistema monolítico, ele continuará monolítico, mesmo que disponibilize serviços;
- **Família da identificação de serviço (*service identification family*):** não abrange o processo de transformação, significando que não ocorre a remodelação do sistema de elementos legados para elementos baseados em serviço. Nesta família a modernização é limitada a identificação de possíveis serviços dentro do sistema legado, através de ténicas de Reengenharia;
- **Família da transformação do modelo de negócio (*business model transformation family*):** os processos de Engenharia Reversa e a Engenharia Avante não são contemplados, sendo a modernização realizada através do processo de transformação, realizado no nível conceitual. Existem duas principais categorias de abordagens de migração nessa família sendo a primeira àquelas que definem meta-processos, cujo objetivo é apoiar na tomada de decisão sobre como fazer a modernização. A segunda categoria executa a Reengenharia sobre os processos de negócio do sistema, para que sirvam de base para o desenvolvimento *top-down* de serviços;
- **Família de transformação de elementos de projeto (*design element transformation family*):** o processo de transformação ocorre somente nos elementos básicos de projeto (por exemplo, módulos ou classes). Caso os processos de Engenharia Reversa e Avante forem abordados se limitarão também somente à esse nível. Nesta família a modernização é limitada a remodular os elementos do sistema legado para elementos baseados em serviço, por exemplo, uma especificação de componente é alterada para especificação de serviço, um módulo é transformado em um serviço, ou um segmento de código da camada de persistência é convertido em um serviço de dados;
- **Família da Engenharia Avante (*forward engineering family*):** abrange completamente o processo de Engenharia Avante, sendo que os processos de transformação e Engenharia Reversa ocorrem somente no nível de elementos básicos de projeto. O foco desta família é o desenvolvimento de sistemas baseados em serviços, tendo como ponto de partida os processos de negócios. A Engenharia Reversa é utilizada apenas para localizar as funcionalidades dos serviços identificados no processo de Engenharia Avante;

- **Família da transformação de projetos e elementos compostos (*design and composite element transformation family*):** os três processos de modernização ocorrem nos níveis de elementos básicos de projeto e elementos de composição de projeto. Engloba a recuperação e refatoração da arquitetura do sistema legado para SOA, além de remodelar os elementos legados para elementos baseados em serviço;

- **Família da transformação de composição baseada em padrões (*pattern-based composition transformation family*):** inclui apenas o processo de transformação no nível de elementos de composição de projeto, implicando que a arquitetura do sistema existente é alterada ou configurada dentro de SOA, geralmente através da utilização de *patterns*;

- **Família da Engenharia Avante com análise de lacunas (*forward engineering with gap analysis family*):** o processo de transformação ocorre nos níveis conceituais, de elmentos de composição de projeto e elementos básicos de projeto. Aqui o processo de Engenharia Avante engloba as atividades de análise e projeto de serviços, enquanto a Engenharia Reversa não é utilizada. O foco principal dessa família é no desenvolvimento *top-down* de serviços, começando com a extração dos modelos de negócio do sistema legado para depois projetar os serviços. O que diferencia esta família da outra que também utiliza desenvolvimento *top-down* (Família da Engenharia Avante) é que nesta são feitas comparações entre os artefatos originais e os gerados, sendo essas comparações feitas em cada nível de abstração (incluindo ós níveis conceituais, de composição e de projeto).

A Tabela 2.1 mostra a divisão das 75 abordagens avaliadas dentro das fámílias definidas, sendo que a coluna **Quantidade** mostra o número de abordagens dentro de cada família e a coluna **Percentual** mostra seu percentual equivalente em relação ao total de abordagens analisadas.

Tabela 2.1 – Visão geral das famílias de modernização para SOA (RAZAVIAN; LAGO, 2010)

Família	Quantidade	Percentual
Família da transformação de código	12	16%
Família da identificação de serviço	12	16%
Família da transformação do modelo de negócio	5	7%
Família de transformação de elementos de projeto	21	28%
Família da Engenharia Avante	8	10%
Família da transformação de projetos e elementos compostos	10	14%
Família da transformação de composição baseada em padrões	3	4%
Família da Engenharia Avante com análise de lacunas	4	5%

2.3 Considerações finais

A categorização das metodologias de modernização de sistemas para SOA, apresentada na seção anterior, permite uma visão geral dos vários tipos de abordagens existentes, mostrando diversos tipos de modernizações possíveis para atingir o mesmo objetivo, embora, em alguns casos, possuam níveis de abstração diferentes.

Com base nesta categorização, é possível incluir a proposta deste trabalho dentro da família de identificação de serviço (*service identification service*), pois este é o seu principal objetivo. Porém, embora o foco da abordagem de modernização proposta foque na identificação dos serviços, existe uma etapa específica para o tratamento dos *web services*, que define as regras que eles devem seguir no momento de suas criações, sem definir, entretanto, técnicas ou ferramentas específicas. O Capítulo 4 mostrará os detalhes da abordagem proposta.

2.4 Resumo do Capítulo

Este capítulo apresentou a terminologia e conceitos da Engenharia de Software que serão utilizados neste trabalho. Também mostrou uma categorização das abordagens existentes, de acordo com as atividades realizadas, o que possibilitou a inclusão da abordagem deste trabalho em uma delas.

O próximo capítulo apresenta as arquiteturas que esta proposta abrange, desde as existentes no sistema de origem, até SOA, que é a arquitetura de destino da abordagem de modernização proposta.

3 ARQUITETURAS DE SOFTWARE

Este capítulo aborda os conceitos sobre arquitetura de software e detalha as arquiteturas referenciadas na abordagem de modernização, sendo elas a arquitetura monolítica na Seção 3.2, baseadas em serviços na Seção 3.3 e a em camadas na Seção 3.4.

3.1 Conceitos Básicos

Bass, Clements e Kazman (2012) define arquitetura de software como sendo um conjunto de estruturas necessários de um sistema, compreendendo elementos de software, relacionamentos entre eles e as propriedades de ambos.

Dentre as estruturas de software que compõem uma arquitetura é importante ressaltar três delas:

- **Estruturas estáticas:** unidades de implementação (módulos) que se concentram na forma como a funcionalidade do sistema é dividida e atribuída a equipes de implementação;
- **Estruturas dinâmicas:** se concentram na forma como os elementos interagem uns com os outros em tempo de execução para executar as funções do sistema (ex: conjunto de componentes e/ou serviços);
- **Estruturas de alocação:** descreve o mapeamento das estruturas de software com os ambientes de organização, desenvolvimento, instalação e execução do sistema (ex: atribuição de módulo/componente a um time de desenvolvimento).

Rotem-Gal-Oz (2012) apresenta uma outra definição descrevendo a arquitetura de software como uma coleção de decisões fundamentais sobre um produto ou solução, projetado para atender os atributos de qualidade do projeto (requisitos da arquitetura). A arquitetura inclui os principais componente e atributos, além de suas interações e comportamentos para atender os atributos de qualidade.

Com base nas definições apresentadas, pode-se inferir que uma arquitetura deve estabelecer os componentes de um software, suas interações e limites, servindo de guia no desenvolvimento na distribuição das estruturas criadas durante o ciclo de vida do sistema. Nas seções a seguir serão apresentadas alguns exemplos de arquiteturas, mostrando os principais componentes e a forma em que interagem entre si.

3.2 Arquitetura monolítica

Villamizar et al. (2015) define que uma aplicação monolítica é um sistema que possui um único arquivo como resultado, que oferece dezenas ou centenas de serviços usando diferentes interfaces como páginas HTML e *web services*.

Conforme observado pela definição, a arquitetura monolítica está ligada à forma em que as estruturas de software serão publicadas, existindo nesse caso apenas uma. Dentro desta estrutura pode haver outros níveis de arquitetura, relacionadas à organização de seus elementos, como a divisão por camadas, ou até mesmo a disponibilização de serviços.

O uso desta arquitetura possui várias vantagens, podendo-se destacar:

- Simplicidade no desenvolvimento, já que muitas das atuais ferramentas e IDEs foram projetadas para o desenvolvimento de aplicações monolíticas;
- Simplicidade de publicação, pois existe apenas um arquivo, ou estrutura de diretórios, e
- Simplicidade de escalonamento, uma vez que basta criar múltiplas cópias da aplicação acessadas por um balanceador de carga.

Porém, apesar da simplicidade que a utilização desta arquitetura traz, também existe as limitações impostas por ela. Dragoni et al. (2016) cita um conjunto dessas limitações, sendo algumas delas apresentadas a seguir:

- Quanto maior o tamanho da aplicação monolítica mais difícil é mantê-la e evoluí-la, devido ao aumento de sua complexidade;
- Limitação na escalabilidade do sistema, uma vez que para escalar deve-se criar novas instâncias de toda a aplicação, mesmo que o tráfego seja intenso em apenas um grupo pequeno de módulos, ou mesmo apenas um;
- Dificuldade de evoluir as tecnologias utilizadas, devido a necessidade de utilização de uma mesma linguagem e frameworks da aplicação original.

Com base nas vantagens e desvantagens dessa arquitetura, o engenheiro de software deve tomar a decisão se vale a pena utilizá-la para criar um novo sistema, assim como decidir o momento em que ela passa a ser prejudicial em um sistema já existente, sendo necessário sua modernização para uma nova arquitetura.

3.3 SOA

Erl (2005) define a arquitetura orientada a serviços (SOA) como um estilo arquitetural que define o uso de serviços de software com baixo acoplamento e interoperacionais para atender os requisitos dos processos de negócio e usuários. Segundo Daigneau (2012), o serviço se refere a qualquer função de software que executa uma operação de negócio.

Apesar de ser possível utilizar tecnologias como CORBA e DCOM para disponibilizar serviços através de componentes, o foco deste trabalho será na utilização de *web services*, pois provêm os meios para integrar sistemas diferentes e expõem funções reutilizáveis através de HTTP (DAIGNEAU, 2012), utilizando-se de padrões abertos e interoperáveis em diferentes plataformas de computação e independentes das tecnologias de execução subjacentes.

De acordo com Daigneau (2012), os *web services* podem utilizar o HTTP (*Hypertext Transfer Protocol*) de duas maneiras, sendo a primeira para a troca de dados, empregando para isso padrões como XML e JSON (serviços SOAP/WSDL). A segunda usando o HTTP como protocolo de aplicação que define semânticas para o comportamento dos serviços (serviços REST).

REST (*REpresentational State Transfer*), inicialmente definido por Fielding (2000), é um estilo de arquitetura de software para sistemas de hipermídia distribuídos, ou sistemas em que texto, imagens, áudios, e outras mídias são armazenadas através da rede e interconectadas através de *hyperlinks* (KALIM, 2013). *REST web services* são baseados em URLs e nos quatro métodos do protocolo HTTP, utilizando *POST* para inserir um novo registro, *PUT* para alterar um registro existente, *DELETE* para excluir um registro e *GET* para pegar as informações de um registro.

SOAP (*Simple Object Access Protocol*) é um protocolo para troca de mensagens, utilizado para traduzir as informações de um *web service* (por exemplo *request* e *response*), sendo as mensagens documentos XML (KHAN; ABBASI, 2015). SOAP também é conhecido como Envelope SOAP, já que seu elemento raiz é o Envelope. Seu formato segue o padrão mostrado na Figura 3.1.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 ↪   xmlns:ser="http://servicos.estoque.knight.com/"/>

    <!-- Aqui pode ter, ou não, um elemento soapenv:Header -->
    <soapenv:Body>

    </soapenv:Body>
</soapenv:Envelope>
```

Figura 3.1 – Exemplo de um documento SOAP (SAUDATE, 2013)

Segundo Saudate (2013) o elemento Envelope é um container para os elmentos *Header*, que contém metadados relativos à requisição (ex: informações de autenticação e

endereço de retorno), e *Body*, que possui o corpo da requisição (ex: nome da operação e seus parâmetros). O *Header* também permite a adição de especificações de recursos adicionais relativos a *web services*, como o WS-Transaction, que permite um controle extra de transações entre serviços, e o WS-Security que adiciona camadas de segurança ao *web service*.

A existência dessas extensões, em especial a WS-Transaction (detalhado na Seção 4.2.3) foi o principal motivo para a escolha do protocolo SOAP ao invés de REST para os *web services* a serem criados pela proposta de modernização de arquitetura. Apesar de existirem formas para controle de transações entre *REST web services*, isso geralmente não é conseguido de forma tão transparente, sendo necessária a adição de uma nova camada ou framework, como por exemplo o Atomikos[1], e a criação de funções específicas para desfazer as operações já concluídas quando ocorrer um erro. Essa adição de operações acrescenta um ponto crítico à modernização, pois o código acrescentado, se feito de forma incorreta, pode prejudicar a confiabilidade dos dados, além de ser necessário um maior conhecimento do sistema a ser modernizada. Apesar de dificuldades adicionais existentes com utilização de REST, esta ainda é uma opção viável, desde que observados os pontos levantados. Porém, para simplificar o processo de modernização e aumentar a confiabilidade do sistema modernizado, todos os *web services* citados neste trabalho estão relacionados a SOAP.

Além de SOA existem outras arquiteturas baseadas em serviço, sendo que a de microserviços merece mais destaque. A diferenciação entre a arquitetura de microserviços e SOA é um ponto de discussão delicado entre os pesquisadores, pois ambas utilizam serviços como principal componente de implementação e execução de funcionalidades (RICHARDS, 2015). Além disso, existem autores que inclusive colocam microserviço como uma abordagem SOA (NEWMAN, 2015).

Embora possa se discutir se microserviços é ou não é SOA, existem algumas características que uma arquitetura deve possuir para receber tal classificação. Newman (2015) define microserviços como serviços pequenos e autônomos que trabalham em conjunto. Com base nessa definição, é possível inferir que a proposta da utilização de serviços neste trabalho não atende estes requisitos, isso porque nem o tamanho ou sua independência estão entre as restrições impostas pela abordagem de modernização proposta, podendo gerar serviços de vários tamanhos, tanto independentes quanto relacionados entre si. A criação destes serviços é apresentada com mais detalhes na Seção 4.2.3.

3.4 Arquitetura em Camadas

A divisão do sistema em camadas é uma das arquiteturas mais comuns usadas por desenvolvedores para separar as complexidades de um sistema, de acordo com Fowler

[1] Disponível em <https://www.atomikos.com/>

(2002). Nessa arquitetura os elementos do sistema são organizados em camadas horizontais, sendo que cada camada executa papéis específicos da aplicação. A troca de informações é feita somente entre as camadas diretamente conectadas, sendo que a responsável por fazer as requisições será sempre a camada superior. Essa arquitetura pode ser visualizado na Figura 3.2.

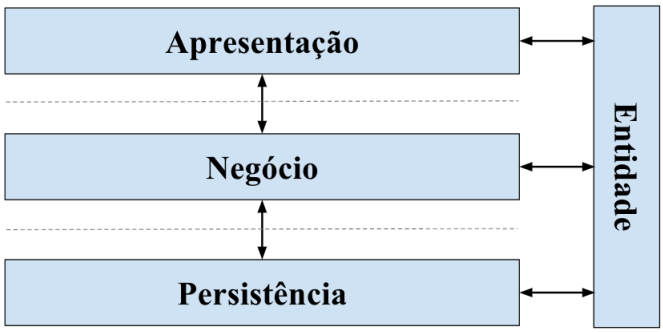


Figura 3.2 – Arquitetura em Camadas

Embora a arquitetura não defina a quantidade de camadas que deve existir, as principais utilizadas são três, sendo elas: apresentação, negócio e persistência, podendo cada uma delas ser definidas como (FOWLER, 2002):

- **Apresentação:** gerencia todas as interfaces com o usuário e lógica de comunicação;
- **Negócio:** executa regras de negócio específicas para cada requisição;
- **Persistência:** faz a comunicação com o banco de dados e gerencia transações.

Apesar de ser a camada de apresentação que mostra os dados para o usuário, ela não tem acesso direto a eles, mas sim a persistência. Como apenas a camada superior pode fazer requisições a camada abaixo dela, uma requisição da apresentação deve passar pela camada de negócio, que processará e retransmitirá para a camada subjacente que é a persistência. A resposta para essa requisição fará o caminho inverso.

Em sistemas orientados a objetos, geralmente é usada uma outra estrutura para representar os dados do banco de dados (Entidade da Figura 3.2), sendo que ela trafega entre as camadas. Pelo fato dessa arquitetura estar relacionada a separação das estruturas estáticas, ela pode estar presente tanto em um sistema monolítico quanto dentro de um serviço SOA.

3.5 Considerações Finais

O motivo da escolha da arquitetura em camadas como requisito para a modernização se deve ao fato de que embora esta seja a arquitetura mais utilizada para a maioria das

aplicações *Java Enterprise Edition* (BASS; CLEMENTS; KAZMAN, 2012), muitas das pesquisas relacionadas à modernização de sistemas legados para componentes ou serviços não levam em consideração a existência dessas camadas.

Essa declaração pode ser observada na pesquisa de Wang et al. (2008), que evidencia a desconsideração da existência de camadas. Apesar disso não estar explícito em outras pesquisas (ADJOYAN; SERIAI; SHATNAWI, 2014; CONSTANTINOU et al., 2015; BUDHKAR; GOPAL, 2012; YOUSEF; ADWAN; ABUSHARIAH, 2014; Liang Bao et al., 2010; Eunjoo Lee et al., 2003; WANG et al., 2008), o mesmo fato pode ser constatado. Pois estas outras pesquisas observam apenas os relacionamentos entre as classes, podendo por exemplo gerar componentes/serviços onde pode existir uma classe de negócio e uma de persistência, mas sem as entidades que trafegarão as informações, ou uma classe entidade e uma de negócio, sem uma persistência para salvar as informações.

As arquiteturas baseadas em serviços resolvem os problemas listados na Seção 3.2. Porém, mesmo utilizando essas novas arquiteturas para os novos desenvolvimentos, ainda restam os sistemas legados, escritos monoliticamente, que precisam ter sua arquitetura migrada de forma a aproveitar de seus benefícios, sendo o auxílio a essa modernização o foco deste trabalho.

```
class ClasseN1{
    public void metodo1(){
        ClasseN2 classeN2 = new ClasseN2();
        classeN2.metodo2();

        ClasseP1 classeP1 = new ClasseP1();
        classeP1.alterar();
    }

    public void metodo3(){
        ClasseP1 classeP1 = new ClasseP1();
        classeP1.alterar();
    }
}
```

```
class ClasseN2{
    public void metodo2(){
        ClasseP2 classeP2 = new ClasseP2();
        classeP2.alterar();
    }

    public void metodo4(){
        ClasseN1 classeN1 = new ClasseN1();
        classeN1.metodo3();

        ClasseP2 classeP2 = new ClasseP2();
        classeP2.alterar();
    }
}
```

Figura 3.3 – Exemplo de arquiteturas em camadas

Apesar da proposta de modernização ser transformar um sistema monolítico em SOA, a estrutura em camadas será mantida dentro dos serviços criados, diminuindo assim a quantidade de refatorações necessárias. As Figuras 3.3 e 3.4 exemplificam, através de trechos de pseudo-código, a arquitetura em camadas e SOA, respectivamente, sendo que os serviços do exemplo SOA também possuem uma arquitetura em camadas dentro delas. A diferenciação do código entre as arquiteturas e suas figuras se faz através do destaque nas linhas de código que sofrem alteração. Nessas figuras as classes ClasseN1 e ClasseN2 pertencem à camada de negócio, e as classes ClasseP1 e ClasseP2 à camada de persistência, sendo que as figuras detalham somente a camada de negócio.

```
@WebService
class ClasseN1{
    public void metodo1(){
        //ServicoClasseN2 referencia ao web service da
        ↪  classe ClasseN2
        ServicoClasseN2 servicoClasseN2 = new
        ↪  ServicoClasseN2();
        servicoClasseN2.metodo2();

        ClasseP1 classeP1 = new ClasseP1();
        classeP1.alterar();
    }

    public void metodo3(){
        ClasseP1 classeP1 = new ClasseP1();
        classeP1.alterar();
    }
}
```

```
@WebService
class ClasseN2{
    public void metodo2(){
        ClasseP2 classeP2 = new ClasseP2();
        classeP2.alterar();
    }

    public void metodo4(){
        //ServicoClasseN1 referencia ao web service da
        ↪  classe ClasseN1
        ServicoClasseN1 servicoClasseN1 = new
        ↪  ServicoClasseN1();
        servicoClasseN1.metodo3();

        ClasseP2 classeP2 = new ClasseP2();
        classeP2.alterar();
    }
}
```

Figura 3.4 – Exemplo de SOA

3.6 Resumo do Capítulo

Este capítulo descreveu as arquiteturas referenciadas pela proposta de modernização, apresentando suas características e detalhando os motivos que levaram a escolhê-las como arquiteturas de origem e destino desta proposta.

No próximo capítulo é apresentado os detalhes propostos de modernização, detalhando os passos que permitirão que um sistema monolítico, orientado a objetos e desenvolvido com uma arquitetura de três camadas, seja transformado em um sistema com arquitetura SOA.

4 MODERNIZAÇÃO DE SISTEMAS MONOLÍTICOS PARA ARQUITETURA ORIENTADA A SERVIÇOS

Nos capítulos anteriores foram apresentados os principais conceitos envolvidos nesta pesquisa, que proporcionaram o fundamento teórico para a criação de uma abordagem que auxilie a modernização, para SOA, de sistemas monolíticos, orientados a objetos e desenvolvidos com uma arquitetura de três camadas.

A abordagem é composta de três etapas sendo a primeira a diminuição de dependência, que reduzirá a quantidade de relacionamentos entre as classes, a segunda é a "clusterização", responsável pelo agrupamento dessas classes e a terceira e última a criação dos serviços, que definirá critérios a serem observados na criação dos *web services*. Dentro delas serão definidas técnicas e fórmulas que poderão ser utilizaras por ferramentas para automatizar suas realizações. Antes de iniciar com a descrição de cada um das etapas será apresentado os fundamentos que serão utilizados.

4.1 Mecanismos da Abordagem proposta

A abordagem proposta utiliza técnicas já existentes na literatura, adequando-as conforme a necessidade para melhor atender os requisitos desejados.

4.1.1 Força de Conectividade

Para auxiliar no agrupamento das classes há a necessidade de identificar o grau de ligação existente entre elas, de forma a unir aquelas com maior ligação. Para atingir esse objetivo encontram-se na literatura várias pesquisas, podendo citar Adjoyan, Seriai e Shatnawi (2014), Constantinou et al. (2015), Budhkar e Gopal (2012), Yousef, Adwan e Abushariah (2014), Liang Bao et al. (2010), Eunjoo Lee et al. (2003), Wang et al. (2008).

Escolheu-se a fórmula definida por Eunjoo Lee et al. (2003) e aprimorada por Wang et al. (2008), a qual foi atribuída o nome **força de conectividade** (*connectivity strength*), pois ela leva em consideração apenas as ligações existentes no código fonte, que, em grande parte dos sistemas legados, é o único documento existente, ou pelo menos o único confiável (ERDEMIR; TEKIN; BUZLUCA, 2011; ERDEMIR; BUZLUCA, 2014).

Essa técnica baseia-se na quantidade e tipos de parâmetros dos métodos de uma classe que são utilizados por outra, sendo que quanto maior a quantidade e/ou a complexi-

dade dos parâmetros, maior será a ligação entre as classes. Sua fórmula é descrita através da equação:

$$FC(N_1, P_1) = \sum_{M_n \in MSET(N_1)} \sum_{M_p \in MSET(P_1)} FC(M_n, M_p) \tag{4.1}$$

$$FC(M_n, M_p) = \begin{cases} \begin{pmatrix} Pricount(M_p) * w_{pri} + \sum_{i=0}^{Abscount(M_p)} COX(P_i) & \text{, se } M_n \text{ chama } M_p \\ 0 & \text{, caso contrário} \end{pmatrix} \end{cases} \tag{4.2}$$

$$COX(P_i) = \begin{cases} \sum_{a \in TYPE(P_i)} w_{pri} & \text{, se a é primitivo} \\ w_{abs} & \text{, se a é membro de } P_i \\ COX(a) * w_{abs} & \text{, caso contrário} \end{cases} \tag{4.3}$$

Em que:

- N_1, P_1 = Classe de negócio e persistência respectivamente;
- MSET(N_1) = Conjunto de métodos da classe N_1;
- M_n, M_p = método específico da classe;
- Pricount(M_p) = quantidade de parâmetros do tipo primitivo no método Mp;
- Abscount(M_p) = conjunto de parâmetros que são classes criadas pelo desenvolvedor;
- w_{pri}, w_{abs} = pesos para parâmetros primitivo e criados pelo desenvolvedor (w_{pri} + w_{abs}. = 1, geralmente $w_{abs} > w_{pri}$, pois tipos criados costumam ser mais complexos que tipos primitivos). Esses pesos devem ser definidos pelo Engenheiro de Sofware responsável pela modernização do sistema legado (Eunjoo Lee et al., 2003);
- P_i= classe passada como parâmetro para o método Mp e que foi criada pelo desenvolvedor e
- COX(P_i) = valor da complexidade da classe criada pelo desenvolvedor.

Para exemplificar, será feito o cálculo da força de conectividade entre as classes N1 e P1 do trecho do código mostrado na Figura 4.1. O primeiro passo é o cálculo da complexidade (COX) dos parâmetros dos métodos de P1 e que são executados por N1, sendo esses parâmetros uma entidade criada no projeto (E1), uma classe externa ao projeto, pertencente a uma biblioteca (Ex), e dois parâmetros de tipos primitivos, um inteiro (i1) e *float* (flt1).

Como E1 possui dois atributos primitivos, o valor de sua complexidade é COX(E1) = 2 * w_{pri} ⇒ 2 * 0,3 ⇒ **0,6**. Não é calculada a complexidade para o tipo EX, pois sua

implementação, assim como sua complexidade, não é de responsabilidade do sistema, sendo considerado para esse caso, apenas o peso w_{abs} para representar essa complexidade. Para os parâmetros primitivos, i1 e flt1, terão como complexidade o peso w_{pri}.

```
public class N1{
    public void metodoN1(){
        P1 p1 = new P1();
        p1.metodoP1(e1, 0.85);
        p1.metodoP3(ex, 7);

        ...
    }
}
```

```
public class P1{
    public void metodoP1(E1 e1, float flt1){
        ...
    }

    public void metodoP3(Ex ex, int i1){
        ...
    }
}
```

```
public class E1{
    private int attrInt;
    private String attrStr;

    ...

}
```

Figura 4.1 – Exemplo de código para cálculo da força de conectividade

Com essas informações é possível calcular a força de conectividade entre N1 e P1, somando as complexidades dos parâmetros de cada método de P1 executado por N1, sendo o cálculo apresentado a seguir:

$$FC(N1, P1) = (COX(E1)+w_{pri})+(w_{abs}+w_{pri}) \Rightarrow (0,6+0,3)+(0,7+0,3) \Rightarrow 0,9+1 \Rightarrow \mathbf{1,9}$$

Na abordagem proposta, a força de conectividade será utilizada como insumo para a refatoração da camada de negócio, em relação à execução de classes da camada de persistência, explicada da Seção 4.2.1.

4.1.2 Algoritmo Fast Community

Apesar de o agrupamento de classes através de seus relacionamentos não ser uma tarefa muito complexa, às vezes, é necessário fazer a divisão do componente gerado, de forma a diminuir seu tamanho. Para realizar essa divisão, utilizou-se o algoritmo de "clusterização"chamado *fast community*, definido por Newman (2003). O objetivo principal desse algoritmo, executado sobre grafos, é encontrar as *communities*, sendo que *community structure* é uma organização especial, em que vértices formam grupos com alta densidade nas arestas internas e baixa densidade nas arestas externas (ERDEMIR; TEKIN; BUZLUCA, 2011).

Segundo Erdemir e Buzluca (2014), este algoritmo apresentou-se superior a outros da literatura, quando observados sob os critérios de *authoritativeness*, *stability* e *extremity of cluster distribution*, significando (ERDEMIR; TEKIN; BUZLUCA, 2011; ERDEMIR; BUZLUCA, 2014):

- ***Authoritativenes*:** similaridade entre a decomposição feita por profissionais e a automatizada por um algoritmo de "clusterização";
- ***Stability*:** o resultado da extração automática não deve produzir efeitos drasticamente diferentes quando executados sobre versões similares de um software com pequenas alterações e

- ***Extremity of cluster distribution*:** diz respeito à equivalência do tamanho dos componentes, pois, não se deve produzir componentes muito grandes ou muito pequenos por não serem comuns na arquitetura dos mesmos, além de poder ocasionar a diminuição da coesão e o aumento do acoplamento.

Apesar desse algoritmo não ter sido inicialmente idealizado para a "clusterização"de classes em sistemas orientados a objetos, Erdemir, Tekin e Buzluca (2011), Erdemir e Buzluca (2014) demonstram essa possibilidade, criando um grafo a partir do código fonte do sistema, sendo que cada vértice representa uma classe, as arestas seus relacionamentos e a seta pertencente as arestas indica que a classe de origem executa métodos da classe de destino. Um exemplo desse grafo pode ser visualizado através da Figura 4.2, sendo que o código que o originou é composto por quatro classes, N1, N2, N3 e N4, a classe N2 executa métodos das classes N1, N3 e N4, e a classe N3 executa métodos de N4.

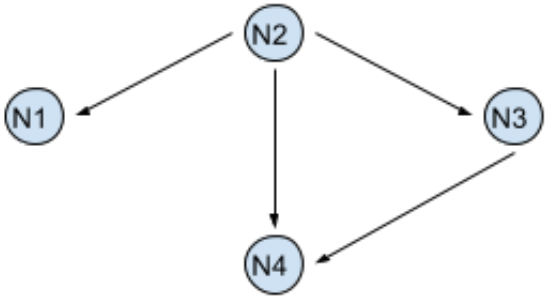


Figura 4.2 – Exemplo de grafo

O algoritmo *fast community* é descrito no Algoritmo 1 (NEWMAN, 2003; ERDEMIR; TEKIN; BUZLUCA, 2011; SHIOKAWA; FUJIWARA; ONIZUKA, 2013; ERDEMIR; BUZLUCA, 2014).

```
1 Considere cada vértice como um cluster diferente;
2 while existir mais de um cluster do
3 |   Junte os dois clusters que tiveram o maior crescimento, ou menor redução da
  |    modularidade;
  end
4 Selecione o ponto de corte do dendrograma resultante, analisando o maior valor de
    da modularidade;
```

Algoritmo 1: Algoritmo *Fast Community*.

Conforme pode ser observado no algoritmo, a "clusterização"é feita em iterações e, em cada uma delas, dois clusters são agrupados até que reste apenas um único cluster. Essa sequencia de agrupamentos gera um dendrograma, que mostra a ordem em que os agrupamentos foram feitos.

A Figura 4.3 mostra o dendrograma relacionado ao grafo da Figura 4.2, sendo que a régua na parte superior indica o número de iterações que o algoritmo teve. Nela

é possível verificar que o primeiro agrupamento foi o das classes N2 e N1, pois, com ele, conseguiu-se o maior ganho de modularidade (ΔW=0,6250) em relação as outras possibilidades de agrupamento (N2-N3, N2-N4 e N3-N4). Na segunda iteração o maior ganho da modularidade foi com a junção de N2 com N3 (ΔW=0,500), e para a última iteração restou a junção dos dois únicos componentes existentes, N1-N2 e N2-N3, transformando tudo em um único cluster e encerrando o algoritmo.

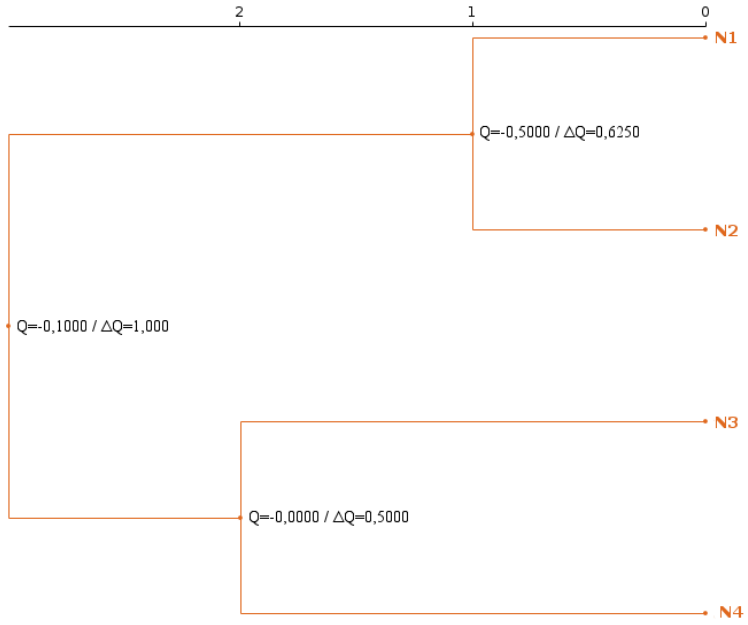


Figura 4.3 – Clusterização

A escolha do ponto de corte será baseada na modularização que cada iteração gera, de acordo com a modularidade que o sistema possui em cada iteração. Neste exemplo, optando-se pelo ponto de corte nas junções N1-N2 ou N3-N4 o mesmo resultado será gerado, ou seja, produzirá dois componentes, um contendo as classes N1 e N2 e outro contendo as classes N3 e N4. É importante ressaltar que pode haver casos em que o melhor é não selecionar nenhum ponto de corte, mantendo todas as classes em um único componente. O cálculo da modularidade é feito através da seguinte fórmula (ERDEMIR; TEKIN; BUZLUCA, 2011; SHIOKAWA; FUJIWARA; ONIZUKA, 2013; ERDEMIR; BUZLUCA, 2014):

- $Qw = \sum_i (C(i))$
- $C(i) = e_{ii} - ai^2$

Em que:

- Qw = modularidade;

- e_{ii} = quantidade de arestas internas dividida pelo total de arestas do grafo (fração de arestas internas ao cluster) e
- a_i^2 = quantidade de arestas externas dividida pelo total de arestas do grafo (fração de arestas externas que se ligam ao cluster).

A escolha do ponto de corte é baseada no valor da modularidade do sistema em cada iteração (variável "Q"da Figura 4.2), sendo os critérios utilizados para tal seleção serão detalhados na Seção 4.2.2.

4.2 Proposta de Metodologia de Modernização

A abordagem demonstrada nesta pesquisa foi definida a partir da utilização de mecanismos de identificação de componentes/serviços. Pois a pesquisa objetiva auxiliar na modernização, para SOA, de sistemas orientados a objetos, monolíticos, e que também possuam uma arquitetura de três camadas. A proposta é composta por três etapas, conforme pode ser observada pela Figura 4.4, que foi criada seguindo a notação BPMN (Object Management Group (OMG), 2011).

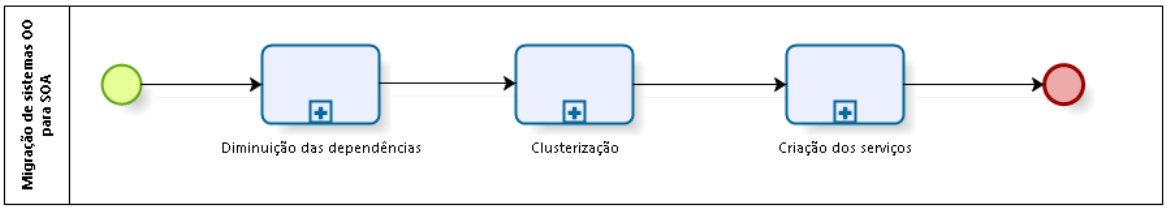


Figura 4.4 – Macro Fluxo da abordagem de modernização de sistemas OO para SOA

Como boa parte dos sistemas legados não possuem documentação ou estas estão desatualizadas ou incompletas (LEWIS; MORRIS; SMITH, 2005; KHADKA et al., 2013; ERDEMIR; BUZLUCA, 2014), todas as técnicas apresentadas usam como entrada apenas o próprio código fonte do sistema, analisando-o e/ou refatorando-o para atingir o objetivo final.

Na primeira etapa é feita uma análise do código, verificando as ligações entre as classes e calculando a força de conectividade (FC) entre as classes de negócio e as classes de persistência utilizadas. A FC é utilizada como base nas refatorações para diminuir a quantidade de relacionamentos existentes. Na segunda etapa, a partir do código refatorado, é executado o agrupamento das classes de acordo com seus relacionamentos. Para os casos necessários, o engenheiro de software executa o algoritmo de "clusterização"*fast community* (Seção 4.1.2), responsável pela divisão em grupos menores. A etapa final culmina na criação de *web services* a partir das classes de negócio. Nesta última etapa não são definidas técnicas específicas, mas critérios para auxiliar na tomada de decisão do engenheiro na

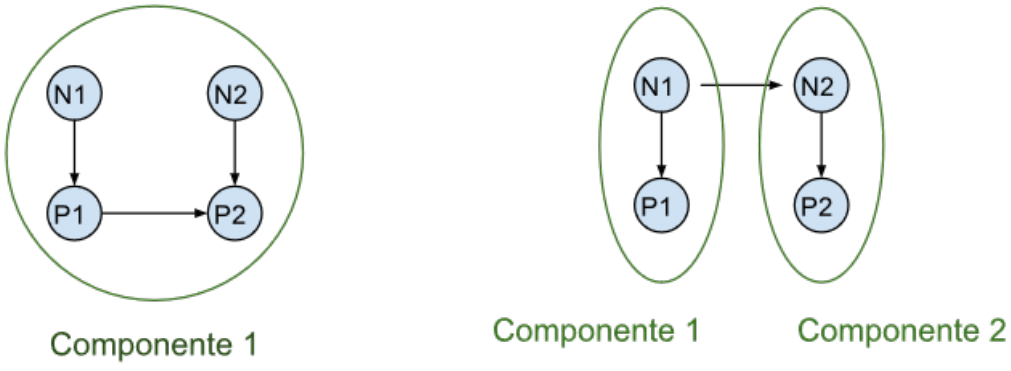
criação dos serviços. Devido a variedade de formas possíveis que os desenvolvedores podem utilizar para determinar se uma classe é de negócio ou persistência, como inclusão em pacotes específicos ou estender determinada classe, a identificação da camada das classes ficará a cargo do Engenheiro de Software encarregado pela modernização do sistema.

Nas seções a seguir serão detalhadas as etapas da abordagem, suas técnicas e ferramentas utilizadas, além de demonstrá-las através de sua execução sobre o sistema SIGA-EPCT (Sistema Integrado de Gestão Acadêmica da Educação Profissional e Tecnológica). Esse sistema atende os requisitos iniciais e é utilizado por alguns Institutos Federais de Educação, Ciência e Tecnologia do país.

4.2.1 Diminuição das dependências

Com o intuito de melhorar a coesão e o acoplamento dos *web services* resultantes, o primeiro passo é melhorar esses atributos nas classes existentes. Para guiar esta etapa, definiu-se que será mantida, para cada componente gerado (conjunto de classes), a arquitetura em camadas existente no sistema original. A camada de negócio, que é a camada de mais alto nível dentro de cada componente, irá disponibilizar seus métodos como serviços.

Partindo-se desta premissa, foram feitas análises de cada camada, iniciando pela persistência e, assim, identificou-se a existência de relacionamentos entre suas classes. Isto é um indício de que elas estão tratando de regras de negócio, sendo necessárias refatorações, transportando parte do código para a camada superior. Esta alteração possibilita a criação futura de componentes menores, conforme demonstrado através da Figura 4.5, que contém a representação do código original na Figura 4.5a e do código refatorado na Figura 4.5b.



(a) Componente Original

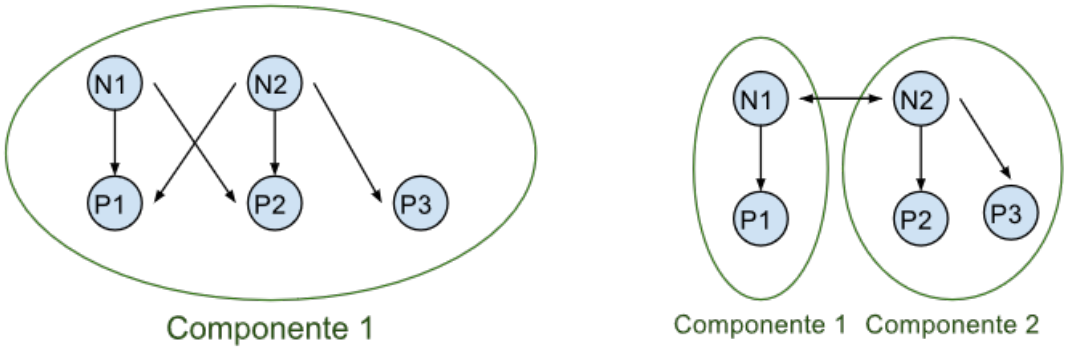
(b) Componente Refatorado

Figura 4.5 – Refatoração das classes de persistência

Após essas refatorações, passou-se para a análise da camada de negócio e seus relacionamentos com a persistência. Percebeu-se que no negócio existiam diferentes classes relacionadas a uma mesma classe de persistência. Embora essa utilização não quebre o

conceito de camadas (FOWLER, 2002), essa prática dificulta a criação das classes em componentes menores, conforme ilustrado pela Figura 4.6a.

Para tentar diminuir o tamanho do componente a ser gerado, definiu-se mais uma refatoração, na qual cada classe de persistência irá se relacionar com apenas uma classe de negócio, conforme ilustrado na Figura 4.6b.



(a) Componente Original

(b) Componente Refatorado

Figura 4.6 – Refatoração das classes de negócio

Porém, para realizar a operação proposta primeiro deve-se identificar qual classe de negócio será responsável por qual classe de persistência. Isto é feito calculando a FC entre cada classe das camadas de negócio e persistência, sendo que a fórmula para o cálculo da FC foi explicada na Seção 4.1.1. Com base na FC, cada classe de persistência deverá se relacionar com a classe de negócio que possuir a maior conectividade.

Para exemplificar essa refatoração, supõem-se um sistema com duas classes de negócio (N1 e N2), duas de persistência (P1 e P2) e três de entidade (E1, E2 e E3), conforme mostrado na Figura 4.7.

Inicialmente é necessário calcular a complexidade (COX) das classes usadas como parâmetros entre N1 e P1, ou seja, as entidades E1 e E2. Como E1 tem dois atributos primitivos então sua complexidade é COX(E1) = 2 * w_{pri} ⇒ 2 * 0,3 ⇒ **0,6**. Para o cálculo referente a classe E2, temos COX(E2) = 1 * w_{pri} + COX(E3) * w_{abs} ⇒ 1 * 0,3 + (1 * 0,3) * 0,7 ⇒ **0,51**. Para esse exemplo usou-se os valores 0,3 e 0,7 para os pesos w_{pri} e w_{abs} respectivamente, conforme definidos por Eunjoo Lee et al. (2003).

Com o valor das complexidades obtidas anteriormente é possível calcular a força de conectividade entre N1→P1, N1→P2, N2→P1, N2→P2, tendo:

- $FC(N1, P1) = 0 * w_{pri} + ((COX(E1) + COX(E2)) + (COX(E2) + 1 * w_{pri})) \Rightarrow 0 * 0,7 + (0,6 + 0,51) + (0,51 + 1 * 0,3) \Rightarrow 1,11 + 0,81 \Rightarrow \mathbf{1,92}$
- $FC(N1, P2) = 1 * w_{pri} \Rightarrow 1 * 0,3 \Rightarrow \mathbf{0,3}$

```java
public class N1{
    public void metodoN1(E1 e1, E2 e2){
        P1 p1 = new P1();
        p1.metodoP1(e1, e2);
        p1.metodoP3(e2, 7);
        ...
        P2 p2 = new P2();
        p2.metodoP2(100000);
        ...
    }
}
```

```java
public class N2{
    public void metodoN2(E1 e1, E2 e2){
        P1 p1 = new P1();
        p1.metodoP3(e2, 5);
        ...
        P2 p2 = new P2();
        p2.metodoP4(e1, e2);
        ...
    }
}
```

```java
public class P1{
    public void metodoP1(E1 e1, E2 e2){
        ...
    }

    public    void metodoP3(E2 e2, int i1){
        ...
    }
}
```

```java
public class P2{
    public void metodoP2(long l1){
        ...
    }

    public void metodoP4(E1 e1, E2 e2){
        ...
    }
}
```

```java
public class E1{
    private int attrInt;
    private String attrStr;
}
```

```java
public class E2{
    private long attrLng;
    private E3 attrE3;
}
```

```java
public class E3{
    private boolean attrBool;
}
```

Figura 4.7 – Exemplo de código

- $FC(N2, P1) = 1 * w_{pri} + COX(E2) \Rightarrow 1 * 0,3 + 0,51 \Rightarrow \mathbf{0,81}$
- $FC(N2, P2) = 0 * w_{pri} + (COX(E1) + COX(E2)) \Rightarrow 0 * 0,7 + (0,6 + 0,51) \Rightarrow \mathbf{1,11}$

Comparando as forças de conectividade, tem-se que N1 possui uma conectividade maior com P1 (FC= 1,92) e N2 com P2 (FC= 1,11). Com essa informação, segue-se para a refatoração das classes N1 e N2, para que elas utilizem apenas suas respectivas persistências, e caso necessário, passem a relacionar entre si.

Nessa refatoração, inclui-se os métodos que fazem chamadas a persistência, caso eles não existam. Então deve-se substituir as chamadas originais para que utilizem os métodos recém inseridos. O resultado pode ser observado na Figura 4.8.

Após essa alteração, verificou-se que alguns métodos tinham em seu corpo apenas uma chamada à classe de persistência pertencente a outra classe de negócio, não fazendo sentido sua permanência e optando-se por excluí-los. Como consequência, algumas classes de negócio ficaram sem nenhum método em seu corpo, sendo possíveis suas retiradas do projeto, após devida verificação de sua inutilização pela camada de apresentação. A exemplificação desses procedimentos pode ser visualizada na Figura 4.9.

O processo completo desta etapa pode ser visto na Figura 4.10. Na etapa seguinte, o novo código é analisado e, a partir de seus relacionamentos, é feita a "clusterização"das classes, formando grupo de classes coesos e com baixo acoplamento.

Código Original

```
public class N1{
    public void metodoN1(E1 e1, E2 e2){
        P1 p1 = new P1();
        p1.metodoP1(e1, e2);
        p1.metodoP3(e2, 7);
        ...
        P2 p2 = new P2();
        p2.metodoP2(100000);
        ...
    }

    ...

}
```

```
public class N2{
    public void metodoN2(E1 e1, E2 e2){
        P1 p1 = new P1();
        p1.metodoP3(e2, 5);
        ...
        P2 p2 = new P2();
        p2.metodoP4(e1, e2);
        ...
    }

    ...

}
```

Código Refatorado

```
public class N1{
    public void metodoN1(E1 e1, E2 e2){
        P1 p1 = new P1();
        p1.metodoP1(e1, e2);
        p1.metodoP3(e2, 7);
        ...
        N2 n2 = new N2();
        n2.metodoP2(100000);
        ...
    }

    public void metodoP3(E2 e2, int i1){
        P1 p1 = new P1();
        p1.metodoP3(e2, i1);
    }

    ...
}
```

```
public class N2{
    public void metodoN2(E1 e1, E2 e2){
        N1 n1 = new N1();
        n1.metodoP3(e2, 5);
        ...
        P2 p2 = new P2();
        p2.metodoP4(e1, e2);
        ...
    }

    public void metodoP2(long l1){
        P2 p2 = new P2();
        p2.metodoP2(l1);
    }

    ...
}
```

Figura 4.8 – Refatoração: Inclusão de Métodos

Código Original

```
public class N3{
    public void metodoP4(E2 e2,
    ↪   int i1){
        P4 p4 = new P4();
        p4.metodoP6(l1);
    }

    ...

}

public class N4{

    ...

}
```

Migração da Persistência

```
public class N3{
    public void metodoP4(E2 e2, int
    ↪   i1){
        N4 n4 = new N4();
        n4.metodoP6(l1);
    }

    ...

}

public class N4{

    ...

    public void metodoP6(long l1){
        P4 p4 = new P4();
        p4.metodoP6(l1);
    }
}
```

Exclusão de Método

```
public class N3{

    ...

}

public class N4{

    ...

    public void metodoP6(long l1){
        P4 p4 = new P4();
        p4.metodoP6(l1);
    }
}
```

Figura 4.9 – Refatoração: Alteração das chamadas aos métodos

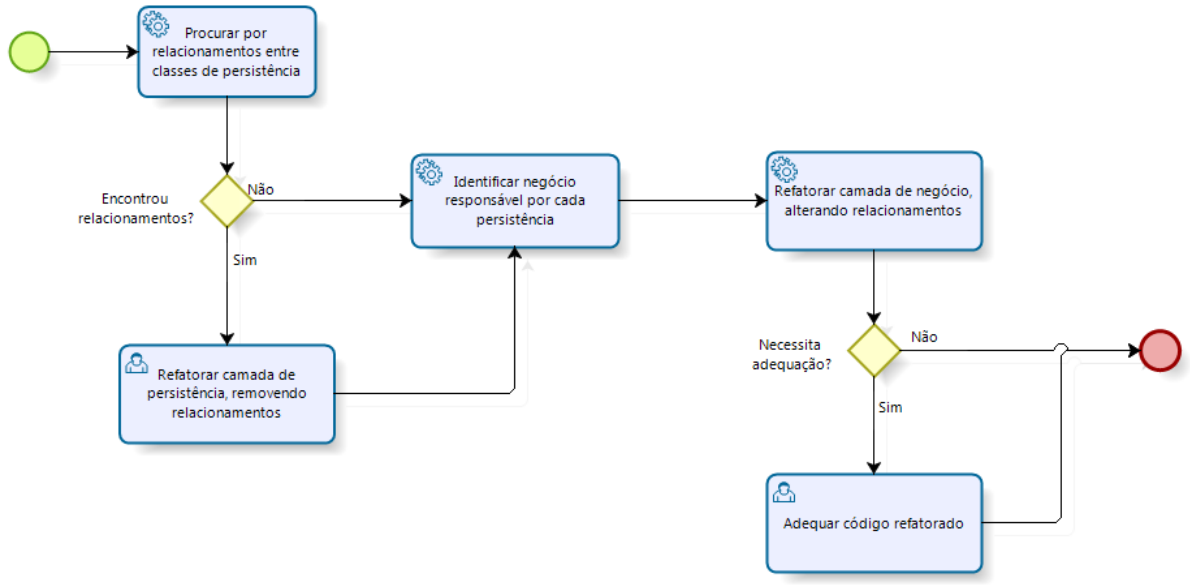


Figura 4.10 – Fluxo do processo de diminuição das dependências das classes

4.2.2 Clusterização

Após as refatorações para diminuição das dependências, deve-se reanalisar o código fonte, de forma a agrupar as classes para que, posteriormente, venham a formar componentes que irão disponibilizar serviços. Como cada classe de negócio já engloba um conjunto específico de persistências, torna-se possível simplificar essa análise, focando apenas na camada de negócio.

O objetivo dessa etapa é unir as classes que possuem relacionamento, criando grupos independentes. Porém, pode ser necessário que alguns desses componentes precisem ser divididos, diminuindo, assim, seus tamanhos. Como uma tentativa dessa redução, é executado o algoritmo apresentado na Seção 4.1.2.

Para exemplificar a execução desse algoritmo, primeiramente deve-se criar um grafo a partir do código do sistema. A Figura 4.11 mostra o grafo contendo as classes de negócio do código da Figura 4.7, sendo incluído outras arestas nesse grafo para melhor ilustrar a execução do algoritmo de "clusterização".

Para esse grafo, inicialmente é efetuado o cálculo de sua modularidade, considerando cada aresta como um cluster diferente, obtendo os valores:

- $C(N1) = \frac{0}{4} - \left(\frac{2}{4}\right)^2 \Rightarrow \mathbf{-0,25}$
- $C(N2) = \frac{0}{4} - \left(\frac{3}{4}\right)^2 \Rightarrow \mathbf{-0,5625}$
- $C(N3) = \frac{0}{4} - \left(\frac{2}{4}\right)^2 \Rightarrow \mathbf{-0,25}$

- $C(N4) = \frac{0}{4} - \left(\frac{1}{4}\right)^2 \Rightarrow \mathbf{-0,0625}$
- $Qw = C(N1) + C(N2) + C(N3) + C(N4) = -0,25 + -0,5625 + -0,25 + -0,0625 \Rightarrow \mathbf{-1,125}$

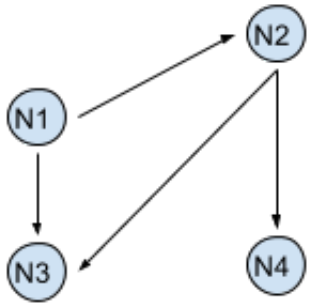


Figura 4.11 – Exemplo de grafo

A modularidade Qw acima reflete o valor quando considera que cada aresta, que representa uma classe, é um cluster diferente, sendo Qw obtido através da soma do cálculo de C(N1), C(N2), C(N3) e C(N4), que levam em consideração as arestas internas e externas de cada cluster.

O próximo passo do algoritmo é agrupar os dois clusters que geraram um maior ganho de modularidade. Essa variação da modularidade deve ser calculada agrupando as classes que possuem arestas entre si (N1→N2, N1→N3, N2→N3 e N2→N4), sendo o valor da modularidade Qw para cada agrupamento demonstrado a seguir.

A união de N1 a N2 resulta em:

- $C(N1, N2) = \frac{1}{4} - \left(\frac{3}{4}\right)^2 = -0,3125$
- $Qw_1 = C(N1, N2) + C(N3) + C(N4) = -0,3125 + -0,25 + -0,0625 \Rightarrow -0,625$
- $\Delta Q_1 = Qw_1 - Qw = -0,625 - (-1,125) \Rightarrow \mathbf{0,50}$

A união de N1 a N2 resulta em:

- $C(N1, N3) = \frac{1}{4} - \left(\frac{2}{4}\right)^2 = 0$
- $Qw_2 = C(N1, N3) + C(N2) + C(N4) = 0 + -0,5625 + -0,0625 \Rightarrow -0,625$
- $\Delta Q_2 = Qw_2 - Qw = -0,625 - (-1,125) \Rightarrow \mathbf{0,50}$

A união de N1 a N2 resulta em:

- $C(N2, N3) = \frac{1}{4} - \left(\frac{3}{4}\right)^2 = -0,3125$

- $Qw_3 = C(N2, N3) + C(N1) + C(N4) = -0,3125 + -0,25 + -0,0625 \Rightarrow -0,625$
- $\Delta Q_3 = Qw_3 - Qw = -0,625 - (-1,125) \Rightarrow \mathbf{0,50}$

A união de N1 a N2 resulta em:

- $C(N2, N4) = \frac{1}{4} - \left(\frac{2}{4}\right)^2 = 0$
- $Qw_4 = C(N2, N4) + C(N1) + C(N3) = 0 + -0,25 + -0,25 \Rightarrow -0,50$
- $\Delta Q_4 = Qw_4 - Qw = -0,50 - (-1,125) \Rightarrow \mathbf{0,625}$

Com base nas simulações apresentadas acima, a junção que trouxe o maior ganho de modularidade foi a junção de N2 com N4 que obteve um ganho de 0,625 ($\Delta Q_4 = 0,625$). Essa junção encerra a primeira iteração do algoritmo, lembrando que as iterações acabam somente quando todas as arestas estiverem dentro do mesmo cluster.

Para a segunda iteração deve-se fazer novamente todas as junções possíveis, mas agora considerando N2 e N4 como um único cluster. Dessa forma as junções possíveis são N1→N3, N1→N2,N4 e N2,N4→N3. Nessa iteração a modularidade a ser utilizada como base deve ser a que gerou a junção, ou seja Qw_4.

A união de N1 a N2 resulta em:

- $C(N1, N3) = \frac{1}{4} - \left(\frac{2}{4}\right)^2 = 0$
- $Qw_5 = C(N1, N3) + C(N2, N4) = 0 + 0 \Rightarrow 0$
- $\Delta Q_5 = Qw_5 - Qw_4 = 0 - (-0,50) \Rightarrow \mathbf{0,50}$

A união de N1 a N2 resulta em:

- $C(N1, N2 - N4) = \frac{2}{4} - \left(\frac{2}{4}\right)^2 = 0,25$
- $Qw_6 = C(N1, N2 - N4) + C(N3) = 0,25 + -0,25 \Rightarrow 0$
- $\Delta Q_6 = Qw_6 - Qw_4 = 0 - (-0,50) \Rightarrow \mathbf{0,50}$

A união de N1 a N2 resulta em:

- $C(N2 - N4, N3) = \frac{2}{4} - \left(\frac{2}{4}\right)^2 = 0,25$
- $Qw_7 = C(N2 - N4, N3) + C(N1) = 0,25 + -0,25 \Rightarrow 0$
- $\Delta Q_7 = Qw_7 - Qw_4 = 0 - (-0,50) \Rightarrow \mathbf{0,50}$

Como todas as junções da segunda iteração gerou um mesmo ganho na modularidade ($\Delta Q_5 = \Delta Q_6 = \Delta Q_7 = 0,50$), pode-se escolher qualquer uma delas, sendo que nesse exemplo será escolhido a primeira junção, de N1 com N3. Feito essa escolha resta apenas juntar os *clusters* N2-N4 com o N1-N3 e teremos todas as arestas em um único cluster, encerrando, assim, o algoritmo que gerará o dendrograma da Figura 4.12.

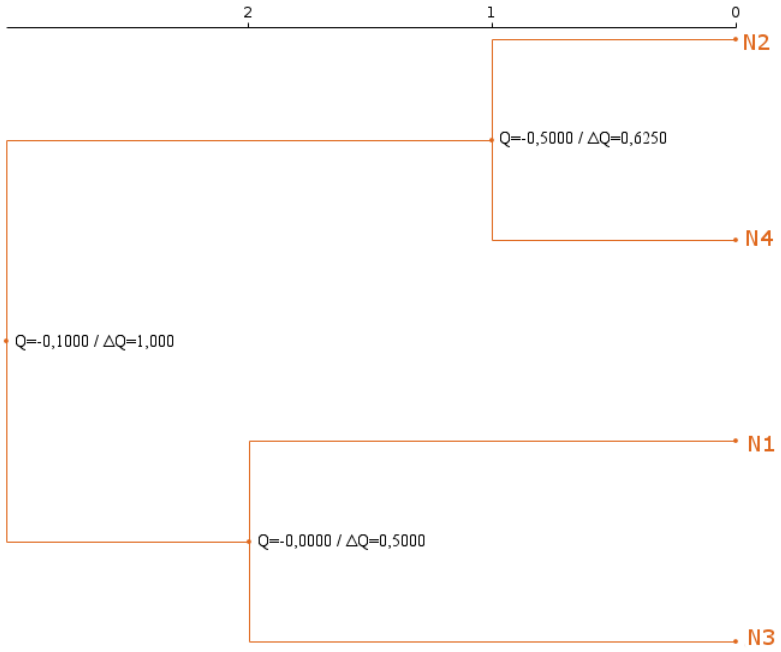


Figura 4.12 – Clusterização

Segundo Girvan e Newman (2003) o valor de Qw [-0,5, +1] geralmente varia entre 0,3 e 0,7, sendo que o valor para estruturas com forte ligação varia entre 0,6 e 0,7. Com base nesses dados, verifica-se que, para o exemplo da Figura 4.12, o melhor é deixar todas as classes em um único componente, pois não foi encontrado um ponto de corte em que o valor da modularidade esteja dentro da variação citada.

Utilizando das informações obtidas nessa etapa, é possível separar os grupos de classes de negócio e suas respectivas dependências em projetos separados. Cada projeto terá classes de negócio e de persistência exclusivas. Porém, como a camada de entidade é a representação da base de dados, as entidades utilizadas serão copiadas para cada projeto, de acordo com a utilização, podendo haver duplicação de classes. Contudo, não é o foco desse trabalho tratar da separação das entidades. Caso haja interfaces e superclasses na camada de negócio e/ou de persistência, elas também serão copiadas, conforme sugerido por Wang et al. (2008).

O processo de "clusterização"dessa seção pode ser visto na Figura 4.13. O próximo passo consiste na a criação e disponibilização dos serviços para cada um dos projetos criados, de forma que possam ser acessados através da internet.

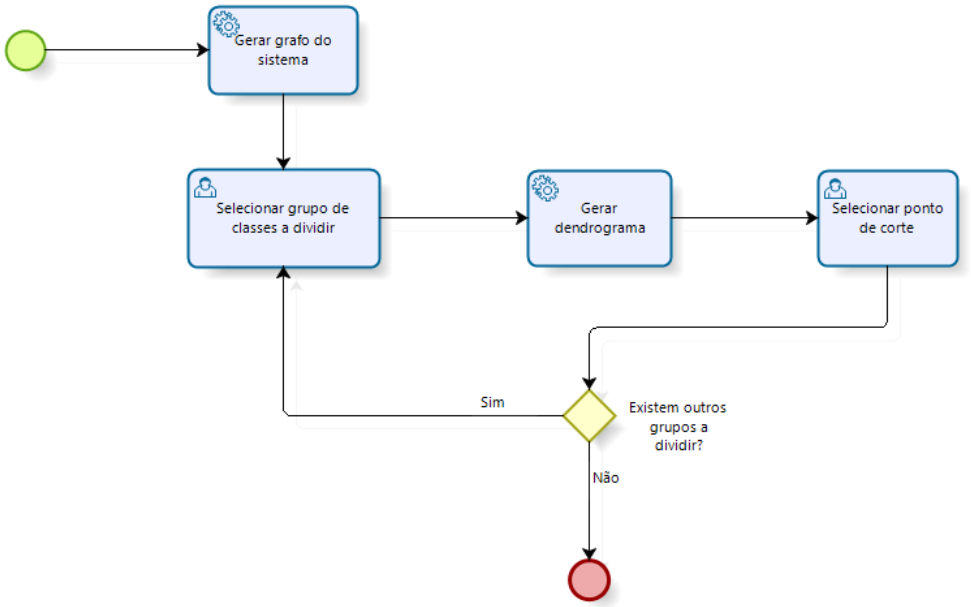


Figura 4.13 – Fluxo do processo de clusterização

4.2.3 Criação dos serviços

Seguindo as etapas descritas anteriormente têm-se vários grupos de classes que poderão disponibilizar serviços. Além disso, conforme dito anteriormente, os métodos a serem disponibilizados serão os existentes na camada de negócio de cada componente.

Cada linguagem especifica anotações, bibliotecas e códigos a serem inseridos para que sejam disponibilizados os *web services*. Além disso, existem várias IDEs que agilizam esse processo, de forma que não será detalhado uma técnica específica para isso, sendo essa seção responsável pela discussão de pontos importantes a serem observados.

Independente da forma em que os *web services* forem criados, será necessário alterar as chamadas feitas à classes que agora pertencem a outro serviço. Uma ilustração dessa refatoração pode ser visualizada na Figura 4.14.

Para que a refatoração citada acima possa ser realizada, primeiramente os *web services* precisam criados. Porém, existem algumas restrições a serem observadas no momento da criação desses serviços, conforme levantado por Guo et al. (2005), sendo elas:

1. O tipo do método deve ser público;
2. Métodos abstratos não podem ser publicados porque esse tipo de método não contém um corpo com sua implementação, ficando esta implementação a cargo de suas subclasses que poderão publicar seus métodos;

Chamada entre classes

```
class Classe1{
    public void metodo1(){
        //Classe2 e Classe2 pertencem ao mesmo projeto
        ...
        Classe2 classe2 = new Classe2();
        classe2.metodo2();
        ...
        Classe3 classe3 = new Classe3();
        classe3.metodo3();
        ...
    }
}
```

Chamada entre serviços

```
class Classe1{
    public void metodo1(){
        ...
        //Classe2 migrou para outro servico
        ServicoClasse2 classe2 = new ServicoClasse2();
        classe2.metodo2();
        ...
        //Classe3 continua no mesmo projeto
        Classe3 classe3 = new Classe3();
        classe3.metodo3();
        ...
    }
}
```

Figura 4.14 – Refatoração chamada entre serviços

3. Métodos com mesmo nome devem possuir nomes de serviço diferentes, pois o padrão WSDL, responsável pela descrição e localização dos serviços, não leva em consideração as assinaturas dos métodos, somente seus nomes;

4. Se um método possui transação, mas não é a raiz dessa transação, não deve ser disponibilizado como serviço, pois ele possui apenas parte da operação a ser realizada.

A última restrição só é relevante se um método que não é raiz de uma transação não contiver uma operação completa, o que nem sempre acontece. Pode haver métodos que realizam uma operação completa, mas que também compõem uma funcionalidade maior ao mesmo tempo. Porém, independente dele ser executado sozinho ou em conjunto com outros métodos, em ambos os casos pode existir a necessidade de controlar a transação. No exemplo da Figura 4.14, se o **metodo1** fizer uso de transação, esse controle deve ser mantido mesmo depois da refatoração, já que não houve mudança da funcionalidade.

Como um dos objetivos dessa abordagem é que a funcionalidade original seja mantida no sistema modernizado, é importante que as funcionalidades em que todo um conjunto de operações são concluídas, ou nenhum de seus resultados são efetivados, mantenham essas mesmas propriedades. Isso é conseguido através do uso de transações que são mecanismos para garantir que todos os participantes de uma aplicação alcancem um resultado de comum acordo. Ela tem sido historicamente definida através das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), sendo essas propriedades definidas como (REUTER; GRAY, 1993):

- **Atomicidade:** todas operações são concluídas ou nenhuma operação é concluída.
- **Consistência:** a aplicação deve sair de um estado consistente para outro estado também consistente.
- **Isolamento:** os efeitos de uma operação não são compartilhados para fora da transação até que seja concluída com sucesso.

- **Durabilidade:** uma vez que a transação for concluída com sucesso, as alterações devem permanecer mesmo em caso de futuras falhas.

A Seção 4.2.3.1 traz mais detalhes sobre o controle de transações entre serviços diferentes.

4.2.3.1 Transações entre serviços

Segundo Snell (2002), transações são conceitos fundamentais na construção de aplicações distribuídas confiáveis, porém, nenhuma das principais especificações de *web services* (SOAP, WSDL, UDDI, etc), foram projetadas para prover mecanismos que permitam a eles se conectarem para criar soluções dependentes e confiáveis.

Para resolver esse problema a IBM, juntamente com a Microsoft, definiram duas especificações complementares, a WS-Coordination (NEWCOMER; ROBINSON, 2009b), e a WS-Transaction (NEWCOMER; ROBINSON, 2009a). A WS-Coordination provê mecanismos para criar e registrar serviços, usando os protocolos definidos pela WS-Transaction (FREUND; STOREY, 2002; LANGWORTHY et al., 2004), sendo seus níveis de operação visualizados na Figura 4.15.

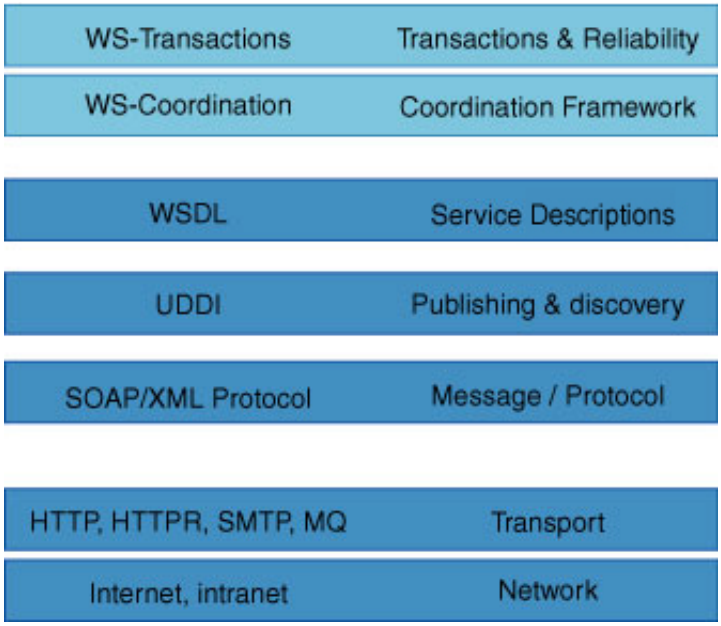


Figura 4.15 – Padrões de descrição de web services (FREUND; STOREY, 2002)

WS-Coordination

Segundo Freund e Storey (2002), Langworthy et al. (2004), o *framework* definido pela WS-Coordination é composto por três elementos:

- **Serviço de ativação:** cria uma atividade e especifica seu protocolo de ativação disponível;
- **Serviço de registro:** coordena a seleção de protocolos e registra os participantes e
- **Serviço de coordenação:** controla o processo de conclusão das atividades, utilizando para isso o protocolo de coordenação selecionado para a transação (definidos pela especificação WS-Transaction).

Para cada nova atividade criada, o serviço de ativação retorna um CoordinationContext (elemento XML utilizado em uma mensagem como convite para participar de uma atividade), que contem os seguintes campos (LANGWORTHY et al., 2004):

- Identificador da atividade;
- Tipo da transação (atômica ou de negócio);
- Endereço do serviço de registro;
- Tempo de expiração da atividade (opcional) e
- Elementos extendidos, que permitem que outras informações sejam comunicadas.

Temos, então, que o *framework* de coordenação provê um sistema para gerenciar comunicações entre *web services*, além de poder trabalhar com sistemas que utilizam transações ACID, assim como outras formas de transação. De modo que fica a cargo da coordenação de protocolos (definida pela especificação WS-Transaction) implementar as transações ACID (FREUND; STOREY, 2002).

WS-Transaction

Segundo Freund e Storey (2002), Langworthy et al. (2004), a especificação WS-Transaction define os protocolos de coordenação atômicos e de negócio.

O protocolo para transações atômicas é utilizado para tratar atividades com tempo de vida curto. No escopo desse protocolo é observado todo o conjunto de operações a serem executadas, sendo que ou todas são concluídas com sucesso, ou em caso de falha de alguma delas, nenhuma operação é efetivada. Esse objetivo é atingido utilizando o protocolo *Two-Phase Commit* (2PC).

O protocolo 2PC coordena o registro dos serviços para poder tomar a decisão de efetivar ou cancelar as operações e informa todos os serviços do resultado final, sendo esta decisão a mesma para todos os serviços envolvidos. Essa tomada de decisão é feita nas duas fases descritas a seguir (LANGWORTHY et al., 2004):

- **Fase de preparação:** todos os participantes são avisados para aguardarem sinal de conclusão ou cancelamento de suas operações e, então, votar no resultado final. Esse voto é propagado para o coordenador geral da transação para tomar a decisão final.
- **Fase de efetivação:** se todos os participantes votarem pela conclusão, então as operações são efetivadas, caso contrário são abortadas.

O protocolo para transações de negócio trata de atividades de longa duração, mas, para diminuir a espera pela utilização dos recursos, os resultados das operações intermediárias devem ser liberados mesmo antes do término de todo o processo. Mecanismos de gerenciamento de falha e compensação são geralmente utilizados para reverter os efeitos de atividades anteriormente completadas.

É possível usar ambos protocolos em combinação, conforme pode ser observado na Figura 4.16, sendo que a atividade 2 utiliza transações atômicas e a atividade 3 utiliza ambas as transações, atômicas e de negócio. A imagem mostra também a relação existente entre os protocolos WS-Transaction e WS-Coordination.

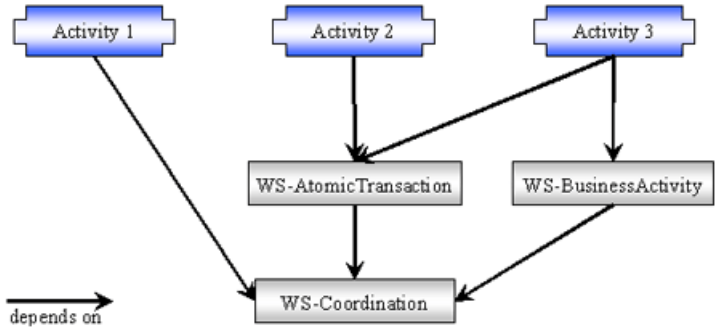


Figura 4.16 – Utilização dos protocolos de transação (LANGWORTHY et al., 2004)

O fluxo dessa etapa pode ser visualizada na Figura 4.17.

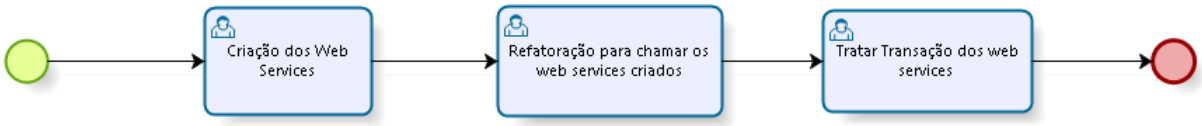


Figura 4.17 – Fluxo para criação dos serviços

4.3 Resumo do Capítulo

Conforme detalhado neste capítulo, a abordagem proposta para modernização de sistemas monolíticos, orientados a objetos e que também possuem arquitetura de

três camadas, é composta por três etapas (Seção 4.1). A primeira é a diminuição de dependências (Seção 4.2.1), que irá reduzir a quantidade de relacionamentos entre as classes, tanto na camada de persistência quanto na de negócio. A segunda etapa é a "clusterização", (5.3.2) que agrupará as classes relacionadas, e a última define critérios a serem seguidos durante a criação dos *web services*.

Para auxiliar na etapa de diminuição de dependências, é utilizada a fórmula da força de conectividade (Seção 4.1.1), que calcula a força de ligação entre as classes. Na etapa de "clusterização"é executado o algoritmo *fast community* (Seção 4.1.2) que agrupa as classes relacionadas permitindo a divisão dos grupos com tamanhos indesejados.

No próximo capítulo é apresentado o estudo de caso, feito sobre um sistema acadêmico, que executa as três etapas apresentadas, demonstrando a efetividade das mesmas. Foram utilizadas ferramentas que automatizam a execução das duas primeiras etapas, diminuição de dependências e clusterização, mostrando a eficácia da semi-automatização dessas etapas. Para a etapa de criação de serviços, é apresentada a implementação dos critérios definidos, para a criação dos *web services*, dentro do sistema acadêmico.

5 ESTUDO DE CASO E AVALIAÇÃO DA ABORDAGEM

O principal objetivo deste estudo de caso é analisar a eficácia do modelo de modernização proposto dentro do contexto de sistemas monolíticos, orientados a objetos e que possuem também uma arquitetura de três camadas, sendo que foi executado seguindo a metodologia *Goal/Question/Metric* (BASILI; CALDIERA; ROMBACH, 1994).

5.1 Contexto

Segundo Kitchenham e Pickard (1998), o estudo de caso define o conjunto de objetivos e limitações em que ele deve ser executado. Para definir esses objetivos, a metodologia GQM *(Goal/Question/Metric)* foi utilizada, derivando em um conjunto de questões que devem ser respondidas para determinar se o objetivo foi alcançado.

Objetivo (*Goal*): Avaliar a abordagem proposta, em relação a sua eficácia, do ponto de vista do engenheiro de software e no contexto de sistemas monolíticos, orientados a objetos e que tenham sido desenvolvidos com arquitetura de três camadas.

Definido o objetivo, ele é refinado em questões para caracterizar a forma em que a avaliação do objetivo será realizada (BASILI; CALDIERA; ROMBACH, 1994). As questões (*questions*) definidas foram:

Questão 1: Que tipo de melhoria a refatoração das classes traz?

Nesta questão, tenta-se identificar a melhoria obtida com a etapa de Diminuição de dependências (Seção 4.2.1).

Questão 2: Quais foram as melhorias obtidas nos componentes gerados?

Nesta questão, tenta-se identificar os benefícios obtidos com a etapa de "Clusterização"(Seção 5.3.2).

Questão 3: A funcionalidade original é mantida após a criação dos serviços?

Nesta questão, tenta-se verificar se as funcionalidades originais foram mantidas após a modernização de arquitetura.

Para responder às perguntas, um conjunto de métricas (*metrics*) são definidas e associadas às questões (WOHLIN et al., 2000), sendo elas apresentadas na Tabela 5.1, na qual a coluna **Medida** mostra a medida utilizada para o cálculo, a coluna **Questões** mostra à qual questão a métrica está relacionada e a coluna **Resultados Esperados** indica se é esperado que os percentuais aumentem (⇑) ou diminuam (⇓), para as métricas

M1 a M4. Para a métrica M5, a coluna de resultados indica o percentual de testes de regressão executados com sucesso.

Tabela 5.1 – Lista de métricas e relação com as questões do paradigma GQM.

	Métrica	**Medida**	**Questão**	**Resultado Esperado**
M1	% de variação no acoplamento médio das classes	ACCL	Q1	⇓
M2	% de variação no tamanho dos componentes	TACO	Q2	⇓
M3	% de variação na complexidade dos componentes	COXP	Q2	⇓
M4	% de variação no acoplamento entre componentes	ACCO	Q2	⇑
M5	% de testes de regressão realizados com sucesso	TEST	Q3	100%

A medida ACCL (acoplamento entre classes) é calculada somando-se a quantidade de classes de negócio ou persistência referenciadas por uma classe de negócio, desconsiderando o número de vezes em que a referência acontece. TACO (tamanho do componente) é a quantidade de classes de negócio ou persistência que fazem parte um componente. ACCO (acoplamento entre componentes) é a quantidade de outros componentes que ele referencia, independente do número de vezes que isso acontece. TEST é a execução de um caso de teste no qual a comparação da funcionalidade original com a do *web service* geram o mesmo resultado. Vale ressaltar que as medidas ACCL e TACO consideram somente as classes de negócio e persistência devido ao fato de ser essas as classes a serem distribuídas entre os componentes, sendo que as demais serão copiadas de acordo com a necessidade.

COXP (complexidade de um componente) foi definida por Cho, Kim e Kim (2001) e representa a complexidade estática de um componente (CSC). Ela leva em consideração os relacionamentos entre as classes contidas no componente. Sua fórmula é definida como:

$$CSC = \sum_{i=1}^{m}(Count(R_i) * W(R_i))$$

Em que:

- Count(R_i) = Quantidade de cada tipo de relacionamento entre as classes (Dependência, Agregação, Generalização, Composição) e
- W(R_i) = Peso atribuído para cada tipo de relacionamento (Tabela 5.2).

5.2 Planejamento

Para o teste da proposta foi escolhido o sistema acadêmico SIGA-EPCT[1][2] (Sistema Integrado de Gestão Acadêmica da Educação Profissional e Tecnológica), na qual as

[1]Site do sistema: <http://colaboracao.sigaepct.net/>
[2]Foi utilizada nessa pesquisa foi a versão em desenvolvimento 11.1 (commits do dia 05/10/2016)

Tabela 5.2 – Pesos com base no tipo de relacionamento para o cálculo do CSC (CHO; KIM; KIM, 2001).

Tipo do Relacionamento	**Peso**
Dependência	2
Associação	4
Generalização	6
Agregação	8
Composição	10

técnicas apresentadas foram testadas. Este sistema foi idealizado inicialmente através de um projeto de pesquisa da SETEC/MEC que financiou o seu desenvolvimento através de parcerias com vários Institutos Federais em todo o Brasil. Hoje, é o sistema acadêmico utilizado por alguns desses institutos.

A escolha desse sistema se deve ao fato dele atender aos requisitos da pesquisa sendo um sistema Java, desenvolvido sob a arquitetura três camadas e monolítico, pois possui apenas um arquivo EAR (*Enterprise Application aRchive*) a ser publicado em um servidor de aplicação. Ele possui 200 classes de negócio, 244 persistências e 474 entidades, totalizando 1148 classes e 77.092 linhas de código, desconsiderando a camada de apresentação.

5.2.1 Ferramentas utilizadas

Para automatizar os cálculos necessários foram utilizadas duas ferramentas, sendo que a primeira foi desenvolvida no decorrer desta pesquisa, nomeada de JCluster, e é responsável pelos cálculos citados anteriormente (seções 4.1.1 e 4.1.2). A segunda é a ferramenta JTransformer[3], encarregada de analisar padrões de código e realizar refatorações.

Ambas as ferramentas, JCluster e JTransformer, são plugins da IDE Eclipse. Elas executam suas operações utilizando como entrada o projeto Java aberto na IDE. A limitação dessas ferramentas é que elas funcionam apenas para sistemas desenvolvidos em Java. Para a utilização dessas ferramentas em outros sistemas Java será necessário que o Engenheiro de Software faça algumas configurações sobre a identificação das camadas de negócio e persistência. Nas seções subsequentes suas funcionalidades serão brevemente descritas, ficando a explicação de suas utilizações para a seção do estudo de caso (Capítulo 5).

[3]Site da ferramenta <https://sewiki.iai.uni-bonn.de/research/jtransformer/start>

5.2.1.1 JCluster

O plugin idealizado e desenvolvido durante esta pesquisa[4] possui as funcionalidades listadas abaixo, sendo que todas elas usam como base o código do projeto aberto na IDE Eclipse. Além disso, esse plugin identifica uma classe como pertencente à camada de negócio ou persistência se faz através do pacote em que a classe pertence.

- **Cálculo da força de conectividade:** calcula a FC entre as classes, seguindo as fórmulas da Seção 4.1.1. Com base nesses cálculos, também são identificadas as maiores ligações entre as classes de negócio e persistência, detalhado na Seção 4.2.1.
- **Geração de grafo:** a partir do código-fonte do sistema é gerado um grafo, no qual cada classe de negócio é representada por um vértice e seus relacionamentos por arestas. Cada conjunto de classes relacionadas tem a mesma cor para facilitar a identificação. O exemplo de um grafo gerado pode ser visualizado na Figura 5.7 da Seção 5.3.2, no qual será descrito a utilização desse plugin em um estudo de caso.
- **Geração de dendrograma:**[5] ao clicar em qualquer uma das classes do grafo, é gerado um dendrograma, seguindo o algoritmo *fast community* (Seção 4.1.2), baseando-se em todas as arestas que contém a mesma cor da selecionada. Ao clicar com o botão direito em um dos pontos de junção é possível pre-visualizar a quantidade de grupos a serem gerados diferenciando-os através das cores. Ao clicar duas vezes em uma junção, as cores dos grupos do dendrograma são repassadas para o grafo. A visualização dessa funcionalidade está ilustrada na Figura 5.9 da Seção 5.3.2, na qual é descrita a utilização desse plugin em um estudo de caso.
- **Separação dos componentes:** para cada conjunto de cores do grafo é criada uma pasta com todas as classes de mesma cor, além de suas dependências. Com isso é possível criar projetos independentes para cada componente, observando apenas as bibliotecas necessárias.

Dentre as funcionalidades citadas acima, vale detalhar o algoritmo usado para a coloração dos vértices no grafo, sendo ele descrito no Algoritmo 2.

O if/else da linha 6 do algoritmo 2 faz com que um vértice que possua mais de uma origem não tenha a mesma cor de nenhuma delas. Esse passo foi definido para que o engenheiro de software não seja induzido a englobar esses vértices a nenhuma das origens primárias.

[4]Disponibilizado no github através do link <https://github.com/aborgesrodrigues/hierarchical-clustering>

[5]Foi utilizado como base o sistema já existente disponível através do link <https://github.com/lbehnke/hierarchical-clustering-java>:

```
1 while existir vértice sem cor do
2 |  Selecione um dos vértices sem cor;
3 |  Defina uma cor para o vértice selecionado;
4 |  while existir vértices referenciados pelo vértice selecionado do
5 |  |  Selecione um dos vértices referenciados;
6 |  |  if vértice não possuir cor then
7 |  |  |  Insira mesma cor do vértice selecionado;
  |  |  else
8 |  |  |  Insira uma nova cor para o vértice selecionado;
  |  |  end
  |  end
   end
```

Algoritmo 2: Algoritmo de coloração dos vértices do grafo.

5.2.1.2 JTransformer

Essa ferramenta é um plugin do Eclipse que permite a análise e as transformações de códigos Java (KNIESEL; HANNEMANN; RHO, 2007; ALVES; HAGE; RADEMAKER, 2011; BINUN; KNIESEL, 2012). O JTransformer analisa o código fonte, suas dependências com projetos e bibliotecas, além de criar uma representação do código em Prolog[6]. Suas análises podem ser expressas em um nível de abstração bastante elevado. Kniesel, Hannemann e Rho (2007) compararam-na com um conjunto de outras ferramentas de análise e transformação de código, sendo que o JTransformer foi melhor nos seguintes aspectos:

- **Expressividade:** não limita as análises e transformações que podem ser realizadas;
- ***Turnaround*:** suporta um nível de abstração que promove o rápido desenvolvimento sem limitar a expressividade;
- **Integração:** realiza análise e integração sem necessidade de ferramentas externas;
- **Performance:** rapidez nas análises individuais (de milisegundos a segundos);
- **Escalabilidade:** a performance nas análises individuais acontecem mesmo em sistemas com dezenas de milhares de classes;
- **Suporte a Multi-projetos:** permite a análise e as transformações de múltiplos projetos que relacionam entre si e
- **Disponibilidade:** possibilita baixar versões do software e documentação apropriada.

Alves, Hage e Rademaker (2011) também compararam algumas ferramentas, analisando os items abaixo, sendo que o resultado pode ser visualizado na Tabela 5.3, sendo que

[6]Detalhes da linguagem Prolog pode ser encontrado no link <http://lpn.swi-prolog.org/lpnpage.php?pageid=online>

o **"x"** e **-"** representam se a ferramenta atende ou não determinado critério, respectivamente, seguindo os seguintes critérios:

- **Paradigma** em que a linguagem de busca é baseada;
- **Tipos** de dados suportados pela linguagem;
- **Parametrização** que indica se o comportamento das consultas podem depender de parâmetros;
- **Polimorfismo** que especifica se as consultas podem ser abstraídas dos tipos em que as relações são construídas;
- **Modularidade** que determina a extensão em que é possível reutilizar uma consulta específica para construir outras consultas e
- **Bibliotecas** que determina a possibilidade de usar e/ou criar bibliotecas de consultas genéricas.

Tabela 5.3 – Comparação entre ferramentas de análise de código (ALVES; HAGE; RADEMAKER, 2011).

Criterio vs. Ferramentas		**Grok**	**Rscript**	**JRelCal**	**SemmleCode**	**CrocoPat**	**JGraLab**	**JTransformer**
Paradigma		Relational	Relational and Comprehensions	API Relational	OO and SQL-like	FO-logic Imperative	SQL-like and Path Expr	FO-logic
Tipos	String	x	x	x	x	x	x	x
	Int	x	x	x	x	x	x	x
	Real	x	-	x	x	x	x	x
	Bool	-	x	x	x	x	x	x
	Other	-	Composite and Location	Java	Object	-	Edges and Node	Logic terms
Parametrização		-	x	x	-	x	x	x
Polimorfismo		-	x	x	x	-	x	x
Modularidade		x	x	x	x	x	-	x
Bibliotecas		-	-	x	x	-	-	x

A Figura 5.1 mostra alguns dos predicados básicos do JTransformer (aqueles terminados com T maiúsculo, com marcação onde aparecem). As palavras iniciadas em maiúsculo são variáveis, o *underscore* é uma pseudo-variável que indica atributos irrelevantes para a análise/transformação.

Para facilitar o entendimento de quem não é familiarizado com os termos da programação lógica, Kniesel, Hannemann e Rho (2007) fazem uma comparação desse tipo de programação com suas contrapartes do banco relacional (entre parênteses).

Cada nó AST é um *fact base* (tupla do banco de dados). O predicado (nome da relação), representa o tipo do nó, o primeiro parâmetro (atributo) é um identificador único (chave) do respectivo nó e os outros parâmetros são valores primitivos ou identificadores de outros nós, representando referência, sendo que o segundo parâmetro referencia o nó superior. Diferentemente dos bancos relacionais, os programas lógicos não armazenam *facts* para cada relação, mas define predicados através de derivação de regras e podem ser definidos recursivamente.

Para exemplificar esses conceitos, a seguir será dada uma breve explicação dos termos destacados da Figura 5.1, lembrando que o caractere *underscore* (_) pode ser utilizado em qualquer um dos parâmetros esperados, para o caso de se fazer buscas genéricas:

- **fieldDefT (linha 6):** identifica a declaração de um atributo dentro de classes (**InClass**), pertencente a determinado tipo (**T**);
- **methodT (linha 10):** identifica métodos, pertencentes a classes (**InClass**), que possuam parâmetros (**Args**) e tipo de retorno (**T**) específicos;
- **getFieldT (linha 18):** identifica o acesso a um atributo dentro de um método (**MName**) por uma variável ou sendo enviado como parâmetro a outro método (**OnRecv**);
- **execT (linha 21):** identifica uma execução dentro de um método (**CallingM**);
- **NewClassT (linha 24):** identifica a instanciação de uma classe, dentro de um método (**CalledM**), que tenha recebido alguns parâmetros (**Args**);
- **paramT (linha 27):** identifica os parâmetros *parametrized*, por exemplo, em **String[]**, **Class<T>**, os colchetes ([]) e a sequência <T>, serão os itens identificados;
- **forLoopT (linha 30):** identifica a instrução *for* executada em determinado método (**InMethod**);

Nas próximas seções serão mostradas a abordagem proposta para modernização de sistemas orientados a objetos para SOA e a utilização dos mecanismos apresentados dentro dessa abordagem.

5.3 Execução

Nessa seção, será descrita a execução da abordagem de modernização sobre o sistema SIGA-EPCT, detalhando cada uma das etapas, facilitando o entendimento das questões, das métricas e de seus cálculos.

5.3.1 Diminuição das dependências

Conforme explicado na Seção 4.1, os primeiros passos da abordagem proposta é a diminuição das dependências entre as classes, de forma que possa obter componentes menores nas etapas posteriores.

```
1  type(Type,Name) :-
2  classDefT(Type, _,Name, _).
3
4  field(Field, InClass, FType, FName) :-
5  type(_ ,FType, _),
6  fieldT(Field,InClass,T,FName, _).
7
8  method(Meth,InClass,MName,Args,RetType) :-
9  type( _,RetType, _),
10 methodT(Meth,InClass,MName,Args,T, _, _).
11
12 instanceMethod(Method,InClass,Name,Args,Type) :-
13 method(Method,InClass,Name,Args,Type),
14 not(Name = '<init>'),
15 not(Name = '<cinit>').
16
17 accesses(AccessingM,InBlock,OnRecv,AccessedField) :-
18 getFieldT( _,InBlock,AccessingM,OnRecv,AccessedField).
19
20 calls(CallingM,InBlock,CalledM,Args) :- // %Inst. method
21 execT(Exec,InBlock,CallingM, Call),
22 applyT(Call,Exec,CallingM, _, _,Args,CalledM).
23 calls(CallingM,InBlock,CalledM,Args) :- // %Constructor
24 newClassT( _,InBlock,CallingM,CalledM,Args, _, _, _).
25
26 param(Param,InMethod,Type) :-
27 paramT(Param,InMethod,type( _,Type, _), _).
28
29 forLoopBody(For,InMethod, LoopBody) :-
30 forLoopT(For, _,InMethod, _, _, _,LoopBody).
```

Figura 5.1 – Exemplo de script do JTransformer (KNIESEL; HANNEMANN; RHO, 2007)

Inicialmente procura-se por classes de persistência que executem métodos de outras classes também de persistências. Para isso utilizou-se o plugin JTransformer (Seção 5.2.1.2) para realizar essa pesquisa dentro do código fonte do sistema SIGA-EPCT.

A Figura 5.2 mostra o código desse script, cuja explicação segue abaixo:

- Linha 1: nome do método;
- Linha 5: identifica toda chamada a método feita, atribuindo o método de origem e o receptor da chamada às variáveis MethodCall e Receiver respectivamente;
- Linhas 7 e 8: identificam todas as classes de persistência (classes que extendem de GenericDAO);
- Linhas 10 a 16: filtram somente os receptores que são classes de persistência;
- Linhas 18 a 20: filtram apenas os métodos de classe de persistência;
- Linha 22: ignora chamadas a métodos dentro da própria classe e
- Linha 24: ignora chamadas a métodos da superclasse.

O resultado da execução do script é mostrado na Figura 5.3, sendo que, no ponto 1, tem-se o código da classe com a identificação do ponto onde o script encontrou a

chamada entre persistências. O ponto 2 mostra os scripts existentes e a quantidade de ocorrências encontradas na execução de cada um deles. Já o ponto 3 contém as próprias ocorrências, indicando a classe e o número da linha em que ela ocorre. Ao clicar em uma dessas ocorrências é aberto o arquivo com o cursor já na linha identificada, como mostrado no ponto 1.

No caso da classe **RegraAcademicaDAO** da Figura 5.3, a refatoração para remover a dependência entre as classes de persistência, consiste em retirar o método **inserir** dessa classe e migrá-lo para a classe de negócio correspondente, que é a **ManterRegraAcademicaEJB**. O resultado dessas operações pode ser observado através dos trechos de código mostrado na Figura 5.4. Cada uma das ocorrências encontradas deve ser analisada individualmente, pelo engenheiro de software, para encontrar uma melhor forma de retirar esses relacionamentos indesejados da camada de persistência, sendo essa uma limitação que pode ser abordada em um trabalho futuro.

```
1  :- module(persistence__persistence_analysis, [ persistence_persistence_call/1 ]).
2
3  persistence_persistence_call(CallId) :-
4      %Identifica cada chamada de metodo
5      callT(CallId, _, MethodCall, Receiver, _, _, _, _),
6      %Identifica classes de persistência
7      fully_qualified_name(GenericDAO, 'org.sigaept.nucleo.dao.GenericDAO'),
8      subtype(DAO, GenericDAO),
9      %filtra somente chamadas feitas a classes de persistência
10     (
11         (identT(Receiver, CallId, MethodCall, Local), localT(Local, _, _, DAO, _, _))
12         ;
13         newT(Receiver, _, _, _, _, _, _, DAO, _)
14         ;
15         callT(Receiver, _, _, _, _, _, _, DAO)
16     ),
17     % busca somente em métodos de classes de persistência
18     methodT(MethodCall, DAOClass, _, _, _, _, _, _),
19     subtype(DAOClass, GenericDAO),
20     classT(DAOClass,_,_,_,_),
21     %ignorar chamadas dentro da própria classe
22     DAOClass \== DAO,
23     %ignorar chamadas a superclasse
24     DAO \== GenericDAO.
```

Figura 5.2 – Script para encontrar chamadas entre classes de persistência

A próxima etapa para diminuição das dependências é fazer com que uma classe de persistência fique relacionada a somente uma classe de negócio, conforme explicado na Seção 4.2.1. Para isso, primeiramente é necessário identificar as persistências e a classe de negócio correspondente a cada uma delas. Assim, foi desenvolvido o plugin JCluster (Seção 5.2.1.1) que analisa o código e calcula a força de conectividade (Seção 4.1.1) entre todas as classes de negócio e persistência. Ao final do processo é gerado um arquivo contendo essa relação, conforme pode ser observado na Tabela 5.4.

Com essas informações pode-se fazer a refatoração no código, alterando as classes de negócio que aparecem no arquivo gerado, de forma que elas executem somente as suas classes de persistência, modificando as demais chamadas para as classes de negócio adequada.

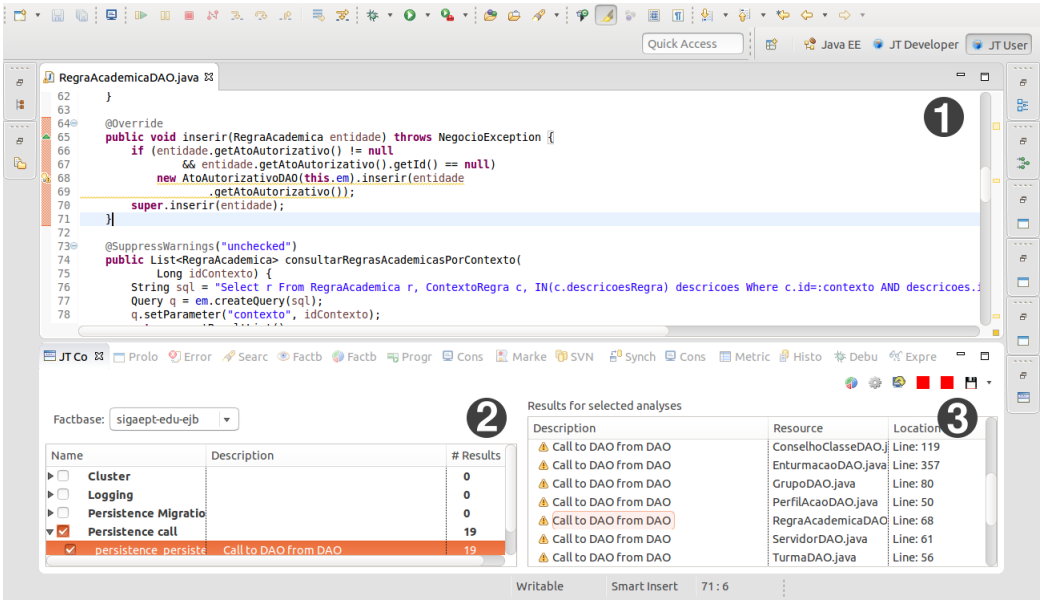


Figura 5.3 – Funcionamento da ferramenta JTransformer

Método na classe de persistência

```
public class RegraAcademicaDAO extends
 ↪   GenericDAO<RegraAcademica> {
    ...

    @Override
    public void inserir(RegraAcademica entidade)
     ↪   throws NegocioException {
        if (entidade.getAtoAutorizativo() != null &&
         ↪   entidade.getAtoAutorizativo().getId()
         ↪   == null)
            new AtoAutorizativoDAO(this.em).inserir(
             ↪   entidade.getAtoAutorizativo());

            super.inserir(entidade);
            }

            ...
    }
```

Método migrado para classe de negócio

```
public class ManterRegraAcademicaEJB extends
 ↪   GenericCrudEJB<RegraAcademica,
 ↪   RegraAcademicaDAO> implements
 ↪   IManterRegraAcademicaEJB{
    ...

    @Override
    public void inserir(RegraAcademica entidade)
     ↪   throws NegocioException {
        if (entidade.getAtoAutorizativo() != null &&
         ↪   entidade.getAtoAutorizativo().getId()
         ↪   == null)
            new AtoAutorizativoDAO(this.em).inserir(
             ↪   entidade.getAtoAutorizativo());

            super.inserir(entidade);
        }

        ...
        }
```

Figura 5.4 – Refatoração para retirada de dependência entre persistências

Essa refatoração é feita, em sua maior parte, também pela ferramenta JTransformer (Seção 5.2.1.2), através da utilização de scripts e pode ser representado pelo Algoritmo 3.

A exclusão citada nas linhas 5 e 6 se faz possível porque existe somente um método com apenas uma chamada para outra classe de negócio sem agregar nem processar nenhuma informação. Dessa forma, esse método torna-se irrelevante, já que é possível obter a mesma informação apenas chamando a classe de destino diretamente. Conforme dito antes, ao realizar essas exclusões, pode acontecer de algumas classes ficarem sem nenhum método em seu corpo, podendo, então, ser excluídas do projeto, sendo que no sistema SIGA-EPCT, sete classes de negócio foram removidas.

Tabela 5.4 – Segmento do resultado da identificação das classes persistências com suas respectivas classes de negócios

Classe de Persistência	Classe de Negócio	FC
RegraAcademicaDAO	ManterRegraAcademicaEJB	81.82
MatrizCurricularPeriodoDAO	ManterMatrizCurricularEJB	28.05
ModalidadeAprendizagemDAO	ManterCursoEJB	0.15
LotacaoServidorDAO	ManterServidorEJB	129.35
ParticipacaoEventoExternoDAO	ManterEventoExternoEJB	0.90
ContaCorrentePagamentoDAO	ManterServidorEJB	1.94
CDUDAO	ManterCDUEJB	0.30
AlunoDAO	ManterRelatorioAlunoEJB	523.80
ProjetoExtensaoDAO	ManterProjetoExtensaoEJB	64.56
ReaberturaTurmaDAO	ReaberturaTurmaClasseEJB	33.62

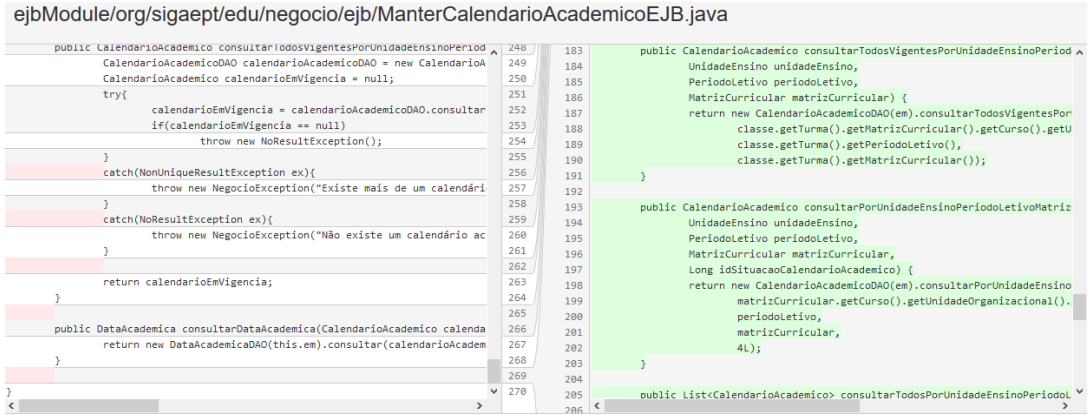
```
1 for cada classe de negócio do
2     Criação de métodos na classe de negócio para disponibilizar suas persistências,
       caso não existam;
3     for cada método da classe que possui chamadas à persistências de outra classe de
       negócio do
4         Migrar chamadas a persistências não pertencentes a ela para os métodos do
           negócio de destino;
5         if método contiver somente essa chamada then
6             Excluir método;
          end
      end
  end
```

Algoritmo 3: Algoritmo para alteração das chamadas da camada de negócio

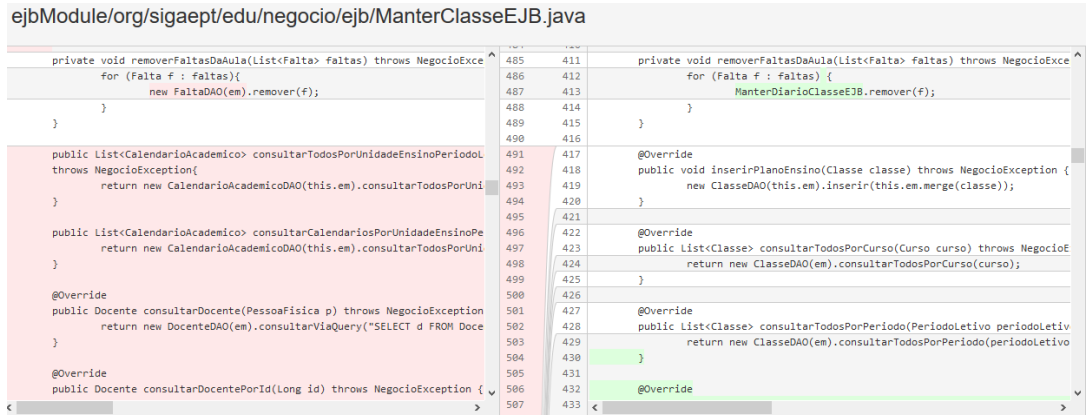
A Figura 5.5 mostra parte do resultado das etapas da refatoração citada anteriormente, sendo que a 5.5a exibe alguns métodos inseridos na classe **ManterCalendarioAcademicoEJB**, referentes a chamadas de sua persistência que estavam incluídas em outras classes. A Figura 5.5b demonstra a exclusão de métodos que não agregavam informação ou processamento ao resultado da chamada (item 2.2 das etapas de refatoração). Além disso, também revela o resultado da alteração do método **removerFaltasDaAula**, que teve a chamada ao método **remover** da classe **FaltaDAO**, alterado para uma chamada ao método de mesmo nome mas pertencente à classe **ManterDiarioClasseEJB**, que é a responsável pela persistência.

Apesar de agilizar bastante o processo, ainda é necessário fazer pequenas adequações manuais no código após as refatorações, como adequação dos *imports* e adaptação dos métodos duplicados que podem ter sido inseridos, além da exclusão, quando possível, de classes que ficaram sem nenhum método, entre outros.

Parte desse script, responsável pela substituição da chamada de um método de uma classe da persistência pelo método equivalente de sua respectiva classe de negócio,



(a) Inclusão de métodos



(b) Alteração de chamada e exclusão de métodos

Figura 5.5 – Resultado das refatorações da modernização onde uma persistência está associada à apenas um negócio

pode ser observado na Figura 5.6, sendo a explicação desse script detalhada a seguir:

- Linha 8: Variável CallId identifica uma chamada a uma persistência que não pertence a classe de negócio, e BusinessTarget representa a classe de negócio para a qual a persistência foi migrada;
- Linhas 9 a 15: Cria uma referência para o atributo da classe BusinessTarget;
- Linha 18: Cria a declaração de uma variável do tipo BusinessTarget no código;
- Linhas 20 e 21: altera a chamada a um método de uma persistência para a classe de negócio BusinessTarget.

Ao terminar as refatorações pode-se iniciar o processo de "clusterização"das classes. O novo código refatorado será utilizado como base para a execução do próximo passo.

```
:- module(persistence_migration_transformation, []).

:- multifile(user:ct/3).

%CallId - Chamada a uma persistência que não pertence à classe de negócio
%Business - Classe de negócio onde ocorre a chamada
%BusinessTarget - Classe de negócio para onde foi a persistência chamada
user:ct( replaceCalls(CallId, Business, BusinessTarget),
    (
        %cria a variável do negócio a ser chamado
        classT(BusinessTarget, _, NameBusinessTarget, _, _),
        implementsT(_, BusinessTarget, BusinessTargetInterface),
        fieldT(FieldEJB, Business, BusinessTargetInterface, NameBusinessTarget, null),
        new_id(NewGetFieldEJB)
    ),
    (
        %insere a variável no código
        add(fieldAccessT(NewGetFieldEJB,_,_,_,FieldEJB,_)),
        %substitui a chamada da persistência para o negócio
        replace(callT(CallId, Parent, Encl, _, Args, Method, TypeParams, Type),
            callT(CallId, Parent, Encl, NewGetFieldEJB, Args, Method, TypeParams, Type))
    )
).
```

Figura 5.6 – Parte do script de refatoração das classes de negócio

5.3.2 Clusterização

Seguindo o procedimento detalhado na Seção 4.2.2, foi criado o grafo do sistema SIGA-EPCT, através da ferramenta JCluster (Seção 5.2.1.1), que representa as classes de negócio e seus relacionamentos, sendo possível visualizar parte dele na Figura 5.7. Nessa figura cada conjunto de cores representa um possível componente a ser gerado, sendo que a coloração seguiu o Algoritmo 2 da Seção 5.2.1.1.

Na Figura 5.7 é possível ver a existência de alguns grupos de vértices isolados, porém verifica-se que em grande parte eles estão inter-relacionados (dentro da marcação). Para esse caso, e mesmo para os grupos menores, pode-se executar o algoritmo de "clusterização"*fast community*, descrito na Seção 4.1.2, gerando um dendrograma para tentar identificar uma melhor divisão das classes. O algoritmo não é executado sobre todo o sistema de uma vez, sendo necessário que o engenheiro de software escolha cada conjunto de classes que deseja tratar. A Figura 5.8 mostra um zoom maior nas classes presentes dentro da marcação, para melhor visualização.

O dendrograma da Figura 5.9 representa a "clusterização"dos vértices contidos dentro da marcação da Figura 5.7 e é gerado ao se clicar em qualquer um desses vértices.

Nessa imagem é possível verificar a modularidade em cada iteração e sua variação, conforme explicado na Seção 4.1.2. Com base nas informações disponibilizadas, o engenheiro de software pode selecionar um dos pontos de junção e visualizar a quantidade de componentes que esse ponto de corte irá gerar, através das cores apresentadas (Figura 5.9).

Ao escolher um ponto de corte, as cores do dendrograma são repassadas para o grafo. Apesar do Engenheiro de Software ter liberdade na escolha do ponto de corte, Girvan e Newman (2003) sugere que a modularidade do ponto escolhido seja igual ou superior a 0,6, conforme explicado na Seção 4.2.2. O engenheiro deverá fazer esse procedimento em todos os grupos de classes nos quais deseja diminuir seu tamanho.

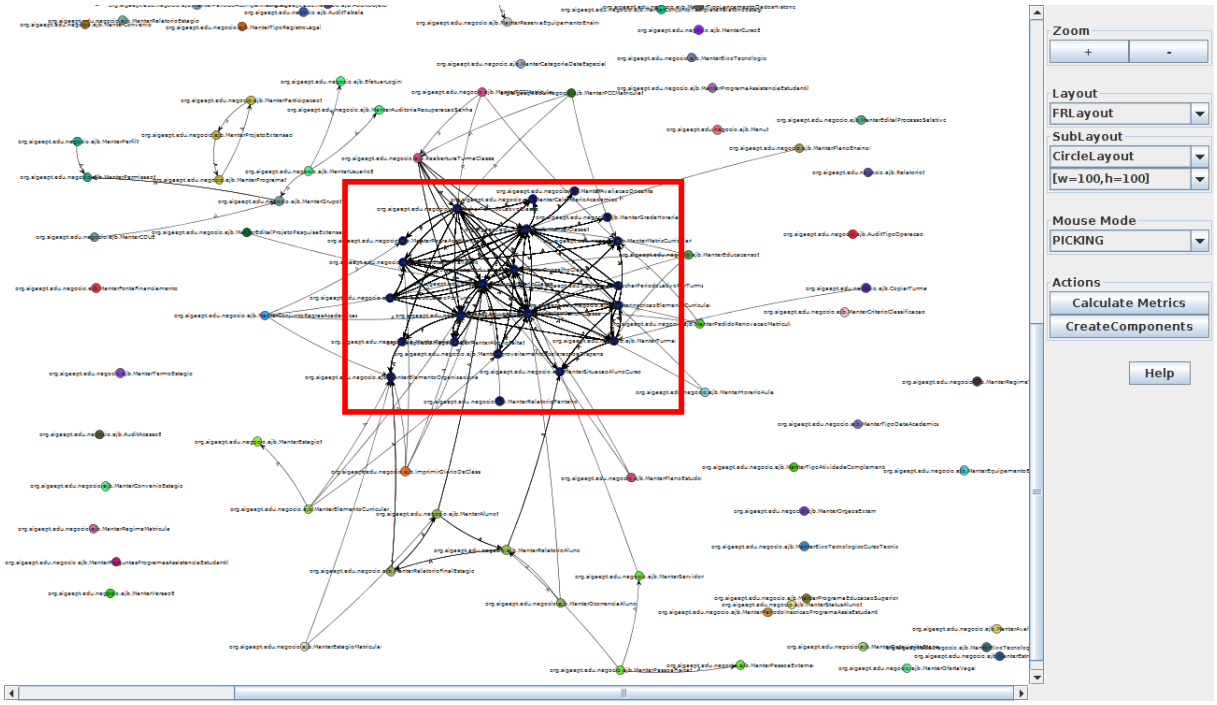


Figura 5.7 – Grafo representando o sistema SIGA-EPCT

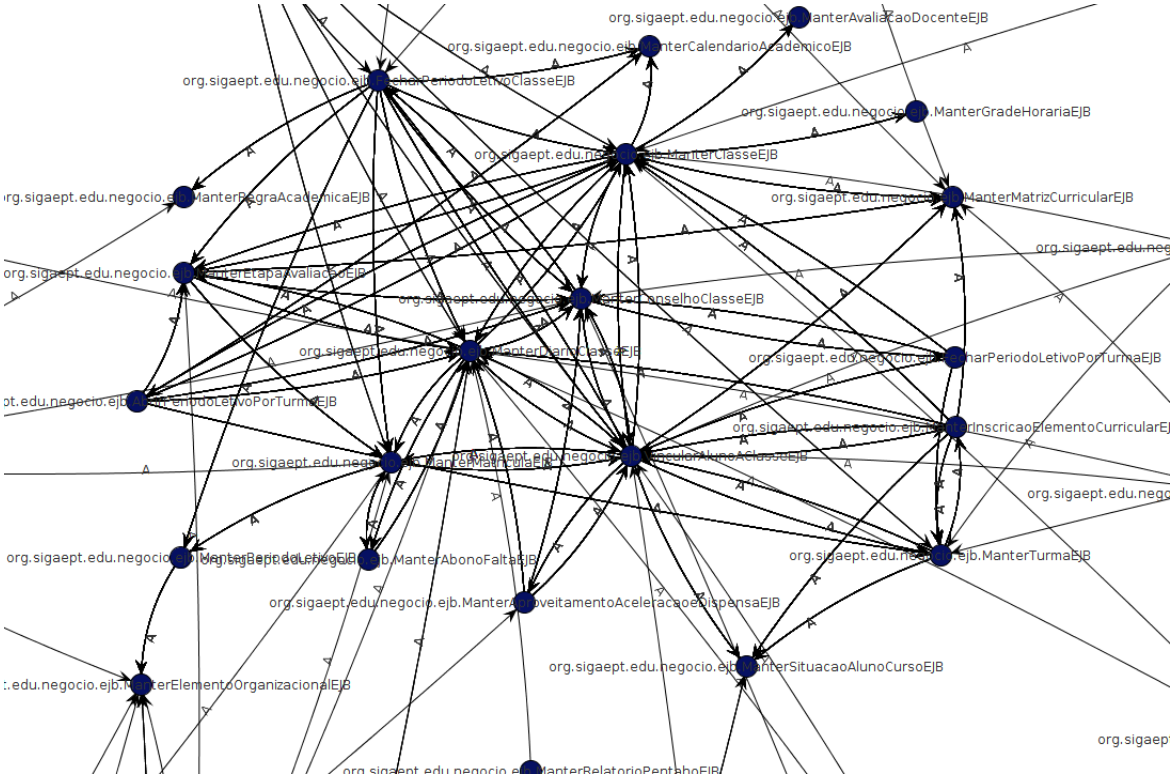


Figura 5.8 – Zoom do grafo representando o sistema SIGA-EPCT

Após o término das clusterizações, JCluster (Seção 5.2.1.1) cria diretórios separados para cada grupo de cores existentes no grafo e move para elas cada conjunto que possuem a mesma cor, levando além das classes de negócio, suas persistências, e copiando as entidades, interfaces e superclasses utilizadas. Cada pasta conterá todas as classes necessárias para criação de um projeto que irá criar e disponibilizar os serviços, sendo necessária a verificação das bibliotecas extras que devem ser inseridas ou excluídas.

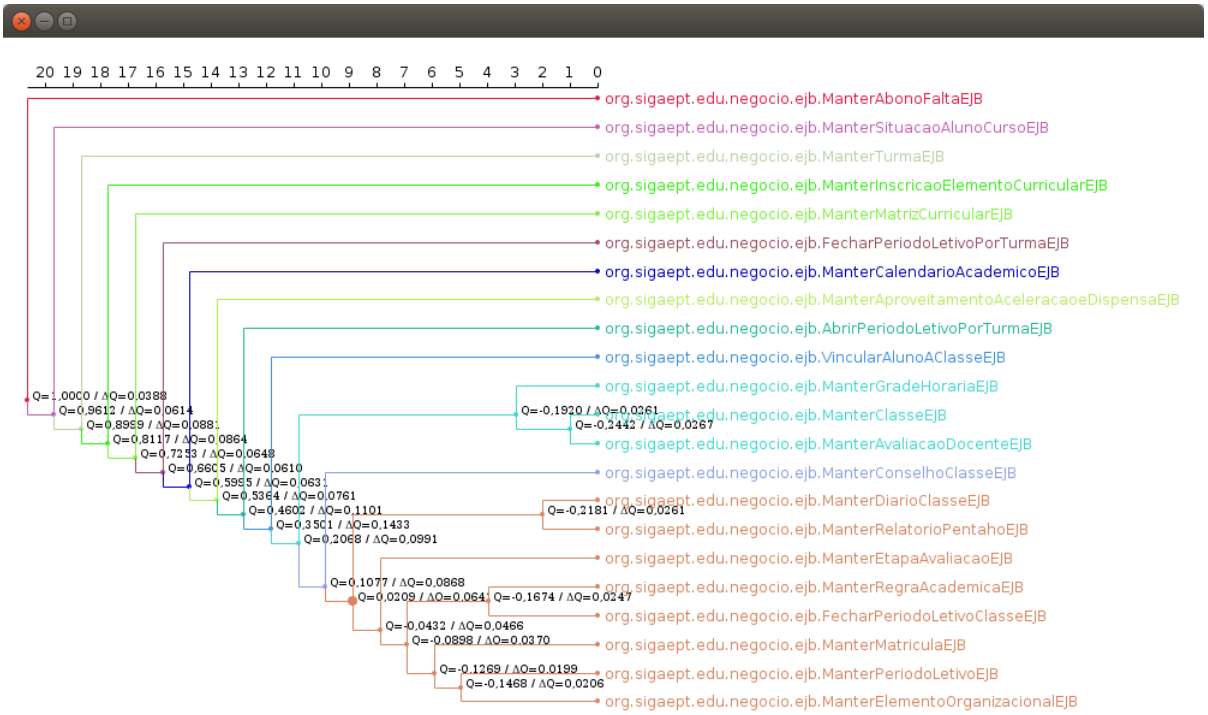


Figura 5.9 – Clusterização das classes

5.3.3 Criação dos serviços

Conforme explicado na Seção 4.2.3, não será apresentada técnica de automatização para criação e disponibilização dos serviços, mas será exemplificado, dentro do estudo de caso, as questões abordadas na referida seção.

O primeiro passo é a disponibilização dos métodos como serviços. Como foi definido que são as classes de negócio que irão disponibilizar seus métodos, deve-se incluir a anotação **@javax.jws.WebService** nessas classes. Essa anotação é responsável por definir uma classe como *web service* e todos os métodos públicos existentes poderão ser acessados. Apesar de não ser obrigatório, é recomendado a utilização da anotação **@javax.jws.WebMethod** nos métodos a serem disponibilizados (KALIM, 2013). Um exemplo de uma classe do sistema SIGA-EPCT contendo essas anotações pode ser visualizada na Figura 5.10.

Com os *web services* criados, é necessário verificar a existência de métodos com mesmo nome, mesmo tendo assinaturas diferentes, conforme restrições apresentadas na Seção 4.2.3. É possível agilizar a identificação desses métodos com a utilização do plugin JTransformer, sendo necessário a criação de um script para isso, conforme pode ser visto na Figura 5.11.

Uma vez identificados os métodos de mesmo nome, fica a cargo do engenheiro de software alterar o nome do método, ou o nome a ser disponibilizado como serviço, incluindo o atributo **operationName** à anotação **@javax.jws.WebMethod**. A Figura

```java
@WebService
public class ManterStatusAlunoEJB implements IManterStatusAlunoEJB {

    @PersistenceContext(unitName = "siga")
    private EntityManager em;

    @WebMethod
    public List<TipoStatusAluno> consultarTodosTipoStatusAluno() throws NegocioException {
        return new TipoStatusAlunoDAO(this.em).consultarTodos();
    }

}
```

Figura 5.10 – Criação de web service

```
:- module(duplicated_method_names_analysis, [ duplicated_method_names_finder/2 ]).

duplicated_method_names_finder(MethodId, MethodIdAux) :-
    %filtrar somente as classes de negócio
    packageT(Package, 'org.sigaept.edu.negocio.ejb'),
    compilationUnitT(CompilationUnit, Package, _, _, [ClassId]),
    classT(ClassId, CompilationUnit, _, _, _),
    %compara os métodos dentro de uma mesma classe
    methodT(MethodId, ClassId, MethodName, _, _, _, _, _),
    methodT(MethodIdAux, ClassId, MethodNameAux, _, _, _, _, _),
    %ignora a comparação de um método com ele mesmo
    MethodId \== MethodIdAux,
    %compara método com mesmo nome
    MethodName == MethodNameAux.
```

Figura 5.11 – Script para encontrar nomes de métodos duplicados

5.12 mostra ambas refatorações, que produzirão o mesmo resultado, feitas na classe **ManterAbonoFaltaEJB**, que possui dois métodos com o nome **pesquisaSimples**, mas assinaturas diferentes.

Tratados os nomes dos métodos, é necessário refatorar as chamadas às classes que agora pertencem a outro componente e, consequentemente, a outro *web service*. Essa refatoração pode ser visualizada no trecho código da Figura 5.13. Uma explicação mais detalhada sobre utilização de web services pode ser obtida em Oracle (2016).

Feito os passos citados, resta ainda verificar a necessidade de controle de transação dos métodos, sendo necessária a análise individual de cada método pelo engenheiro de software. Apesar de existir as transações atômicas e de negócio, dentro do sistema SIGA-EPCT, tem-se a necessidade de utilizar apenas a primeira, pois não existe nenhuma operação de longa duração. A diferença entre esses tipos de transação é explicado na Seção 4.2.3.

O primeiro passo para habilitar a transação é adicionar a anotação **@com.sun.xml.ws.api.tx.at.Transactional** na classe ou método. Ao inserir a anotação na classe, todos

Alteração do nome do método

```
@WebService
public class ManterAbonoFaltaEJB{
    ...
    @WebMethod
    public List<AbonoFalta> pesquisaSimples(...) {
        ...
    }

    @WebMethod
    public List<AbonoFalta> pesquisaSimples2(...){
        ...
    }
    ...
}
```

Alteração do nome do serviço

```
@WebService
public class ManterAbonoFaltaEJB{
    ...
    @WebMethod
    public List<AbonoFalta> pesquisaSimples(...) {
        ...
    }

    @WebMethod(operationName="pesquisaSimples2")
    public List<AbonoFalta> pesquisaSimples(...) {
        ...
    }
    ...
}
```

Figura 5.12 – Mudança dos nomes dos métodos dos web services

Código Original

```
public class ManterEducacensoEJB{
    @EJB
    private IManterTurmaEJB ManterTurmaEJB;

    ...
    private List<RegistrosComposto>
     ↪   getDadosProfissionalDocencia(Docente
     ↪   docente) {
        ...
        List<Turma> listaTurma = ManterTurmaEJB
        .consultarTodosTurmasPorDocente(docente);
        ...
    }
    ...
}
```

Código Refatorado

```
public class ManterEducacensoEJB {
    ManterTurmaEJBService manterTurmaEJBService;

    ...
    private List<RegistrosComposto>
     ↪   getDadosProfissionalDocencia(Docente
     ↪   docente) {
        ...
        List<Turma> listaTurma =
        manterTurmaEJBService
        .getManterTurmaEJBPort()
        .consultarTodosTurmasPorDocente(docente)
        ...
    }
    ...
}
```

Figura 5.13 – Refatoração para chamada a *web services*

os métodos dela seguirão as mesmas configurações. Adicionando diretamente nos métodos, é possível definir configurações diferentes para uma mesma classe. Oracle (2016) descreve as opções de configuração existentes para essa anotação:

- **Versão:** versão do contexto da coordenação de transação atômica utilizado pelo *web service* e seus clientes. A versão especificada deve ser a mesmo através da transação inteira, sendo os seguintes valores possíveis: WSAT10, WSAT11, WSAT12 e DEFAULT
- **Tipo do fluxo:** indica se o contexto da coordenação de transação atômica será passado adiante durante o fluxo da transação. Seus possíveis valores podem ser encontrados na Tabela 5.5.

O código final de um *web service* da classe **ManterTurmaEJB**, contendo as anotações descritas acima pode ser vista na Figura 5.14.

Tabela 5.5 – Valores de tipo de fluxo existentes (ORACLE, 2016).

Valor	**Cliente web service**	**Web service**
NEVER	Não exporta o contexto da transação mesmo que possua uma transação	Não importa o contexto da transação mesmo que já exista um fluxo de transação
SUPPORTS (Valor padrão)	Exporta o contexto de transação somente se já existir uma transação	Importa o contexto da transação somente se já existir um fluxo de transação
MANDATORY	Se não houver uma transação a ser exportada, um erro é reportado	Se não houver um contexto de transação a ser importado, um erro é reportado

```
@WebService
@Transactional(value=Transactional.TransactionFlowType.SUPPORTS, version=Transactional.Version.DEFAULT)
public class ManterTurmaEJB extends GenericCrudEJB<Turma, TurmaDAO> implements IManterTurmaEJB {
    ...
    @WebMethod
    public List<Turma> consultarTodosTurmasPorDocente(Docente docente) {
        return new TurmaDAO(em).consultarTodosTurmasPorDocente(docente);
    }
    ...
}
```

Figura 5.14 – *Web service* com controle de transação

Seguindo as orientações apresentadas, é possível disponibilizar os *web services* referentes a camada de negócio de cada componente identificado na seção anterior. É importante lembrar que embora não abordada por esta pesquisa, ainda será necessário a refatoração da camada de apresentação do sistema, de forma a usar os serviços criados, ficando esta tarefa para um trabalho futuro.

5.4 Análise e Resultados

Nesta seção serão analisadas os valores das métricas calculadas após a execução da abordagem, utilizando-as como base para avaliar se o objetivo de eficácia definido anteriormente foi atingido.

5.4.1 Questão 1: Que tipo de melhoria a refatoração das classes traz?

Com a execução da etapa de diminuição das dependências, foi possível observar a redução geral do acoplamento das classes através da Medida ACCL da Tabela 5.1. A Tabela 5.6 mostra a quantidade média de dependências das classes, antes e depois das refatorações, dividindo os resultados por camadas. Quando observada somente a camada de negócio, o aumento do acoplamento encontrado ocorre porque originalmente não havia muitas dependências entre classes de negócio, mas sim a reutilização do mesmo método

da mesma classe de persistência. Apesar disso, ainda é possível observar uma diminuição média de 24% no acoplamento (métrica M1 da Tabela 5.1) existentes nas classes.

Tabela 5.6 – Cálculo da quantidade média de dependências das classes por camada

	Código Original	**Código Refatorado**
Negócio - Negócio	0,08	1,05
Negócio - Persistência	4,34	2,30
Geral	4,42	3,36

Um exemplo que demonstra essa realidade é apresentado pela Figura 5.15, mostrando que originalmente a classe **CopiarTurmaEJB** fazia uso de três classes de persistência, **PeriodoLetivoDAO**, **CursoDAO**, **TurmaDAO**, e depois da refatoração passou a referenciar apenas uma única classe de negócio **ManterTurmaEJB**. Essa figura também mostra a remoção de métodos que deixaram de ser relevantes, pois passaram a existir na classe de negócio **ManterTurmaEJB**.

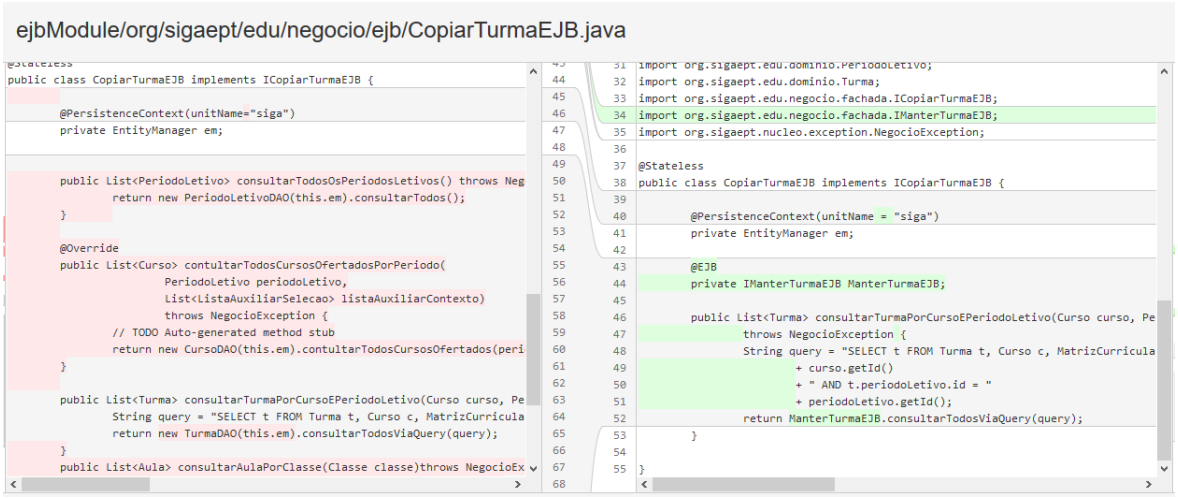


Figura 5.15 – Refatoração da classe CopiarTurmaEJB

5.4.2 Questão 2: Quais foram as melhorias obtidas nos componentes gerados?

Com a técnica de "clusterização", responsável pela divisão dos componentes, conseguiu-se uma diminuição da complexidade (medida COXP) e do tamanho dos componentes gerados (medida TACO), podendo ser facilmente verificado pela redução da quantidade de classes em cada componente. Porém ao se dividir um componente, provavelmente irão ser criados outros com dependências entre si (medida ACCO), já que originalmente eram um só componente. Entretanto, como o ponto de corte, responsável pela divisão do componente, é de livre escolha do engenheiro de software, essa redução pode variar significativamente.

Para exemplificar, se o algoritmo de "clusterização"for executado no maior componente do sistema SIGA-EPCT (cor azul da Figura 5.7), criado automaticamente a partir dos relacionamentos das classes, teremos valores diferentes para as métricas citadas acima.

A Tabela 5.7 mostra essa variação em relação as medidas definidas na Tabela 5.1, sendo que os dados dessa tabela refletem ao componente e dendrograma das Figuras 5.8 e 5.9, respectivamente, com exceção da medida COXP que é relacionado a complexidade média de todo o sistema. As colunas da tabela 5.7 representa o componente original e quatro conjuntos de componentes gerados após a escolha de quatro dos possíveis pontos de corte do dendrograma, todos com modularidade superior a 0,6, conforme sugerido por Girvan e Newman (2003). Para cada um desses pontos de corte, as linhas da tabela representam:

- **Modularidade:** valor da modularidade no ponto de corte;
- **Qtde de clusters:** quantidade de componentes que o cluster original irá derivar;
- **TACO**: quantidade média de classes pertencentes a cada componente derivado;
- **ACCO**: acoplamento médio entre os componentes derivados;

Tabela 5.7 – Variação dos valores das métricas de acordo com ponto de corte da clusterização

		Após Clusterização			
	Original	**Ponto de Corte 1**	**Ponto de Corte 2**	**Ponto de Corte 3**	**Ponto de Corte 4**
Modularidade	1	0,9612	0,8999	0,8117	0,7253
Qtde de componentes	1	2	3	4	5
COXP	14,20	13,96	13,73	13,51	13,30
TACO	22	11	7,33	5,5	4,4
ACCO	0	0,50	1,33	1,50	1,80

Com base nas medidas da Tabela 5.7 é possível calcular as métricas M2, M3 e M4, também calculadas a partir dos pontos de corte definidos nessa tabela (1 ao 4). Temos então:

- **M2:** redução variou entre 50% e 80%
- **M3:** redução variou entre 1,70% e 6,33%
- **M4:** aumento variou entre 166% e 260%

Para a métrica M4 foi utilizado como base o valor de ACCO calculado para o ponto de corte 1, já que originalmente não existe dependência com outro componente (ACCO = 0). Essas dependências passam a existir após a divisão do componente original, que gera componentes menores que se relacionam.

5.4.3 Questão 3: A funcionalidade original é mantida após a criação dos serviços?

Conforme dito anteriormente, as alterações propostas não alteram os algoritmos do sistema, de forma que as mudanças são feitas apenas nas chamadas e/ou nomes e não nos corpos dos métodos, conforme explicitado nas Figuras 5.5 e 5.13.

Para o cálculo da métrica M3 foram criados 5 casos de teste[7] nos quais foram executados métodos do sistema antes e depois da modernização. A Tabela 5.8 mostra o resultado da execução dos testes, sendo que a coluna **Classe** representa a classe original do sistema, **Serviço** representa o *web service* criado a partir da classe, **Método** é o método tanto da classe quanto do *web service* que foi executado, **Ind.** informa se o *web service* é independente, ou seja, contém todo o código dentro de si ou se depende de outro serviço e a coluna **Resultado** mostra o resultado dos testes.

Tabela 5.8 – Resultado dos casos de teste

Classe	Serviço	Método	Ind.	Resultado
ManterCursoEJB	ManterCursoService	consultarTodosEtapaEnsino	não	sucesso
ManterTCCMatriculaEJB	ManterTCCMatriculaService	remover	não	sucesso
ManterTCCMatriculaEJB	ManterTCCMatriculaService	consultarTodosTCCs	sim	sucesso
ManterMatrizCurricularEJB	ManterMatrizCurricularService	consultarTodos	não	sucesso*
ReaberturaTurmaClasseEJB	ReaberturaTurmaClasseService	reabrirTurma	sim	sucesso*

Em todos os casos de teste foram obtidos sucesso nas comparações, indicando que a funcionalidade continuou trazendo os mesmos resultados. Os asteriscos (*) nos dois últimos resultados da Tabela 5.8 são para informar que foi necessário uma refatoração antes de sua execução. O motivo foi pelo fato de as entidades utilizadas como parâmetros formarem um ciclo a partir de seus atributos, que geram erro no momento da montagem dos XMLs para troca de mensagens, sendo necessário retirar os ciclos encontrados. Um exemplo de ciclo encontrado pode ser verificado pelas linhas destacadas dos trechos de códigos mostrados na Figura 5.16, onde para esse caso foi retirado o atributo **private List<ProjetoPedagogicoCurso> projetosPedagogicosCurso** da classe **Curso**.

```
public class Curso extends GenericEntidadeId {
    @CampoUnico(descricao="Código")
    @Column(name="codigo")
    private String codigo;

    ...

    @OneToMany(mappedBy="curso")
    private List<ProjetoPedagogicoCurso>
     ↪   projetosPedagogicosCurso;

    ...
}
```

```
public class ProjetoPedagogicoCurso extends
 ↪   GenericEntidadeId{

    ...

    @Column(name="nome_arquivo")
    private String nomeArquivo;

    @ManyToOne
    @JoinColumn(name = "curso_id")
    private Curso curso;

    ...
}
```

Figura 5.16 – Exemplo de ciclo

[7] O ambiente de teste criado pode ser acessado através do link <https://github.com/aborgesrodrigues/ambiente_teste_siga>.

Apesar de a alteração para retirada dos ciclos das entidades não ter uma complexidade alta, pode afetar várias partes do sistema, onde é necessário substituir as chamadas aos atributos retirados por novos métodos de consulta para trazer os mesmos dados.

5.5 Discussão

Com base nas métricas calculadas para as questões definidas anteriormente pode-se observar a eficácia da proposta de modernização, sendo esse o objetivo principal deste estudo de caso. Uma das principais características está na redução do tamanho dos *web services* gerados. Isso é feito tanto na primeira etapa, diminuição de dependências, que reduz o acoplamento entre as classes, quanto na etapa de "clusterização", que permite a quebra dos grupos de classes. Outro ponto é a inclusão do controle de transações entre serviços, que ajuda a garantir que as funcionalidades originais continuarão funcionando da mesma forma.

Apesar dos benefícios obtidos, existem alguns pontos críticos a serem observados nesta pesquisa. O primeiro é relacionados ao uso da fórmula da força de conectividade (Seção 4.1.1) para a diminuição do acoplamento das classes, e o segundo na utilização do algoritmo *fast community* na "clusterização"das classes(Seção 4.1.2). Apesar de serem técnicas sobre as quais já foram comprovadas suas eficácias, não é possível assegurar que são as melhores opções existentes para subsidiar essas operações, pois embora existam várias abordagens que podem ser utilizadas para atingir o mesmo objetivo, faltam trabalhos que os comparem de forma a auxiliar na escolha. Devido a complexidade existente em realizar essas comparações, essa atividade não foi incluída no escopo desse trabalho.

Outro ponto crítico está relacionado aos testes. Não foi utilizado nenhum framework para a execução dos testes da Seção 5.4.3, mas criou-se um ambiente integrado com o sistema original e com os *web services* gerados, onde foi possível executar ambos e comparar os resultados. A não utilização de um framework de testes como Junit[8] ou Arquillian[9] foi por causa de problemas técnicos encontrados, como a chamada de classes EJB (*Enterprise JavaBeans*) de dentro dos casos de teste. Também não foi testado a performance dos *web services* criados, ficando a cargo do engenheiro de software definir uma forma de fazer tais validações, além de realizar um conjunto de testes que abranjam mais áreas do sistema.

Apesar da abordagem proposta não exigir que o Engenheiro de Software possua um profundo conhecimento sobre sistema a ser modernizado, o código sistema SIGA-EPCT, do estudo de caso foi conseguido pelo fato do autor pertencer a equipe de desenvolvimento, o que prejudica essa avaliação. Apesar disso, a utilização do sistema SIGA-EPCT como estudo se caso foi devido a dificuldade de se conseguir acesso ao código fonte de sistemas

[8]Site do Junit: <http://junit.org/junit4/>
[9]Site do Arquillian: <http://arquillian.org/>

razoavelmente complexos.

A metodologia proposta não tem a pretensão de gerar uma arquitetura ideal de serviços ao final do processo, mas garantir que esse novo código seja funcional, mantenha as mesmas funcionalidades do sistema original e facilite a manutenção dos serviços disponibilizados. Um importante resultado a ser obtido é permitir ao engenheiro de software, a partir do código refatorado, analisar pontualmente a performance de cada serviço, identificando possíveis gargalos. A partir de uma análise mais restrita, e não da totalidade do sistema, é possível fazer novas refatorações que se julgarem necessárias, podendo alterar a arquitetura ou até mesmo a linguagem utilizada em cada serviço sem afetar os demais *web services*, desde que não sejam alteradas as assinaturas dos métodos disponibilizados.

5.6 Resumo do Capítulo

Neste capítulo a abordagem de modernização proposta foi executada no sistema SIGA-EPCT como forma de um estudo de caso. Para guiar essa execução foi utilizado o paradigma GQM, com o objetivo de mostrar a eficácia da abordagem, que de acordo com as métricas estabelecidas, foi alcançado.

No próximo capítulo serão apresentados as considerações finais da abordagem de modernização proposta, apresentando suas principais contribuições, as oportunidades de trabalhos futuros que permite, e os trabalhos correlatos existentes.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esta dissertação apresentou uma abordagem para modernização de sistemas monolíticos, orientados a objetos, e desenvolvidos com uma arquitetura de três camadas, para SOA. Diferentes técnicas foram estudadas e aplicadas, como cálculo da Força de Conectividade, "Clusterização"e Refatorações, que dão suporte à Reengenharia desses sistemas.

6.1 Trabalhos correlatos

Conforme observado na Seção 2.2 existem várias famílias de abordagens existentes na literatura para modernizar sistemas legados para SOA. Contudo, nesta seção serão apresentadas as abordagens referentes à mesma família da apresentada nesta dissertação (Família da identificação de serviço), focando naquelas que utilizam sistemas orientados a objetos como origem e que definem técnicas de semi-automatização do processo.

O primeiro trabalho é o Eunjoo Lee et al. (2003), que define a fórmula para o cálculo da força de conectividade, utilizada na primeira etapa da abordagem proposta nesta dissertação. Wang et al. (2008) utiliza o trabalho anterior como base e aprimora o cálculo da força de conectividade, com a utilização de pesos diferentes para atributos primitivos e complexos. Com base na força de conectividade é feita a "clusterização"hierárquica das classes, que gerará um dendrograma com os agrupamentos realizados em cada iteração, sendo que os novos valores da FC encontrados em cada iteração serão utilizados para definir o ponto de corte. Na abordagem desta dissertação optou-se pela utilização do algoritmo *fast community* para a "clusterização". Embora os trabalhos de Eunjoo Lee et al. (2003) e Wang et al. (2008) tenham como objetivo a identificação de componentes e não serviços, sua comparação é válida, pois o foco deles também é na identificação de grupos de classes.

Budhkar e Gopal (2012) utiliza a similaridade existente entre as classes para realizar um algoritmo de "clusterização"hierárquico, utilizando-se de um *threshold*, definido pelo engenheiro de software, como ponto de encerramento nas iterações de agrupamento. Apesar de não apresentar a fórmula para o cálculo da similaridade, deixa claro que é baseado nos tipos de relacionamentos existentes entre as classes, como herança, composição, execução de métodos, etc.

Adjoyan, Seriai e Shatnawi (2014) define outra função para calcular a ligação

entre as classes, a *fitness function (FF)*, que leva em conta os cálculos de *functionality*, *composability* e *self- containment*. Neste trabalho também é utilizado um algoritmo de "clusterização"hierárquico com base nos valores das FF obtidos. Para definir o ponto de corte é utilizado o algoritmo *depth first search (DFS)*, sendo que inicialmente, no nó raiz, é comparado a similaridade do nó corrente com a similaridade dos nós filhos, sendo que quando a similaridade do nó corrente for maior que a média da similaridade dos nós filhos, este nó será o ponto de corte.

Tanto o agrupamento das classes quanto a clusterização delas se assemelham nas propostas apresentadas acima, inclusive em relação a proposta desta pesquisa. Todas essas abordagens, apesar de definirem critérios distintos, agrupam as classes através destes critérios e utilizam algoritmos de "clusterização"hierárquica para o agrupamento dessas classes. Essas clusterizações são todas feitas de forma hierárquica, sendo que algumas geram grafos, criam dendrogramas e escolhem um ponto de corte (Eunjoo Lee et al., 2003; WANG et al., 2008; ADJOYAN; SERIAI; SHATNAWI, 2014), ou agrupam hierarquicamente as classes, analisando o valor que esse agrupamento gera até que se atinja um ponto de parada, ou *treshold*, que assemelha-se ao uso do ponto de corte (BUDHKAR; GOPAL, 2012).

Uma das vantagens da abordagem de modernização proposta em relação as apresentadas anteriormente, são as refatorações definidas com o intuito de diminuir o tamanho dos serviços a serem criados, enquanto as demais não possuem esta etapa. Outra vantagem é que a abordagem desta pesquisa leva em consideração a existência de uma arquitetura de camadas no sistema legado a ser modernizado, enquanto as demais abordagens desconsideram esse fato, podendo causar agrupamentos de classes inviáveis de se manter, como entidades e persistência sem negócio, ou negócio e persistência sem entidades, tornando-as difíceis de se aplicar nos sistemas com esse tipo de arquitetura.

6.2 Contribuições

A principal contribuição é a definição de uma abordagem para a modernização para SOA de sistemas monolíticos, orientados a objetos e que também possuam uma arquitetura em camadas, através das etapas e técnicas semi-automatizadas que foram definidas nela.

A primeira etapa dessa abordagem, diminuição das dependências, contribui com técnicas que permitem a semi-automatização de refatorações que reduzirá a quantidade de dependências que cada classe possui, em relação a outras classes, tanto na camada de persistência quanto na de negócio, de modo a permitir um menor acoplamento nos serviços a serem gerados.

A segunda etapa, "clusterização", também apresenta técnicas que possibilitam

a semi-automatização da identificação dos grupos de classes de negócio, que poderão disponibilizar seus métodos como serviços. Além disso, também é apresentada técnicas que permitem a divisão desses grupos, caso seja necessário, devido à quantidade de classes que possuíram em um primeiro momento.

Apesar da abordagem não ser direcionada a uma linguagem de programação específica, foi criado o plugin JCluster que implementa as técnicas das duas primeiras etapas da abordagem, semi-automatizando o processo para os sistemas desenvolvidos na linguagem Java. A descrição da utilização do plugin JTransformer para a análise e refatoração de código contribui para elucidar seu funcionamento, que pode ser utilizado das mais diversas formas.

A terceira etapa, criação de serviços, apesar de não descrever nenhuma técnica para a criação dos *web services*, contribui com a descrição de como eles devem ser criados, e as características que eles devem possuir.

6.3 Limitações e trabalhos futuros

A abordagem de modernização proposta abre espaço para novas contribuições que poderão complementá-las.

Como explicitado anteriormente, a abordagem de modernização definida não abrange a adequação da camada de apresentação, que necessitará ser refatorada para que possa usufruir dos *web services* gerados. Essas refatorações são bastante complexas e trabalhosas e precisam de técnicas e/ou ferramentas que permitam a automatização ou semi-automatização do trabalho. Uma solução seria a construção de uma ferramenta para auxiliar essas refatorações, além de identificar os pontos a serem refatorados. Para deixá-la independente de linguagem será necessário que essa ferramenta permita a inserção de padrões das chamadas aos *web services* para as diversas linguagens, de acordo com a necessidade do Engenheiro de Software. Essa ferramente também pode ser utilizada para refatorar as chamadas entre os *web services*, após suas criações, na etapa de criação de serviços.

Outro ponto não abordado é a separação do banco de dados por serviço. Essa separação permitiria que cada serviço possuísse uma base de dados própria. Como as entidades são a representação do banco de dados, estas também provavelmente serão afetadas, de forma que cada *web service* terá seu próprio conjunto exclusivo de entidades. Para realizar essa tarefa será necessário descobrir quais tabelas são acessadas por quais *web services*, montando um mapeamento. Além disso, será necessário a exclusão das chaves estrangeiras entre tabelas de *web services* diferentes, sendo necessário a inclusão dessa *constraint* dentro dos *web services*. Deverão ser refatoradas as entidades, retirando os atributos que referenciam entidades de outros *web services*, além de refatorar os scripts

SQL (ou HQL), que utilizem essas referências que foram removidas.

Na análise dos relacionamentos entre classes de persistência, da etapa de diminuição de dependências, não foi definido uma técnica para automatizar as refatorações necessárias. Apesar de ser possível agilizar a identificação dessas classes através de ferramentas como a JTransformer, a migração desse tipo de chamada para a camada de negócio, conforme exemplificado pela Figura 5.4, exige conhecimento do sistema por parte do Engenheiro de Software responsável pela modernização. Seria necessário a criação/utilização de uma ferramenta que agilizasse tal refatoração, permitindo ao Engenheiro de Software interagir com essa ferramenta para fazer as devidas configurações, como a escolha da classe de negócio que receberá o código removido da persistência.

Conforme apresentado na Seção 5.5, não só este trabalho mas, a literatura em geral sobre modernizações para SOA, carece de estudos de comparação das técnicas utilizadas tanto para calcular a força de ligação entre as classes, quanto para realizar a "clusterização"das mesmas. Para analisar as técnicas que possibilitam o agrupamento de classes pode-se utilizar a técnica MoJo (TZERPOS; HOLT, 1999), que é uma métrica que pode ser usada para avaliar a similaridade entre decomposições de um sistema. Uma forma de avaliar os modelos de "clusterização"é a utilizada por Erdemir e Buzluca (2014), que avalia as abordagens sobre os critérios de *authoritativeness*, *stability* e *extremity of cluster distribution*.

Pelo fato desta pesquisa focar nos critérios a serem observados na criação de *web services*, mas não estabelecer técnicas para criá-los de fato, os testes destes *web services* foram retirados do escopo da pesquisa, mas pode ser incorporado em um trabalho futuro. Um importante tipo de teste seria o teste de regressão dos *web services*, comparando os resultados do sistema monolítico original com o SOA refatorado, através de um conjunto amplo de casos de teste e uso de ferramentas para auxiliar. Outros testes também podem ser feitos como teste de performance, que analisa o comportamento dos *web services* através de métricas e valores de referência pré-definidos, e o teste de stress, que verifica volume de dados que *web services* consegue atender. O uso de ferramentas é fundamental na execução de testes, como por exemplo as ferramentas SOAPUI [1], TestingWhiz [2] e SOAtest [3].

O plugin JCluster está disponível para atualizações, podendo melhorar a distribuição dos vértices (classes) no grafo gerado por ele, de forma a facilitar a visualização. Se for definida técnicas específicas, é possível adicionar a funcionalidade de criação dos *web services* e a refatoração das chamadas.

[1]Disponível em <https://www.soapui.org>
[2]Disponível em <http://www.testing-whiz.com/>
[3]Disponível em <https://www.parasoft.com/product/soatest/>

REFERÊNCIAS

ADJOYAN, S.; SERIAI, A.-D.; SHATNAWI, A. Service Identification Based on Quality Metrics. *Proceedings of the 26 International Conference on Software Engineering & Knowledge Engineering (SEKE2014)*, p. 1–6, 2014. Citado 4 vezes nas páginas 30, 32, 76 e 77.

ALVES, T. L.; HAGE, J.; RADEMAKER, P. A comparative study of code query technologies. In: *Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011*. [S.l.: s.n.], 2011. p. 145–154. ISBN 9780769543475. Citado 3 vezes nas páginas 11, 56 e 57.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. *Encyclopedia of Software Engineering*, v. 2, p. 528–532, 1994. Citado na página 52.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice (3rd Edition) (SEI Series in Software Engineering)*. [S.l.: s.n.], 2012. 640 p. ISBN 0321815734. Citado 2 vezes nas páginas 25 e 30.

BINUN, A.; KNIESEL, G. Joining forces for higher precision and recall of design pattern detection. *. . . , Technical report IAI-TR-2012-01*, 2012. Citado na página 56.

BUDHKAR, S.; GOPAL, A. Component-based architecture recovery from object oriented systems using existing dependencies among classes. *International Journal of Computational Intelligence Techniques*, v. 3, n. 1, p. 56–59, 2012. Citado 4 vezes nas páginas 30, 32, 76 e 77.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, v. 7, n. 1, p. 13–17, 1990. ISSN 07407459. Citado 3 vezes nas páginas 9, 20 e 21.

CHO, E. S.; KIM, M. S.; KIM, S. D. Component metrics to measure component quality. *Proceedings Eighth Asia-Pacific Software Engineering Conference*, p. 419–426, 2001. ISSN 1530-1362. Citado 3 vezes nas páginas 11, 53 e 54.

CONNALL, D.; BURNS, D. Reverse Engineering: Getting a Grip on Legacy Systems. *Data Management Review*, v. 24, n. 7, 1993. Citado 2 vezes nas páginas 15 e 19.

CONSTANTINOU, E. et al. Extracting reusable components: A semi-automated approach for complex structures. *Information Processing Letters*, Elsevier B.V., v. 115, n. 3, p. 414–417, 2015. ISSN 00200190. Citado 2 vezes nas páginas 30 e 32.

DAIGNEAU, R. *Service Design Pattern*. [S.l.: s.n.], 2012. XXXIII. 81–87 p. ISSN 0717-6163. ISBN 9780874216561. Citado na página 27.

Dos Santos Brito, K. et al. LIFT - A Legacy information retrieval tool. *Journal of Universal Computer Science*, v. 14, n. 8, p. 1256–1284, 2008. ISSN 0958695X. Disponível em: <http://www.scopus.com/inward/record.url?eid=2-s2.0-45949097336{&}partnerID=40{&}md5=717a584161cfa00c2741519582>. Citado na página 15.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. 2016. Citado 2 vezes nas páginas 15 e 26.

ERDEMIR, U.; BUZLUCA, F. A learning-based module extraction method for object-oriented systems. *Journal of Systems and Software*, Elsevier Inc., v. 97, p. 156–177, 2014. ISSN 01641212. Citado 7 vezes nas páginas 16, 32, 34, 35, 36, 37 e 79.

ERDEMIR, U.; TEKIN, U.; BUZLUCA, F. Object Oriented Software Clustering Based on Community Structure. *Proceedings - 18th Asia-Pacific Software Engineering Conference, APSEC 2011*, p. 315–321, 2011. ISSN 1530-1362. Citado 4 vezes nas páginas 32, 34, 35 e 36.

ERL, T. Service-Oriented Architecture: Concepts, Technology, and Design. *City*, p. 760, 2005. ISSN 0131858580. Citado na página 27.

Eunjoo Lee et al. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In: *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. [S.l.]: IEEE Comput. Soc, 2003. p. 336–341. ISBN 0-7695-2020-0. ISSN 0730-3157. Citado 6 vezes nas páginas 30, 32, 33, 39, 76 e 77.

FIELDING, R. T. Architectural Styles and the Design of Network-based Software Architectures. *Building*, v. 54, p. 162, 2000. ISSN 1098-6596. Disponível em: <http://www.ics.uci.edu/{~}fielding/pubs/dissertation/top.h>. Citado na página 27.

FOWLER, M. J. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420. Citado 2 vezes nas páginas 29 e 39.

FREUND, T.; STOREY, T. Transactions in the world of Web services. *Research Paper, IBM*, 2002. Citado 3 vezes nas páginas 9, 48 e 49.

GIRVAN, M.; NEWMAN, M. E. J. Finding and evaluating community structure in networks. *Cond-Mat/0308217*, p. 1–16, 2003. ISSN 1063651X. Citado 3 vezes nas páginas 45, 65 e 72.

GUO, H. et al. Wrapping client-server application to Web services for Internet computing. *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, v. 2005, p. 366–370, 2005. Citado na página 46.

KALIM, M. *Java Web Services: Up and Running, 2nd Edition*. [S.l.: s.n.], 2013. 359 p. ISBN 9781449365110. Citado 2 vezes nas páginas 27 e 67.

KHADKA, R. et al. A structured legacy to SOA migration process and its evaluation in practice. *c2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, MESOCA 2013*, p. 2–11, 2013. ISSN 2326-6910. Citado na página 37.

KHAN, M. W.; ABBASI, E. Differentiating Parameters for Selecting Simple Object Access Protocol (SOAP) vs . Representational State Transfer (REST) Based Architecture. *Journal of Advances in Computer Networks*, v. 3, n. 1, 2015. ISSN 17938244. Citado na página 27.

KITCHENHAM, B. A.; PICKARD, L. M. Evaluating Software Engineering Methods and Tools Part 9: Quantitative Case Study Methodology. *ACM SIGSOFT Software Engineering Notes*, v. 23, n. 1, p. 24–26, 1998. ISSN 01635948. Citado na página 52.

KNIESEL, G.; HANNEMANN, J.; RHO, T. A comparison of logic-based infrastructures for concern detection and extraction. *Proceedings of the 3rd workshop on Linking aspect technology and evolution*, p. 6, 2007. Citado 4 vezes nas páginas 9, 56, 57 e 59.

LANGWORTHY, D. et al. *Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity*. 2004. Disponível em: <http://msdn.microsoft.com/en-us/library/ms996526.aspx>. Acesso em: 29/01/2017. Citado 4 vezes nas páginas 9, 48, 49 e 50.

LEHMAN, M. M.; BELADY, L. A. *Program evolution: processes of software change*. [S.l.]: Academic Press Professional, Inc., 1985. 538 p. ISBN 0-12-442440-6. Citado na página 19.

LEWIS, G.; MORRIS, E.; SMITH, D. Service-Oriented Migration and Reuse Technique (SMART). *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, n. September, p. 222–229, 2005. Citado na página 37.

Liang Bao et al. Extracting reusable services from legacy object-oriented systems. *2010 IEEE International Conference on Software Maintenance*, p. 1–5, 2010. ISSN 1063-6773. Citado 3 vezes nas páginas 16, 30 e 32.

NEWCOMER, E.; ROBINSON, I. Web Services Atomic Transaction (WS-AtomicTransaction). *Oasis*, n. February, p. 1–28, 2009. Citado na página 48.

NEWCOMER, E.; ROBINSON, I. Web services coordination (WS-Coordination) Version 1.2. *Oasis*, n. February, p. 1–26, 2009. Citado na página 48.

NEWMAN, M. E. J. Fast algorithm for detecting community structure in networks. *Physics*, n. 2, p. 1–5, 2003. Citado 2 vezes nas páginas 34 e 35.

NEWMAN, S. *Building Microservices*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015. 280 p. ISBN 1491950358, 9781491950357. Citado na página 28.

Object Management Group (OMG). Business Process Model and Notation (BPMN) Version 2.0. *Business*, v. 50, n. January, p. 170, 2011. ISSN 13507540. Citado na página 37.

ORACLE. *Metro User Guide*. 2016. Disponível em: <https://metro.java.net/guide/>. Acesso em: 29/01/2017. Citado 4 vezes nas páginas 11, 68, 69 e 70.

RAZAVIAN, M.; LAGO, P. Towards a conceptual framework for legacy to SOA migration. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 6275 LNCS, p. 445–455, 2010. ISSN 03029743. Citado 3 vezes nas páginas 11, 21 e 23.

RAZAVIAN, M.; LAGO, P. A systematic literature review on SOA migration. *Journal of Software: Evolution and Process*, v. 27, n. 5, p. 337–372, 2015. ISSN 20477481. Citado na página 21.

REUTER, A.; GRAY, J. *Transaction Processing: Concepts and Techniques*. [S.l.: s.n.], 1993. Citado na página 47.

RICHARDS, M. *Microservices vs. Service-Oriented Architecture*. [s.n.], 2015. 1–55 p. ISBN 9781491952429. Disponível em: <https://www.nginx.com/microservices-soa/>. Citado na página 28.

ROTEM-GAL-OZ, A. *SOA Patterns*. [S.l.: s.n.], 2012. 296 p. ISBN 9781933988269. Citado na página 25.

SAUDATE, A. *SOA Aplicado Integrando com web services e além*. [S.l.: s.n.], 2013. 293 p. ISBN 9788566250152. Citado 2 vezes nas páginas 9 e 27.

SEACORD, R. C.; PLAKOSH, D.; LEWIS, G. A. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. [S.l.: s.n.], 2003. 332 p. ISSN 08953805. ISBN 0321118847. Citado na página 19.

SHIOKAWA, H.; FUJIWARA, Y.; ONIZUKA, M. Fast Algorithm for Modularity-Based Graph Clustering. *Proceeding of the Twenty-Seventh Conference on Artificial Intelligence*, p. 1170–1176, 2013. Citado 2 vezes nas páginas 35 e 36.

SNEED, H. M. Planning the Reengineering of Legacy Systems. *IEEE Software*, v. 12, n. 1, p. 24–34, 1995. ISSN 07407459. Citado na página 19.

SNELL, J. Automating business processes and transactions in Web services. *Research paper, IBM Emerging Technologies*, p. 4–6, 2002. Citado na página 48.

STOJANOVI, Z. *A Method for Component-Based and Service-Oriented Software Systems Engineering*. [S.l.: s.n.], 2005. ISBN 9090191003. Citado na página 15.

TIOBE.COM. *TIOBE Index for January 2017*. 2017. Disponível em: <http://www.tiobe.com/tiobe-index/>. Acesso em: 28/01/2017. Citado na página 15.

TZERPOS, V.; HOLT, R. MoJo: a distance metric for software clusterings. *Sixth Working Conference on Reverse Engineering*, p. 187–193, 1999. Citado na página 79.

ULRICH, W. From Legacy Systems to Strategic Architectures. *Software Engineering Strategies*, v. 2, n. 1, p. 18–30, 1994. Citado 2 vezes nas páginas 15 e 19.

VILLAMIZAR, M. et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube. *10th Computing Colombian Conference*, p. 583–590, 2015. Citado 3 vezes nas páginas 15, 17 e 26.

VISAGGIO, G. Ageing of a Data Intensive Legacy System: Symptoms and Remedies. *Journal of Soft. Maintenance and Evolution*, v. 13, p. 281–308, 2001. Citado na página 19.

WANG, X. et al. A new approach of component identification based on weighted connectivity strength metrics. *Information Technology Journal*, v. 7, n. 1, p. 56–62, 2008. ISSN 18125638. Citado 6 vezes nas páginas 16, 30, 32, 45, 76 e 77.

WOHLIN, C. et al. *Experimentation in Software Engineering: An Introduction*. [S.l.: s.n.], 2000. xx, 204 p. p. ISBN 0792386825. Citado na página 52.

YOUSEF, R.; ADWAN, O.; ABUSHARIAH, M. A. M. Extracting SOA Candidate Software Services from an Organization's Object Oriented Models. *JSEA - Journal of Software Engineering and Applications*, v. 7, n. August, p. 770–778, 2014. ISSN 1945-3116, 1945-3124. Citado 3 vezes nas páginas 16, 30 e 32.