UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

ANDREY RICARDO PIMENTEL

UMA ABORDAGEM PARA PROJETO DE SOFTWARE ORIENTADO A OBJETOS BASEADO NA TEORIA DE PROJETO AXIOMÁTICO

TESE DE DOUTORADO

CURITIBA
MAIO -2007.

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

Tese de Doutorado

apresentada à Universidade Tecnológica Federal do Paraná

como requisito para a obtenção do título de

DOUTOR EM CIÊNCIAS

por

ANDREY RICARDO PIMENTEL

UMA ABORDAGEM PARA PROJETO DE *SOFTWARE* ORIENTADO A OBJETOS BASEADO NA TEORIA DE PROJETO AXIOMÁTICO

Orientador:

Prof. Dr. Paulo Cézar Stadzisz	CPGEI, UTFPR

Membros da Banca:

Profª. Drª. Itana Maria de Souza Gimenes	DIN, UEM
Prof. Dr. João Umberto Furquim de Souza	DEINFO, UEPG
Prof. Dr. Cesar Augusto Tacla	CPGEI, UTFPR
Prof. Dr. Douglas Paulo Bertrand Renaux	DAELN, UTFPR
Prof. Dr. Jean Marcelo Simão	DAELN, UTFPR

Curitiba, Maio de 2007

ANDREY RICARDO PIMENTEL

UMA ABORDAGEM PARA PROJETO DE *SOFTWARE* ORIENTADO A OBJETOS BASEADO NA TEORIA DE PROJETO AXIOMÁTICO

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) da Universidade Tecnológica Federal do Paraná (UTFPR), como parte dos requisitos para obtenção do título de Doutor em Ciências. Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Paulo Cézar Stadzisz

CURITIBA - PR

Maio 2007

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

P644a Pimentel, Andrey Ricardo
Uma abordagem para projeto de software orientado a objetos baseado na teoria de projeto axiomático / Andrey Ricardo Pimentel. Curitiba. UTFPR, 2007
189 f. : il. ; 30 cm

Orientador: Prof. Dr. Paulo Cézar Stadzisz
Tese (Doutorado) – Universidade Tecnológica Federal do Paraná. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2007

1. Engenharia de software. 2. Software – Desenvolvimento. I. Stadzisz, Paulo Cézar, orient. II. Universidade Tecnológica Federal do Paraná. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. III. Título.
CDD: 005.1

À Nossa Senhora da Rosa Mística

Agradecimentos

A Deus.

Aos meus pais e ao meu irmão pelo amor, carinho e apoio que sempre tive.

À minha amada Christiana, pelo amor, carinho, paciência, dedicação e apoio, sem o qual não teria conseguido

Ao prof. Dr. Paulo Cézar Stadzisz pelo privilégio de tê-lo tido como orientador e pela dedicação, paciência e empenho para a realização deste trabalho.

Agradeço aos membros da banca examinadora pela disposição em analisar esta tese e pelas valiosas contribuições para tornar este trabalho melhor.

Agradeço ao meus colegas do LSIP: Evangivaldo, Quinaia, Erik, Malga, Schastai, Luguesi, Jean, Ademir, Leticia, Nico, Fabio, e todos os outros, pelo incentivo, pelo companheirismo, pelas discussões, pelo café e pela sensacional convivência.

À Universidade Tecnológica Federal de Paraná - UTFPR e ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI pela infraestrutura e suporte para a realização deste.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico CNPq pela bolsa, sem a qual seria impossível realizar este trabalho.

Aos meus amigos.

Resumo

Esta tese apresenta uma abordagem que aplica a Teoria de Projeto Axiomático ao projeto de *software* orientado a objetos para ajudar a garantir a qualidade da solução de projeto ao longo do processo de desenvolvimento. A qualidade do projeto é fundamental para a realização de um produto com qualidade. A fundamentação teórica da Teoria de Projeto Axiomático e sua natureza livre de domínio, a tornam uma poderosa ferramenta para ser aplicada em conjunto com metodologias e técnicas de desenvolvimento de *software* orientado a objetos visando auxiliar na garantia de qualidade na solução de projeto. O objetivo desta abordagem é propor e integrar métodos que permitam o uso do Projeto Axiomático em conjunto com um processo de desenvolvimento de *software* orientado a objetos, como o Processo Unificado. A abordagem proposta possui 4 etapas que estão inseridas principalmente na fase de elaboração do Processo Unificado. A abordagem de projeto proposta estabelece formas de aplicar o Axioma da Independência em projetos de *software* orientados a objetos para garantir a qualidade da solução de projeto ao longo do processo de desenvolvimento. São definidas analogias entre os conceitos do Projeto Axiomático e os de projeto de *software* orientado a objetos. É estabelecido um modelo de hierarquia para a decomposição dos requisitos funcionais e dos parâmetros de projeto, baseada em casos de uso. Um processo de *ziguezagueamento* que estende o processo original é criado para poder ser aplicado a um ciclo de vida de desenvolvimento de *software* iterativo e incremental. É criado um arcabouço para a aplicação do Axioma da Informação para projetos de *software* orientado a objetos. Este arcabouço usa métricas de complexidade de *software* bem estabelecidas na literatura como pontos por caso de uso, pontos por função e o conjunto de métricas de Chidamber e Kemerer. Apresenta-se um estudo de caso em que foi criada uma solução de projeto para um sistema embarcado com restrições de tempo, ilustrando como a abordagem proposta pode ser usada para escolher a melhor solução de projeto a cada nível de decomposição garantindo a qualidade da solução de projeto.

Palavras-Chave: Engenharia de *Software*, Orientação a Objetos, Projeto Axiomático, Processo Unificado, Qualidade de Projeto.

Abstract

This thesis presents an approach, that applies the Axiomatic Design Theory to Object-Oriented software design in order to guarantee the quality of the design solution along the development process. The quality of the design is an essential aspect for the construction of a product with quality. The theorethical foundation of the Axiomatic Design Theory and its domain-free nature makes it a powerfull tool to be applied together with Object-Oriented software development metodologies and techniques in order to help on guarantiing the quality of the design solution. The goal of this approach is to propose and integrate methods that allow the use of Axiomatic Design together with an Object-Oriented software development methodology, such as the Unified Process. The proposed approach has 4 stages that are inserted mainly on the elaboration phase of the Unified Process. The proposed design approach establishes ways for applying the Independence Axiom in order to guarantee the quality of the design solution along the design process. Analogies between concepts of Axiomatic Design and Object-Oriented software design are defined. A hierarchical model for the decomposition of functional requirements and design parameters, based on use cases, is defined. A zigzagging process that extends the original zigzagging process, is created to be applied on an iterative and incremental software development life cycle. A framework for the application of the Information Axiom on Object-Oriented software is created. This framework uses software complexity metrics well-established on the literature such as use case points, function poins and the Chidamber and Kemerer metrics suite. A case study is presented where a design solution for an embedded system with time constraints is created. This case study illustrates how the proposed approach can be used to choose the best design solution, at each level of design decomposition, in order to guarantee the quality of the design solution.

Keywords: Software Engineering, Object-Orientation, Axiomatic Design, Unified Process, Design Quality.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

ISO	*International Organization for Standardization*
ABNT	Associação Brasileira de Normas Técnicas
LOC	Linhas de Código
CMM	*Capability Maturity Model*
UML	*Unified Modeling Language*
TRIZ	Teoria Inventiva de Resolução de Problemas
QFD	Desdobramento da Função da Qualidade
FR	Requisito Funcional
DP	Parâmetro de Projeto
CA	Atributos do Cliente
PV	Variável de Processo
GDT	Teoria Geral de Projeto
IEEE	*Institute of Electrical and Electronics Engineers*
OMT	*Object Modeling Technique*
OOSE	*Object-Oriented Software Engineering*
RUP	*Rational Unified Proces*
CPGEI	Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial
UTFPR	Universidade Tecnológica Federal do Paraná
DSQI	*Design Structure Quality Index*
XML	*eXtensible Markup Language*
WDK	*Workshop Design Konstruktion*
HOQ	*House Of Quality*
SQFD	Desdobramento da Função da Qualidade para *Software*
MIT	*Massachusets Institute of Technology*
ADo-oSS	*Axiomatic Design of Object-Oriented Software Systems*
SOA	Arquitetura Orientada a Serviços
WMC	Métodos Ponderados por Classe
DIT	Profundidade da Árvore de Herança
NOC	Número de Subclasses
CBO	Acoplamento entre Objetos
RFC	Resposta para uma Classe
LCOM	Falta de Coesão nos Métodos
RISC	*Reduced Instruction Set Computer*
SRAM	*Static Random Access Memory*
ADC	*Analog to Digital Converter*
DAC	*Digital to Analog Converter*
RTC	*Real-time Clock*
USB	*Universal Serial Bus*
LCD	*Liquid Cristal Display*
CK	Métricas de Chindamber e Kemerer
CASE	*Computer Aided Software Engineering*

Sumário

1 Introdução

Este Capítulo apresenta o panorama global desta tese por meio da descrição do tema do trabalho, do contexto no qual este tema está inserido, além dos principais problemas envolvidos que representam as questões de base do trabalho. Neste Capítulo são também apresentados os objetivos gerais deste trabalho e os objetivos específicos a serem alcançados, bem como a motivação e a justificativa para sua realização. Por fim, apresenta-se a organização dos capítulos do documento.

1.1 Contexto e Questionamento do Trabalho

O desenvolvimento de *software* tem sido objeto de estudos desde a criação do computador na década de 40. Desde então, a capacidade dos processadores tem aumentado em uma taxa de crescimento exponencial, seguindo a "lei de Moore" (MOORE, 1965) (BEZERRA, 2002). Com isso, a relação custo/desempenho do *hardware* pôde ser reduzida drasticamente ocasionando uma grande ampliação na utilização do computador, tanto pelas empresas quanto pela população em geral. O computador passou a ser usado como ferramenta comum de trabalho, equipamento de comunicação e, até mesmo, como forma de entretenimento. A popularização dos computadores provocou um crescimento muito grande na demanda por *software*, cada vez maiores e mais complexos.

Os programas de computador hoje são aplicados nas mais diversas áreas de atividade humana, que vão desde operações bancárias até educação e entretenimento. Existem sistemas computacionais envolvidos em operações críticas, como sistemas que controlam o funcionamento de uma aeronave, o que exige um grau elevado de confiabilidade. Por outro lado, existem sistemas computacionais que demandam interatividade para serem interessantes e apresentarem bons resultados, como os envolvidos com educação e entretenimento. Muitos sistemas computacionais, como os de força de vendas e telemetria, necessitam coletar informação em locais diversos, exigindo mobilidade do sistema. Outro aspecto importante é

que as organizações estão em constante evolução, o que cria uma necessidade constante de evolução e manutenção em seus sistemas de informação.

Como resultado destas diversas exigências, aumentou a necessidade de se construir *software* com maior qualidade, o que elevou os custos de desenvolvimento e manutenção de sistemas computacionais. Por outro lado, não era possível desenvolver *software* que pudesse satisfazer toda a demanda, em termos de diversidade, quantidade e complexidade (BROOKS, 1987). O processo de desenvolvimento de *software* teve que evoluir para fazer frente às pressões por redução de custos e aumento da produtividade (WINCK; GOETTEN, 2006).

Neste contexto, novos paradigmas, linguagens e abordagens foram desenvolvidos e, à medida que o *software* foi se tornando mais importante, novas metodologias e técnicas para seu desenvolvimento foram propostas e melhoradas continuamente. Cada nova metodologia, método ou técnica, representa um esforço para melhorar o processo de desenvolvimento de sistemas de *software*.

Entretanto, a qualidade do produto de *software* não é fácil de ser alcançada. Os sistemas de *software* devem ser projetados corretamente para serem controláveis, confiáveis, produtivos e, então, atingirem seus objetivos (SUH, 2001). Segundo Pressman (2005), os requisitos de *software* são o fundamento para a medição da qualidade e, além disso, "...um modelo que exibe alta qualidade levará a um software que exiba qualidade" (PRESSMAN, 2005). Isto significa que a qualidade do produto de *software* está fortemente ligada à qualidade do projeto. Portanto, é fundamental considerar a qualidade do projeto pois um projeto com qualidade tem uma probabilidade muito maior de gerar um produto de *software* com qualidade. Neste contexto, a qualidade do *software* também depende do processo de desenvolvimento, pois um processo de desenvolvimento de software bem definido e bem gerenciado é uma característica fundamental que diferencia um projeto hiperprodutivo de um malsucedido (JACOBSON; BOOCH; RUMBAUGH, 1999).

Os fatores de qualidade de *software*, descritos no padrão ISO 9126, da *International Organization for Standardization* (ISO), adotado no Brasil pela Associação Brasileira de Normas Técnicas (ABNT) como NBR 13596, estabelecem seis atributos chave de qualidade de *software*, que são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade (ABNT, 1996). Além destes, são freqüentemente citados na literatura como padrões de qualidade de *software*: corretude, integridade, flexibilidade, testabilidade, reusabilidade e interoperabilidade (CAVANO; McCALL, 1978) (PRESSMAN, 2005) (PETERS; PEDRYCZ, 2001).

Além de critérios de qualidade de *software*, existem vários tipos de métricas de

software, entre elas: a métrica linhas de código (LOC) (PRESSMAN, 2005), a Complexidade Ciclomática (McCABE, 1976), métricas orientadas a objetos (LORENZ; KIDD, 1994) (CHIDAMBER; KEMERER, 1994) e métricas orientadas a funcionalidade como Pontos por Função (VAZQUEZ; SIMÕES; ALBERT, 2003). Estas métricas avaliam o tamanho e a complexidade do produto de *software* desenvolvido, servindo de base para estimativas de esforço para futuros desenvolvimentos e para avaliar a produtividade das equipes.

Apesar dos elementos existentes, o grande desafio ainda é construir sistemas de *software* com mais qualidade, dentro dos prazos estabelecidos e, mais importante, dentro de um orçamento limitado, o que resulta em mais maturidade no processo de desenvolvimento (PAULK et al., 1991). Neste sentido, foi criado um padrão de qualidade no desenvolvimento de *software,* chamado *Capability Maturity Model* (CMM)*,* para garantir que as empresas fornecedoras de *software* para o governo dos EUA conseguissem entregá-lo com a qualidade desejada, dentro do prazo e orçamento estipulados (PAULK et al., 1991). O CMM é um modelo que classifica a organização, de acordo com a sua maturidade no processo de desenvolvimento de *software,* em 5 níveis: inicial, repetível, definido, gerenciado e otimizado.

Para lidar com restrições de qualidade, prazos e orçamento, o desenvolvimento de *software* se tornou cada vez menos um processo baseado na experiência ou talento dos desenvolvedores e mais um processo de engenharia, do ponto de vista do rigor das técnicas e padrões aplicados. As metodologias de desenvolvimento de *software* atuais representam melhorias importantes ao processo de desenvolvimento, contemplando: notação unificada para documentação de projetos, melhor organização do processo de desenvolvimento usando etapas, artefatos, atividades bem definidas, métricas e melhor gerenciamento da complexidade do projeto (JACOBSON; BOOCH; RUMBAUGH, 1999).

Existem metodologias de desenvolvimento de *software* como as citadas na seção 1.3, critérios de qualidade de projeto de *software* como os citados na seção 2.1.2, critérios de qualidade de *software* como os citados nesta seção e em (PRESSMAN, 2005), métricas de *software* como as citadas anteriormente e notações de projeto como a *Unified Modeling Language* (UML) (OBJECT MANAGEMENT GROUP, 2005).

Apesar disso, "...mesmo atualmente, falta para a maioria das metodologias de desenvolvimento de *software* a profundidade, flexibilidade e a natureza quantitativa que estão normalmente associadas com disciplinas de projeto de engenharia mais clássicas"[1] (PRESSMAN, 2005). De fato, muitas das decisões de projeto de *software*,como por exemplo

[1] Tradução de "*...even today, most software design methodologies lack the depth, flexibility and quantitative nature that are normally associated with more classical engineering design disciplines*."

a escolha de quais classes irão compor um sistema ou a atribuição de responsabilidades para estas classes, são tomadas sem o uso de critérios quantitativos que as orientem. Estas decisões são tomadas, na maioria das vezes com base apenas em experiências bem sucedidas dos desenvolvedores, o que pode levar à criação de soluções de projeto não adequadas.

São necessárias contribuições no sentido de ajudar o projetista a tomar decisões ao longo do processo de desenvolvimento de *software*. Existem muitas decisões de projeto a serem feitas pelo desenvolvedor, entre elas a escolha da solução de projeto mais adequada em um conjunto de soluções aceitáveis para o projeto. A ausência de um critério preciso para tomada de decisão pode tornar o processo de desenvolvimento menos confiável, uma vez que boas decisões são fundamentais para se criar uma solução de projeto melhor.

Uma base científica para projeto poderá ajudar a aprimorar as atividades de projeto, fornecendo ao projetista uma fundamentação teórica baseada em princípios que caracterizam boas soluções de projeto. As disciplinas de projeto de engenharia normalmente estão associadas a uma base científica e teórica. Existem diversos estudos sobre projeto em engenharia como, por exemplo: a teoria de Projeto Robusto de Taguchi (PADKE, 1989), a Teoria Inventiva de Resolução de Problemas (TRIZ) (SAVRANSKI, 2000), o Desdobramento da Função da Qualidade (QFD) (CLAUSING, 1994) e a Teoria de Projeto Axiomático (SUH, 1990). Uma característica comum nessas técnicas é fornecer critérios para a tomada de decisões de projeto.

Esses critérios podem ser quantitativos como a função de perda de qualidade e os vetores ortogonais do Projeto Robusto (PADKE, 1989), a técnica da Casa da Qualidade (HAUSER; CLAUSING, 1998) e as métricas de independência funcional e de conteúdo de informação do Projeto Axiomático (SUH, 1990). Existem também critérios de tomada de decisões de projeto baseados em princípios de bons projetos como os axiomas, teoremas e corolários da Teoria de Projeto Axiomático e os princípios inventivos e a matriz de contradições da TRIZ (SAVRANSKI, 2000). Além disso, a Teoria de Projeto Axiomático possui uma natureza independente de domínio, o que a torna mais abrangente e facilita sua aplicabilidade no desenvolvimento de *software*.

A Teoria de Projeto Axiomático (SUH, 1990) surgiu como um processo para projeto de produtos, peças e componentes, usado em diversas áreas da engenharia. Este processo é baseado em conceitos de independência entre os requisitos funcionais do projeto e na minimização do conteúdo de informação do projeto. Estes conceitos podem, também, ser aplicados no desenvolvimento de *software* e podem melhorar a qualidade do processo de desenvolvimento, garantindo a qualidade do produto desenvolvido. Com a Teoria de Projeto

Axiomático é possível conseguir parâmetros efetivos para orientar a tomada de decisões de projeto de forma a garantir a qualidade da solução de projeto.

A Teoria de Projeto Axiomático foi concebida inicialmente em 1978, para projetos de engenharia mecânica, principalmente na indústria automobilística (DEO; SUH, 2004), e muitos dos seus conceitos são aplicáveis diretamente ao projeto e produção (manufatura) neste domínio. O Projeto Axiomático passou a ser aplicado em outras áreas como indústria naval (WHITCOMB; SZATKOWSKI, 2000) e indústria eletrônica (DO; PARK, 1996). Entretanto, o desenvolvimento de *software* difere em muitos aspectos do desenvolvimento de produtos tanto em engenharia mecânica como naval e eletrônica. As metodologias e técnicas atuais de projeto de *software* possuem conceitos muito específicos. Uma metodologia de desenvolvimento de *software* está inserida em um processo. O processo de *software* mais usado atualmente é o Processo Unificado (UP) (JACOBSON; BOOCH; RUMBAUGH, 1999).

Assim sendo, o principal questionamento tratado nesta tese é: "É possível e vantajoso aplicar a metodologia de projeto de engenharia Teoria de Projeto Axiomático em conjunto com um processo de desenvolvimento de *software* orientado a objetos como o Processo Unificado?" Existem similaridades entre projetos em engenharia industrial e projetos de *software*. Estas similaridades são traduzidas em princípios de bons projetos que compõem a base da Teoria de Projeto Axiomático. O que indica que os princípios de projeto da Teoria de Projeto Axiomático são aplicáveis em projeto de *software*. Portanto, o que se deseja é utilizar os princípios da Teoria de Projeto Axiomático, seus critérios para tomada de decisões de projeto, seus conceitos, suas ferramentas e sua natureza independente de domínio, para aprimorar o processo de desenvolvimento de *software* orientado a objetos.

1.2 Objetivos

Esta tese de doutorado aborda a integração entre teorias de projeto de engenharia e metodologias e técnicas de desenvolvimento de *software* orientado a objetos. O objetivo geral desta tese de doutorado é:

- Desenvolver uma abordagem baseada na Teoria do Projeto Axiomático para o projeto de *software* orientado a objetos integrando métodos existentes para possibilitar a aplicação do Processo Unificado em conjunto com o Projeto

Axiomático.

Os objetivos específicos deste trabalho são:

1. A avaliação da Teoria de Projeto Axiomático (SUH, 1990), seus princípios, técnicas, ferramentas e sua aplicabilidade em projetos em geral, especialmente em projetos de *software*.
2. A definição de correspondências entre os conceitos do Processo Unificado como casos de uso, classes, objetos, atributos, métodos, responsabilidades e colaborações, e os conceitos da teoria de Projeto Axiomático, dentre eles, atributos do cliente, requisitos funcionais, parâmetros de projeto, restrições e variáveis de processo.
3. A definição de associações entre os domínios da teoria de Projeto Axiomático e as fases do Processo Unificado.
4. A definição de um padrão de decomposição dos requisitos funcionais fortemente baseada em casos de uso para a aplicação em conjunto com o Processo Unificado.
5. O estabelecimento de um padrão para se obter um processo de "ziguezagueamento" mais próximo de um ciclo de vida iterativo e incremental como o do Processo Unificado.
6. A descrição da aplicação dos axiomas na fase de projeto de *software*. A teoria de Projeto Axiomático é baseada em dois Axiomas, além de corolários e teoremas relacionados. Essa descrição é feita através da determinação de como a aplicação destes axiomas, teoremas e corolários, influencia a tomada de decisões de projeto e quais são as conseqüências desta aplicação.
7. A definição de um arcabouço para calcular o conteúdo de informação de um projeto de *software* orientado a objetos necessário para a aplicação dos axiomas, teoremas e corolários da Teoria de Projeto Axiomático. Este arcabouço se baseia em métricas de complexidade de *software* orientado a objetos encontradas na literatura.
8. Desenvolver um estudo de caso para avaliação dos impactos da aplicação da abordagem em um projeto de *software* orientado a objetos.

1.3 Motivação e Justificativa do Trabalho

A motivação para este trabalho é aprimorar o processo de desenvolvimento de *software* orientado a objetos, através da aplicação da Teoria de Projeto Axiomático, provendo critérios, para tomada de decisões de projeto, baseados em princípios gerais de bons projetos. Nesta tese é proposta uma abordagem para a aplicação da teoria de Projeto Axiomático no projeto de *software* orientado a objetos. A teoria de Projeto Axiomático é aplicada com sucesso em várias áreas do conhecimento humano. Este sucesso alcançado pelo Projeto Axiomático torna promissora a contribuição que sua aplicação pode trazer ao projeto de *software* orientado a objetos. Isto se deve em parte a problemas relacionados à tomada de decisões de projeto em engenharia de *software*, mais especificamente envolvendo projeto orientado a objetos. Estes problemas serão discutidos na seção 2.1.3.

O projeto de *software* evoluiu muito desde o surgimento dos primeiros computadores, tendo sido criadas diversas técnicas, como: a metodologia estruturada (GANE; SARSON, 1995), Análise Essencial (McMENAMIM; PALMER, 1991), técnicas orientadas a objetos como a proposta por Booch (BOOCH, 1994), *Object Modeling Technique* (OMT) (RUMBAUGH et al., 1991), *Object-Oriented Software Engineering* (OOSE) (JACOBSON et al., 1992), *Unified Modeling Language* (UML) (RUMBAUGH; JACOBSON; BOOCH, 2004), *Rational Unified Process* (RUP) (JACOBSON; BOOCH; RUMBAUGH, 1999), a proposta por Larman (LARMAN, 2004) e, mais recentemente, *Extreme Programming* (ASTEL; MILLER; NOVAK, 2002), desenvolvimento orientado a aspectos (JACOBSON; NG, 2004) e desenvolvimento baseado em componentes (GIMENES; HUZITA, 2005).

Essas metodologias e técnicas provêem para o desenvolvedor poucas ferramentas para a tomada de decisões de projeto. Existem princípios e diretrizes para realização de bons projetos, como modularidade, coesão e acoplamento, listados em (PRESSMAN, 2005) que são discutidos na Seção 2.1.2. Estes princípios e diretrizes representam linhas gerais a serem seguidas pelo projetista provendo pouco auxilio para tomada de decisões de projeto como a escolha da melhor solução de projeto entre algumas soluções possíveis. Decisões de projeto são tomadas na maioria das vezes com base na experiência ou intuição do desenvolvedor. A experiência do projetista, muitas vezes representada na forma de padrões de projeto (GAMMA et al., 2000), pode auxiliar na escolha de um tipo de solução que já foi aplicado com sucesso em outros projetos mas não garante que ela seja a mais adequada para o caso atual.

Neste contexto, ao se realizar um projeto de *software*, apenas a aplicação das metodologias existentes não garante a qualidade da solução de projeto. Elas provêem critérios para avaliar, através de testes, se o produto de *software* gerado pela solução de projeto satisfaz os requisitos funcionais. Com os testes, pode-se avaliar características de qualidade do *software* como funcionalidade, confiabilidade, eficiência, corretude e integridade mas não características como manutenibilidade, flexibilidade, reusabilidade e interoperabilidade que são derivadas de um projeto bem feito. Portanto, apenas com testes não se consegue garantir a qualidade da solução de projeto.

Para tentar obter qualidade do projeto, começou-se a adotar padrões de *software*, que descrevem a aplicação bem sucedida da experiência de outros desenvolvedores (GAMMA et al., 2000). Existem várias categorias de padrões que podem ser usadas no desenvolvimento de *software*, entre elas: padrões de projeto (GAMMA et al., 2000), padrões arquiteturais (FOWLER, 2003) (QUINAIA, 2005) e padrões de caso de uso (ADOLPH; BRAMBLE, 2003). Os padrões de projeto (GAMMA et al., 2000) provêem ao projetista um repertório de soluções de projeto que vêm sendo aplicadas satisfatoriamente no desenvolvimento de vários sistemas, o que não garante que ela seja adequada para um determinado problema. A própria escolha de qual padrão usar já é uma decisão de projeto baseada fortemente na experiência. Entretanto, uma metodologia de desenvolvimento deve prover mecanismos, como princípios e critérios, para avaliar e garantir que as soluções de projeto tenham a qualidade desejada. Usar sempre uma boa solução de projeto na construção do produto é fundamental para que o produto final tenha qualidade.

A natureza livre de domínio da Teoria de Projeto Axiomático, o alto nível de abstração de seus conceitos de projeto e sua fundamentação teórica fazem do Projeto Axiomático uma ferramenta poderosa para ser aplicada em conjunto com metodologias e técnicas de desenvolvimento orientado a objetos para garantir a qualidade da solução de projeto de *software*. (SUH, 2001)

Na Teoria de Projeto Axiomático existem ferramentas como os axiomas, teoremas e seus corolários, a matriz de projeto, a hierarquia funcional, o "*ziguezagueamento*" e os domínios de projeto (SUH, 1990), que auxiliam a tomada de decisões de projeto. A Teoria de Projeto Axiomático provê ferramentas que permitem comparar soluções alternativas de projeto e escolher a melhor delas em cada uma das diversas etapas de desenvolvimento. Por meio destas comparações, o Projeto Axiomático permite identificar e descartar soluções não promissoras mesmo nas fases iniciais do desenvolvimento. Com isto, garante-se que apenas as boas soluções, sejam desenvolvidas evitando-se gastos desnecessários de recursos ou o

desenvolvimento de soluções inadequadas.

Na Teoria de Projeto Axiomático os requisitos funcionais não estão necessariamente associados a uma utilização do produto ou sistema pelo usuário. Neste caso, a definição de uma abordagem fortemente baseada em casos de uso para a Teoria de Projeto Axiomático pode deixar o processo mais focado na utilização do *software*. Este fato pode auxiliar na aplicação do Projeto Axiomático em projetos em outras áreas do conhecimento, em especial na área de Engenharia de *Software*.

É importante prover um mecanismo para determinar a ordem na qual os módulos de *software* devem ser projetados e quais módulos podem ser projetados em paralelo por equipes diferentes. Muitas vezes é necessária a divisão do trabalho em diferentes equipes de desenvolvimento. A Teoria de Projeto Axiomático fornece ferramentas para tornar esta divisão mais eficiente e, também, formas para fazer com que as equipes trabalhem da maneira mais independente possível. Estas ferramentas são: os axiomas já citados, que permitem tornar os módulos mais independentes, e a matriz de projeto (SUH, 1990).

Esta matriz fornece informações sobre qual parâmetro de projeto satisfaz determinado requisito funcional. Isto permite visualizar quais parâmetros de projeto estão relacionados com o mesmo requisito funcional e, portanto, saber quais são independentes. Esta independência funcional permite que as diferentes equipes possam realizar o desenvolvimento do *software*, independentemente. Caso contrário, haverá a necessidade de comunicação e sincronização entre as equipes durante o desenvolvimento, o que eleva a duração, o custo, e a complexidade do desenvolvimento.

Um aspecto importante para garantir a qualidade de *software* é a capacidade de rastrear os requisitos a partir de algum artefato que satisfaz este requisito, bem como rastrear os artefatos a partir do requisito que eles satizfazem (LEFFINGWELL; WIDRIG, 2003). Esta capacidade de rastreamento é garantida pela teoria de projeto axiomático através de ferramentas como a decomposição e a matriz de projeto.

1.4 Organização do Documento

Esta tese apresenta o trabalho de doutorado realizado no programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) da Universidade Tecnológica Federal do Paraná (UTFPR). Esta tese está organizada em 6 capítulos, além da

Introdução.

O Capítulo 2 apresenta a fundamentação teórica deste trabalho abordando: uma discussão sobre os principais problemas e dificuldades em projeto de *software* orientado a objetos, um resumo das principais teorias de projeto de engenharia e a apresentação e contextualização da Teoria de Projeto Axiomático.

No Capítulo 3 é apresentado um resumo da teoria de projeto axiomático, sua aplicação em *software* e trabalhos relacionados. São apresentados os principais conceitos do Projeto Axiomático relacionando-os com a engenharia de *software*. São apresentados também os principais trabalhos relacionados com projeto axiomático e desenvolvimento de *software*.

A abordagem para projeto de *software* orientado a objetos usando o Projeto Axiomático proposta neste trabalho é apresentada no Capítulo 4. Uma visão geral da abordagem é apresentada, contextualizando-se sua aplicação. São apresentados os aspectos relacionados com a independência funcional (Axioma 1) como: a correspondência entre os conceitos do Projeto Axiomático e os conceitos do Processo Unificado, a correspondência entre as fases do processo Unificado e os domínios do Projeto Axiomático e a aplicação do Axioma 1 no Processo Unificado.

No Capítulo 5 é apresentada um arcabouço para o cálculo do conteúdo de informação (Axioma 2) para a metodologia proposta neste trabalho.

No Capítulo 6 é apresentado um estudo de caso para a análise e avaliação da abordagem proposta. Aborda-se o caso do projeto e implementação de um sistema embarcado de tempo real.

No Capítulo 7 são apresentadas as conclusões da tese e perspectivas de trabalhos futuros.

2 Fundamentação Teórica

Neste Capítulo é apresentada a fundamentação teórica deste trabalho abordando: uma discussão sobre os principais problemas e dificuldades em projeto de *software* orientado a objetos, um resumo das principais teorias de projeto de engenharia que possuem aplicabilidade em projeto de *software* como TRIZ e QFD, sua aplicação em projeto de *software* e a apresentação e contextualização da teoria de Projeto Axiomático.

2.1 Conceitos e Problemas em Projetos de *Software*

Esta seção apresenta uma discussão a respeito dos principais problemas encontrados no projeto de *software* orientado a objetos. A discussão começa pela análise dos principais conceitos de teoria de projeto. A seguir são apresentadas as principais características de um projeto de *software* orientado a objetos para então serem apresentados os principais problemas encontrados nele.

2.1.1 Conceitos de Projeto em Geral

Existem vários conceitos e definições a respeito da atividade de projeto. Projeto é uma atividade humana que está intimamente relacionada com a concepção de um objeto ou serviço novo e a criação de um modelo para orientar a realização deste objeto. Como objeto, pode-se entender uma edificação, um artefato, um produto, um sistema de *software*, um procedimento, entre outros. Projetar é uma atividade que desde a antigüidade vem sendo estudada. O crítico de arquitetura romana Vitruvius, no século I a. C., desenvolveu a noção de que construções bem projetadas eram aquelas que apresentaram firmeza, comodidade e prazer (KAPOR, 1990).

Com isso, surgiu o conceito de que a qualidade de um produto está intimamente ligado com a qualidade do seu projeto. Este conceito é muito importante pois ressalta a

importância de um bom projeto para a construção de um produto de qualidade. O grande objetivo da engenharia de projeto é produzir um modelo ou representação que gere um produto de qualidade (PRESSMAN, 2005).

O ato de projetar está freqüentemente relacionado com a elaboração de um modelo ou plano para orientar a realização de um objeto. Neste sentido, projetar pode ser definido como "...o processo de se aplicar várias técnicas e princípios com o propósito de se definir um dispositivo, um processo ou um sistema com detalhes suficientes para permitir sua realização física" (TAYLOR, 1959). Isto freqüentemente é feito através de uma representação, ou modelo. O grande objetivo do projetista é construir um modelo ou representação de qualquer entidade que será construída posteriormente. A criação deste modelo envolve: intuição e julgamento baseado na experiência em projetos semelhantes, princípios que orientam a criação do modelo, critérios para avaliar a qualidade do produto projetado, além de um processo iterativo (PRESSMAN, 1995).

Um projeto é elaborado a partir da necessidade do objeto a ser construído, de forma que a realização do objeto satisfará esta necessidade. Dentre essas necessidades estão: demanda de mercado por um novo produto, necessidade de se realizar uma atividade que necessita ser planejada, necessidade de um cliente por um novo sistema, entre outros. Um problema existente ou a percepção do cliente para um problema pode gerar a necessidade por um objeto novo a ser projetado para solucionar o problema. Neste sentido, "o processo de projeto transforma a percepção do cliente para um problema no objeto de projeto" (NORLUND, 1996).

Projeto pode ser visto como uma função de mapeamento entre objetos de dois conjuntos. De acordo com a Teoria Geral de Projeto (GDT) (YOSHIKAWA, 1981), "Projeto pode ser definido como o mapeamento do espaço de atributos para o espaço de funções" (TOMIYAMA et al., 1989). O primeiro conjunto representa necessidades humanas e o segundo conjunto representa soluções tecnológicas que irão satisfazer essas necessidades. Nesse sentido, "Projeto é uma disciplina que tenta unir o mundo da tecnologia e o mundo das pessoas e propósitos humanos" (KAPOR, 1990).

Uma outra definição para projeto, relacionando necessidades e a satisfação dessas necessidades diz que "projeto é uma interação entre **o que** nós queremos alcançar e **como** nós queremos alcançá-lo." (SUH, 2001). Ou, em um enunciado mais completo:

> "Projeto pode ser definido formalmente como a criação de soluções sintetizadas na forma de produtos, processos ou sistemas que satisfazem necessidades percebidas através do mapeamento entre os

> requisitos funcionais no domínio funcional e os parâmetros de projeto do domínio físico, através da seleção apropriada dos parâmetros de projeto que satisfazem os requisitos funcionais." (SUH, 1990)

Com este enunciado, Suh relaciona as necessidades percebidas como requisitos funcionais para os produtos, sistemas ou processos, com os parâmetros de projeto devidamente selecionados que irão satisfazê-las. Os parâmetros de projeto podem ser considerados como sendo as soluções tecnológicas para a construção do produto, sistema ou processo.

Para criar as soluções tecnológicas que satisfaçam as necessidades e assim realizar um projeto, o projetista deve praticar a diversificação e convergência (PRESSMAN, 2005). Diversificação é a aquisição de um repertório de alternativas, componentes, soluções e conhecimento, tudo isso contido em catálogos, livros e na mente. Todo esse repertório deve ser montado e organizado e o projetista deve escolher dentro deste repertório as alternativas que melhor satisfazem aos requisitos definidos para o sistema. Essas alternativas devem ser analisadas e descartadas ou aceitas até que haja convergência para uma solução (BELADY, 1981).

Em outras palavras, projetar consiste em dois processos distintos: processo criativo e o processo analítico. No processo criativo, ou diversificação, novas idéias ou soluções são formuladas ou absorvidas. Enquanto que no processo analítico, ou convergência, estas novas soluções são analisadas e descartadas ou adotadas. O processo criativo é subjetivo e depende do conhecimento e da criatividade do projetista. O processo analítico é determinístico e baseado em um conjunto finito de princípios básicos, como os da Teoria de Projeto Axiomático (SUH, 1990).

Em resumo, a finalidade de um projeto é traduzir necessidades, ou requisitos, em termos de soluções tecnológicas que satisfaçam essas necessidades, de forma a obter um produto com qualidade. A escolha do conjunto adequado de requisitos funcionais pelo desenvolvedor é fundamental para a realização de um bom projeto e para a criação de um produto com qualidade, seja ele uma edificação, um automóvel, um computador ou mesmo um sistema de *software*.

O processo de projetar envolve a diversificação, que é a criação de um conjunto de possíveis soluções tecnológicas para satisfazer os requisitos e a convergência, que é a escolha de qual dessas soluções criadas é a melhor para satisfazê-los. Essa escolha, realizada no processo analítico, é feita através de decisões de projeto. A tomada de decisões de projeto corretas é fundamental para a criação de uma boa solução de projeto e, conseqüentemente,

para a criação de um produto de qualidade.

2.1.2 Projeto de *Software*

O projeto de um sistema de *software* tem muitas características em comum com projetos de engenharia, do ponto de vista do mapeamento dos requisitos funcionais em soluções tecnológicas. "Projetar sistemas de *software* significa determinar como os requisitos funcionais são implementados na forma de estruturas de *software*" (PETERS; PEDRYCZ, 2001). Para sistemas de *software*, as necessidades ou requisitos podem ser necessidades de clientes, demanda de mercado, necessidade de alteração ou evolução do sistema de *software*, entre outras. O conjunto dos requisitos para o produto de *software* são mapeados no conjunto das soluções tecnológicas para a construção do *software*.

Os requisitos funcionais representam o comportamento do sistema em termos de entradas, saídas e funcionalidades a serem providas pelo sistema a ser projetado. Entretanto, os requisitos funcionais não são suficientes para representar todos os requisitos de um sistema (LEFFINGWELL; WIDRIG, 2003). Existem características do sistema como desempenho, usabilidade, confiabilidade e suportabilidade que necessitam de um outro tipo de requisito para descrevê-las. Este tipo de requisito é chamado requisito não funcional (GRADY, 1992).

Neste caso, as soluções tecnológicas são estruturas de *software*, necessárias para a codificação do sistema de *software*. Em outras palavras, "...projeto é a única maneira com a qual se consegue traduzir com precisão os requisitos do cliente em um produto de *software* ou sistema finalizado". (PRESSMAN, 2005).

O processo de desenvolvimento de *software* está inserido no ciclo de vida do *software* (PETERS; PEDRYCZ, 2001). Existem vários modelos de ciclos de vida de *software*, como por exemplo: em cascata (ROYCE, 1970), espiral (BOEHM, 1989) e prototipação (GOMAA; SCOTT, 1981). Todos esses modelos de ciclos de vida têm como uma de suas etapas ou atividades o projeto. O padrão do *Institute of Electrical and Electronics Engineers* (IEEE), IEEE 1074.1-1997 para a criação de modelos de ciclo de vida de *software* estabelece que os principais processos da fase de desenvolvimento de *software* são: Requisitos, Projeto e Implementação (IEEE, 1997). Nos modelos de ciclo de vida, a atividade de projeto recebe o resultado da atividade de requisitos e gera um modelo para a atividade de implementação ou codificação.

Durante o projeto de *software* é criada a estrutura interna de um sistema de *software*. Esta estrutura interna é chamada de arquitetura. Arquitetura de *software* pode ser

definida como a estrutura ou organização dos componentes do programa (módulos), a maneira com a qual estes componentes interagem e a estrutura da informação que é usada por estes componentes (PRESSMAN, 2005).

O grande objetivo de um projeto de *software* é levar a um produto de *software* que tenha qualidade. Os principais quesitos de qualidade de *software* estão relacionados na Seção 1.1. Para conseguir alcançar os objetivos de qualidade do produto de *software* gerado, um projeto de *software* deve ter qualidade. Existem diretrizes para a qualidade de um projeto de *software*, e entre elas estão (PRESSMAN, 2005):

1. Um projeto deve ser modular;
2. Um projeto deve conter representações distintas para dados, arquitetura, interfaces e componentes;
3. Um projeto deve levar a componentes que possuam características de independência funcional;
4. Um projeto deve levar a interfaces que reduzam a complexidade das conexões entre os componentes e o ambiente externo.

Diretrizes de qualidade como as citadas acima podem ser usadas como parâmetro para avaliar a qualidade de um projeto de *software*. Mas para atingir essas diretrizes, é fundamental que o projetista tenha disciplina na aplicação de um processo de desenvolvimento, além de seguir princípios de projeto de *software*. Existem princípios de projeto de *software* que um projetista deve seguir para gerar um sistema de *software* com qualidade. Pressman (2005) enuncia princípios de projeto de *software*, que são (PRESSMAN, 2005):

1. O projeto deve ser rastreável para o modelo de análise;
2. Sempre considere a arquitetura do sistema a ser construído;
3. O projeto dos dados é tão importante quanto o projeto das funções de processamento;
4. As interfaces (ambas internas e externas) devem ser projetadas com cuidado;
5. A interface com o usuário deve estar sintonizada com as necessidades do usuário final;
6. O projeto, no nível dos componentes, deve ser funcionalmente independente;
7. Componentes devem ser fracamente acoplados entre eles e com o ambiente externo;
8. As representações (modelos) de projeto devem ser entendidas facilmente;
9. O projeto deve ser desenvolvido iterativamente.

Estimar e calcular a complexidade de um sistema de *software* é um parâmetro importante para o desenvolvimento com qualidade. É importante ter-se uma estimativa do tamanho do sistema para se conseguir estimar o esforço necessário na sua construção. Dentre as métricas de *software* existentes, a mais popular é a contagem de linhas de código (LOC) (PRESSMAN, 2005), devido à facilidade em ser computada. Existem métricas de *software* com várias finalidades como as de complexidade de programa como a "ciência do *software*" (PETERS; PEDRYCZ, 2001) e a "complexidade ciclomática" (McCABE, 1976), métricas da complexidade da arquitetura como DSQI (*Design Structure Quality Index*), métricas de complexidade orientada a objetos como as propostas por Lorenz e Kidd (LORENZ; KIDD, 1994) e por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994) e métricas de complexidade das funcionalidades do *software* como pontos por função (VAZQUEZ; SIMÕES; ALBERT, 2003) e os pontos por casos de uso (ANDA et al., 2001).

Estas métricas avaliam o tamanho e a complexidade do produto *software* desenvolvido, tendo como objetivo servir de base para estimativas de esforço para futuros desenvolvimentos e para avaliar a produtividade das equipes. As métricas de *software* são quantitativas e, por este motivo, não tem como dizer se um conjunto de classes e objetos é o mais adequado para satisfazer um determinado conjunto de requisitos funcionais, mas apenas dizer se um conjunto da classes é mais complexo que outro.

2.2 Desenvolvimento Orientado a Objetos e o Processo Unificado

O desenvolvimento orientado a objetos começou em 1967 com a linguagem Simula-67 (BIRTWISTLE et al., 1975) e desde então surgiram linguagens orientadas a objetos como Smalltalk (GOLDBERG; ROBSON, 1989) e C++ (STROUSTRUP, 1997) entre outras. Nos anos 80 começaram a surgir metodologias de desenvolvimento orientadas a objetos para tirar vantagens deste paradigma. Entre 1989 e 1994 surgiram quase 50 métodos de desenvolvimento orientados a objetos, fenômeno que foi chamado a guerra dos métodos (BOOCH; RUMBAUGH; JACOBSON, 2005).

Entre as mais importantes estavam: o método de G. Booch (BOOCH, 1994), a *Object Modeling Technique* (OMT) de J. Rumbaugh (RUMBAUGH et al., 1991), o método de Shlaer e Mellor (SHLAER; MELLOR, 1988) e o método *Objectory* de I. Jacobson (JACOBSON et al., 1992). I. Jacobson introduziu a modelagem de casos de uso em 1987

(JACOBSON, 1987) e criou o primeiro processo de desenvolvimento de *software* que utiliza casos de uso, chamado *Objectory* (JACOBSON et al., 1992).

Cada método possuía uma notação própria, o que gerou uma infinidade de tipos de diagramas e notações. Isto causava problemas de comunicação, treinamento de pessoal e portabilidade. Um esforço de unificação começou em 1994 quando J. Rumbaugh e, logo após, I. Jacobson, juntaram-se a G. Booch, na empresa *Rational Software Corporation*[2]. O primeiro grande resultado desse esforço foi a criação da *Unified Modeling Language* (UML), apresentada, na sua versão 1.0, em 1997 (BOOCH; RUMBAUGH; JACOBSON, 2005). A UML é uma linguagem criada para visualizar, especificar, construir e documentar os artefatos de um sistema de *software*. A UML é adotada, desde 1997, como padrão internacional pelo OMG (*Object Management Group*) (OBJECT MANAGEMENT GROUP, 2005).

Outro grande resultado deste esforço de unificação de metodologias foi a criação, pela *Rational*, de um processo de desenvolvimento que usa a UML em seus modelos, chamado Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999). O Processo Unificado evoluiu do processo *Rational Objectory*, sendo inicialmente chamado de *Rational Unified Process* (RUP). O Processo Unificado, apesar de não ser um padrão, é amplamente adotado, sendo considerado como um modelo de processo de desenvolvimento de *software* orientado a objetos.

2.2.1 Processo Unificado

Uma das características do Processo Unificado é ter um ciclo de vida iterativo e incremental. O desenvolvimento iterativo e incremental adota o modelo espiral de Boehm (BOEHM, 1989) em que cada fase do desenvolvimento é dividida em iterações (KRUCHTEN, 2003). A cada iteração, uma parte do desenvolvimento é construída, testada, validada e integrada ao restante do projeto. Devido a isto, uma abordagem iterativa e incremental ajuda a lidar com a complexidade do desenvolvimento de um sistema grande, além de tornar o desenvolvimento flexível o suficiente para acomodar novos requisitos, ou mesmo, mudanças nos requisitos existentes (BOOCH; RUMBAUGH; JACOBSON, 2005). Um ciclo de vida iterativo e incremental permite a identificação e correção de riscos e de erros mais cedo durante o processo (KRUCHTEN, 2003).

O Processo Unificado é dito guiado por casos de uso pois a construção do sistema é toda feita com base nos casos de uso (JACOBSON; BOOCH; RUMBAUGH, 1999). A cada

[2] Recentemente adquirida pela IBM

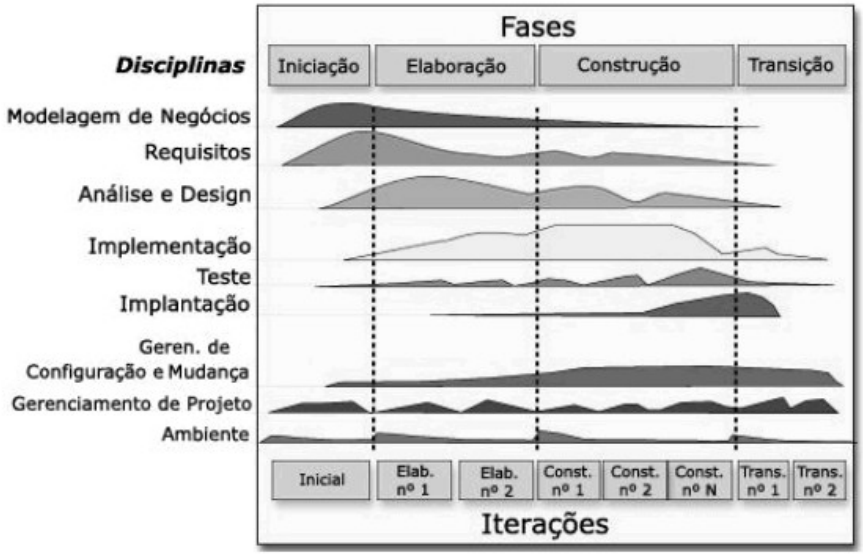
iteração da fase de construção, escolhe-se um pequeno conjunto de casos de uso que serão realizados. Através dos casos de uso é possível identificar e organizar os requisitos funcionais do sistema em função do resultado produzido para o usuário. Para realizar cada caso de uso são identificadas as classes, os objetos e suas interações, que são projetados, construídos, testados e integrados ao restante do sistema. Ao final de todas as iterações da fase de construção é obtido um sistema com todas as funcionalidades implementadas, testadas e integradas.

O Processo Unificado também é dito centrado na arquitetura. Um processo centrado na arquitetura significa que no começo do processo são criadas as linhas gerais de uma arquitetura do *software* que evolui a cada iteração e é usada como linha mestra durante todo o desenvolvimento. Isto torna mais fácil dividir o desenvolvimento em várias equipes, diminuindo o retrabalho, facilitando o reuso de componentes e servindo como uma base sólida para gerenciar o desenvolvimento (BOOCH; RUMBAUGH; JACOBSON, 2005). Além disso, a UML provê mecanismos para que o desenvolvedor tenha várias visões diferentes da arquitetura do *software* segundo sua necessidade (KRUCHTEN, 2003).

2.2.2 Fases do Processo Unificado

O Processo Unificado é dividido em duas dimensões: a das fases e a dos fluxos (KRUCHTEN, 2003). As relações entre as fases do Processo Unificado e seus principais fluxos de atividades estão ilustrados na Figura 1.

A dimensão das fases mostra características do ciclo de vida do Processo Unificado em função do tempo. Uma fase é o espaço de tempo entre dois marcos principais do processo, onde os objetivos da fase são alcançados, artefatos são concluídos e decisões sobre a próxima fase são tomadas (BOOCH; RUMBAUGH; JACOBSON, 2005). Cada fase, por sua vez, poderá ser composta por ciclos, iterações e marcos (KRUCHTEN, 2003). Uma iteração representa um ciclo completo de desenvolvimento que vai desde a análise de requisitos até implementação e testes que resulta em uma versão executável (BOOCH; RUMBAUGH; JACOBSON, 2005). As fases do processo unificado são: concepção (iniciação), elaboração, construção e transição.



Fonte: (KRUCHTEN, 2003)

Figura 1 - Fases, disciplinas e iterações do Processo Unificado

Na fase da concepção, delimita-se o escopo e a visão do sistema a ser construído, e é estabelecido o plano inicial do desenvolvimento onde são estimados prazos, esforços e outras necessidades. Além disso, identificam-se os riscos críticos, principalmente aqueles que podem afetar a capacidade de se construir o sistema (KRUCHTEN, 2003). Nesta fase é importante demostrar para os usuários em potencial que o sistema proposto poderá satisfazer suas necessidades (JACOBSON; BOOCH; RUMBAUGH, 1999). Nesta fase são elaborados: um documento com as exigências e características principais para o projeto, um modelo inicial de casos de uso, uma avaliação de riscos, além de um modelo empresarial, se necessário (KRUCHTEN, 2003).

Na fase de elaboração é criada uma arquitetura estável para o sistema que cubra as funcionalidades mais significativas para os usuários. Nesta fase são identificados, organizados e especificados a maior parte dos atores e casos de uso do sistema. Além disso, planeja-se a fase de construção do sistema e prepara-se o ambiente necessário para o desenvolvimento (KRUCHTEN, 2003). Entre os principais produtos desta fase estão: Um modelo de casos de uso, a descrição dos requisitos não funcionais, um protótipo executável e uma descrição da arquitetura do *software* (KRUCHTEN, 2003).

O objetivo principal da fase de construção é projetar, implementar e testar todo o sistema. Esta fase usualmente é dividida em várias iterações e a cada iteração um grupo de funcionalidades (casos de uso) é projetado, construído e testado, integrando-se ao sistema como um todo. O principal resultado desta fase é o código executável completo do sistema

pronto para os testes de implantação (JACOBSON; BOOCH; RUMBAUGH, 1999).

A fase de transição se inicia com a versão anterior à de distribuição do sistema. Entre as principais atividades desta fase estão: preparar a implantação do sistema, preparar manuais e treinamento para os usuários, testar a ajustar o *software* para as condições reais de operação, corrigir os defeitos identificados nos testes de implantação e implantar o *software* (JACOBSON; BOOCH; RUMBAUGH, 1999).

2.2.3 Fluxos, Disciplinas e Atividades

A dimensão dos fluxos mostra as características do Processo Unificado em função das atividades realizadas. Esta dimensão representa os fluxos principais do processo, agrupando as atividades de acordo com a sua natureza. Esta dimensão representa os trabalhadores, atividades e artefatos que participam de cada fluxo (KRUCHTEN, 2003). Artefatos podem ser: documentos, modelos, partes de modelos, código fonte, código executável, entre outros que são criados ou manipulados na execução dos fluxos.

Por sua vez, os fluxos, ou disciplinas, previstos pelo Processo Unificado são: modelagem de negócios, requisitos, análise e projeto, implementação, testes, distribuição, gerenciamento de configuração e mudança, gerenciamento de projeto e ambiente (KRUCHTEN, 2003). Nesta tese são abordados principalmente os fluxos de requisitos, análise e projeto e implementação. A cada iteração, principalmente na fase de construção, o desenvolvimento passa por vários fluxos, com ênfase nos objetivos da fase. Em uma iteração são realizadas atividades de cada um dos fluxos, pelos trabalhadores, gerando os artefatos necessários para a construção da versão do *software* a ser gerada nesta iteração.

No fluxo de requisitos, os requisitos funcionais e não funcionais do sistema são identificados, organizados e especificados. O artefato principal deste fluxo é o modelo de casos de uso que especifica os requisitos funcionais, a interação com o usuário, além dos requisitos não funcionais (KRUCHTEN, 2003).

O objetivo do fluxo de análise e projeto é mapear os requisitos em modelos que descrevam como implementar o sistema. Na atividade de análise é estabelecida uma arquitetura candidata, através da análise dos casos de uso, que é representada através do modelo de análise (KRUCHTEN, 2003). O modelo de análise representa as classes de análise e suas colaborações, que irão descrever conceitualmente como os casos de uso serão realizados (JACOBSON; BOOCH; RUMBAUGH, 1999).

Na atividade de projeto o modelo de análise é refinado, criando-se o modelo de

projeto. Nesta atividade são projetadas as classes e objetos que irão realizar os casos de uso, bem como seus métodos, atributos, relacionamentos, generalizações e interações (JACOBSON; BOOCH; RUMBAUGH, 1999). Na atividade de projeto também são definidos o modelo físico da base de dados e os itens de projeto que servirão para satisfazer os requisitos não funcionais, como por exemplo, exigências de desempenho (KRUCHTEN, 2003).

No fluxo de implementação, as classes e objetos definidos no projeto são codificadas, compiladas, testadas e integradas ao restante do sistema (JACOBSON; BOOCH; RUMBAUGH, 1999). Neste fluxo são criados o código fonte, o código executável e *scripts*, tanto para protótipos nas fases iniciais como para versões finais do *software*.

2.3 Problemas Relacionados ao Projeto de *Software*

Para realizar um bom projeto, deve-se aplicar os princípios citados na Seção 2.1.1, em especial a diversificação e convergência (PRESSMAN, 2005). Diversificação e convergência, definidas na Seção 2.1.1, exigem intuição e julgamento. Essas qualidades são baseadas na experiência do projetista, em princípios, em critérios de qualidade e em um processo iterativo que levem a um resultado final satisfatório (PRESSMAN, 2005). Apesar disso, mesmo com a aplicação desses critérios, a convergência para uma solução é feita primeiramente com base na experiência do projetista. Além disso, critérios como: o projeto deve ser modular e os componentes devem ter independência funcional (PRESSMAN, 2005) não dão ao projetista critérios quantitativos para a tomada de decisões de projeto.

Segundo Booch (2006) "A vida de um arquiteto de *software* é uma longa e rápida sucessão de decisões subótimas de projeto tomadas parcialmente no escuro" (BOOCH, 2006). Essas decisões, quase sempre, precisam ser tomadas em um espaço de tempo bem curto, devido a um cronograma apertado gerado a partir de prazos estabelecidos com poucos critérios. Isto normalmente ocorre em uma organização com pouca maturidade de capacidade do processo de *software* (PAULK et al., 1991). Essa necessidade de tomar uma decisão de projeto em um curto espaço de tempo faz com que, nem sempre, a decisão tomada precise ser ótima. Além disso, o fato dessas decisões serem tomadas "parcialmente no escuro" significa que faltam critérios mais precisos para orientar decisões de projeto.

Muitas decisões de projeto são tomadas entre a concepção da visão do sistema até

chegar na criação do código executável. Para um sistema de *software* algumas dessas decisões de projeto avançam o progresso do desenvolvimento enquanto outras representam caminhos sem saída ou então que gerarão a necessidade de se refazer partes prontas (BOOCH, 2006). Essas decisões gerarão arquiteturas. Algumas dessas arquiteturas são intencionais, outras são acidentais.

Uma arquitetura intencional é identificada e desenvolvida de forma explícita. Neste caso, o projetista tinha a intenção de criar aquela arquitetura específica. Uma arquitetura acidental simplesmente aparece como conseqüência das decisões de projeto que ocorrem durante o desenvolvimento (BOOCH, 2006). Uma arquitetura acidental pode se tornar uma arquitetura intencional após ter sido comprovada sua eficácia com o decorrer do tempo. Neste caso, uma arquitetura acidental pode vir a ser útil, apesar de ter sido criada sem critérios muito definidos.

Em 1991, Pressman (1991) já alertava na terceira edição de seu livro que falta para as metodologias de projeto de *software* a profundidade, a flexibilidade e a natureza quantitativa das disciplinas de projeto de engenharia mais clássicas (PRESSMAN, 1995). Na sexta edição do mesmo livro, publicado em 2006, Pressman (2006) escreve que "mesmo atualmente, falta para a maioria das metodologias de desenvolvimento de *software* a profundidade, flexibilidade e a natureza quantitativa que estão normalmente associadas com disciplinas de projeto de engenharia mais clássicas" (PRESSMAN, 2005).

Projetar *software* é uma atividade complexa. Grande parte desta complexidade é inerente à própria natureza do *software*. Sistemas de *software* são mais complexos proporcionalmente ao seu tamanho, que qualquer outra construção humana. Computadores são algumas das máquinas mais complexas já construídas pelo homem. Eles possuem um número muito grande de estados. Sistemas de *software* possuem algumas ordens de grandeza a mais de estados que computadores (BROOKS, 1987).

Além de serem complexos por natureza, sistemas de *software* são constantemente sujeitos a pressões por mudança. A maior pressão por mudanças vem da necessidade de modificação da funcionalidade do sistema que é incorporada pelo *software*. Outra razão para essa pressão por mudanças se deve ao fato do *software* ser uma entidade flexível e que pode ser mudado com muito mais facilidade do que produtos manufaturados (BROOKS, 1987).

Para tentar administrar essa complexidade inerente de sistemas de *software*, emergiu um conceito chamado Independência Funcional (STEVENS; MEYERS; CONSTANTINE, 1974). Este conceito está intimamente ligado à modularidade, ocultação de informações e abstração (PRESSMAN, 1995). Em sistemas de *software*, a Independência

Funcional pode ser medida através de dois critérios: coesão e acoplamento.

"Coesão é uma medida da força funcional relativa de um módulo" (PRESSMAN, 1995). Em outras palavras a coesão mede o grau com que as tarefas executadas por um único módulo se relacionam entre si (IEEE, 1990). Para *software* orientado a objetos, coesão também pode ser conceituada como sendo o quanto uma classe encapsula atributos e operações que estão fortemente relacionados uns com os outros (PRESSMAN, 2005).

"Acoplamento é uma medida da interdependência relativa entre os módulos" (PRESSMAN, 1995). Para sistemas de *software* orientados a objetos pode-se definir acoplamento como sendo o grau com o qual classes estão conectadas entre si (PRESSMAN, 2005). Existem métricas definidas na literatura para coesão de um módulo (BIEMAN; OTT, 1994) e acoplamento entre módulos (DHAMA, 1995).

As métricas de coesão e acoplamento, tratam de avaliar como os componentes dependem uns dos outros. Neste caso, não são considerados como os requisitos funcionais, que levaram à criação destes componentes, são dependentes uns dos outros e nem se o conjunto de requisitos funcionais escolhido é apropriado. Quando se escolhe um conjunto de requisitos funcionais muito interdependente, os componentes da solução gerada tendem a ser interdependentes também.

O desenvolvimento de *software* orientado a objetos tem como um dos seus preceitos aumentar a coesão e diminuir o acoplamento entre os módulos do sistema. O desenvolvimento orientado a objetos facilita para o desenvolvedor criar componentes mais reutilizáveis (COLEMAN et al., 1996). Para isso, a orientação a objetos disponibiliza abstrações como classes, objetos, interfaces, atributos e métodos (WINCK; GOETTEN, 2006). Além disso, a orientação a objetos introduziu conceitos como encapsulamento, herança e polimorfismo para aumentar a reutilização e a extensibilidade e facilitar a manutenção de sistemas de *software* (COLEMAN et al., 1996).

Entre as decisões mais importantes em um projeto orientado a objetos estão a identificação das classes que comporão o sistema e a atribuição de responsabilidades para essas classes. Para tentar ajudar o projetista nessa tomada de decisões foram criados padrões. Um padrão é um par composto de um problema e a respectiva solução, que possui um nome e que pode ser aplicado em novos contextos, contendo diretrizes de como aplicá-lo a essas novas situações (LARMAN, 2004).

Os padrões tem por objetivo permitir que o desenvolvedor reutilize soluções já aplicadas e aceitas como boas soluções. Esta reutilização de soluções tem por objetivo diminuir o tempo e o esforço de desenvolvimento (PRESSMAN, 2005). Segundo Gamma et

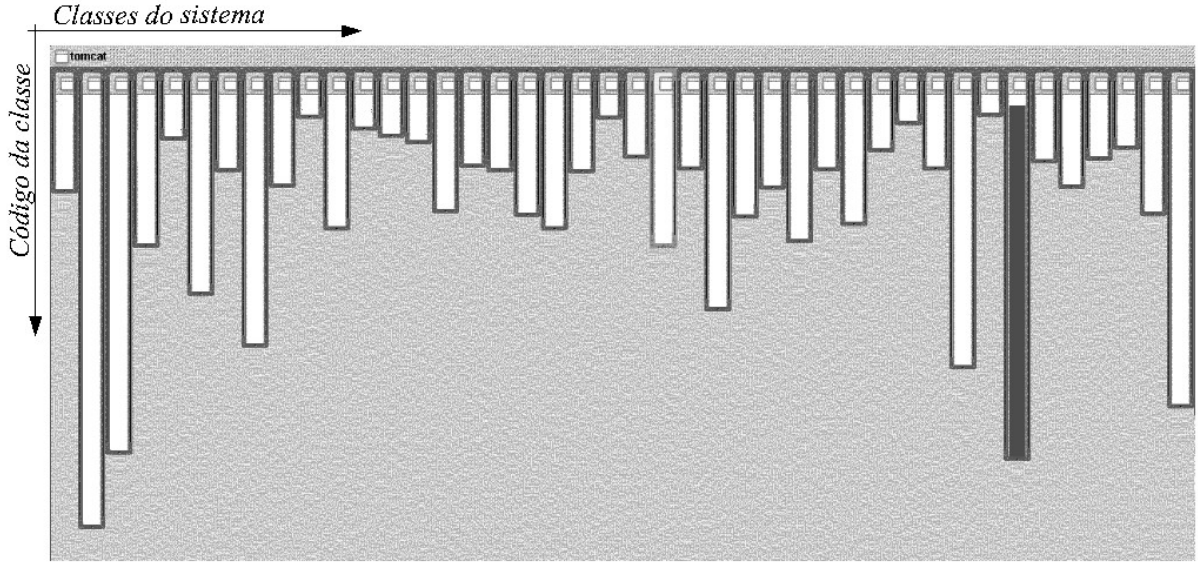
al. (2000), “Padrões de projeto tornam mais fácil reutilizar arquiteturas e experiências bem sucedidas” (GAMMA et al., 2000). Essas soluções podem ser tanto arquiteturas intencionais como acidentais.

No caso de arquiteturas acidentais, essas soluções com o tempo se transformam em arquiteturas intencionais (BOOCH, 2006), ou então, são aceitas até que deixem de ser soluções interessantes. Os padrões de projeto representam soluções de projeto que se mostraram eficientes. No caso de soluções acidentais, muitas vezes elas surgem de decisões de projeto tomadas sem a aplicação de critérios mais precisos. Em certos casos, um padrão de projeto pode representar uma arquitetura acidental, que não necessariamente representa uma solução ótima.

Por outro lado, a orientação a objetos fornece um modo eficaz de representar os elementos inerentes ao domínio do negócio como dos componentes da solução (WINCK; GOETTEN, 2006). As abstrações como classes e objetos permitem que o desenvolvedor crie componentes da solução que representem entidades do mundo real relativas ao negócio a ser automatizado. No entanto, não disponibiliza um modo fácil quando se quer representar elementos que não estão diretamente relacionados com o negócio no mundo real, mas fazem parte da solução (WINCK; GOETTEN, 2006). Esses elementos servem muitas vezes como suporte à solução, como por exemplo o tratamento de exceções.

As características relevantes de uma aplicação, agrupadas por similaridade são chamadas de interesses (*concerns*) (WINCK; GOETTEN, 2006). Os interesses representam requisitos do sistema. Um tipo de interesse fica totalmente contido em um ou dois componentes do sistema. Esse tipo de interesse normalmente está fortemente relacionado com o domínio da aplicação. Por exemplo, no *software* do servidor WEB apache TOMCAT[3], a funcionalidade de interpretação de código XML[4] está implementada em apenas uma classe como ilustrado na Figura 2 (WINCK; GOETTEN, 2006). A Figura 2 e a Figura 3 (HILSDALE; KERSTEN, 2004) representam as classes do *software* do servidor TOMCAT. Cada barra vertical representa uma classe e as regiões escuras em cada uma das barras verticais representam a localização do código responsável pela implementação do interesse em cada uma das classes. Na Figura 2, as regiões escuras representam a implementação da funcionalidade de interpretação de código XML. Pode-se ver que toda a implementação desta funcionalidade está concentrada em uma única classe.

[3] Apache Tomcat é um projeto da Apache *Software* Foundation. http://www.apache.org
[4] A eXtensible Markup Language é um padrão do World Wide Web Consortium (W3C)

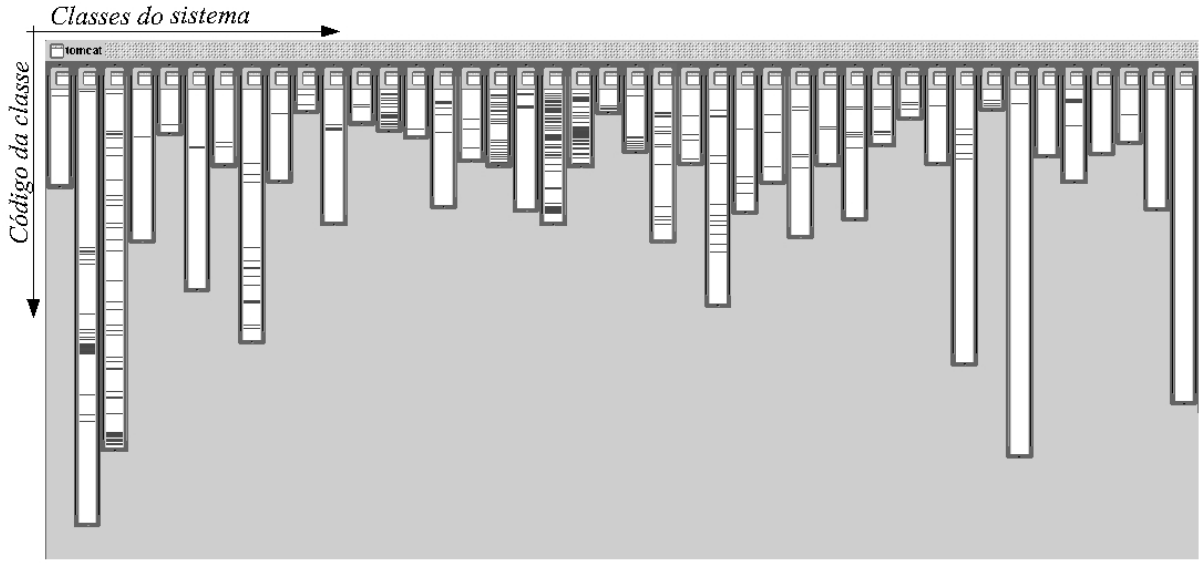


Fonte: (HILSDALE; KERSTEN, 2004)

Figura 2 - Ocorrência da funcionalidade "interpretação de XML" no TOMCAT

Em uma aplicação orientada a objetos existem muitos requisitos que não podem ser colocados em um componente individual e que algumas vezes até impactam em muitos outros componentes (JACOBSON; NG, 2004). Este tipo de requisito é chamado de interesse transversal (*Crosscutting Concern*) ou interesse sistêmico (WINCK; GOETTEN, 2006). São exemplos de interesses transversais: sincronização de objetos concorrentes, distribuição da solução, tratamento de exceções, persistência de objetos, auditoria, entre outros. Por exemplo, no mesmo *software* do servidor TOMCAT, a funcionalidade de Auditoria do sistema está espalhada em várias classes, como ilustrado na Figura 3. Nesta figura, as regiões escuras representam a implementação da funcionalidade de auditoria do sistema. Pode-se ver que as regiões escuras estão presentes em várias barras, e portanto, a implementação desta funcionalidade está espalhada por várias classes do sistema.

A orientação a objetos tem dificuldade em modularizar os interesses transversais. Esta dificuldade gera dois novos problemas que são o código emaranhado (*tangled Code*) e o espalhamento de código (*scattering Code*). O código emaranhado acontece quando vários interesses transversais são implementados em uma única classe, em vez de serem criadas novas classes para tal. Inversamente, o espalhamento de código acontece quando o código, para implementar um interesse, se encontra propagado por várias classes, em vez de estar encapsulado em uma única (WINCK; GOETTEN, 2006).



Fonte: (HILSDALE; KERSTEN, 2004)

Figura 3 - Ocorrência da funcionalidade de auditoria no TOMCAT

Tanto o código emaranhado quanto o espalhamento de código prejudicam a coesão e o acoplamento do sistema. O código emaranhado diminui a coesão porque quando uma classe implementa vários interesses, o grau com que as tarefas realizadas por esta classe estão relacionadas diminui e esta classe poderia ser dividida em classes mais coesas. Por outro lado, o código espalhado aumenta o acoplamento do sistema pois qualquer mudança na implementação do interesse implica na alteração de várias classes. O aparecimento de código espalhado e código emaranhado na implementação de um sistema ocasiona os seguintes problemas: replicação do código, dificuldade de manutenção, redução da capacidade de reutilização de código e aumento da dificuldade de compreensão do código (WINCK; GOETTEN, 2006).

Outro tipo de interesse transversal é a extensão. Uma extensão representa um serviço ou característica adicional a ser implementada no sistema (JACOBSON; NG, 2004). O conceito de interesse de extensão é análogo ao conceito de extensão de casos da uso, onde "o caso de uso que estende define um conjunto modular de incrementos de comportamento que aumentam uma execução do caso de uso estendido sob condições específicas" (OBJECT MANAGEMENT GROUP, 2005).

Em outras palavras, um interesse de extensão aumenta o comportamento de outro interesse sob determinadas condições. Isto gera a necessidade de se colocar um fragmento de código onde não era necessário antes. É o chamado código cola (*glue code*) (JACOBSON; NG, 2004). Adicionar este tipo de código torna as classes originais difíceis de serem

compreendidas e as extensões difíceis de serem identificadas a primeira vista (JACOBSON; NG, 2004).

A inabilidade do desenvolvimento orientado a objetos em representar de forma modular interesses transversais pode estar relacionado com uma falta de critérios mais precisos na hora de identificar e selecionar o conjunto de requisitos funcionais adequado. No desenvolvimento orientado a objetos usando a UML (OBJECT MANAGEMENT GROUP, 2005) e o processo unificado (JACOBSON; BOOCH; RUMBAUGH, 1999), os requisitos funcionais são capturados e organizados através dos casos de uso. Os casos de uso foram introduzidos por Jacobson (JACOBSON, 1987) e se tornaram uma técnica popular entre os desenvolvedores.

Um caso de uso é uma descrição de um conjunto de ações, incluindo variantes, que um sistema executa para produzir um resultado de valor observável para o ator (BOOCH; RUMBAUGH; JACOBSON, 2005). A modelagem de casos de uso é uma técnica que captura e organiza os requisitos funcionais em função do resultado produzido para o usuário. Devido a este fato, usar os casos de uso para capturar e organizar os requisitos funcionais pode gerar muitas dificuldades para lidar com interesses transversais já que estes representam elementos que não estão diretamente relacionados com o negócio no mundo real e servem muitas vezes como suporte à solução.

Para tentar amenizar essa dificuldade (JACOBSON; NG, 2004) propuseram novos tipos de casos de uso como os casos de uso de utilidade (*utility use cases*) e os casos de uso de infra-estrutura (*infraestructure use cases*). Casos de uso de utilidade representam funcionalidades que não estão diretamente relacionadas com a interação com o usuário mas servem como suporte à execução do sistema. Casos de uso de infraestrutura representam requisitos não funcionais do sistema que necessitam de tratamento pelo sistema.

A escolha incorreta do conjunto de requisitos funcionais para o projeto, pode causar muitos problemas para o desenvolvimento do *software*. Ela pode prejudicar a independência funcional dos componentes que é uma diretriz de um bom projeto. Portanto, a falta de critérios para escolher um conjunto adequado de requisitos funcionais pode comprometer a qualidade do produto de *software* a ser gerado pelo projeto.

2.4 Técnicas de Engenharia

A pesquisa sobre projeto de engenharia começou, de forma mais sistemática, na Alemanha, na metade do século XIX. Muitas técnicas de projeto de engenharia foram desenvolvidas desde então. Entre elas estão teorias como a da Resolução Inventiva de Problemas (TRIZ) (SAVRANSKI, 2000), Desdobramento de Função de Qualidade[5] (QFD) (CLAUSING, 1994), o Projeto Robusto de Taguchi (PADKE, 1989) e a Teoria do Projeto Axiomático (SUH, 1990).

Além dos trabalhos citados acima, existem trabalhos que procuram estabelecer uma teoria de projeto. Eles abordam o problema geral de projeto, estabelecendo um modelo geral de projeto para deduzir, através de lógica, modelos para problemas concretos de projeto. Entre estes trabalhos estão: a teoria geral de projeto (YOSHIKAWA, 1981), a teoria formal de projeto (BRAHA; MAIMON, 1998), a teoria axiomática de informação de projeto (SALUSTRI; VENTER, 1992) e a teoria axiomática de modelagem de projeto (ZENG, 2003).

Das técnicas de projeto de engenharia citadas acima, poucas possuem trabalhos a respeito de sua aplicabilidade ao desenvolvimento de *software*, entre essas, as mais importantes são: o Projeto Axiomático (SUH, 1990), a TRIZ (SAVRANSKI, 2000) e o QFD (CLAUSING, 1994). O desdobramento da função de qualidade (QFD) é citado por Pressman (2006) como uma técnica para elicitar e organizar requisitos para um sistema de *software* (PRESSMAN, 2005). A aplicação da TRIZ envolve um processo criativo do projeto, que inclui a criação de várias soluções para um problema. Entretanto, nenhuma delas fornece critérios para tomada de decisões de projeto como selecionar a melhor entre as várias possíveis soluções. Neste caso, a Teoria de Projeto Axiomático (SUH, 1990), a ser apresentada na Seção 2.5 é a técnica que fornece estes critérios para tomada de decisão. Existem alguns trabalhos que relacionam a teoria inventiva de resolução de problemas (TRIZ) com o desenvolvimento de *software* (RAWLINSON, 2001)(REA, 2001a)(REA, 2001b)(REA, 2002)(WISSE, 2001). Esta seção apresenta um resumo do QFD e da TRIZ e uma discussão sobre sua aplicabilidade no desenvolvimento de *software*.

2.4.1 Desdobramento da Função de Qualidade (QFD)

O método do desdobramento da função de qualidade (QFD) foi desenvolvido na empresa Mitsubishi no Japão por Y. Akao e S. Mizuno (PRASAD, 1996) (AKAO, 1990).

[5] Do inglês *Quality Function Deployment*

Mais tarde, a Toyota usou a técnica da Casa da Qualidade[6] (HOQ) do QFD para identificar e priorizar as necessidades do cliente, relacioná-las com características de engenharia, compará-las com os produtos concorrentes e estabelecer quais características de engenharia são mais importantes e quais são as áreas importantes de melhoria (SUH, 2001)(GUMMUS, 2005). Com a aplicação do QFD e HOQ nos anos entre 1977 e 1984, a Toyota reduziu os custos de desenvolvimento de produto em 61%, o ciclo de desenvolvimento em um terço e virtualmente eliminou os problemas de garantia relacionado com ferrugem (SULLIVAN, 1986).

QFD é um método que engloba todo o processo de desenvolvimento de um produto (PRASAD, 1996). A Casa da Qualidade é a técnica mais conhecida e mais usada do QFD. Entre as outras técnicas do QFD tem-se: diagramas de afinidade, diagramas de inter-relacionamentos, árvore de hierarquia, matrizes e tabelas, diagramas de programa de decisão de processo e processo analítico hierárquico (GUMMUS, 2005).

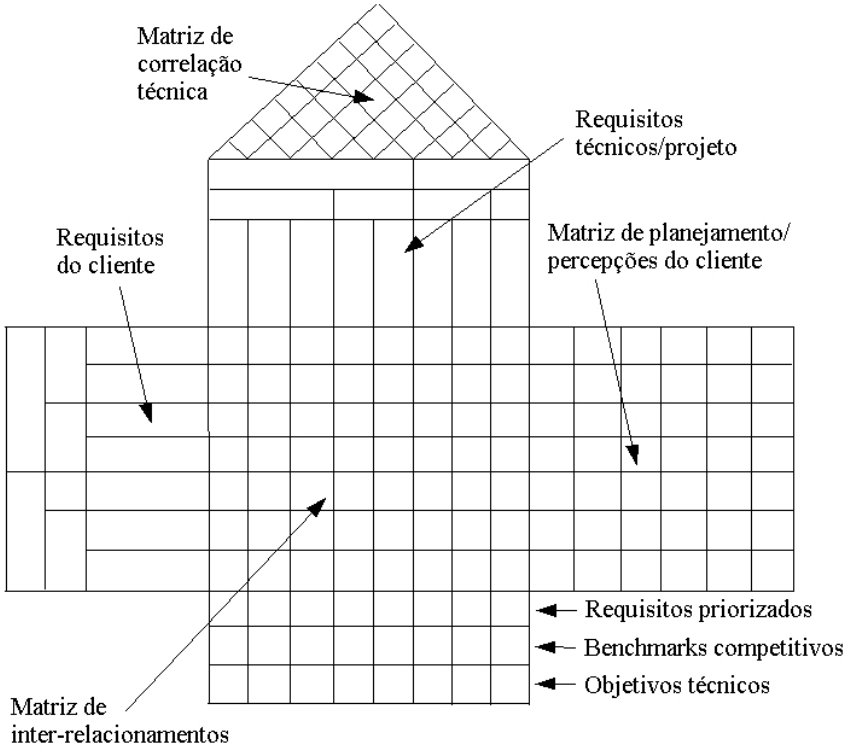
2.4.1.1 Casa da Qualidade (HOQ)

A Casa da Qualidade é uma matriz que traduz o conjunto dos requisitos do cliente, pesquisas de mercado e informações de comparações técnicas em um conjunto de características de engenharia organizadas de acordo com sua prioridade que serão usadas no desenvolvimento de um novo produto. Esta matriz, apresentada na Figura 4, é dividida em diferentes regiões: requisitos do cliente, requisitos técnicos, matriz de planejamento, matriz de inter-relacionamentos, matriz de correlação técnica e prioridades, comparações e alvos técnicos (GUMMUS, 2005).

A Casa da Qualidade funciona da seguinte maneira. As necessidades do cliente são organizadas na forma de uma lista na região dos requisitos do cliente. Estes requisitos do cliente são quantificados em termos de prioridade e percepção de desempenho em relação a outros produtos na região da matriz de planejamento. Os requisitos técnicos do produto são listados e organizados na região de requisitos técnicos. A região da matriz de correlação técnica é usada para representar como os requisitos técnicos interferem uns com os outros. Em seguida as relações entre os requisitos do cliente e os requisitos técnicos são quantificadas em termos de sua importância para o projeto na matriz de inter-relacionamentos. As conclusões são sumarizadas na parte de prioridades, comparações e alvos técnicos.

---

[6] Do inglês *House of Quality*



Fonte: (GUMMUS, 2005)

Figura 4 - Estrutura da casa da qualidade (HOQ)

2.4.1.2 QFD e o Desenvolvimento de *Software*

O QFD é uma técnica que pode ser usada para traduzir as necessidades dos clientes em requisitos técnicos de *software* (PRESSMAN, 2005). Segundo Zultner (1993) "QFD se concentra em maximizar a satisfação do cliente a partir do processo de engenharia de *software*" (ZULTNER, 1993). Uma técnica para aplicar QFD para o desenvolvimento de *software* é a SQFD. O objetivo da *Software Quality Function Deployment* (SQFD) é melhorar o desenvolvimento de *software* aplicando técnicas de melhoria de qualidade durante a especificação de requisitos. Para isto, as necessidades dos clientes são confrontadas com as restrições próprias do projeto de forma a concentrar os melhores esforços nos aspectos com maior importância (KROGSTIE, 1999). Haag; Raja e Schkade fizeram um comparativo sobre as vantagens do uso de SQFD em projetos de *software* onde o SQFD se mostrou mais eficiente que as metodologias tradicionais (HAAG; RAJA; SCHKADE, 1996). O SQFD se apresentou vantajoso na comunicação com o pessoal técnico, com os usuários e gerentes, em encontrar os requisitos dos usuários, em desenvolver sistemas relativamente livres de erros e

em criar uma documentação mais completa e consistente (HAAG; RAJA; SCHKADE, 1996).

2.4.2 Teoria Inventiva de Resolução de Problemas (TRIZ)

TRIZ é um acrônimo para Teoria Inventiva para Resolução de Problemas[7]. A TRIZ pode ser conceituada como "uma metodologia sistemática, orientada ao ser humano, baseada em conhecimento para a solução de problemas inventivos" (SAVRANSKI, 2000). É uma metodologia que fornece heurísticas genéricas para a solução de problemas. A TRIZ é voltada para a solução de problemas cuja solução exige um certo grau de inventividade por parte do projetista. Com a aplicação da TRIZ é possível identificar soluções diferentes das soluções convencionais para um problema. A TRIZ tem por objetivo aplicar o conhecimento acumulado a respeito de projetos e patentes já realizados durante anos para auxiliar o projetista a criar uma nova solução para um problema.

Em meados do século XX, o engenheiro russo Altshuller, que era um especialista em patentes da marinha soviética na época, começou a analisar mais de 400.000 patentes procurando por problemas inventivos e como estes foram solucionados (SAVRANSKI, 2000). Com base neste trabalho de análise e categorização das soluções, Altshuller formulou uma teoria de projeto para produtos e invenções que visa ajudar os projetistas a encontrar soluções que não seriam consideradas usando-se as técnicas convencionais de projeto.

A TRIZ é baseada em alguns princípios básicos e ferramentas, entre eles: idealidade, princípios inventivos, matriz de contradições, análise Campo-Substância, soluções padrão e as leis de evolução. Segundo (SAVRANSKI, 2000), estes conceitos são a pedra fundamental da TRIZ.

O princípio da idealidade é usado para que se possa ter um parâmetro de comparação entre as soluções encontradas e a solução ideal. Segundo Carvalho e Black, os princípios inventivos são heurísticas, ou sugestões, de possíveis soluções para um determinado problema (CARVALHO; BLACK, 2001).

A matriz de contradições é uma ferramenta para facilitar a identificação dos princípios inventivos que podem ser aplicados na solução de um problema (SAVRANSKI, 2000). A matriz de contradições, encontra princípios inventivos que podem ser aplicados ao projeto para solucionar uma contradição física, formulada em termos dos parâmetros de engenharia.

O objetivo da análise Campo-Substância é modelar problemas relacionados com

[7]Do russo transliterado "*Teoriya Resheniya Izobretatelskikh Zadatch*".

sistemas tecnológicos existentes. Ela pode ser aplicada, tanto para sistemas complexos como para partes de um sistema. A análise Campo-Substância permite a identificação de falhas ou deficiências nos processos do sistema.

Há 76 soluções padrão que são soluções genéricas aplicáveis como modelo para solucionar o problema relacionado. Elas contêm a descrição concisa para a situação genérica e a descrição das restrições. As leis de evolução da técnica, por sua vez, são padrões de evolução de sistemas técnicos. Estes padrões de evolução podem ser usados para ver o progresso evolucionário do sistema e antecipar possíveis efeitos indesejáveis (SAVRANSKI, 2000).

2.4.2.1 TRIZ e Projeto de *Software*

A TRIZ foi inicialmente formulada baseada em projetos de engenharia mecânica. Devido a esse fato, os princípios inventivos e os parâmetros de engenharia são voltados para objetos reais, como parafusos, chapas de aço, energia elétrica e calor. Rawlinson (2001) apresenta uma forma para a aplicação dos conceitos da TRIZ no desenvolvimento de *software* (RAWLINSON, 2001). Nesse artigo são propostas analogias entre conceitos da TRIZ e conceitos de desenvolvimento de *software*, considerando como recursos os componentes do sistema (*software* e *hardware*) (RAWLINSON, 2001).

Rea (2001a) (2001b) apresentou analogias para os 40 princípios inventivos no contexto de *software* (REA, 2001a)(REA, 2001b). Em outro artigo, Rea (2002) propõe a extensão dos conceitos da análise campo-substância para poderem ser usados em problemas de *software* (REA, 2002). Como substância, ele entende, também, qualquer tipo de objeto ou dado de um sistema, e como campo, qualquer tipo de ação sobre um objeto ou dado (procedimento, função ou método). Neste mesmo artigo, é proposta a utilização de *MetaPatterns* (WISSE, 2001) para a modelagem dos objetos, em função do tempo e do contexto onde estão sendo utilizados, visando aprimorar a análise campo-substância (REA, 2002).

A TRIZ pode ser uma ferramenta muito útil para a identificação e solução de problemas. Com a TRIZ é possível encontrar soluções que não seriam identificadas por outras metodologias como o Desdobramento da Função de Qualidade. Estas soluções inovadoras, não fazem parte das soluções usadas pelos especialistas da área, e podem trazer vantagens sobre as soluções tradicionais.

2.5 Introdução ao Projeto Axiomático

Nesta Seção apresenta-se uma introdução à Teoria de Projeto Axiomático. O contexto histórico da criação do Projeto Axiomático é apresentado e discutido na subseção 2.5.1. A subseção 2.5.2 apresenta os objetivos principais e características do Projeto Axiomático, incluindo um exemplo. A subseção 2.5.3 ilustra algumas aplicações do Projeto Axiomático na indústria.

2.5.1 Contexto Histórico

A Teoria de Projeto Axiomático começou a ser desenvolvida no final dos anos 70. No final dos anos 50 e início dos anos 60 existia uma percepção de que os Estados Unidos da América do Norte estavam cientificamente atrasados com relação à União Soviética devido ao sucesso do programa espacial soviético. Esta percepção de atraso foi atribuída à falta de se enfatizar os aspectos científicos no ensino de engenharia. Nesse sentido, se fazia necessário estabelecer uma base científica para que os campos de projeto e manufatura, que normalmente possuem um embasamento empírico, se tornem mais próximos de disciplinas científicas (SUH, 1990).

Para estabelecer essa base científica, Suh decidiu tentar identificar possíveis axiomas para a ciência de projeto (SUH, 1990). Este trabalho foi realizado no departamento de engenharia mecânica do *Massachusets Institute of Technology* (MIT). Do ponto de vista da lógica matemática, os axiomas representam as fórmulas básicas de uma teoria (CASANOVA; GIORNO; FURTADO, 1987). Axiomas são verdades que não podem ser derivadas mas para as quais não existem contra-exemplos e exceções (SUH, 2001). Para as ciências naturais como física, química e biologia, os axiomas representam princípios fundamentais, como por exemplo as leis da mecânica de Newton, e são tomados como base para a dedução de outras propriedades (SUH, 2001).

Existem similaridades entre projetos feitos para as mais diversas áreas da atividade humana. Por exemplo, o que existe em comum entre o projeto de um produto mecânico e o projeto de um órgão governamental é que as necessidades são satisfeitas por entidades que provêem o resultado desejado (SUH, 1990). Essas similaridades vêm da existência de princípios básicos inerentes à atividade de projeto e não ao domínio específico de cada projeto.

Na tentativa de identificar a existência de princípios básicos para a atividade de

projeto, a Teoria de Projeto Axiomático de Suh começou com a seguinte pergunta: "Dado um conjunto de requisitos funcionais para um determinado produto, existem axiomas de aplicação genérica que levam a decisões corretas [...] para planejar um sistema de produção ótimo?" (SUH; BELL; GOSSARD, 1978).

Para responder a esta pergunta foi elaborado um conjunto de axiomas que foram testados e avaliados em estudos de caso. Os 12 axiomas iniciais foram reduzidos a apenas dois axiomas e um conjunto de corolários e teoremas (SUH, 1990). Os dois axiomas, os corolários e os teoremas derivados provêem para o projetista um conjunto de princípios, regras e orientações para facilitar a tomada de decisões durante o projeto.

2.5.2 Objetivos da Teoria de Projeto Axiomático

Segundo Suh (2001), o objetivo principal da Teoria de Projeto Axiomático é "...estabelecer uma base científica para projeto e aprimorar as atividades de projeto provendo ao projetista uma fundamentação teórica baseada em ferramentas e processos do pensamento lógico e racional." (SUH, 2001) Além disso, esta teoria deve ter uma natureza independente de domínio para poder ser aplicada em projetos nas mais diversas áreas do conhecimento humano. Dentre os objetivos mais específicos, Suh (2001) enumera os seguintes: tornar projetistas humanos mais criativos, reduzir o processo randômico de busca de novas soluções, minimizar o processo iterativo de tentativa e erro e determinar as melhores soluções de projeto entre as que foram propostas (SUH, 2001).

Segundo Suh (2001), o Projeto Axiomático aprimora a criatividade do projetista porque exige do projetista uma definição clara dos objetivos do projeto e, também, porque fornece critérios para ajudar a eliminar idéias ruins o mais cedo possível, possibilitando que o projetista se concentre nas idéias mais promissoras (SUH, 2001). Caldenfors (1998) mostra através de estudos sistemáticos que, na questão de aprimorar a criatividade em projetos, os estudantes que aprenderam a Teoria de Projeto Axiomático obtiveram resultados melhores que estudantes que aprenderam outros métodos (CALDENFORS, 1998).

Um exemplo de aplicação da Teoria de Projeto Axiomático é o caso do projeto de uma torneira para água quente e fria onde se quer controlar a temperatura da água e a sua vazão (SUH, 2001). Neste exemplo, são consideradas duas alternativas para o projeto da torneira e se quer determinar qual das duas soluções é o melhor projeto. As duas alternativas estão apresentadas na Figura 5.

Na solução 1 o controle da temperatura e da vazão é feito através da regulagem

simultânea da abertura dos registros de entrada de água quente e de entrada de água fria. Neste caso, qualquer ação em qualquer um dos registros afeta tanto a temperatura quanto a vazão. Na solução 2, a torneira possui um registro que controla independentemente a vazão e a temperatura. Se o registro é movido verticalmente, ele controla somente a vazão e se movido horizontalmente ele controla somente a temperatura. Neste caso, consegue-se aumentar ou diminuir a temperatura sem afetar a vazão e vice-versa.

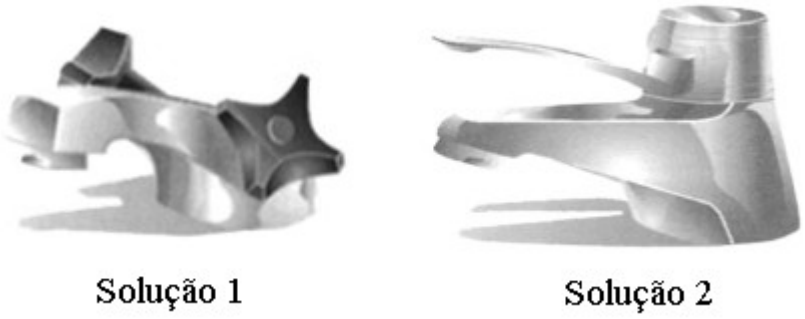


Figura 5 - Duas soluções alternativas para o projeto da torneira

Para se definir qual das duas soluções é a que satisfaz melhor as necessidades, é necessário definir quais são as necessidades. Para este caso as necessidades são: (1) controlar a temperatura da água e (2) controlar a vazão da água. A teoria de Projeto Axiomático fornece critérios para determinar qual dessas soluções é a melhor.

Um dos axiomas é o Axioma da Independência (Axioma 1)[8] que diz que os requisitos funcionais devem ser mantidos independentes. Para esta análise, as necessidades (1) e (2) serão consideradas como requisitos funcionais. Na solução 1, quando se abre ou fecha qualquer um dos registros, isto afeta os dois requisitos funcionais ao mesmo tempo, o que não permite satisfazer os requisitos funcionais de maneira independente. Entretanto, na solução 2 pode-se satisfazer o requisito funcional de controle da temperatura sem afetar a vazão da água e vice-versa. Portanto, apenas na solução 2 os requisitos são independentes, o que a caracteriza como uma boa solução de projeto.

2.5.3 Utilização da Teoria de Projeto Axiomático

Os axiomas, teoremas e corolários da Teoria de Projeto Axiomático fornecem princípios e critérios práticos para tomada de decisão de projeto. Esta característica, e o fato de ter sido inicialmente desenvolvida no escopo da engenharia mecânica, levou o Projeto

[8] Os axiomas da teoria de projeto axiomático estão definidos no capítulo 3

Axiomático a ter diversas aplicações na indústria, sendo as mais significativas na indústria automobilística. Por exemplo, trabalhos em cooperação com grandes empresas automobilísticas como a FORD e a FIAT para o projeto de partes de veículos são apresentados em (ARCIDIACONO et al, 2004) e (DEO; SUH, 2004). Em outros campos, a aplicação do Projeto axiomático vai até, por exemplo, a indústria naval militar (WHITCOMB; SZATKOWSKI, 2000).

A natureza flexível e independente de domínio da Teoria de Projeto Axiomático permite que ela seja aplicada em conjunto com outras metodologias de projeto como TRIZ e Projeto Robusto (HU; YANG; TAGUCHI, 2000) e o QFD (GUMMUS, 2005). Esta mesma razão permite uma fácil integração com metodologias de gerenciamento de processo como o *Six Sigma* (ARCIDIACONO, 2006).

Entre as principais características da Teoria de Projeto Axiomático estão: fornecer critérios práticos e precisos para a tomada de decisão de projeto, ser independente de domínio e ser facilmente adaptável a outras teorias e metodologias. Estas características tornam o Projeto Axiomático uma abordagem muito promissora para ser aplicada em conjunto com o Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999) para aprimorar o desenvolvimento de *software* orientado a objetos.

2.6 Conclusões

Este Capítulo apresentou uma discussão sobre o que é projeto, sua importância e a importância da qualidade do projeto na qualidade do produto. As principais características de qualidade de um projeto de *software* foram apresentadas e relacionadas aos principais problemas e dificuldades em projeto de *software* orientado a objetos. Foi apresentado um resumo das principais teorias de projeto de engenharia que possuem aplicabilidade em projeto de *software* como TRIZ e QFD, sua aplicação em projeto de *software* e a apresentação e contextualização da teoria de Projeto Axiomático.

A finalidade de um projeto é traduzir necessidades, ou requisitos, em termos de soluções tecnológicas que satisfaçam essas necessidades, de forma a obter um produto com qualidade. A escolha do conjunto adequado de requisitos funcionais é fundamental para a realização de um bom projeto e para a criação de um produto com qualidade, seja ele uma edificação, um automóvel, um computador ou mesmo um sistema de *software*. Outra escolha

fundamental é a de qual dentre as possíveis soluções tecnológicas identificadas é a melhor para para satisfazer os requisitos funcionais.

Das técnicas de projeto de engenharia citadas neste Capítulo, poucas possuem trabalhos a respeito de sua aplicabilidade no desenvolvimento de *software.* Dentre estas técnicas, a Teoria de Projeto Axiomático se mostra a mais promissora para a aplicação em conjunto com um processo de desenvolvimento de *software* orientado a objetos porque: provê critérios práticos e precisos para a tomada de decisão de projeto, é independente de domínio, é facilmente adaptável a outras teorias e metodologias e possui vários trabalhos a respeito da sua aplicação em projeto de *software*.

3 Projeto Axiomático

A Teoria de Projeto Axiomático tem como base um conjunto de axiomas, corolários e teoremas que representam princípios, critérios e regras para a realização de bons projetos (SUH, 1990). Para a sua aplicação em projetos, além dos axiomas, teoremas e corolários, são também necessários definições de conceitos, processos e artefatos.

Este Capítulo apresenta um resumo da Teoria de Projeto Axiomático. A primeira Seção apresenta uma introdução à teoria e suas definições básicas. Na segunda Seção descrevem-se os principais conceitos do Projeto Axiomático relacionados com o Axioma da Independência. Na terceira Seção são apresentados os principais conceitos relacionados com o Axioma da Informação. Na quarta Seção são apresentados trabalhos relacionados com a Teoria de Projeto Axiomático e desenvolvimento de *software*.

3.1 Definições Iniciais

A Teoria de Projeto Axiomático tem como base dois axiomas e um conjunto de oito corolários e teoremas relacionados (ver Anexo 1). Estes axiomas, corolários e teoremas dependem de definições de conceitos básicos para que possam ser compreendidos e aplicados corretamente. Nesta Seção são apresentados os principais conceitos e definições, além dos dois axiomas que formam o núcleo da Teoria da Projeto Axiomático.

3.1.1 Requisitos Funcionais, Parâmetros de Projeto e Restrições

Entre os conceitos que servem de base para a Teoria de Projeto Axiomático estão:

- Atributos do cliente
- Requisitos funcionais
- Parâmetros de projeto
- Variáveis de processo

- Restrições

Os atributos do cliente (CA, do inglês, *customer attributes*) representam as necessidades dos clientes. Estas necessidades freqüentemente nascem de um problema ou da percepção do cliente para um problema. Estas necessidades podem ser descritas através de características ou atributos, do sistema ou produto, desejados pelo cliente. São exemplos de atributos do cliente: desempenho, satisfação dos clientes, lucro, tolerância, flexibilidade, funcionalidades do *software* e respostas desejadas do *software*. Como as necessidades do cliente refletem o negócio do cliente, elas frequentemente são colocadas de maneira genérica, necessitando serem organizadas e, às vezes, refinadas. Isto pode ser feito mais formalmente, usando-se métodos como QFD (ver seção 2.2.1).

Suh (1990) define o termo "função", ou funcionalidade, como sendo a saída ou resposta desejada. Para Suh, os requisitos funcionais são "...a caracterização pelo projetista das necessidades percebidas para um produto" (SUH, 1990). Então, os atributos do cliente (CAs) são mapeados em termos dos requisitos funcionais (FR) (SUH, 2001). Nestes termos, um requisito funcional (FR, do inglês, *functional requirement*) pode ser definido como sendo uma saída ou resposta requerida de um produto que caracteriza uma necessidade percebida para este produto. Para sistemas de *software*, um requisito funcional expressa o comportamento do sistema, suas entradas, saídas e a funcionalidade provida ao usuário (LEFFINGWELL; WIDRIG, 2003).

O conjunto dos requisitos funcionais (FRs) é o conjunto mínimo de requisitos independentes que especificam completamente as necessidades do produto no domínio funcional (SUH, 2001). Como exemplo de requisitos funcionais tem-se: prover acesso ao interior, movimentar o veículo, suportar a carga e manter a carga imóvel. No caso de sistemas de *software*, podem ser considerados requisitos funcionais (FR): registrar um cliente, registrar uma venda, mostrar o relatório de acessos, entre outros. Os requisitos funcionais (FR) têm um papel importante na engenharia de *software*. Na maioria dos modelos de ciclo de vida para *software* existe uma etapa ou atividade relacionada com a definição e especificação de requisitos funcionais e não funcionais (PETERS; PEDRYCZ, 2001).

Segundo Suh (2001), os parâmetros de projeto (DP, do inglês, *design parameter*) são as variáveis chaves no domínio físico que caracterizam que o projeto satisfaz os requisitos funcionais (FRs) especificados (SUH, 2001). Os parâmetros de projeto são as variáveis que são criadas de forma a satisfazer os requisitos funcionais. Como exemplo, pode-se ter: porta, motor, material de isolamento, eixos, paredes, suportes e, no caso de sistemas de *software*, objetos, funções, programas, algoritmos, entre outros.

Variáveis de processo (PV, do inglês, *process variable*) são as variáveis chaves no domínio de processo que caracterizam como o processo de produção pode gerar, ou realizar, os parâmetros de projeto (DP) especificados. No caso de sistemas de *software*, as variáveis de processo podem ser: linhas de código, variáveis locais, compiladores, entre outros.

As Restrições (C, do inglês, *constraint*) são limites para as soluções aceitáveis. Existem dois tipos de restrições: de entrada e de sistema. As restrições de entrada são colocadas como parte das especificações do projeto. As restrições de sistema são impostas pelo sistema no qual a solução do projeto deverá funcionar (SUH, 2001). Em engenharia de *software* as restrições são consideradas como requisitos não funcionais e são uma parte importante da Especificação de Requisitos de *Software* (PETERS; PEDRYCZ, 2001).

3.1.2 Domínios

Na Teoria de Projeto Axiomático, o desenvolvimento do projeto é realizado através do mapeamento entre os domínios do cliente, funcional, físico e de processo (SUH, 2001). Este mapeamento é ilustrado na Figura 6.

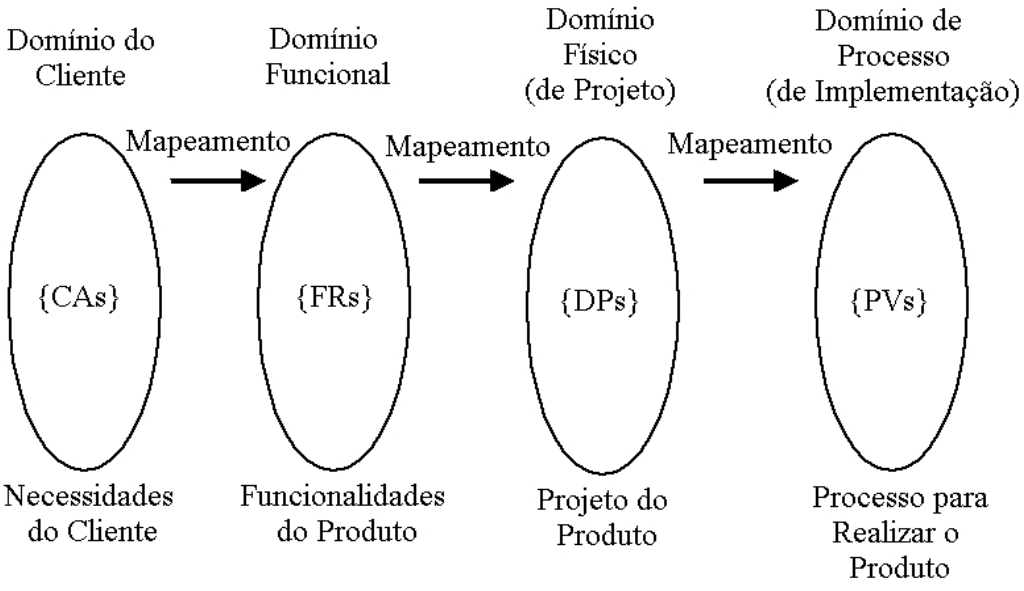


Figura 6 - Domínios do Projeto Axiomático

O domínio do cliente representa o conjunto das necessidades do cliente representadas pelos atributos dos clientes (CA). O domínio funcional representa as exigências impostas ao produto a partir das necessidades dos clientes que são especificadas pelo conjunto

dos requisitos funcionais (FR) do sistema. Neste domínio estarão representadas as funcionalidades desejadas para o sistema. O domínio físico, ou domínio de projeto, representa o conjunto dos parâmetros de projeto que satisfazem os requisitos funcionais. Neste domínio estão representados os componentes da solução do projeto.

O domínio de processo, ou domínio de implementação, é o conjunto dos processos, representados pelas variáveis de processo, que realizam os parâmetros de projeto (SUH, 2001). Neste domínio estão representados aspectos relacionados com a realização do projeto que envolve a manufatura do produto.

Mapeamento é o processo que relaciona um elemento de um domínio com um, ou mais elementos do outro domínio. Por exemplo, o mapeamento entre o domínio funcional e o físico relaciona um requisito funcional com um parâmetro de projeto, usando a matriz de projeto. O mapeamento entre os domínios é realizado em uma determinada seqüência. Inicialmente, é feito o mapeamento entre o domínio do cliente e o domínio funcional. A seguir é feito o mapeamento entre o domínio funcional e o físico. Por fim, é feito o mapeamento entre o domínio físico e o de processo.

O processo de mapeamento pode também ser descrito em termos dos conceitos pertencentes a cada domínio. Primeiramente são identificadas as necessidades ou, atributos do cliente (CAs), do domínio do cliente. Os atributos do cliente (CAs) do domínio do cliente são especificados em termos dos requisitos funcionais (FRs) no domínio funcional. Para satisfazer os requisitos funcionais (FRs) no domínio funcional, são criados parâmetros de projeto (DPs) no domínio físico. Então, para manufaturar um produto especificado em termos dos parâmetro de projeto (DPs) no domínio físico é desenvolvido um processo caracterizado pelas variáveis de processo (PVs) no domínio de processo (SUH, 2001).

O processo de projeto, propriamente dito, é realizado através do mapeamento entre os requisitos funcionais (FRs) no domínio funcional e os parâmetros de projeto (DPs) no domínio físico. O mapeamento entre os domínios funcional e físico é realizado através de um processo chamado “ziguezagueamento”.

O processo de “ziguezagueamento” é uma das características do Projeto Axiomático. Ele une o processo de mapeamento entre, por exemplo, os requisitos funcionais (FRs) e os parâmetros de projeto (DP) e o processo de decomposição funcional. Os processos de decomposição funcional e “ziguezagueamento” são explicados com detalhes nas seções 3.2.3 e 3.2.4, respectivamente. O “ziguezagueamento” normalmente só é realizado entre os domínios funcional e físico, entretanto, em um processo de engenharia concorrente, seria possível realizar um “ziguezagueamento” entre mais de dois domínios (STADZISZ, 1997).

3.1.3 Axiomas

No livro "*The Principles of Design*", Nam. P. Suh definiu os dois axiomas que embasam a Teoria de Projeto Axiomático, mais oito corolários e dezesseis teoremas relacionados (SUH, 1990). Em um livro mais recente, "*Axiomatic Design: Advances and Applications*", Suh definiu mais vinte e quatro teoremas (SUH, 2001). Estes corolários e teoremas são apresentados no Anexo 1. Os dois axiomas propostos, que formam a base do Projeto Axiomático, são:

Axioma 1 (Axioma da Independência) *Mantenha a independência dos requisitos funcionais (SUH, 2001).*

Axioma 2 (Axioma da Informação) *Minimize o conteúdo de informação do projeto (SUH, 2001).*

O primeiro axioma define um projeto "bom" como sendo aquele em que os requisitos funcionais são independentes entre si com relação aos parâmetros de projeto que os realizam. Isto significa que em uma solução de projeto aceitável os parâmetros de projeto e os requisitos funcionais estão relacionados de tal forma que um determinado parâmetro de projeto pode ser ajustado para satisfazer um determinado requisito funcional sem afetar outros requisitos funcionais (SUH, 1990).

Este axioma permite decidir qual a melhor solução de projeto, usando o conceito de independência funcional. "Entre duas soluções de projeto possíveis, a que possuir a maior independência funcional é a melhor" (SUH, 1990). A aplicação do Axioma da Independência em projetos e as conseqüências dessa aplicação além de métricas para o cálculo da independência funcional são apresentadas na Seção 3.2.

O segundo axioma estabelece que o "melhor projeto" é aquele que possui o menor conteúdo de informação (SUH, 1990). O conteúdo de informação de uma solução de projeto está relacionado com a probabilidade de sucesso da realização desta solução. Em outras palavras, quanto maior o conteúdo de informação mais difícil será para transformar essa solução de projeto em produto. O conceito de conteúdo de informação e métricas para seu cálculo são discutidos na Seção 3.3.

3.2 Axioma 1 - Axioma da Independência

Apesar do desenvolvimento ser um processo que passa pelos quatro domínios, a atividade de projeto em si é representada pelos mapeamentos entre o domínio funcional e o domínio físico. Neste mapeamento, para cada requisito funcional (FR), é criado um único parâmetro de projeto (DP) correspondente, de forma a satisfazer o Axioma 1 ou Axioma da Independência. O enunciado deste axioma é "Manter a independência dos requisitos funcionais" (SUH, 2001). Um enunciado alternativo para este axioma é "Em um *projeto* aceitável, os parâmetros de projeto (DPs) e os requisitos funcionais (FRs) estão relacionados de tal forma que um DP específico pode ser ajustado para satisfazer um FR sem afetar outros FRs" (SUH, 2001).

O Axioma da Independência estabelece que quando existem dois ou mais requisitos funcionais (FRs), a solução deve ser tal que cada requisito funcional (FR) deve ser satisfeito sem afetar os outros (SUH, 2001). Isto significa que o conjunto de parâmetros de projeto (DPs) deve ser identificado de forma a satisfazer os requisitos funcionais (FRs) e manter sua independência. Quando a solução de projeto não satisfaz o Axioma da Independência, uma nova solução deve ser criada.

Neste caso, o projetista deve criar novos parâmetros de projeto (DPs) para satisfazer os requisitos funcionais (FRs) tentando mantê-los independentes. Em alguns casos, o conjunto de requisitos funcionais (FR) identificado não é adequado, de tal forma que a identificação de um conjunto de parâmetros de projeto (DP) que satisfaçam o Axioma da Independência nem sempre é possível. Neste caso, o conjunto de requisitos funcionais (FRs) deve ser modificado, de forma a encontrar um novo conjunto de requisitos funcionais (FRs), que possa produzir um projeto mais independente.

3.2.1 Matriz de Projeto

Um dos instrumentos usados no Projeto Axiomático é a matriz de projeto. Ela representa o mapeamento entre dois domínios, principalmente entre o domínio funcional e o físico. Em uma das dimensões da matriz estão representados os requisitos funcionais (FRs) e na outra, os parâmetros de projeto (DPs).

Cada elemento da matriz de projeto representa a relação entre um determinado requisito funcional (FR) e um determinado parâmetro de projeto. Esta relação, denotada pelo elemento A_{ij}, representa o grau com que o parâmetro de projeto DP_j é usado para satisfazer o

requisito funcional FR_i. Cada elemento da matriz representa que um determinado parâmetro de projeto (DP) está relacionado a um determinado requisito funcional (FR). A relação que representa a matriz de projeto é apresentada na Equação (1) (SUH, 1990), onde {FR} é o vetor dos requisitos funcionais, {DP} é o vetor dos parâmetros de projeto e [A] é a matriz de projeto que relaciona os requisitos funcionais com os parâmetros de projeto que os satisfazem.

$$\{FR\}=[A]\{DP\} \tag{1}$$

$$\begin{aligned} FR_1 &= A_{11}\,DP_1 + A_{12}\,DP_2 + A_{13}\,DP_3 + \cdots + A_{1n}\,DP_n \\ FR_2 &= A_{21}\,DP_1 + A_{22}\,DP_2 + A_{23}\,DP_3 + \cdots + A_{2n}\,DP_n \\ FR_3 &= A_{31}\,DP_1 + A_{32}\,DP_2 + A_{33}\,DP_3 + \cdots + A_{3n}\,DP_n \\ &\vdots \\ FR_n &= A_{n1}\,DP_1 + A_{n2}\,DP_2 + A_{n3}\,DP_3 + \cdots + A_{nn}\,DP_n \end{aligned} \tag{2}$$

A relação apresentada na Equação (1) pode ser representada sob a forma de um sistema de equações, como na Equação (2) (SUH, 1990). Neste sistema, os elementos A_{ij} representam o relacionamento entre o requisito funcional FR_i e o parâmetro de projeto DP_j. Quando $A_{ij} \neq 0$, então DP_j é usado para satisfazer FR_i. Caso $A_{ij} = 0$, DP_j não está relacionado com a satisfação de Fr_i. Por exemplo, se o elemento A_{12} for diferente de zero, isto significa que o parâmetro de projeto DP_2 é usado para satisfazer o requisito funcional FR_1.

Cada linha do sistema linear apresentado na Equação (2) representa um requisito funcional e os correspondentes parâmetros de projeto que o satisfazem. Cada linha do sistema também representa um módulo do sistema que está sendo projetado, relacionando o requisito funcional e os parâmetros de projeto usados na sua realização (SUH, 2001). Neste caso, módulo corresponde ao conjunto de parâmetros de projeto usados para satizfazer um determinado requisito funcional. Esta relação está representada na Equação (3).

$$FR_i = \sum_j A_{ij}\,DP_j \tag{3}$$

Do ponto de vista do Axioma de Independência, existem três tipos de matrizes de projeto. Na Equação (4), cada parâmetro de projeto (DP) é usado para satisfazer um único requisito funcional (FR). Esta configuração satisfaz completamente o Axioma da

Independência. Neste caso, o projeto ou solução é dito "desacoplado", do inglês, *uncoupled*. Este é considerado o melhor tipo de projeto, pois um ajuste em um parâmetro de projeto (DP) não afeta outros requisitos funcionais (FRs) (SUH, 1990).

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & A_{33} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix} \tag{4}$$

O tipo de projeto mais comum é o projeto dito "semi-acoplado", do inglês, *decoupled* (SUH, 1990). Neste tipo de projeto, um requisito funcional é satisfeito por mais de um parâmetro de projeto (DP). Isto significa que, além do parâmetro de projeto (DP) correspondente ao requisito funcional (FR), outros parâmetros de projeto (DPs) já criados são usados para satisfazê-lo. Neste caso, a matriz de projeto assume o formato semelhante a uma matriz triangular inferior como apresentado na Equação (5) (SUH, 1990).

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix} \tag{5}$$

Nem sempre a matriz de projeto assume a forma triangular inferior. Neste caso, a ordem dos requisitos funcionais (FRs) e a ordem dos parâmetros de projeto (DPs) podem ser alteradas desde que seja mantida a correlação entre cada requisito funcional (FR) e os parâmetros de projeto (DPs) criados para satizfazê-lo, até que a matriz de projeto assuma a forma triangular inferior. Este fato indica que existirá uma ordem na qual os parâmetros de projeto (DPs) deverão ser realizados. Soluções de projeto do tipo semi-acoplados são as mais comuns e, segundo o Axioma da Independência, são consideradas soluções aceitáveis (SUH, 1990).

A situação a ser evitada em um projeto é aquela na qual os parâmetros de projeto não só estão relacionados com os requisitos funcionais (FRs) que estão representados na matriz, abaixo deste, mas, também, com os que estão representados na matriz, acima deste. Por exemplo, na Equação (6), os requisitos funcionais FR_1, FR_2 e FR_3 são todos satisfeitos pelos parâmetros de projeto DP_1, DP_2 e DP_3.

Esta situação é problemática pois supondo que uma mudança em FR_1 implique na alteração em DP_1, isto pode influenciar FR_2 e pode demandar alterações em DP_2 e DP_3 o que

por sua vez pode influenciar FR_1 e FR_3. Não existe, portanto, uma ordem de realização dos parâmetros de projeto (DPs) que resolva este problema. Este tipo de projeto é o chamado acoplado, do inglês *coupled*, e a matriz assume um formato completo, como na Equação (6) (SUH, 1990).

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix} \tag{6}$$

Como exemplo de aplicação do Axioma 1 usando a matriz de projeto, é analisado o projeto da torneira de água quente/fria apresentado na seção 2.3.2. A solução para este projeto é apresentada em (SUH, 2001). Para este projeto foram especificados dois requisitos funcionais: "FR1- controlar a temperatura" e "FR2 - controlar a vazão da água".

Na primeira solução foram criados dois parâmetros de projeto: "DP1 - válvula C1 que controla a saída de água quente" e "DP2 - válvula C2 que controla a saída de água fria" (ver Figura 7). Neste caso, C1 e C2 são usadas tanto para controlar a temperatura (FR1) como para controlar a vazão (FR2). Construindo-se a matriz de projeto para essa solução, tem-se um projeto acoplado, e portanto, ruim.

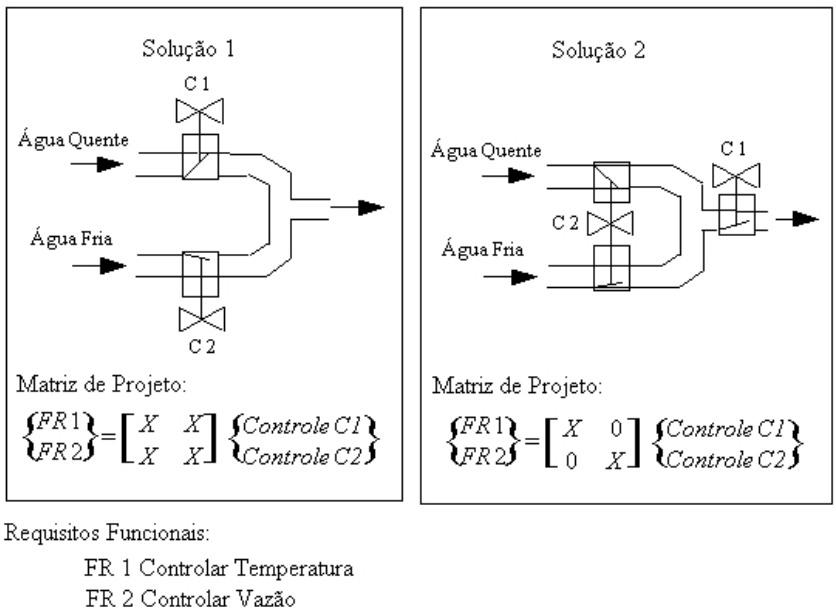


Figura 7 - Duas soluções para o projeto da torneira

Para a segunda solução foram identificados dois outros parâmetros de projeto: "DP1

- válvula C1, que controla apenas a vazão da água" e "DP2 - válvula C2 que aumenta a água fria e diminui a água quente e vice-versa sem alterar a vazão". A matriz de projeto para esta solução é desacoplada (ver Figura 7) porque DP1 satisfaz apenas FR1 e DP2 satisfaz apenas FR2, de forma independente.

Em (CAPPETTI; NADEO; PELLEGRINO, 2004) é apresentada uma metodologia para reduzir o acoplamento da matriz de projeto. Esta metodologia é baseada na lógica nebulosa (*Fuzzy*) e na teoria da representação (ZADEH et al., 1975). É proposto que associando-se valores de função de pertinência para cada célula da matriz de projeto, é possível reduzir o grau de dependência entre parâmetros de projeto (DP) e requisitos funcionais (FR). Com isto é possível reduzir o grau de acoplamento da matriz de projeto, otimizando um projeto acoplado, quando não for possível obter um projeto desacoplado (CAPPETTI; NADEO; PELLEGRINO, 2004).

Uma estratégia para reduzir o acoplamento da matriz de projeto é apresentada em (LEE, 2006). Esta estratégia tem por objetivo identificar os elementos da matriz que causam acoplamento e permitir que o projetista modifique apenas estes elementos do projeto. Esta estratégia é aplicada transformando a matriz de projeto em um grafo. Através de manipulações com a matriz de adjacências deste grafo, encontram-se os ciclos que indicam os elementos que causam o acoplamento na solução de projeto.

3.2.1.1 Matrizes de Projeto Não Quadradas

Segundo o Teorema 4 (ver Anexo 1), em um projeto ideal o número de requisitos funcionais (FRs) deve ser igual ao número de parâmetros de projeto (DPs) (SUH, 1990). Como nem todas as soluções de projetos criadas são ideais, existem casos em que a matriz de projeto não assume a forma quadrada. Existem dois tipos de projetos com essa característica: projeto acoplado por número insuficiente de parâmetros de projeto (DPs) e projeto redundante.

No primeiro caso, o número de requisitos funcionais (FRs) é maior que o número de parâmetros de projeto (DPs) (ver Equação (7)). Isto resulta em um projeto acoplado, segundo o Teorema 1 (ver Anexo 1) (SUH, 1990). Este tipo de projeto é resultado de muitos parâmetros de projeto (DPs) criados ou de requisitos funcionais (FRs) que não foram identificados. Nestes casos, o Teorema 2 (ver Anexo 1) indica que deve-se identificar mais parâmetros de projeto (DP) para tornar a matriz desacoplada ou, pelo menos, semi-acoplada (SUH, 1990).

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \end{Bmatrix} \tag{7}$$

No caso do número de parâmetros de projeto (DPs) ser maior que o número de requisitos funcionais (FRs) (ver Equação (8)), pelo Teorema 3 (ver Anexo 1), o projeto é dito "redundante" ou acoplado (SUH, 1990). Neste caso, pode-se tentar agrupar parâmetros de projeto (DP) que afetem um determinado requisito funcional (FR) para remover a redundância. Cada parâmetro de projeto (DP) que pertence a esse agrupamento pode ser tratado com a decomposição do requisito funcional (FR) correspondente.

$$\begin{Bmatrix} FR_1 \\ FR_2 \end{Bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix} \tag{8}$$

3.2.2 Métricas Para o Cálculo da Independência Funcional

O enunciado original do Axioma da Independência permite avaliar se uma solução de projeto é boa (matriz de projeto desacoplada), aceitável (matriz de projeto semi-acoplada) ou inaceitável (matriz de projeto acoplada). Em muitos casos, as diferentes soluções propostas são apenas aceitáveis (matrizes de projeto semi-acopladas), sendo necessário um critério para decidir qual a melhor solução. Para isso, o Axioma 1 pode ser enunciado de uma maneira diferente para permitir vários graus de independência funcional, ou seja: "De duas soluções de projeto aceitáveis, a que possuir a maior independência funcional é a melhor"(SUH, 1990).

Para encontrar qual a solução com a maior independência funcional, é necessário uma medida quantitativa para se fazer a comparação. É possível encontrar uma medida quantitativa de independência funcional. Em (SUH, 1990) são deduzidas e descritas duas medidas de independência funcional com base na matriz de projeto. São elas a "reangularidade" (do inglês *reangularity*), denotada por R, e a "semangularidade" (do inglês *semangularity*), denotada por S.

"Reangularidade" mede a ortogonalidade entre os parâmetros de projeto (DPs) e pode ser considerada como uma medida de interdependência entre os parâmetros de projeto (DPs) (OLEWNIK; LEWIS, 2003). Uma fórmula para o cálculo da "reangularidade" é apresentada na Equação (9).

$$R = \prod_{\substack{i=1,n-1 \\ j=1+i,\,n}} \left(1 - \frac{\left(\sum_{k=1}^{n} A_{ki} A_{kj}\right)^2}{\left(\sum_{k=1}^{n} A_{ki}^2\right)\left(\sum_{k=1}^{n} A_{kj}^2\right)} \right)^{1/2} \tag{9}$$

A "semangularidade" mede o relacionamento angular entre os eixos correspondentes de parâmetros de projeto (DPs) e de requisitos funcionais (FRs) e pode ser considerada como sendo uma medida da correlação entre um requisito funcional (FR) e qualquer conjunto de parâmetros de projeto (DPs) (OLEWNIK; LEWIS, 2003). Uma fórmula para o cálculo da "semangularidade" é apresentada na Equação (10).

Ambas, possuem um valor máximo de 1, que corresponde a um projeto desacoplado (ideal) e também, ambas possuem um valor 0 para projetos acoplados (inaceitáveis). Quando a independência funcional diminui, o valor de "reangularidade" e de "semangularidade" decresce (OLEWNIK; LEWIS, 2003).

$$S = \prod_{j=1}^{n} \left(\frac{|A_{jj}|}{\left(\sum_{k=1}^{n} A_{kj}^2\right)^{1/2}} \right) \tag{10}$$

A reangularidade e a semangularidade possuem propriedades estabelecidas em teoremas (SUH, 1990). Se uma matriz de projeto for particionada em submatrizes quadradas, segundo o Teorema 10 (ver Anexo 1), o produto das reangularidades das submatrizes não zero é igual a reangularidade da matriz total. Esta propriedade também vale para a semangularidade. A reangularidade e a semangularidade da matriz de projeto permanecem inalteradas com a mudança na ordem das linhas da matriz de projeto desde que sejam preservadas as relações entre requisitos funcionais (FRs) e parâmetros de projeto (DPs), segundo o Teorema 11 (ver Anexo 1) (SUH, 1990).

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix} \tag{11}$$

Como exemplo é apresentado o cálculo das métricas de independência funcional para a matriz de projeto apresentada na Equação (11). Para esta matriz, tanto o valor da reangularidade quanto o valor da semangularidade são iguais a 0,7071068. A reangularidade

mede a interdependência entre os parâmetros de projeto, o componente maior da independência funcional. Pode ser medida, também, a relação entre os eixos correspondentes de FRs e DPs, dada pela semangularidade (SUH, 1990).

3.2.3 Hierarquia Funcional e Decomposição Funcional

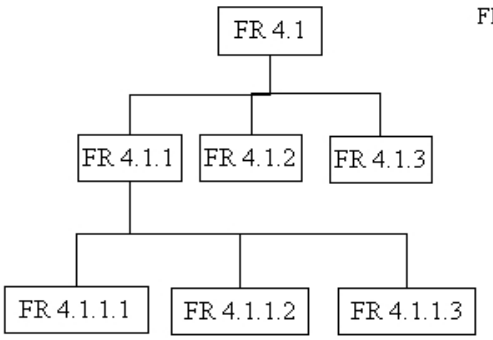
Quando os requisitos funcionais (FR) são identificados, eles fornecem as especificações necessárias para o projeto no nível conceitual. Mas, para a realização do projeto, é necessário um nível de detalhamento maior dos requisitos funcionais. É necessário decompor esses requisitos em unidades com maior detalhamento. Para isso, os requisitos funcionais (FR) são estruturados em uma hierarquia funcional. Esta hierarquia é obtida através da decomposição funcional de cada requisito funcional (FR) em sub-requisitos.

A decomposição funcional é um processo no qual, após o mapeamento de todos os requisitos funcionais em parâmetros de projeto de um nível da hierarquia, cada requisito funcional (FR) é dividido em funcionalidades menores, chamadas sub-requisitos (sub-FR). Esta decomposição deve atender a algumas restrições, de forma a não alterar as relações de independência funcional representadas na matriz de projeto (TATE, 1999).

Em (TATE, 1999) foi definida uma seqüência de etapas a serem consideradas para realizar as decomposições funcionais durante um projeto. Além disso, foram definidas restrições a serem consideradas para manter a coerência entre as decomposições, nos diversos níveis da hierarquia funcional, durante o processo de "ziguezagueamento" (TATE, 1999). Essas restrições dizem respeito à escolha apropriada dos sub-requisitos funcionais (sub-FR) de tal forma que a matriz de projeto mantenha as mesmas características de acoplamento do nível anterior. Em outras palavras, se, por exemplo, o parâmetro de projeto DP_j não estiver associado ao requisito funcional FR_i, seu subparâmetros também não deverão estar relacionados com os sub-requisitos de FR_i.

Um exemplo de decomposição funcional é apresentado por Stadzisz (1997) para o projeto de uma buzina automobilística (STADZISZ, 1997). A árvore de hierarquia dos requisitos funcionais (FR) é apresentada na Figura 8. Um dos requisitos funcionais do projeto é "FR 4.1 - Permitir a fixação da buzina ao veículo". Este requisito funcional é decomposto em outros requisitos funcionais cada vez mais especializados, passando por dois níveis de decomposição até chegar ao requisito funcional "FR 4.1.1.2 - Aumentar a pressão de fixação do parafuso e da porca de fixação da caixa" (STADZISZ, 1997). Neste caso, cada sub-requisito funcional está hierarquicamente associado com seu superior, o que permite que cada

requisito funcional de nível mais baixo esteja relacionado com seu correspondente de nível mais alto.



FR 4.1 - Permitir a fixação da buzina ao veículo
FR 4.1.1 - Permitir a fixação da caixa da buzina à haste de fixação
FR 4.1.1.1 - Sustentar e posicionar a caixa em relação à haste
FR 4.1.1.2 – Aumentar a pressão de fixação do parafuso e da porca de fixação da caixa
FR 4.1.1.3 – Apertar o parafuso de fixação da caixa
FR 4.1.2 - Permitir o posicionamento e a orientação da buzina em relação ao ponto de fixação na carroceria do veículo
FR 4.1.3 - Permitir a fixação da haste à carroceria do veículo

Figura 8 - Exemplo de hierarquia de decomposição funcional

Um requisito funcional pode ser decomposto em sub-requisitos de vários tipos. Segundo Hintersteiner (1999), eles podem ser classificados em funções de processo, de comando e controle, e de suporte e integração (HINTERSTEINER, 1999). As funções de processo estão relacionadas com o processamento propriamente dito dos dados além do transporte destes através do sistema. As funções de comando e controle representam a lógica necessária para coordenar os diferentes processos no subsistema. As funções de suporte e de integração mantém o subsistema funcionando de forma coordenada (HINTERSTEINER, 1999).

A decomposição funcional tem por objetivo refinar as funcionalidades etapa por etapa para dar ao projetista vários níveis de abstração dos requisitos funcionais (FRs). A decomposição funcional deve ser feita até que sejam satisfeitas as necessidades de representação do projetista para o projeto. Muitas vezes, decompõem-se os requisitos funcionais até que estes correspondam a funcionalidade de um componente físico indivisível, como por exemplo, a funcionalidade esperada de uma arruela de pressão (ver FR 4.1.1.2 do exemplo da Figura 8).

A decomposição dos requisitos funcionais não é um processo que depende apenas dos requisitos funcionais do nível anterior, mas, também, dos parâmetros de projeto criados para satisfazer estes requisitos, em um processo chamado de ziguezagueamento, do inglês, *zigzagging* (SUH, 2001). O ziguezagueamento vincula a decomposição dos requisitos

funcionais à criação dos parâmetros de projeto (a partir da decomposição dos parâmetros de projeto de nível superior) estabelecendo uma ligação forte entre os domínios funcional e físico.

3.2.4 Ziguezagueamento

Para cada requisito funcional identificado em um determinado nível de decomposição, é identificado um parâmetro de projeto que representa sua realização. Após a identificação de todos os parâmetros de projeto de um nível da hierarquia funcional, é feita a decomposição de cada requisito funcional em sub-requisitos, fazendo surgir um novo nível da hierarquia funcional. Então, o processo é realizado novamente, agora para este novo nível. A decomposição é feita até que o projeto esteja completo (SUH, 2001).

Para cada novo nível de decomposição dos requisitos funcionais encontrados, é realizado um mapeamento entre os domínios e, após o mapeamento, é feita uma nova decomposição. Este processo de passar do domínio funcional para o domínio físico e novamente para o funcional em um outro nível da hierarquia é chamado de "ziguezagueamento" (SUH, 2001). A Figura 9 ilustra este processo.

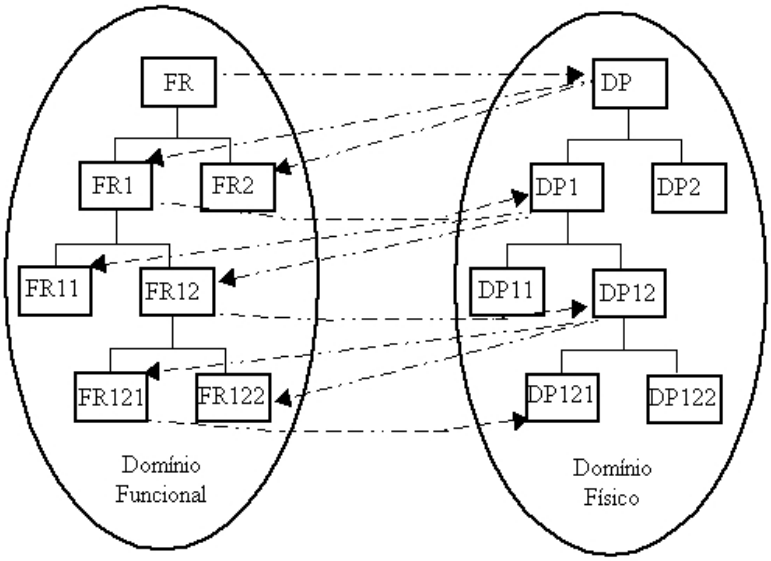


Figura 9 - Processo de "Ziguezagueamento"

Não somente os requisitos funcionais (FRs) orientam a definição ou invenção dos parâmetros de projeto (DPs), mas também, os parâmetros de projeto orientam a decomposição dos requisitos funcionais. Isto acontece devido ao fato de que para decompor os requisitos

funcionais (FRs) é necessário que os parâmetros de projeto (DPs) correspondentes sejam identificados. A identificação do conjunto de sub-requisitos funcionais (sub-FR) terá como ponto de partida a solução delineada pelos parâmetros projeto (DPs) identificados no nível anterior.

Em (SCHREYER; TSENG, 2000) é apresentada uma notação para representar a decomposição funcional e o processo de "ziguezagueamento". Esta notação é baseada nos *statecharts* apresentados em (HAREL, 1987). Esta notação ajuda a identificar estados principais, além de outros parâmetros importantes, como tempo, sinais de sensores e informação de memória compartilhada. É apresentada a matriz de transições de estados, que relaciona as condições de entrada com as ações de saída para cada estado (SCHREYER; TSENG, 2000).

3.2.5 Diagrama de Fluxo e Diagrama de Junção de Módulos

Grandes sistemas são caracterizados por possuírem um grande número de componentes, sejam eles de *hardware* ou *software*. Para o projeto de grandes sistemas, existe a necessidade de se entender e representar a arquitetura do sistema e cada um de seus módulos. Para este tipo de representação foram desenvolvidas duas ferramentas: o diagrama de fluxo e o diagrama de junção de módulos (SUH, 2001).

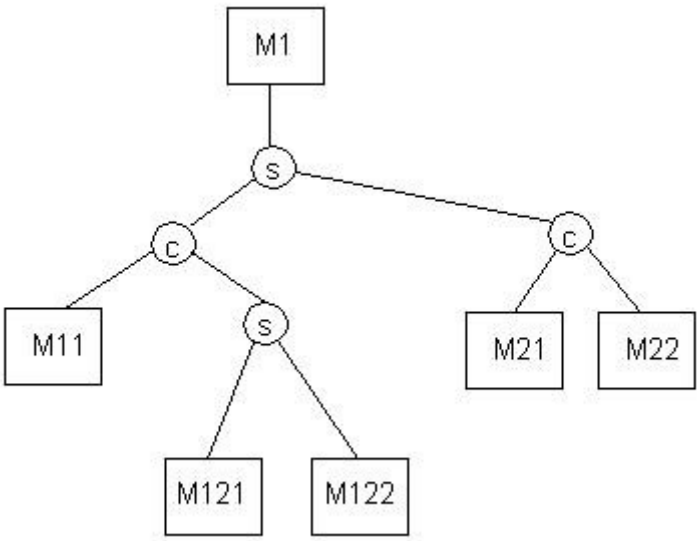


Figura 10 - Diagrama de junção de módulo

Um módulo é definido como sendo uma linha da matriz de projeto e relaciona o requisito funcional (FR) e os parâmetros de projeto (DP) que o satisfazem. O diagrama de junção de módulos representa o relacionamento entre os módulos através dos níveis da hierarquia. Um exemplo de diagrama de junção de módulos é apresentado na Figura 10. Neste diagrama, quando os módulos aparecem relacionados por um conectivo "s", eles são independentes e a matriz de projeto que os relaciona é desacoplada. Quando os módulos estão relacionados por um conectivo "c", existe uma relação de controle entre eles, o que significa uma matriz de projeto semi-acoplada (SUH, 2001).

O diagrama de fluxo permite fazer uma representação da arquitetura do sistema. Esta representação é baseada na independência entre os módulos. Quando a matriz de projeto é desacoplada, os módulos são representados em paralelo, indicando a independência entre eles, e relacionados pelo conector "s", como na Figura 11. Quando a matriz de projeto é semi-acoplada existe uma dependência, que é representada colocando os módulos em seqüência, conectados por "c" (SUH, 2001). Neste tipo de representação não são consideradas matrizes de projeto acopladas, caso em que o projeto deverá ser modificado.

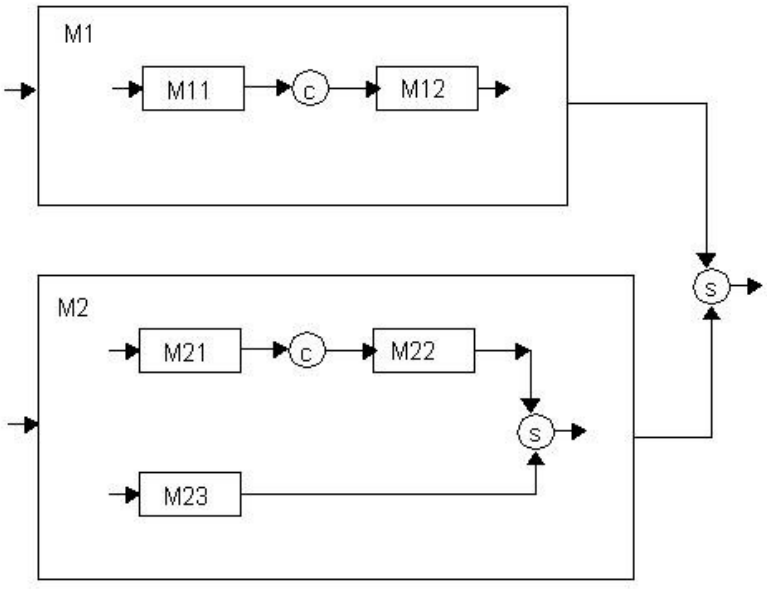


Figura 11 - Diagrama de fluxo

3.3 Axioma 2 - Axioma da Informação

O Axioma 2 estabelece que "o melhor projeto é um projeto funcionalmente desacoplado que tem o conteúdo de informação mínimo" (SUH, 1990). Em um enunciado mais simples, o Axioma 2 consiste em "Minimizar o conteúdo de informação do projeto" (SUH, 1990). O conteúdo de informação é um parâmetro importante para a definição de um bom projeto. Segundo Suh (2001), "o Axioma da Informação provê uma medida quantitativa de mérito de um dado projeto, além de prover uma base teórica para a otimização e robustez do projeto" (SUH, 2001). O conteúdo de informação pode ser usado como parâmetro para auxiliar na identificação do conjunto de parâmetros de projeto mais adequado para satisfazer um determinado conjunto de requisitos funcionais.

"Informação é a medida do conhecimento requerido para satisfazer um dado FR em um determinado nível da hierarquia de FRs" (SUH, 1990). Isto significa que quanto mais informação for necessária para satisfazer um requisito funcional, maior será o conteúdo de informação. Quanto maior a informação necessária para satisfazer os requisitos funcionais (FRs), mais difícil será para realizar o projeto com sucesso, pois a complexidade do projeto e da sua fabricação ou implementação aumentará. Assim, o conteúdo de informação está intimamente relacionado com a probabilidade de sucesso no projeto e fabricação de um produto. Esta relação é ilustrada em (SUH, 2001), onde é dito que "o Axioma de Informação estabelece que o projeto que tem a maior probabilidade de sucesso é o melhor projeto".

O conteúdo de informação é calculado com base na probabilidade de que o parâmetro de projeto (DP) satisfaça o requisito funcional (FR) correspondente. Este cálculo pode ser feito com base na razão entre o valor esperado para este parâmetro de projeto (DP) e a variação possível deste valor. Esta probabilidade pode ser calculada com base na complexidade da tarefa (SUH, 1990). Segundo Suh (1990), sendo p a probabilidade de que o DP satisfaça o FR, o conteúdo de informação I pode ser calculado como indicado na Equação (12).

$$I = \frac{1}{p} \tag{12}$$

A Equação (12) mostra que o conteúdo de informação é inversamente proporcional à probabilidade de sucesso. Para facilitar o cálculo de probabilidades relacionadas, o conteúdo de informação é calculado de forma logarítmica conforme apresentado na Equação (13)

(SUH, 1990).

$$I = \log\left(\frac{1}{p}\right) \tag{13}$$

O inverso da probabilidade de sucesso pode ser interpretado de diversas formas. Uma forma de interpretá-la é como a dificuldade ou complexidade em projetar e construir o produto. Por exemplo, quanto maior a precisão exigida pelos requisitos funcionais e restrições, mais difícil será projetar e construir um produto ou peça. Então, o inverso da probabilidade de sucesso é definido em termos da precisão requerida dos parâmetros de projeto. Neste caso, um parâmetro de projeto pode assumir uma determinada faixa de valores, chamada em (SUH, 1990) de *system range*. Para este mesmo parâmetro de projeto existe uma restrição para que os valores assumidos sejam considerados aceitáveis. Esta outra faixa de valores é chamada de "tolerância" (*tolerance*). Então, a dificuldade em projetar e fabricar o produto pode ser considerada em termos da razão entre a faixa de valores que o parâmetro pode assumir e a faixa de valores aceitáveis. Assim, foi proposto em (SUH, 1990) a fórmula mostrada na Equação (14) para o cálculo do conteúdo de informação.

$$I = \log\left(\frac{system\,range}{tolerance}\right) \tag{14}$$

O conteúdo de informação pode ser calculado como a composição dos conteúdos de informação dos requisitos funcionais do sistema. Esta composição é feita através da composição das probabilidades. Como o conteúdo de informação é calculado de forma logarítmica, a composição dos conteúdos de informação dos requisitos funcionais do sistema, se estes forem independentes, pode ser dada como a soma destes conteúdos[9], como apresentada na Equação (15).

$$I = -\sum_{i=1}^{n} p_i \log p_i \tag{15}$$

Neste caso o cálculo do conteúdo de informação não usa fatores ponderados. Um dos motivos é que se forem usados fatores ponderados na soma do conteúdo de informação, este não representará mais a probabilidade total de sucesso. Também é considerado que o cálculo da informação do requisito funcional (FR) é baseado na faixa de variação de projeto deste requisito funcional. Segundo SUH (2001), "Se esta faixa de variação for corretamente

[9]ver Teorema 13, no Anexo 1

especificada, ela já definirá a importância relativa de cada requisito funcional (FR)" (SUH, 2001).

Um exemplo do cálculo do conteúdo de informação é apresentado no estudo realizado por Nakazawa (1987) para determinar o melhor carro subcompacto à venda no Japão, na época. Para cada requisito funcional (FR) foi definida uma faixa de valores toleráveis de projeto (tolerância). Um dos quesitos considerados é o consumo (dado em km/l), e cujo valor tolerável deveria estar entre 17 e 24,6 km/l. O modelo A consome entre 12,4 e 24,6 km/l e o modelo B consome entre 11,1 e 21,9 km/l. Usando a Equação (14) para calcular o conteúdo de informação relativo a este quesito, obtém-se para o modelo A o valor 0,473 e para o modelo B, o valor 0,790. Então, relativamente ao quesito consumo, o modelo A possui um conteúdo de informação menor, sendo uma solução melhor (NAKAZAWA, 1987).

Poucos trabalhos tratam do cálculo do conteúdo de informação para projetos de sistemas de *software*. Suh (2001) relaciona, brevemente, o conteúdo de informação com o número de linhas de código do *software*, dizendo que quanto menos linhas de código o *software* tiver, maior será a probabilidade dele realizar as funções desejadas corretamente (SUH, 2001).

Em (SHIN et al., 2004) são apresentadas formas de se calcular o conteúdo de informação de um projeto. São apresentados dois métodos, o método gráfico, para projetos com dois requisitos funcionais e o por integração, para projetos com mais de dois requisitos funcionais. O cálculo é feito com base na associação entre o *system range* e a tolerância. O método gráfico usa uma distribuição uniforme de probabilidade para a associação. No método por integração podem ser usados quaisquer tipos de distribuição de probabilidade (SHIN et al., 2004).

3.4 Complexidade no Projeto Axiomático

Complexidade é um conceito muito usado nas mais variadas áreas. Normalmente está associado à dificuldade em se compreender e lidar com um determinado objeto ou conceito (LEE, 2003). A Teoria de Projeto Axiomático discute e define a complexidade de um projeto e os diferentes aspectos envolvidos. A complexidade de um projeto, do ponto de vista da Teoria de Projeto Axiomático, pode ser definida como sendo "a medida da incerteza em se alcançar um conjunto desejado de requisitos funcionais" (SUH, 2001). Esta incerteza

pode estar relacionada com fatores dependentes de tempo ou não, sendo classificada em complexidade independente de tempo e complexidade dependente de tempo (SUH, 2001).

3.4.1 Complexidade Independente do Tempo

Para a Teoria de Projeto Axiomático a complexidade é discutida em relação aos requisitos funcionais (LEE, 2003). A complexidade independente de tempo avalia a complexidade de um sistema cuja satisfação dos requisitos funcionais não está relacionada com aspectos temporais (LEE, 2003). Este tipo de complexidade está dividido em dois componentes: a complexidade real e a complexidade imaginária. A complexidade real é definida como sendo a dificuldade real para que os requisitos funcionais do sistema sejam satisfeitos. Segundo Suh (2001), a complexidade real é igual ao conteúdo de informação da solução de projeto, apresentado na seção 3.3 (SUH, 2001).

A complexidade imaginária pode ser definida como a "incerteza causada pela falta de conhecimento e entendimento do projetista a respeito de uma solução de projeto" (LEE, 2003). Para efeito de cálculo deste tipo de complexidade, seu escopo foi reduzido à falta de conhecimento a respeito da matriz de projeto (LEE, 2003). Por exemplo, em um projeto semi-acoplado, segundo o Teorema 7 (ver Anexo 1), existe uma ordem para que os parâmetros de projeto (DP) sejam alterados para satisfazer um conjunto de requisitos funcionais (FR). Se o projetista não reconhecer que o projeto é semi-acoplado, a dificuldade devido ao desconhecimento deste fato (complexidade imaginária) é igual à probabilidade em se encontrar a ordem correta para a mudança nos parâmetros de projeto. A complexidade independente do tempo, também chamada de complexidade absoluta, de um projeto pode ser calculada pela soma da complexidade real e da complexidade imaginária (SUH, 2001).

3.4.2 Complexidade Dependente do Tempo

Na complexidade dependente do tempo, a incerteza muda em função do tempo. A incerteza pode diminuir ou aumentar em função do tempo, ou mesmo, possuir um comportamento periódico. Se a incerteza a respeito do projeto diminui em função do tempo, ele pode ser tratado como se fosse independente de tempo, com algumas considerações a respeito de estados transitórios. Os tipos de complexidade dependente do tempo que demandam uma atenção mais cuidadosa no projeto podem ser: complexidade combinatória e complexidade periódica (SUH, 2001).

A complexidade dependente de tempo periódica ocorre quando as incertezas

inerentes ao sistema voltam a seus valores iniciais de tempos em tempos, isto é, periodicamente, e portanto, não cresce indefinidamente. Por exemplo, o problema de escalonamento das saídas de vôos. Atrasos podem ocorrer, devido a falhas, acidentes ou mesmo devido a um escalonamento ruim, e as incertezas a respeito do escalonamento vão aumentando durante o dia. No dia seguinte, as incertezas começam com os valores iniciais (SUH, 2001).

Na complexidade dependente de tempo combinatória, a incerteza cresce indefinidamente. Isto pode ocorrer por duas razões principais: a faixa de valores do sistema (*system range*) se afasta continuamente da faixa de valores aceitáveis para o projeto (tolerância) e o comportamento imprevisível do conjunto de requisitos funcionais (FR) em função do tempo (LEE, 2003). Por exemplo, as peças de um sistema mecânico podem se desgastar de maneira imprevisível, pois o sistema pode ser usado de maneira imprevisível pelo usuário, e isto aumenta a incerteza de que os requisitos funcionais continuarão a ser satisfeitos. Outro exemplo é o fato de que alterações imprevisíveis na legislação tributária podem causar um comportamento imprevisível no conjunto de requisitos funcionais (FR), aumentando a incerteza sobre o comportamento de um sistema de *software* de controle da folha de pagamento de uma empresa.

No caso de sistemas em que o conjunto de requisitos funcionais pode variar durante o ciclo de vida do sistema, a complexidade do projeto pode aumentar devido a estas mudanças. A Teoria da Flexibilidade (VALCKENAERS, 1993) introduz diretrizes para o projeto deste tipo de sistema. Esta teoria enuncia uma importante regra geral para projeto (terceiro Axioma de projeto): o projetista deve efetivar uma decisão de projeto o mais tarde possível no processo de desenvolvimento e esta efetivação deve ter o menor grau de severidade possível (WYNS, 1999).

3.5 Projeto Axiomático e Desenvolvimento de *Software*

A natureza livre de domínio do Projeto Axiomático facilitou sua aplicação nas mais variadas áreas, como apresentado na seção 2.3, inclusive na área de desenvolvimento de *software*. Segundo Suh e Do (2000), o primeiro trabalho que usou o Projeto Axiomático no desenvolvimento de *software* foi apresentado em 1991 (KIM; SUH; KIM, 1991). Outro

trabalho relacionado foi a dissertação de mestrado de M. Angiulo[10] (1996) que, segundo Kumar e Campion, aplicou o Projeto Axiomático no desenvolvimento do *Microsoft FrontPage*[11] (ANGIULO, 1997) (KUMAR; CAMPION, 2004).

Hintersteiner e Nain (1999) usaram a Teoria de Projeto Axiomático para o desenvolvimento de *software* com foco em sistemas integrados de *software* e *hardware*. É proposta uma metodologia para facilitar o desenvolvimento de *software* de controle em conjunto com seu *hardware* correspondente. Cada requisito funcional é mapeado em um item do domínio de *software* e um item do domínio de *hardware*, o mesmo acontecendo com cada parâmetro de projeto (HINTERSTEINER; NAIN, 1999).

Um dos primeiros trabalhos envolvendo projeto de *software* orientado a objetos usando o Projeto Axiomático foi apresentado por Do e Suh (1999). É proposto que, partindo das necessidades do cliente, deve-se encontrar os componentes a serem implementados, definindo-se assim as classes e objetos que farão parte do sistema (DO; SUH, 1999). Outra aplicação da Teoria de Projeto Axiomático no desenvolvimento de *software* orientado a objetos está descrita em (CLAPIS; HINTERSTEINER, 2000) onde é apresentado um aperfeiçoamento à OMT (RUMBAUGH et al., 1991), propondo uma decomposição de objetos em objetos menores agregados aos maiores e uma análise da interdependência entre esses objetos através da matriz de projeto.

Em (SUH; DO, 2000) é apresentada uma abordagem para projeto de *software* orientado a objetos chamada "*Axiomatic Design of Object-Oriented Software Systems*" (ADo-oSS). Trata-se de uma abordagem baseada na metodologia OMT (RUMBAUGH et al., 1991). Neste trabalho não são utilizados conceitos da UML (BOOCH; RUMBAUGH; JACOBSON, 2005) como, por exemplo, casos de uso. É apresentado um mapeamento entre a matriz de projeto e o diagrama de classes onde os requisitos funcionais (FRs) representam objetos ou classes, os parâmetros de projeto representam os atributos destes objetos e as células da matriz representam os métodos dos objetos (SUH; DO, 2000).

O trabalho apresentado em (SUH; DO, 2000) serviu de base para o capítulo 5 do livro "*Axiomatic Design: Advances and Applications*" (SUH, 2001), onde é descrita a metodologia ADo-oSS. É apresentada com detalhes a utilização da matriz de projeto, do diagrama de classes, do "ziguezagueamento", do diagrama de junção de módulos e do diagrama de fluxo e as relações entre estes artefatos. Além disso, é apresentado como estudo de caso o projeto do sistema *ACCLARO*[12].

[10] Atualmente gerente de projeto do *Microsoft Project*.
[11] Marca registrada da *Microsoft Corporation*
[12] *ACCLARO* é marca registrada de *Axiomatic Design Software, Inc.*.

A ferramenta de projeto *ACCLARO* foi desenvolvida para auxiliar na elaboração de projetos usando a Teoria de Projeto Axiomático. Ela foi dividida em duas ferramentas, o *ACCLARO Designer* e o *ACCLARO Scheduller*. O *ACCLARO Designer* é uma ferramenta de *software* que auxilia o projeto de produtos e sistemas. Ele permite a decomposição funcional e constrói automaticamente a matriz de projeto. Permite, também, o rastreamento dos requisitos funcionais devido a mudanças, a análise de acoplamento do projeto, entre outras características. O *ACCLARO Scheduller* é uma ferramenta integrada com o *IBM Rational Rose*[13] para gerenciar o andamento do projeto de *software*. Permite, entre outras coisas, identificar o impacto das mudanças de projeto, gerar um plano de desenvolvimento para o projeto, estimar custos e identificar possíveis gargalos no andamento do projeto. O *ACCLARO Designer* tem sua aplicação mais voltada para projetos de engenharia enquanto que o *ACCLARO Scheduller* tem sua aplicação voltada para o gerenciamento de projetos de *software*.

Em (DO, 2004) foi apresentada a utilização da Teoria de Projeto Axiomático para o gerenciamento do processo de desenvolvimento de *software* visando garantir a qualidade do projeto e do produto. É apresentada uma extensão da Teoria de Projeto Axiomático, que consiste em domínios complementares. Estes domínios complementares são usados para representar conceitos próprios do domínio do projeto. Um domínio complementar importante é o domínio de casos de uso empregado para modelar os requisitos funcionais. Além disso, é apresentada a utilização da matriz de estrutura de projeto (ULRICH; EPPINGER, 2003) para possibilitar análises de interdependência dentro de um mesmo domínio (DO, 2004).

O desenvolvimento de *software* orientado a componentes procura melhorar a qualidade de *software*, através da interconexão de componentes, muitas vezes já existentes (GIMENES; HUZITA, 2005). Recentemente, Togay, Aktung et al. (2006) e Togay, Dogru et al. (2006) usaram a Teoria de Projeto Axiomático no desenvolvimento de *software* orientado a componentes. Em (TOGAY; AKTUNC et al., 2006) é apresentada uma métrica para a usabilidade semântica dos componentes de *software*. Em (TOGAY; DOGRU et al., 2006) é proposto um método de prototipação rápida de simulações para construção de *software* usando componentes maduros existentes.

Uma outra abordagem para aprimorar a reutilização no desenvolvimento de *software* é a Arquitetura Orientada a Serviços (SOA) (FERGUSON; STOCKTON, 2005), que procura fazer com que serviços, ou agentes, já existentes trabalhem em conjunto para satisfazer um conjunto de requisitos de um sistema. BIÇER; TOGAY; DOGRU, (2006)

[13] IBM *Rational Rose* é marca registrada de IBM

usaram a Teoria de Projeto Axiomático para representar e organizar os requisitos funcionais através de um conjunto de serviços na WEB e suas operações (BIÇER; TOGAY; DOGRU, 2006).

3.6 Conclusões

Este Capítulo apresentou a Teoria de Projeto Axiomático, descrevendo seus principais conceitos, técnicas e ferramentas com ênfase na sua aplicação no desenvolvimento de *software* orientado a objetos. Foi apresentada uma introdução à teoria e suas definições básicas, os principais conceitos do Projeto Axiomático relacionados com o Axioma da Independência, os principais conceitos relacionados com o Axioma da Informação, além dos principais trabalhos relacionados com a Teoria de Projeto Axiomático e desenvolvimento de *software* orientado a objetos.

A Teoria de Projeto Axiomático provê critérios para tomada de decisões de projeto através dos seus axiomas, teoremas e corolários que visam garantir a qualidade da solução de projeto. Com estes critérios garante-se que o projeto mantenha seus requisitos funcionais independentes (Axioma da Independência) e mantenha alta a probabilidade de que o objeto ou sistema produzido satisfaça esses requisitos (Axioma da Informação).

Apesar de ter sido criada para projetos de engenharia mecânica, a Teoria de Projeto Axiomático usa em seus axiomas, teoremas, corolários, ferramentas e processos, conceitos comuns a todos os domínios de projeto. Este fato faz com que o projeto Axiomático tenha uma natureza livre de domínio, que o torna promissor para ser aplicado nas mais variadas áreas do conhecimento, especialmente no desenvolvimento de *software*.

4 Abordagem Proposta

Neste Capítulo é feita a descrição da abordagem de projeto de *software* proposta nesta tese. Esta abordagem aplica a Teoria de Projeto Axiomático ao desenvolvimento de *software* orientado a objetos baseado no Processo Unificado. A Seção 4.1 apresenta uma introdução à abordagem proposta, incluindo seus objetivos, principais aplicações e vantagens. Na Seção 4.2 são definidas as correspondências entre o Projeto Axiomático e o Processo Unificado. Nesta Seção, uma correspondência entre os domínios do Projeto Axiomático e as fases do Processo Unificado é estabelecida e são definidas as correspondências entre os conceitos básicos do Projeto Axiomático e os conceito básicos do Processo Unificado. Na Seção 4.3 são apresentados o contexto no qual a abordagem proposta se insere no desenvolvimento de *software*, as etapas da abordagem proposta e as atividades realizadas em cada etapa. A aplicação do Axioma da Independência em projetos de *software* é analisada e discutida na Seção 4.4. Na Seção 4.5, os processos de decomposição funcional e *ziguezagueamento* do Projeto Axiomático são adaptados para a aplicação em conjunto com o Processo Unificado sendo estabelecido um arcabouço para a decomposição funcional específico para o desenvolvimento orientado a objetos.

4.1 Introdução à Abordagem Proposta

Um dos objetivos principais deste trabalho é propor uma abordagem que permita aplicar a Teoria de Projeto Axiomático (SUH, 1990) no desenvolvimento de *software* orientado a objetos, em conjunto com um processo estabelecido como o Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999). Esta abordagem de projeto proposta tem o objetivo de ser tão próxima quanto possível das metodologias de desenvolvimento de *software* orientadas a objetos usadas atualmente com vistas a facilitar seu aprendizado e sua aplicação pelos desenvolvedores. Versões anteriores desta abordagem foram apresentadas em (PIMENTEL; STADZISZ, 2005), (PIMENTEL; STADZISZ, 2006a) e (PIMENTEL;

STADZISZ, 2006b).

A criação de uma abordagem que aplica a Teoria de Projeto Axiomático ao projeto de *software* em conjunto com o Processo Unificado tem como objetivo trazer vantagens ao processo de desenvolvimento. Entre estas vantagens estão o estabelecimento de critérios para tomada de decisões de projeto como escolher o conjunto mais apropriado de requisitos funcionais (casos de uso) a ser realizado e escolher a melhor solução de projeto (classes, objetos e interações) entre as possíveis alternativas para satisfazer o conjunto de requisitos.

A abordagem de projeto proposta nesta tese foi definida seguindo algumas etapas. Primeiramente, foram definidas as correspondências entre os conceitos básicos do Projeto Axiomático e os conceitos básicos do Processo Unificado. Uma relação de correspondência entre os domínios do Projeto Axiomático e as fases do Processo Unificado foi estabelecida. As etapas e atividades da metodologia proposta foram definidas bem como sua relação com as fases e fluxos de processos do Processo Unificado. Os processos de decomposição funcional e *ziguezagueamento* do Projeto Axiomático foram adaptados para a aplicação em conjunto com o Processo Unificado, tendo sido estabelecido um arcabouço para a decomposição funcional, específico para o desenvolvimento orientado a objetos. A aplicação do Axioma da Independência em projetos de *software* foi analisada e discutida. Por fim, um arcabouço para o cálculo do conteúdo de informação para projetos de *software* orientado a objetos foi definido (ver capítulo 5).

Em grandes sistemas de informação, as tarefas de desenvolvimento são freqüentemente divididas entre várias equipes. Muitas vezes estas equipes estão distribuídas pelo mundo, o que caracteriza o chamado Desenvolvimento Global de *Software* (CARMEL, 1999). Neste contexto é muito importante reduzir a dependência entre os times ao mínimo indispensável para aumentar a autonomia das equipes. A aplicação do Axioma da Independência no desenvolvimento de *software* pode ajudar a garantir que uma mudança em um parâmetro de projeto (DP) que satisfaz um requisito funcional (FR) não afete outros requisitos funcionais (FR). Em outra palavras, se o projeto satisfaz o Axioma da Independência, a alteração de uma classe do sistema por uma equipe, pouco ou nada influenciaria o trabalho das outras equipes de desenvolvimento. Por esta razão, é mais fácil coordenar os esforço de várias equipes de desenvolvimento se a solução de projeto satisfaz o Axioma da Independência.

Uma das características de projetos de *software* é que os requisitos funcionais (FR) freqüentemente sofrem mudanças durante o ciclo de vida do *software* (BROOKS, 1987). Estas mudanças podem ocorrer devido à necessidade de evolução do *software*, a mudanças na

organização na qual o *software* é aplicado ou à necessidade de manutenção, entre outras. Uma mudança em um requisito funcional (FR) freqüentemente gera mudanças nos parâmetros de projeto (DP) correspondentes, como classes, por exemplo. Esta mudança pode causar um impacto em outros requisitos funcionais. Este impacto pode ser minimizado com a aplicação do Axioma da Independência (SUH, 1990).

A priorização de casos de uso é uma das atividades previstas no Processo Unificado. O objetivo desta atividade é guiar a fase de construção, determinando quais casos de uso podem ser realizados (analisados, projetados, implementados e testados) nas primeiras iterações e quais podem ser realizados mais tarde (JACOBSON; BOOCH; RUMBAUGH, 1999). Se uma solução de projeto de *software* orientado a objetos é semi-acoplada, de acordo com Suh (2001), "...a independência dos requisitos funcionais (FR) pode ser garantida se e somente se os parâmetros de projeto (DP) estiverem determinados em uma seqüência apropriada" (SUH, 2001). Portanto, a aplicação do Projeto Axiomático, com seus axiomas, teoremas, corolários e ferramentas, pode auxiliar a atividade de priorização de casos de uso, facilitando a identificação de uma ordem apropriada para a realização dos casos de uso.

Nas abordagens tradicionais de projeto de sistemas de *software* não se costuma fazer decomposição funcional detalhada mas apenas decomposição estrutural, Diferentemente, na Teoria de Projeto Axiomático é proposta a decomposição dos requisitos funcionais em uma estrutura hierárquica detalhada. A decomposição dos requisitos funcionais, e não apenas dos componentes do sistema, aprimora a atividade de análise permitindo a validação dos requisitos funcionais identificados em relação aos dois axiomas. Isto é importante para permitir a verificação da qualidade da solução de projeto desde o início do processo de projeto. Este tipo de validação permite descartar soluções inadequadas logo nas fases iniciais do desenvolvimento, evitando esforços desnecessários mais tarde. Além disso, a decomposição dos requisitos funcionais provê um número maior de níveis de abstração, permitindo um maior detalhamento para as atividades de análise e projeto.

Nas metodologias de desenvolvimento de *software* tradicionais os requisitos funcionais (FR) são elicitados e organizados por meio de casos de uso (JACOBSON; BOOCH; RUMBAUGH, 1999), sem levar em conta como serão satisfeitos. Só após o término da sua especificação é que o projeto será iniciado, com os casos de uso identificados servindo de base para a criação da estrutura do *software*. Este fato pode ocasionar a criação de classes com base em um conjunto inapropriado de requisitos funcionais, o que pode causar problemas para a realização do *software* como a criação de uma arquitetura que exigirá muito esforço para ser implementada e que pode não satisfazer aos requisitos funcionais (FR) (ver Seção

2.1.3).

Na abordagem proposta, a independência funcional não avalia apenas as relações entre os componentes do sistema e a sua coesão, como nas metodologias tradicionais de desenvolvimento de *software* (PRESSMAN, 2005), mas avalia como os requisitos funcionais são dependentes entre si, com relação aos parâmetros de projeto que os satisfazem. O conceito de independência funcional da Teoria de Projeto Axiomático avalia o conjunto de requisitos funcionais (FR) (casos de uso), concomitantemente com o conjunto de parâmetros de projeto (DPs) (classes) criado em função destes requisitos. Isto aplicado ao projeto de *software* orientado a objetos, vincula fortemente as classes aos casos de uso que elas devem realizar.

A abordagem proposta nesta tese, trabalha em conjunto a identificação dos requisitos funcionais (FR) e a criação dos parâmetros de projeto (DP) correspondentes, tanto no processo de desenvolvimento da solução quanto como critério de qualidade de projeto. Por exemplo, para escolher a melhor solução para um projeto entre dois conjuntos de classes, objetos e suas interações, essas classes devem ser analisadas com relação aos casos de uso que elas realizam. Ao mesmo tempo, para identificar o conjunto de casos de uso mais adequado para o sistema, os casos de uso devem ser analisados com relação às classes que os realizam. A Teoria de Projeto Axiomático avalia os requisitos funcionais (FR) a partir dos parâmetros de projeto (DP) que os satisfazem, mantendo sempre essa correspondência. Além disso, estes parâmetros de projeto (DP) servem de base para a decomposição dos requisitos funcionais (FR) em um processo onde os parâmetros de projeto (DP) dependem dos requisitos funcionais (FR) e os novos requisitos funcionais (FR) dependem dos parâmetros de projeto (DP).

O uso das ferramentas do Projeto Axiomático como a matriz de projeto e a hierarquia funcional facilita a rastreabilidade dos parâmetros de projeto (DP) com relação aos requisitos funcionais (FRs) correspondentes. Para cada requisito funcional (FR) é identificado um conjunto de parâmetros de projetos (DPs) que o satisfaz (ver Capítulo 3). Isto relaciona cada parâmetro de projeto com os requisitos funcionais correspondentes. Esta correspondência está mapeada na matriz de projeto. Por sua vez, cada requisito funcional de baixo nível está associado aos requisitos de alto nível de abstração através da hierarquia funcional. Este mecanismo permite relacionar um componente do *software* ao requisito funcional de alto nível que ele satisfaz.

Esta capacidade de rastreabilidade dos parâmetros de projeto, como classes, objetos e suas interações para seus requisitos funcionais de alto nível, é importante no projeto de *software*. A rastreabilidade é uma técnica comprovada que pode aumentar a qualidade e a confiabilidade do *software*, minimizando o impacto de mudanças (LEFFINGWELL;

WIDRIG, 2003). Esta capacidade é especialmente desejada em sistemas críticos como os de aviônica. A norma DO - 178 B (RTCA, 1999) exige a rastreabilidade entre vários níveis de requisitos e o código fonte. Segundo esta norma, deve haver rastreabilidade entre os requisitos do sistema e os requisitos de alto nível, entre os requisitos de alto nível e os requisitos de baixo nível e entre os requisitos de baixo nível e o código fonte (ESTEREL, 2006). A norma "*Software Considerations in Airborne Systems and Equipment Certification*" (DO – 178 B) (RTCA, 1999) foi estabelecida como norma para *software* de aviação pela *Federal Aviation Administration* (FAA) do governo dos EUA e é considerada a norma de *software* mais exigente na atualidade.

Entre outros objetivos, este tipo de rastreabilidade visa estabelecer vínculos entre os vários estágios de testes, como por exemplo, vincular os testes de unidade aos testes de integração, estes por sua vez aos casos de teste e estes aos testes de sistema. A abordagem proposta nesta tese pode ajudar a se atingir este nível de rastreabilidade através da aplicação dos axiomas, teoremas, ferramentas e processos da Teoria de Projeto Axiomático no projeto de *software*.

4.2 Projeto Axiomático e Projeto de *Software* Orientado a Objetos

Como apresentado no Capítulo anterior, a Teoria de Projeto Axiomático, devido à sua natureza independente de domínio, é aplicável a vários tipos de projeto, desde o projeto de um carro até o projeto de um departamento acadêmico (SUH, 2001). Para a sua aplicação no desenvolvimento de *software* orientado a objetos, mais especificamente em conjunto com o Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999), são estabelecidos nesta tese, relacionamentos entre os principais conceitos do Projeto Axiomático e os principais conceitos de projeto de *software* orientado a objetos. Além disso, para que a Teoria de Projeto Axiomático se adapte ao Processo Unificado, esta tese define as relações entre as fases do Processo Unificado e os domínios do Projeto Axiomático conforme descrito a seguir.

4.2.1 Conceitos de Projeto Orientado a Objetos e Conceitos do Projeto Axiomático

De acordo com Suh (2001), "requisitos funcionais são o conjunto mínimo de requisitos independentes que caracterizam as necessidades funcionais do produto" (SUH,

2001). Segundo Jacobson; Booch; Rumbaugh (1999), "...casos de uso [...] oferecem uma forma sistemática e intuitiva para capturar requisitos funcionais com o foco no valor adicionado ao usuário" (JACOBSON; BOOCH; RUMBAUGH, 1999). Portanto, casos de uso podem ser usados para representar o conceito de requisito funcional (FR) do Projeto Axiomático.

Em trabalhos anteriores, Suh (2001) e Suh; Do (2000), descreveram e utilizaram um processo de desenvolvimento orientado a objetos (SUH, 2001), (SUH; DO, 2000). Neste processo, chamado "*Axiomatic Design for Object-Oriented Software System*" (Ado-oSS), uma relação estreita entre métodos, atributos (componentes de objetos) e parâmetros de projeto (DP) foi estabelecida (SUH, 2001). Apesar disso, nestes dois trabalhos, objetos do sistema são considerados como requisitos funcionais (FRs) o que difere, tanto do conceito de requisito funcional, como da forma de utilização de um objeto, não sendo compatível com as metodologias atuais de desenvolvimento de *software*. Do (2004) apresentou um modelo de gerenciamento de ciclos de vida de *software* baseado em casos de uso que aplica a abordagem axiomática, além de estabelecer uma relação de mapeamento entre casos de uso e requisitos funcionais (DO, 2004).

Os requisitos funcionais são representados por diferentes conceitos da UML de acordo com o nível de abstração, ou mesmo, de acordo com o nível na hierarquia funcional. De acordo com o nível de hierarquia funcional, os requisitos funcionais são representados nesta tese por: casos de uso, subcasos de uso independentes de características técnicas, sub-casos de uso dependentes de características técnicas e serviços técnicos.

De acordo com a especificação da UML (versão 2.0), "um caso de uso é a especificação de um conjunto de ações executados pelo sistema que produz um resultado observável que é, tipicamente, de valor para um ou mais atores" (OBJECT MANAGEMENT GROUP, 2005). Esta definição permite considerar qualquer tipo ou conjunto de ações do sistema como um caso de uso mesmo que não esteja relacionado com um ator, ou mesmo representar uma parte, pequena ou grande, de uma funcionalidade do sistema.

A especificação da UML (versão 2.0) também afirma que o sujeito de um caso de uso pode ser um sistema físico ou qualquer outro elemento de um sistema como componentes, subsistemas ou classes (OBJECT MANAGEMENT GROUP, 2005). Então a definição de casos de uso pode se estender até mesmo a um comportamento esperado de uma classe ou objeto. Devido a esta flexibilidade na definição de casos de uso e para melhor categorizar os diferentes níveis de decomposição funcional, esta tese utilizará uma definição mais restrita para casos de uso.

Definição 1 (Casos de Uso) *Um caso de uso é a especificação de um conjunto de ações que representa uma utilização completa do sistema por um ou mais atores.*

Uma utilização completa do sistema pode ser definida como o uso de um sistema por um ator para realizar um processo do começo ao fim que traga um resultado de valor para ele. A definição de casos de uso como sendo uma utilização completa do sistema, também contempla a funcionalidade de subsistemas e suas divisões que serão também considerados como casos de uso. Isto permite que as funcionalidades de alto nível como as providas por subsistemas sejam consideradas como casos de uso, como sugerido em (LEFFINGWELL; WIDRIG, 2003). Quando a partir da decomposição de um caso de uso forem obtidos requisitos funcionais que representem utilizações do sistema que não são completas, estes requisitos funcionais serão chamados neste trabalho de "subcasos de uso".

Definição 2 (Subcasos de Uso) *Um subcaso de uso é a especificação de um conjunto de ações que representa uma utilização não completa do sistema ou subsistema. Um subcaso de uso representa uma parte de uma utilização completa do sistema, ou seja, parte de um caso de uso.*

Um subcaso de uso é a representação de uma funcionalidade que compõe um caso de uso. Esta funcionalidade pode ser uma utilização não completa do sistema por um ator, uma funcionalidade provida por um componente de *software*, ou alguma funcionalidade importante do sistema mesmo que não esteja relacionada diretamente à interação com atores.

No início do projeto de um sistema, as funcionalidades ou requisitos funcionais são identificados sem considerar características técnicas da solução, de forma que o projetista possa se concentrar nas funcionalidades mais importantes. Nas etapas seguintes, as características técnicas da solução se tornam uma importante fonte de informação para o projeto, como por exemplo na identificação de restrições. Nesta fase, a identificação dos requisitos funcionais deve levar cada vez mais em consideração as características técnicas da solução. Com isso, podemos classificar os subcasos de uso em: independentes de características técnicas e dependentes de características técnicas.

Definição 3 (Subcasos de uso independentes) *Um subcaso de uso independente de características técnicas é a especificação de um conjunto de ações que um sistema executa que representa uma utilização não completa do sistema (uma parte de um caso de uso) que é especificada sem levar em consideração características técnicas empregadas na solução.*

Característica técnica é uma característica ligada à forma de realização de um requisitos funcional. Representa uma forma técnica de implementar a solução, como por exemplo: entrar na fila de conexões de um sistema gerenciador de bancos de dados, efetuar a verificação de redundância cíclica (CRC) dos dados vindos pela porta serial, entre outros.

Definição 4 (Subcasos de uso dependentes) *Um subcaso de uso dependente de características técnicas é a especificação de um conjunto de ações que um sistema executa que representa uma utilização não completa do sistema (uma parte de um caso de uso) que é especificada considerando fortemente características técnicas empregadas na solução.*

De acordo com as necessidades dos projetistas, os subcasos de uso podem representar conceitos diferentes. O refinamento dos casos de uso pode ser feito através de várias formas. Segundo a especificação da UML na versão 2.0, os casos de uso podem ser especificados através de mudanças de estado, de atividades ou na forma de texto em linguagem natural como cenários ou pré-condições e pós-condições (OBJECT MANAGEMENT GROUP, 2005). Também podem ser usados para este refinamento casos de uso incluídos, casos de uso de extensão e fluxos de eventos (LEFFINGWELL; WIDRIG, 2003). Podem ser usados também, casos de uso de utilidade e casos de uso de infra-estrutura (JACOBSON; NG, 2004). Pode-se usar para o refinamento de casos de uso, inclusive, representações de comportamento de partes significativas do sistema.

A definição de subcasos de uso, adotada nesta tese, permite que o projetista represente o refinamento dos casos de uso na forma que desejar sem prejuízo para o processo da abordagem proposta. Os subcasos de uso independentes de características técnicas podem representar os refinamentos indicados para casos de uso como casos de uso incluídos, casos de uso de extensão e fluxos de eventos (LEFFINGWELL; WIDRIG, 2003), bem como os tipos de refinamentos de casos de uso indicados por Jacobson; NG (2004) que são os casos de usos de infra-estrutura e os casos de uso de utilidade (JACOBSON; NG, 2004). Já os subcasos de uso dependentes de características técnicas podem representar conceitos que aparecem em etapas posteriores no refinamento dos casos de uso, como as operações de sistema. As operações de sistema representam operações realizadas pelo sistema na sua reação a um evento de interface causado pelo usuário ou ator (LARMAN, 2004).

Enquanto o projetista está identificando casos de uso e subcasos de uso, ele está tratando de funcionalidades do sistema como um todo ou de partes, componentes ou conjunto de classes do sistema. A partir de uma certa fase, o projetista tem a necessidade de representar

as funcionalidades relativas a cada uma das classes do sistema. A partir desta fase, serão consideradas as funcionalidades esperadas das classes ou objetos do sistema, sendo os requisitos funcionais (FRs), agora, representados pelos serviços técnicos.

A definição de um serviço técnico é adaptada da definição de função de serviço (STADZISZ, 1997). Segundo STADZISZ (1997), "Uma função de serviço exprime uma ação esperada do produto sobre um elemento do meio exterior, em benefício de outro elemento deste meio, dentro de uma fase de utilização" (STADZISZ, 1997). Para a definição de serviço técnico, um objeto do sistema será considerado como um produto e os outros objetos, com os quais interage, seu meio externo. Então, um serviço técnico é um serviço esperado de um objeto por outro para realizar uma colaboração.

Definição 5 (Serviços Técnicos) *Um serviço técnico é um serviço esperado por um objeto, classe ou componente do sistema de outro objeto, classe ou componente do sistema*.

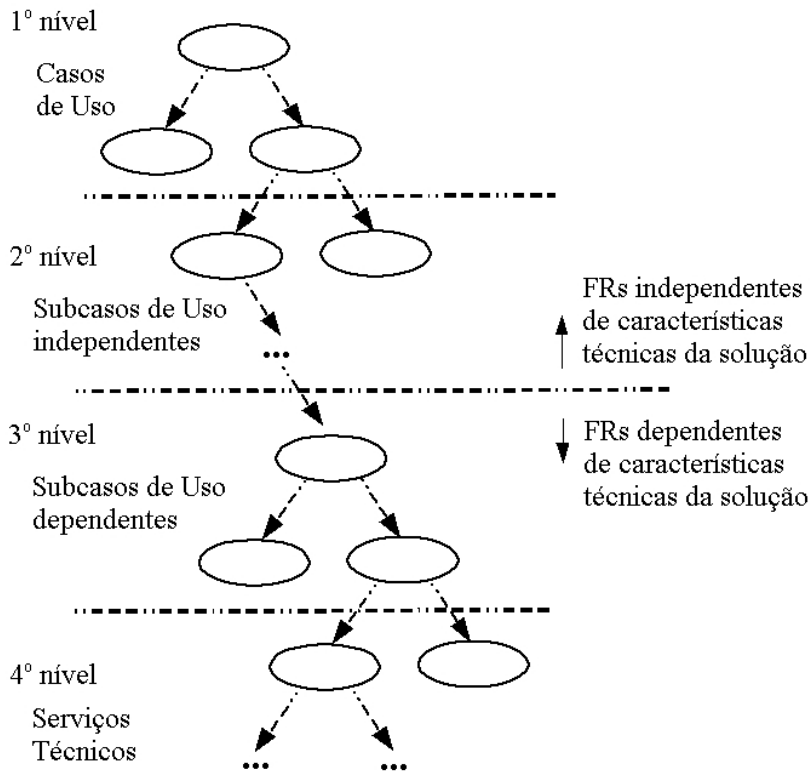


Figura 12 - Níveis da Hierarquia funcional proposta

O conceito de serviço técnico é semelhante ao conceito de responsabilidade de uma classe, definido por Beck e Cunningham, em que uma responsabilidade define um problema a

ser resolvido pela classe (BECK; CUNNINGHAM, 1989). A Figura 12 apresenta uma hierarquia funcional com os diferentes conceitos que, na abordagem proposta nesta tese, representam os requisitos funcionais em cada nível de abstração.

O modelo de hierarquia funcional, apresentado na Figura 12, é obtido através de um processo de decomposição dos requisitos funcionais. Como se trata de um processo de decomposição, os requisitos funcionais de níveis mais baixos agrupados devem descrever na totalidade os requisitos funcionais de nível mais alto (RTCA, 1999).

Parâmetros de projeto (DPs) são as variáveis-chave que caracterizam o projeto e que satisfazem os requisitos funcionais (FRs) específicos (SUH, 2001). O conceito de colaboração é definido pela UML 2.0. Uma colaboração descreve uma estrutura de elementos (papéis) que colaboram entre si para realizar uma determinada funcionalidade (OBJECT MANAGEMENT GROUP, 2005).

Uma colaboração é o conceito da UML responsável por representar a realização de um caso de uso (OBJECT MANAGEMENT GROUP, 2005). Isto significa que as colaborações podem ser consideradas como parâmetros de projeto (DPs) em níveis de abstração mais altos do projeto. As colaborações podem ser divididas em subcolaborações que representam interações menores que compõem as colaborações originais.

Com a identificação das colaborações, são identificadas, também, os papéis (*roles*) participantes na colaboração que interagem para realizar o caso de uso. Um papel em uma colaboração é a descrição de um participante na colaboração (RUMBAUGH; JACOBSON; BOOCH, 2004). Estes papéis, freqüentemente serão tomadas como base para a criação dos objetos e classes em níveis de abstração mais baixos. Um exemplo de colaboração e seus papéis usados na realização de um caso de uso está representado na Figura 13.

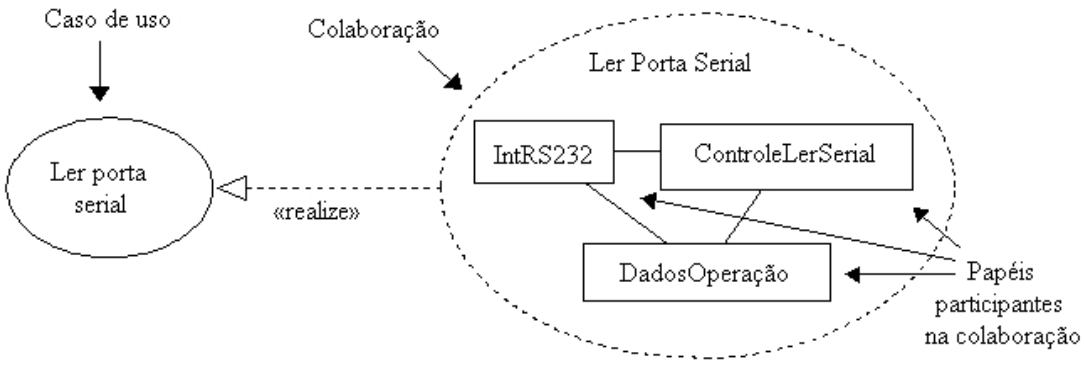


Figura 13 - Caso de uso, colaboração e papéis da colaboração

Segundo Booch; Rumbaugh; Jacobson (2005), "... classes são os blocos de construção mais importantes de qualquer sistema de *software* orientado a objetos" (BOOCH; RUMBAUGH; JACOBSON, 2005). Uma classe descreve um conjunto de objetos que compartilham os mesmos atributos, operações e relacionamentos. As classes e suas instâncias (objetos) que participam em colaborações para realizar os casos de uso do sistema representam os parâmetros de projeto (DPs) da solução. Diagramas de classes representam o modelo estrutural das classes do sistema enquanto que diagramas de interação e de estados, entre outros, compõem o modelo dinâmico das classes do sistema (BOOCH; RUMBAUGH; JACOBSON, 2005).

A Tabela 1 apresenta um quadro que relaciona os requisitos funcionais (FR) de acordo com o nível de abstração do projeto e os correspondentes parâmetros de projeto (DP). A abordagem proposta nesta tese define quatro níveis de abstração com relação aos requisitos funcionais. Estes níveis de abstração servirão de base para o processo de decomposição funcional (ver Seção 4.5.1) e para a definição das etapas da abordagem (ver Seção 4.3.1).

Tabela 1 - Requisitos Funcionais e Parâmetros de Projeto correspondentes

Nível de abstração	**Requisitos Funcionais (FR)**	**Parâmetros de Projeto (DP)**
1	Casos de uso	Colaborações e seus papéis
2	Subcasos de uso independentes de características técnicas	Subcolaborações e seus papéis
3	Subcasos de uso dependentes de características técnicas	Subcolaborações e seus papéis
4	Serviços técnicos	Objetos e Classes

No primeiro nível de abstração, os requisitos funcionais (FRs) são representados pelos casos de uso e os correspondentes parâmetros de projeto (DP) pelas colaborações e seus papéis. Os subcasos de uso independentes de características técnicas representam os requisitos funcionais (FRs) no segundo nível de abstração com os correspondentes parâmetros de projeto (DP) sendo as colaborações e seus papéis. No terceiro nível de abstração, os requisitos funcionais (FRs) são representados pelos subcasos de uso dependentes de características técnicas com os correspondentes parâmetros de projeto (DP) sendo as colaborações e seus papéis. No quarto nível estão os serviços técnicos como requisitos funcionais (FR) e classes e objetos como parâmetros de projeto (DP).

Restrições (Cs) são limites para soluções aceitáveis. As restrições de sistema são

restrições impostas pelo contexto do sistema no qual a solução de projeto deve funcionar (SUH, 2001). Requisitos funcionais (FR) podem ser definidos para lidar com estas restrições. Similares às restrições, requisitos não funcionais usualmente necessitam de algum tipo de funcionalidade do sistema para serem satisfeitos. Estas funcionalidades em particular, também chamadas de mecanismos de infra-estrutura, podem ser modeladas por um tipo de casos de uso chamados casos de uso de infra-estrutura (JACOBSON; NG, 2004).

4.2.2 Fases do Processo Unificado e os Domínios do Projeto Axiomático

Pode-se estabelecer uma relação entre as fases do Processo Unificado e os domínios do Projeto Axiomático. De acordo com Suh (SUH, 2001), o processo de desenvolvimento de um produto envolve quatro domínios diferentes como mostrado na Figura 14: o domínio do cliente, o domínio funcional, o domínio físico e o domínio de processo (ver Capítulo 3).

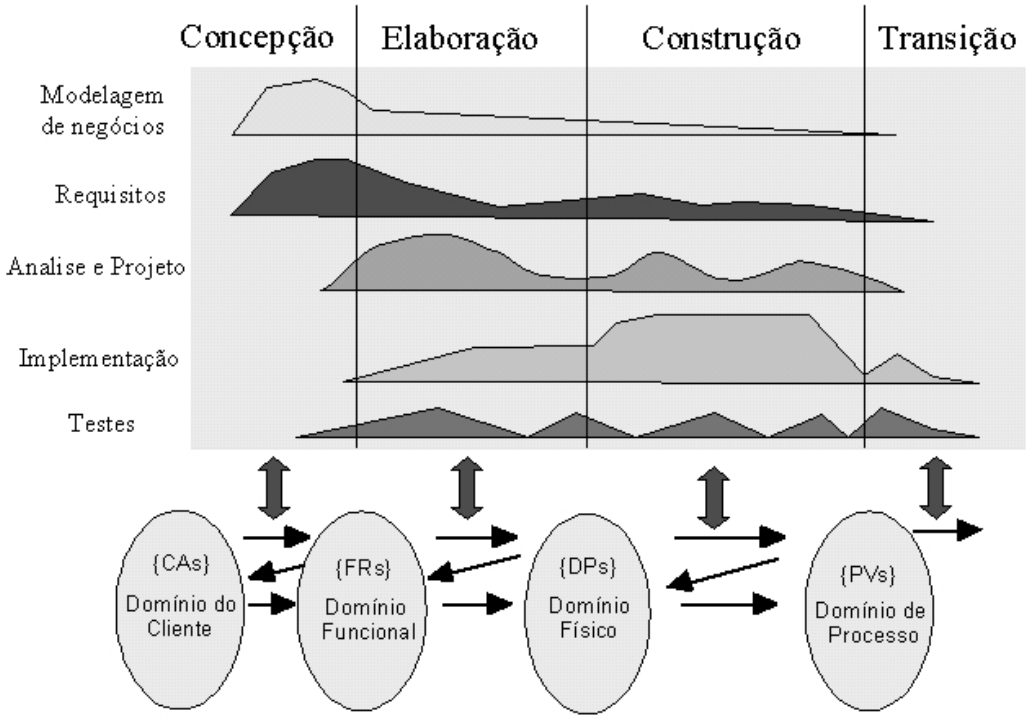


Figura 14 - Domínios de Projeto Axiomático e as fases do Processo Unificado

O domínio do cliente representa as necessidades dos cliente por meio dos atributos dos clientes (CAs). A atividade de projeto propriamente dita é realizada pelo mapeamento entre os requisitos funcionais (FRs) no domínio funcional e os parâmetros de projeto (DPs) no domínio físico. O domínio de processo representa os meios para realizar o produto através das

variáveis de processo (PVs) (SUH, 2001).

O ciclo de vida do Processo Unificado é composto por quatro fases: concepção[14], elaboração, construção e transição (JACOBSON; BOOCH; RUMBAUGH, 1999). Na fase de concepção são criados um modelo do negócio que descreve a organização alvo do sistema, uma visão do sistema que descreve o contexto, escopo e as características do *software*, e a partir deles, a identificação dos casos de uso principais do sistema, criando um modelo inicial de casos de uso (JACOBSON; BOOCH; RUMBAUGH, 1999).

O modelo do negócio e a visão do sistema criados nesta fase representam as necessidades do cliente (CAs) para o sistema, e os casos de uso principais representam os requisitos funcionais (FRs) iniciais do sistema. Do ponto de vista do Projeto Axiomático, as necessidades do cliente (CAs) descritas através do modelo de negócio estão inseridas no domínio do cliente e os casos de uso principais estão representados no domínio funcional. A fase de concepção representa então o mapeamento entre o domínio do cliente e o domínio funcional.

Um dos objetivos principais da fase de elaboração é capturar especificar e refinar a maior parte dos requisitos funcionais (FRs) e formulá-los como casos de uso (JACOBSON; BOOCH; RUMBAUGH, 1999). Além disso, na fase de elaboração é criada uma versão da arquitetura do sistema (componentes, classes, objetos e interações). No domínio funcional do Projeto Axiomático, os requisitos funcionais (FRs) são decompostos ao mesmo tempo que são criados parâmetros de projeto (DPs) no domínio físico (SUH, 2001). Portanto, casos de uso representam o mesmo papel que os requisitos funcionais (FRs) do Projeto Axiomático e a arquitetura do sistema, com seus componentes, classes, objetos e interações representam parâmetros de projeto (DPs) no domínio físico. A fase de elaboração do Processo Unificado tem como um de seus produtos o refinamento do modelo de casos de uso (domínio funcional) e uma versão da arquitetura do sistema (domínio físico). Isto significa que a fase de elaboração está relacionada com o mapeamento entre os domínios funcional e físico do Projeto Axiomático.

De acordo com (JACOBSON; BOOCH; RUMBAUGH, 1999), na fase de construção do Processo Unificado, a arquitetura do sistema, representada pelas classes do sistema, objetos e a interação entre eles, é refinada, construída (implementada em código) e testada, culminando em uma versão estável do *software*. A fase de construção começa com uma versão da arquitetura já com quase a totalidade das classes do sistema, objetos e interações, considerados como parâmetros de projeto (DP) e representados no domínio físico

[14] Traduzido do inglês *inception*, pelo autor

do Projeto Axiomático. Os compiladores e ferramentas de teste usados na sua construção podem ser consideradas como variáveis de processo (PVs) do domínio de processo. Portanto, a fase de construção do Processo Unificado corresponde ao mapeamento entre os domínios físico e de processo do Projeto Axiomático.

Na fase de transição, a partir de uma versão estável do produto de *software*, este é instalado configurado e distribuído. São construídos alguns artefatos, e entre eles: *software* de instalação, manuais e uma descrição da arquitetura do *software*.

4.3 Contexto, Etapas e Atividades da Abordagem Proposta

Além das 4 fases, o Processo Unificado possui fluxos de processo que são realizados passando por elas. Os fluxos de processo centrais, definidos por Jacobson; Booch; Rumbaugh (1999), são: fluxo de requisitos, fluxo de análise, fluxo de projeto, fluxo de implementação e fluxo de teste (JACOBSON; BOOCH; RUMBAUGH, 1999).

De acordo com Suh (2001), o desenvolvimento de um produto é realizado através de um mapeamento entre os quatro domínios (do Cliente, Funcional, Físico e de Processo). Entretanto, a atividade de projeto propriamente dita é realizada através do mapeamento entre os domínios funcional e físico apenas (SUH, 2001). A abordagem proposta nesta tese se concentra na atividade de projeto de *software*, ou seja, se concentra no mapeamento entre os domínios funcional e físico, já que as maiores contribuições da teoria de projeto axiomático, axiomas, teoremas, corolários e ferramentas, estão focadas no mapeamento entre estes dois domínios.

No mapeamento entre os domínios funcional e físico, além de se definir um conjunto adequado de requisitos funcionais, um dos objetivos é criar parâmetros de projeto para satisfazer esses requisitos. No Processo Unificado, este mapeamento fica inserido no processo de realização dos casos de uso. A realização dos casos de uso é composta de vários fluxos de processo, entre eles: fluxo de requisitos, fluxo de análise, fluxo de projeto, fluxo de implementação e fluxo de teste (JACOBSON; BOOCH; RUMBAUGH, 1999). A abordagem proposta nesta tese está inserida no contexto dos fluxos de requisitos, de análise e de projeto, não contemplando aos fluxos de implementação e testes. A Figura 15 representa os fluxos de processo da realização de casos de uso e mostra a relação da abordagem proposta com estes fluxos.

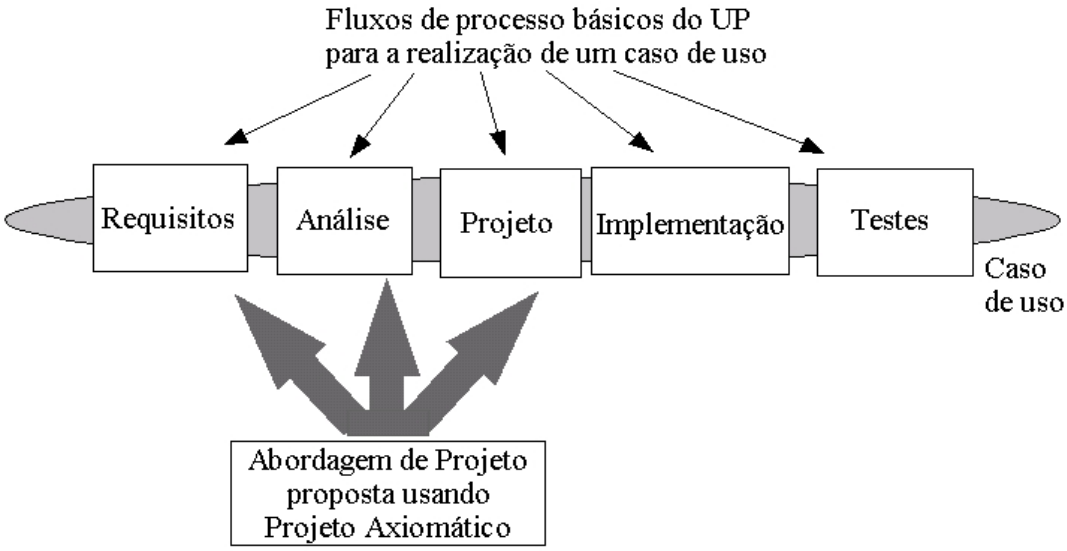


Figura 15 - A abordagem proposta na realização de casos de uso no Processo Unificado

4.3.1 Etapas da Abordagem

A abordagem proposta segue um processo de decomposição dos requisitos funcionais proposto pela Teoria de Projeto Axiomático, adaptado para o projeto de *software* orientado a objetos. Esta decomposição está definida em função dos níveis de abstração propostos na Seção 4.2.1. As etapas da abordagem são baseadas na decomposição funcional proposta, em que cada etapa corresponde à decomposição funcional em cada um destes níveis de abstração. Assim, a abordagem possui quatro etapas distintas que são:

- Etapa 1 – Modelagem de casos de uso e colaborações.
- Etapa 2 – Modelagem de subcasos de uso independentes e subcolaborações.
- Etapa 3 – Modelagem de subcasos de uso dependentes e subcolaborações.
- Etapa 4 – Modelagem de serviços técnicos, objetos e interações.

As etapas da abordagem proposta, suas entradas, saídas e controles estão representados, em um diagrama SADT, na Figura 16. A Seção 4.5 apresenta uma descrição detalhada das etapas, atividades executadas e processo de decomposição funcional realizados em cada etapa.

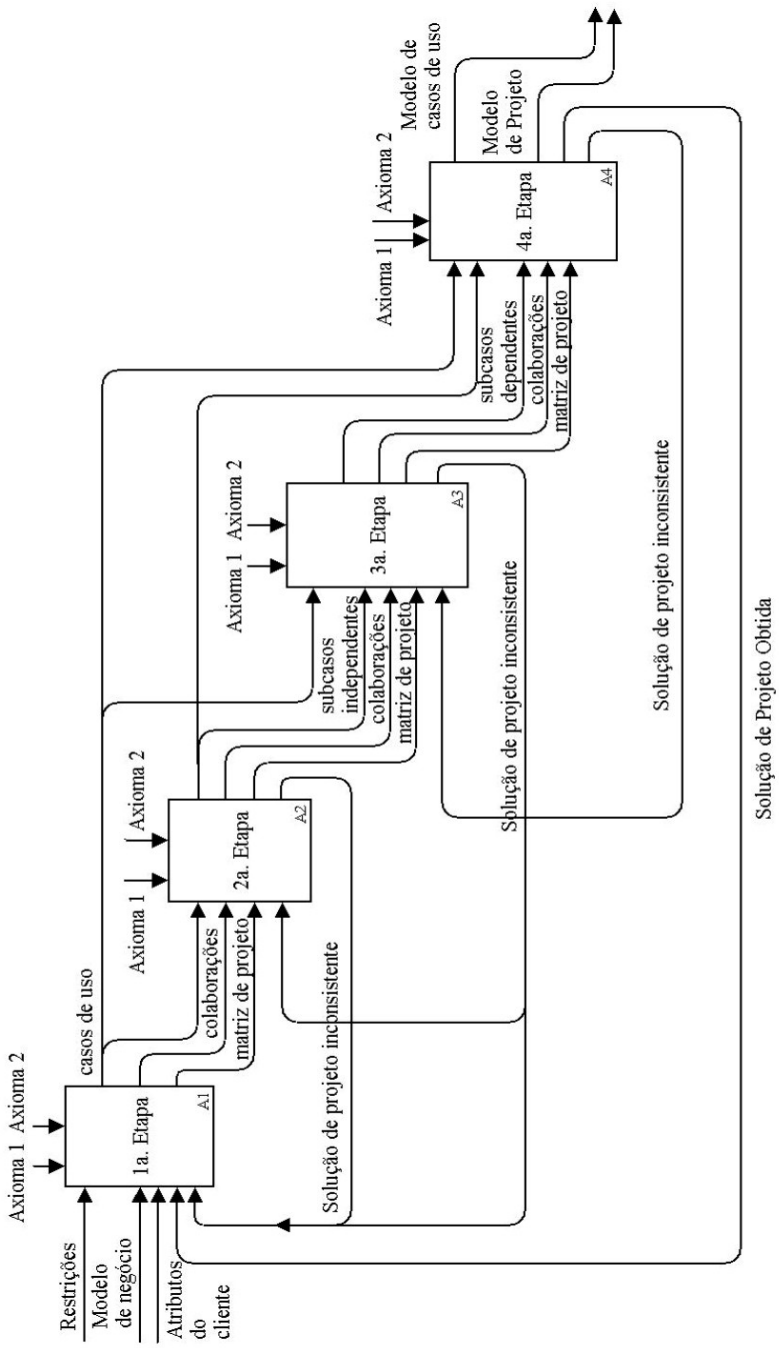


Figura 16 - Etapas da abordagem proposta

As quatro etapas da abordagem estão relacionadas em uma seqüência formando um ciclo. Este ciclo pode ser executado uma ou mais vezes durante o projeto, dependendo do ciclo de vida adotado. Na Proposta original da Teoria de Projeto Axiomático, o ciclo é executado uma só vez, sendo que as decomposições em todos os níveis são feitas em largura. Na decomposição em largura, todos os requisitos funcionais de um nível são decompostos antes de se partir para um outro nível. No caso de um ciclo de vida iterativo e incremental, o ciclo pode ser executado mais de uma vez, sendo que cada execução corresponde a realização

de um ou mais casos de uso do sistema. A decomposição dos requisitos funcionais, neste caso, é feita em profundidade onde um ramo da árvore de hierarquia é totalmente decomposto antes de se partir para o próximo.

4.3.1.1 Modelagem de Casos de Uso - 1ª. Etapa

A primeira etapa envolve a modelagem de casos de uso e colaborações. Nesta etapa são modelados os casos de uso (de alto nível) do sistema, que são identificados a partir do modelo de negócio, e suas colaborações correspondentes. Nesta etapa podem ser identificados casos de uso referentes ao sistema e subsistemas, que podem ser decompostos enquanto a definição de casos de uso for respeitada, ou seja, enquanto os casos de uso obtidos ainda representarem utilizações completas do sistema por um ator (ver **definição 1)**.

Para cada caso de uso, que correspondem a um requisito funcional (FR), são identificadas colaborações e seus papéis, que correspondem aos parâmetros de projeto (DPs). As entradas principais desta etapa são o modelo de negócio, restrições, atributos do cliente e as versões do projeto já realizadas em iterações anteriores.

Uma entrada secundária pode ser uma inconsistência nas matrizes de projeto obtidas nas etapas seguintes que serve como realimentação para se refazer as relações da matriz de projeto nesta etapa. As saídas principais são os casos de uso, as colaborações correspondentes e a matriz de projeto. Os controles desta etapa são os Axiomas 1 e 2, que indicam quando a etapa pode terminar uma vez que a solução de projeto obtida seja uma boa solução.

4.3.1.2 Modelagem de Subcasos de Uso Independentes de Características Técnicas - 2ª. Etapa

A segunda etapa da abordagem consiste na modelagem de subcasos de uso independentes de características técnicas (ver **definição** 3) e suas colaborações. Os subcasos de uso independentes são identificados com base na decomposição dos casos de uso, e nas colaborações correspondentes. As subcolaborações são criadas para atender os subcasos identificados. Os subcasos de uso independentes podem ser identificados, também, através da descrição da interação do sistema com o usuário. Nesta etapa, os papéis das colaborações já podem servir de base para a identificação das classes de análise, feita no fluxo de análise do Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999).

Nesta etapa está prevista a decomposição dos subcasos de uso independentes até

que se torne necessário considerar as características técnicas adotadas para a solução. É neste nível de abstração que são identificados os casos de uso incluídos, os casos de uso extensão, os casos de uso de infra-estrutura e os casos de uso de utilidade (JACOBSON; NG, 2004). As entradas principais desta etapa são: os casos de uso, as colaborações e a matriz de projeto.

Uma entrada secundária pode ser uma inconsistência nas matrizes de projeto obtidas nas etapas seguintes que serve como realimentação para se refazer as relações da matriz de projeto nesta etapa. As saídas principais são os subcasos de uso independentes e suas decomposições, as colaborações, seus papéis, correspondentes e a matriz de projeto. Os controles desta etapa são os Axiomas 1 e 2, que indicam quando a etapa pode terminar uma vez que a solução de projeto obtida seja uma boa solução.

4.3.1.3 Modelagem de Subcasos de Uso Dependentes de Características Técnicas - 3ª. Etapa

Na terceira etapa, ou etapa de modelagem de subcasos de uso dependentes de características técnicas e colaborações, os subcasos de uso dependentes são identificados modelados com base na decomposição dos subcasos de uso independentes de características técnicas. Os subcasos de uso dependentes podem ser identificados através de características técnicas, normalmente associadas à identificação das restrições (C). Nesta etapa os papéis das colaborações já estabelecidos, podem servir de base para a identificação das classes de análise, feita no fluxo de análise do Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999).

Nesta etapa está prevista a decomposição dos subcasos de uso dependentes até que se torne necessário representar a funcionalidade exigida de uma classe ou objeto do sistema o que caracteriza um serviço técnico (ver **definição** 5). As entradas principais desta etapa são: os casos de uso, os subcasos de uso independentes, as colaborações e a matriz de projeto.

Inconsistências nas matrizes de projeto em níveis seguintes, servem como realimentação para se refazer as relações da matriz de projeto nesta etapa. As saídas principais são os subcasos de uso dependentes e suas decomposições, as colaborações, seus papéis, correspondentes e a matriz de projeto. Nesta etapa, os Axiomas 1 e 2 indicam quando a etapa pode terminar devido a solução de projeto obtida ser uma boa solução.

4.3.1.4 Modelagem de Serviços Técnicos - 4ª. Etapa

Na etapa de modelagem dos serviços técnicos é realizada a identificação e

decomposição dos serviços técnicos além de ser realizada a criação das classes, objetos e iterações que irão compor a solução de projeto para o sistema. As entradas desta etapa são: os casos de uso, os subcasos de uso dependentes e independentes, as colaborações, a matriz de projeto do nível anterior, classes, objetos e interações vindos das iterações anteriores.

Como entrada secundária pode se ter inconsistências nas matrizes de projeto obtidas nas etapas seguintes que indicam que devem ser revistas as relações da matriz de projeto nesta etapa. As saídas principais desta fase são as classes, objetos e interações, que satisfazem os casos de uso correspondentes, representados estruturalmente nos diagramas de classes e dinamicamente nos diagramas de interação e nos *statecharts* (OBJECT MANAGEMENT GROUP, 2005).

4.3.2 Etapas da Abordagem e as Fases do Processo Unificado

As etapas da abordagem proposta estão relacionadas com as fases do Processo Unificado. Na etapa de modelagem de casos de uso, partindo das necessidades dos clientes, da visão do sistema e da modelagem de negócio, os casos de uso são identificados e decompostos. A primeira etapa da abordagem se inicia na fase de concepção e vai até a fase de elaboração.

Na segunda etapa da abordagem, os subcasos de uso independentes de características técnicas são identificados e decompostos. Os subcasos de uso dependentes de características técnicas são identificados na terceira etapa da abordagem. Os papéis das colaborações criadas na segunda e terceira etapas da abordagem, possuem detalhamento em um nível de abstração semelhante às classes do modelo de análise. Neste contexto, a segunda e a terceira etapa da abordagem ocorrem totalmente na fase de elaboração do Processo Unificado.

Na quarta etapa da abordagem, os serviços técnicos são identificados a partir dos subcasos de uso dependentes. Nesta etapa, também, são identificadas as classes, os objetos e suas interações, gerando um modelo equivalente ao modelo de projeto do Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999). Esta etapa termina no início da fase de construção do Processo Unificado.

4.3.3 Atividades da Abordagem Proposta

Cada etapa da abordagem proposta contempla um dos quatro níveis de abstração definidos. Apesar dos níveis de abstração serem diferentes, cada etapa realiza essencialmente

o mesmo conjunto de atividades. A diferença é que a cada etapa, as mesmas atividades são executadas sobre diferentes conceitos. Estas atividades refletem a aplicação da Teoria de Projeto Axiomático e suas ferramentas. Estas atividades estão relacionadas em um ciclo, como representado na Figura 17.

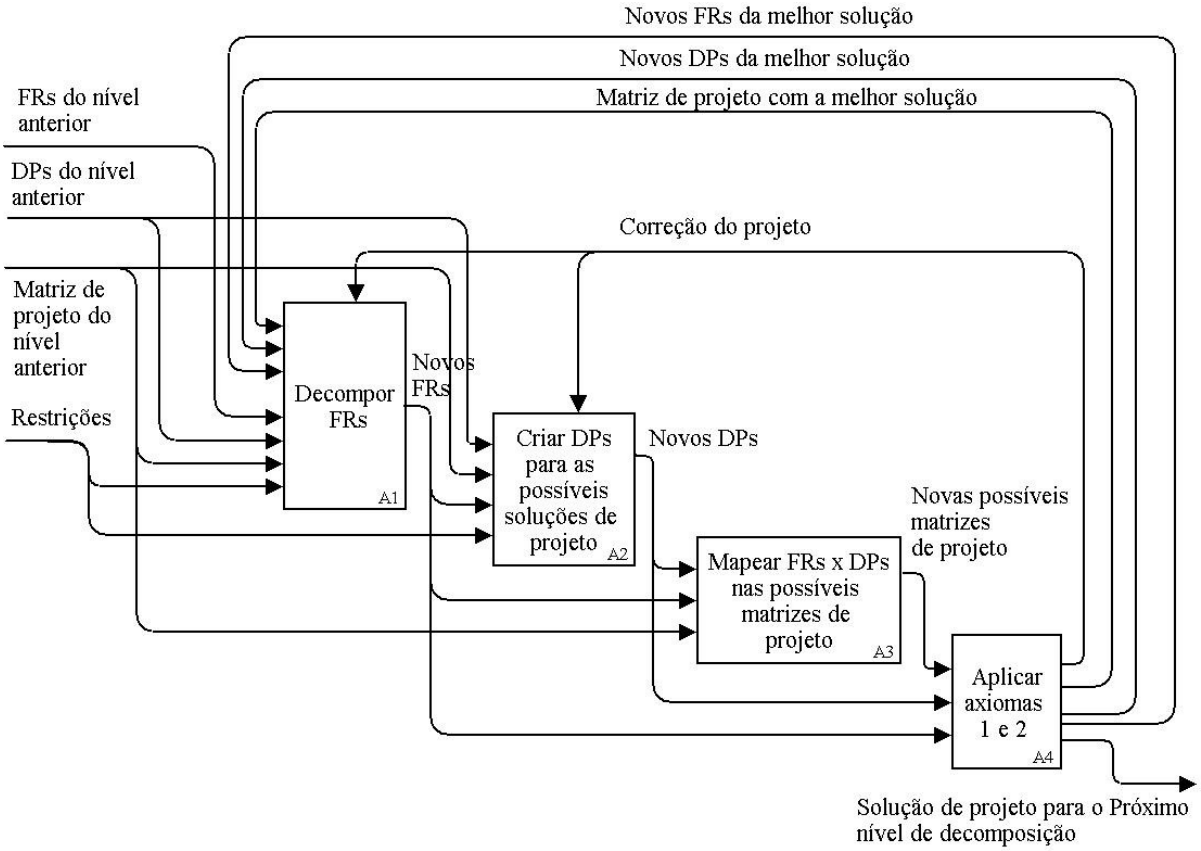


Figura 17 - Atividades realizadas em cada etapa da abordagem proposta

Este ciclo de atividades pode e deve ser realizado tantas vezes quanto desejado para satisfazer as necessidades do projetista para uma determinada etapa. As atividades a serem realizadas em cada uma das etapas são: decompor os requisitos funcionais (FRs), criar parâmetros de projetos (DPs) para satisfazer os requisitos funcionais (FRs) nas possíveis soluções de projeto, mapear as relações entre os requisitos funcionais e os correspondentes parâmetros de projeto nas possíveis matrizes das soluções de projeto e aplicar os Axiomas 1 e 2 para identificar a melhor solução dentre as possíveis para aquele nível de decomposição. A descrição detalhada das atividades executadas em cada uma das etapas é apresentada na Seção 4.5.

Na primeira atividade de cada etapa é identificado um novo conjunto de requisitos funcionais (FRs) através da decomposição dos requisitos anteriores, tomando-se como base os

parâmetros de projeto do nível de decomposição anterior. Como entrada desta etapa tem-se os requisitos funcionais do nível de decomposição anterior, os parâmetros de projeto do nível de decomposição anterior e as restrições. Como saída é obtido um novo conjunto de requisitos funcionais para este nível de decomposição.

Na segunda atividade de cada etapa são criados novos parâmetros de projeto para satisfazer os requisitos funcionais identificados neste nível de decomposição. Estes parâmetros de projeto são criados com base, também, nos parâmetros de projeto da decomposição anterior. Nesta atividade podem ser criadas algumas alternativas de parâmetros de projeto, que serão as possíveis soluções de projeto para este nível. Como entrada desta etapa tem-se os novos requisitos funcionais identificados e os parâmetros de projeto do nível de decomposição anterior. Como saída são obtidos novos conjuntos de parâmetros de projeto para este nível de decomposição.

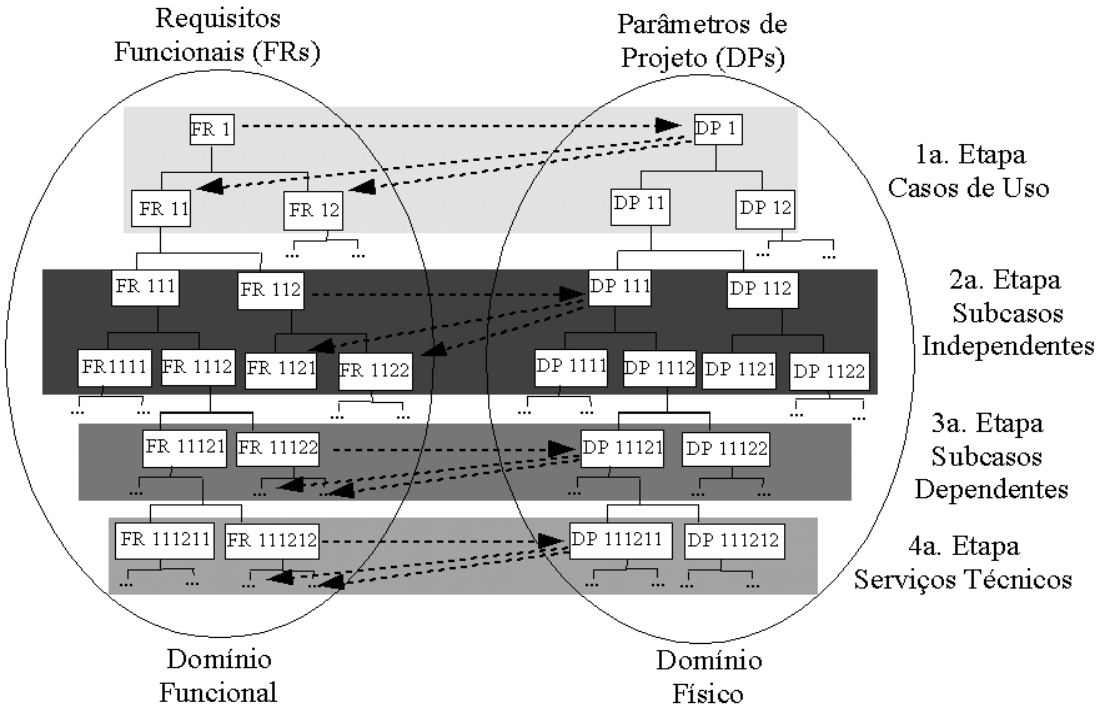


Figura 18 - O processo de "ziguezagueamento" realizado em cada uma das etapas

A terceira atividade consiste em mapear na matriz de projeto os novos requisitos funcionais identificados e os novos parâmetros de projeto criados, identificando-se as dependências entre eles. As entradas desta atividade são os novos requisitos funcionais, os novos parâmetros de projeto e a matriz de projeto da decomposição anterior. Como saída, são obtidas novas matrizes de projeto, consistentes com a matriz de projeto da decomposição

anterior e que representam as possíveis soluções de projeto para este nível. Este mapeamento entre os requisitos funcionais, no domínio funcional, e os parâmetros de projeto no domínio físico para as etapas (níveis de abstração) da abordagem proposta é ilustrado na Figura 18.

Na quarta atividade são aplicados os Axiomas da Independência e da Informação visando a seleção de qual é a melhor entre as soluções criadas para esta decomposição. Após a aplicação dos axiomas, se nenhuma solução for considerada aceitável, novos conjuntos de parâmetros de projeto (DPs) são criados ou até mesmo, um novo conjunto de requisitos funcionais (FRs) pode precisar ser identificado. As entradas desta atividade são as matrizes de projeto que representam as possíveis soluções, o conjunto de requisitos funcionais desta decomposição e os conjuntos de parâmetros de projeto criados nesta decomposição. A saída desta etapa é a melhor solução de projeto adequada, ou, no caso de não haver nenhuma solução aceitável, um aviso para refazer as atividades de criação dos parâmetros de projeto ou mesmo de decomposição dos requisitos funcionais.

O processo realizado pelas atividades em cada etapa, em que um conjunto de requisitos funcionais é identificado a partir da decomposição dos requisitos anteriores, mapeado em parâmetros de projeto, validado pelos Axiomas 1 e 2 e decomposto para criar novos requisitos funcionais representa a aplicação do processo de "*ziguezagueamento*" da Teoria de Projeto Axiomático. Esta relação pode ser visualizada na Figura 18 onde as diferentes etapas podem ser associadas a diferentes níveis de decomposição da hierarquia funcional e aos conceitos de projeto de *software* relacionados a estas etapas. e ao processo de ziguezagueamento.

4.4 Aplicação do Axioma da Independência

A atividade de aplicação dos Axiomas 1 e 2 e os teoremas relacionados é realizada pelo menos uma vez a cada etapa. Esta aplicação visa determinar qual das diversas soluções de projeto criadas deve ser adotada. O Axioma 1 ou Axioma da Independência e teoremas relacionados devem ser aplicados para identificar soluções de projeto não aceitáveis (acopladas). O problema mais visível é o fato da matriz de projeto não ser quadrada, descrito na Seção 3.2.1.1, que contraria o Teorema 4 que diz que a matriz de projeto ideal é uma matriz quadrada (ver Anexo 1) (SUH, 2001). Neste caso pode ser aplicado o Teorema 2 (ver Anexo 1) que indica que deve-se identificar mais parâmetros de projeto (DP), ou pode-se

tentar agrupar parâmetros de projeto (DP) que afetem um determinado requisito funcional (FR) para remover a redundância (SUH, 1990).

Para matrizes quadradas a aplicação principal do Axioma 1 é na identificação de soluções de projeto não adequadas devido a acoplamentos entre requisitos funcionais. Soluções de projeto desacopladas ou semi-acopladas (ver Seção 3.2.1) são consideradas aceitáveis pelo Axioma 1 (SUH, 1990). Para detectar se uma solução de projeto é acoplada, pode ser usada a reangularidade (ver Seção 3.2.2), calculada pela Equação (9) ou pode-se usar a estratégia para eliminar acoplamentos descrita em (LEE, 2006). Se o valor da reangularidade for igual a zero, o projeto é acoplado mas não é possível identificar onde está o acoplamento. A estratégia de Lee (2006) permite identificar, além da existência do acoplamento, onde exatamente está o acoplamento na matriz (LEE, 2006). As soluções de projeto acopladas devem ser descartadas ou ao menos refeitas pois podem causar efeitos indesejáveis no desenvolvimento, principalmente na alteração de parâmetros de projeto (DP).

FRs \ DPs	Classe TelaRegistro	Classe ControleRegistro	Classe Registro
Registrar clientes	X	X	X
Registrar produtos	X	X	X
Registrar pedidos	X	X	X

Figura 19 - Um exemplo de projeto de *software* orientado a objetos acoplado

O uso de técnicas orientadas a objetos como casos de uso, objetos e classes pode não gerar uma boa solução de projeto. No exemplo mostrado na Figura 19, foram identificados os casos de uso “registrar clientes”, “registrar produtos” e “registrar pedidos” para um sistema de vendas hipotético. Para realizar estes casos de uso foram criadas as seguintes classes: uma classe de interface com o usuário chamada “TelaRegistro”, uma classe controle, “ControleRegistro” e uma classe “Registro” que é usada para armazenar na memória e manipular as informações de todos os tipos de registro. Estas três classes são usadas na realização de todos os casos de uso. Como todos os casos de uso dependem destas três classes, a solução de projeto é uma solução acoplada e não satisfaz o Axioma da

Independência. Neste caso, mesmo aplicando técnicas e conceitos de orientação a objetos como casos de uso, classes, objetos e MVC (modelo-visão-controlador) (KRASNER; POPE, 1988), não se consegue garantir que a solução de projeto seja boa.

Um exemplo de solução acoplada que não é tão fácil de ser detectada é a matriz apresentada na Figura 20. Nesta solução, para o projeto de um sistema de bibliotecas foram identificados dois casos de uso acoplados: "FR 3 Emprestar Exemplar" e "FR 4 Confirmar Empréstimo". As colaborações que realizam estes casos de uso compartilham algumas classes, como: "TelaDeEmpréstimo" e "Empréstimo" que geram o acoplamento identificado. Neste caso a classe "TelaDeEmpréstimo" foi criada como papel na colaboração "DP3 Emprestar Exemplar" e a classe "Empréstimo " foi criada como papel na colaboração "DP4 Confirmar Empréstimo".

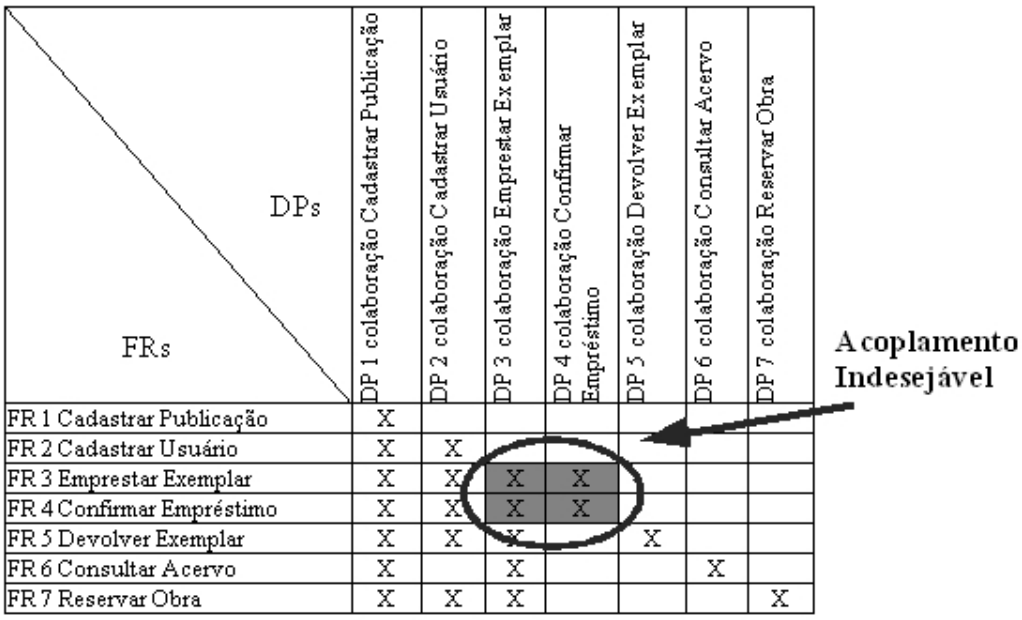


Figura 20 - Matriz de projeto com acoplamento entre requisitos funcionais (FRs)

As melhores soluções de projeto, segundo o Axioma 1, são as desacopladas, entretanto as mais comuns são as semi-acopladas. No caso de apenas existirem soluções semi-acopladas, que são as mais comuns, identifica-se a melhor solução através de uma medida de independência funcional como a reangularidade (ver Seção 3.2.2). Para cada matriz de projeto é calculada a reangularidade. Se a matriz for desacoplada, a reangularidade é igual a 1, segundo o Corolário 8 (ver Anexo 1) (SUH, 1990). Se houver um acoplamento, o valor da reangularidade é igual a 0. No caso de matrizes semi-acopladas, o projeto que tiver maior reangularidade é o que possui maior independência funcional e portanto o melhor projeto.

A reangularidade de um projeto é calculada através da Equação (9). Para efeitos do cálculo da reangularidade, os elementos nulos da matriz terão um valor 0. Os elementos não nulos terão um valor diferente de 0. Este valor representa o grau de dependência entre o requisito funcional e o parâmetro de projeto. Por exemplo, se uma colaboração é totalmente usada para realizar um caso de uso, o valor para o elemento da matriz é 1. No caso de ser usada apenas uma parte da colaboração para realizar o caso de uso, o valor do elemento pode ser menor que um. Na Figura 21 é apresentada a matriz de projeto referente a um sistema de controle de ponto que será usada como exemplo do cálculo da reangularidade. Para efetuar este cálculo, serão considerados os valores 0 para elementos nulos e 1 para elementos não nulos. Aplicando-se a Equação (9) obtém-se o valor *R=0,25*. Este valor indica que a matriz é semi-acoplada e pode ser uma solução aceitável. Este valor também permitiria uma comparação com outras possíveis soluções de projeto.

FRs \ DPs	DP 1 colab. Cadastrar Empregado	DP 2 colab. Registrar Ponto	DP 3 colab. Consultar Relatórios Gerenciais	DP 4 colab. Consultar Relatório de Ponto do Empregado
FR 1 Cadastrar Empregado	X			
FR 2 Registrar Ponto	X	X		
FR 3 Consultar Relatórios Gerenciais	X	X	X	
FR 4 Consultar Relatório de Ponto do Empregado	X	X		X

Figura 21 - Matriz de projeto para um sistema de controle de ponto

A reangularidade é calculada através de um produto de relações entre colunas da matriz tomadas duas a duas (ver Equação (9)). Este fato provoca uma dificuldade que é comparar matrizes com mesmas características de acoplamento mas com tamanhos muito diferentes. O resultado é que matrizes muito maiores possuem valor de reangularidade muito pequenos, o que gera uma distorção nos resultados. Entretanto, o que se deseja é comparar soluções de projeto, de um mesmo nível de decomposição, geradas por conjuntos similares de requisitos funcionais. Estas matrizes, possuem tamanhos muito parecidos e, portanto, a reangularidade pode ser aplicada e fornecer resultados significativos. Exemplos da aplicação da reangularidade no projeto de *software* são apresentados na Seção 4.5.1 e no estudo de caso apresentado no Capítulo 6.

4.5 Decomposição Funcional para Projeto de *Software* Orientado a Objetos

O Processo Unificado é dito ser "guiado por" casos de uso. Isto significa que "o processo de desenvolvimento segue um fluxo que deriva dos casos de uso" (JACOBSON; BOOCH; RUMBAUGH, 1999). Em outras palavras, um ou um grupo pequeno de casos de uso fortemente relacionados pode ser escolhido para ser realizado, de forma que, todo o processo de construção, incluindo projeto, implementação e testes, é realizado para este caso de uso (ou grupo de casos de uso).

Apesar de ser flexível, o Processo Unificado coloca o fluxo de atividades para realização dos casos de uso como um processo *top-down* baseado nos casos de uso. Os casos de uso são identificados e podem ser especificados em detalhes através de cenários, fluxos de eventos, diagramas de estados ou de atividades. O Processo Unificado prevê a especificação dos casos de uso em termos de seus fluxos de eventos. Estes fluxos de eventos podem ser especificados em termos de uma seqüência de interações entre o sistema e os atores. Este detalhamento faz parte do modelo de casos de uso e a partir deste modelo é feita a atividade de análise (JACOBSON; BOOCH; RUMBAUGH, 1999).

Na análise, as classes principais na realização dos casos de uso são identificadas juntamente com suas responsabilidades, gerando o modelo de análise. A partir do modelo de análise, o modelo de classes é refinado e as seqüências de interações entre objetos que realizam os casos de uso podem ser especificados usando-se diagramas de interação da UML (BOOCH; RUMBAUGH; JACOBSON, 2005). Baseado nisto, uma hierarquia para decomposição dos requisitos funcionais para projeto de *software* orientado a objetos pode ser estabelecida.

O ciclo de vida do Processo Unificado é dito ser iterativo e incremental, e sua fase de construção é guiada por casos de uso (JACOBSON; BOOCH; RUMBAUGH, 1999). Isto significa que a fase de construção é dividida em iterações e, a cada iteração, um ou alguns casos de uso escolhidos são realizados, com as atividades de análise, projeto, implementação e teste, incrementando a funcionalidade do sistema. Só então, evolui-se para um outro grupo de casos de uso na próxima interação.

A Teoria de Projeto Axiomático propõe que a decomposição seja feita em amplitude (SUH, 2001). Isto significa que, para a Teoria de Projeto Axiomático, antes de ser efetuada uma decomposição funcional, todos os parâmetros de projeto deste nível já deverão

ter sido identificados e mapeados na matriz de projeto. O processo de decomposição dos requisitos funcionais da abordagem proposta nesta tese segue os princípios básicos do Projeto Axiomático, mas deve ser flexível o suficiente para poder ser adaptado às necessidades do ciclo de vida do Processo Unificado.

Um processo iterativo, incremental e guiado por casos de uso induz uma decomposição em profundidade de um ramo da árvore de hierarquia antes de passar para outros ramos. Esta abordagem em profundidade é possível se o projetista mantiver a cada nível de decomposição, a matriz de projeto consistente com a matriz de projeto do nível anterior (TATE, 1999). Uma decomposição consistente é definida como sendo uma decomposição em que o nível inferior da hierarquia de projeto corresponde ao nível superior (TATE, 1999).

Em outras palavras, se um elemento da matriz de projeto for nulo, a submatriz correspondente em um nível mais baixo da decomposição deverá ser nula. O surgimento de um elemento não nulo da matriz de projeto em uma submatriz correspondente a um elemento nulo em um nível superior de decomposição indica uma inconsistência no projeto. Neste caso o projetista deverá rever o projeto e as decomposições nos níveis anteriores (TATE, 1999).

4.5.1 Decomposição Funcional na Etapa de Modelagem de Casos de Uso

Na primeira etapa da abordagem proposta nesta tese, os requisitos funcionais são representados por casos de uso e os parâmetros de projeto são representados por colaborações e seus papéis. Nesta etapa são realizadas as seguintes atividades: identificar, decompor ou refinar os casos de uso, criar os parâmetros de projeto (colaborações), mapear as relações entre os casos de uso e as colaborações na matriz de projeto e aplicar os axiomas e teoremas para identificar e validar a melhor solução de projeto criada. Os papéis das colaborações criadas nesta etapa auxiliam na identificação das primeiras classes do modelo de análise.

A primeira atividade se inicia com a identificação dos casos de uso, tendo como principais entradas o documento de visão do sistema e o modelo de negócio. O Processo Unificado sugere que os casos de uso podem ser derivados do documento de visão do sistema (JACOBSON; BOOCH; RUMBAUGH, 1999). Este processo de derivação é descrito em detalhes em (LEFFINGWELL; WIDRIG, 2003). A identificação dos casos de uso começa pela identificação dos atores, o que ajuda a definir a fronteira do sistema, como sugerido em (BITNER; SPENCE, 2003). A partir dos atores, os casos de uso são identificados. A Figura 22 apresenta um modelo de casos para um sistema de biblioteca, com os atores e os casos de

uso relacionados com cada ator.

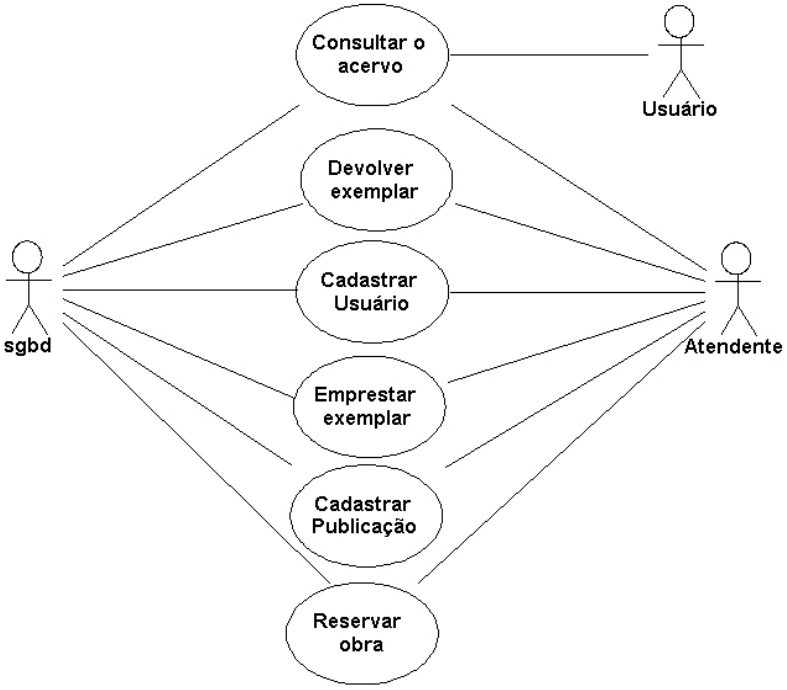


Figura 22 - Casos de uso para o sistema de biblioteca

O Processo Unificado descreve com um grande nível de detalhamento, as atividades, fluxos de processo, as fases e os papéis desempenhados, o que facilita sua aplicação em grandes projetos de sistemas de informação. Em sistemas muito grandes, compostos por vários subsistemas, uma visão global da utilização do sistema pode ser feita por casos de uso que representam como os usuários interagem com as várias aplicações dos subsistemas em conjunto (LEFFINGWELL; WIDRIG, 2003). A partir daí, as funcionalidades dos subsistemas podem ser decompostas e esta decomposição é feita até que se atinja o nível de casos de uso de cada um dos subsistemas (LEFFINGWELL; WIDRIG, 2003).

Logo após a identificação dos casos de uso, a cada ciclo de decomposição, é realizada a atividade de criar as colaborações e seus papéis que irão realizar estes casos de uso. Os detalhes de uma colaboração podem ser mostrados ou ocultados dependendo do nível de abstração desejado. Na primeira etapa, uma colaboração representará toda uma estrutura de papéis, interagindo entre si, para realizar um caso de uso. Estes papéis passarão a ser considerados como objetos em níveis de abstração mais detalhados. Esta estrutura pode ser decomposta em subcolaborações em outros níveis.

Um exemplo de colaboração é apresentado na Figura 23. Uma colaboração

"Cadastrar Publicação" é criada para realizar o caso de uso "Cadastrar publicação" para o sistema de biblioteca apresentado na Figura 22. A colaboração "Cadastrar Publicação" possui alguns papéis que interagem entre si para satisfazer o caso de uso. Neste nível de abstração, estes papéis não precisam necessariamente representar classes do sistema e podem ser inseridos e removidos até que se atinja um modelo mais estabelecido em outros níveis de decomposição do projeto.

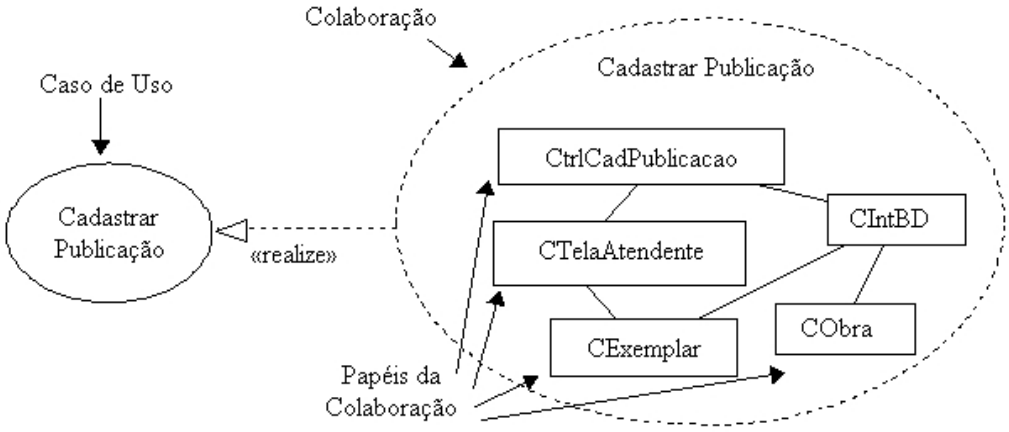


Figura 23 - Colaboração Cadastrar Publicação e seus papéis

Após a criação das colaborações correspondentes aos casos de uso, é realizada a etapa de mapeamento dos casos de uso e das colaborações na matriz de projeto. A matriz de projeto é um recurso importante da Teoria de Projeto Axiomático. A matriz de projeto é apresentada no Capítulo 3 e é descrita com detalhes em (SUH, 1990) e (SUH, 2001). A matriz de projeto facilita principalmente a aplicação do Axioma 1 e teoremas relacionados. Nesta etapa, especificamente, a matriz de projeto conterá em cada uma de suas linhas um caso de uso e nas suas colunas as colaborações correspondentes. As relações entre os casos de uso e as colaborações são representadas em cada elemento da matriz.

Por exemplo, a Figura 24 apresenta a matriz de projeto referente ao modelo de casos de uso apresentado na Figura 22. O "X" no elemento A_{33} indica que a colaboração "DP 3 Emprestar Exemplar" está relacionada com o caso de uso "FR 3 Emprestar Exemplar". Isto significa que esta colaboração é usada para satisfazer (realizar) o caso de uso. Relações parciais podem também ser representadas na matriz de projeto. O "X" no elemento A_{32} representa que um papel na colaboração "DP 2 Cadastrar Usuário", no caso o papel "CUsuário" é usado na realização do caso de uso "FR 3 Emprestar Exemplar". Mapear estas relações parciais é importante pois nos níveis de abstração mais detalhados elas indicarão que

as classes que derivam desses papéis são usados para realizar mais de um caso de uso.

FRs \ DPs	DP 1 colaboração Cadastrar Publicação	DP 2 colaboração Cadastrar Usuário	DP 3 colaboração Emprestar Exemplar	DP 4 colaboração Devolver Exemplar	DP 5 colaboração Consultar Acervo	DP 6 colaboração Reservar Obra
FR 1 Cadastrar Publicação	X					
FR 2 Cadastrar Usuário	X	X				
FR 3 Emprestar Exemplar	X	X	X			
FR 4 Devolver Exemplar	X	X	X	X		
FR 5 Consultar Acervo	X		X		X	
FR 6 Reservar Obra	X	X	X			X

Figura 24 - Matriz de projeto para o sistema de bibliotecas

A atividade a ser realizada a seguir nesta etapa é a aplicação dos Axiomas 1 e 2 e teoremas relacionados. Isto pode ser feito a cada ciclo de decomposição desta etapa ou pelo menos ao final dela. Das atividades anteriores, podem ser identificados alguns conjuntos diferentes de colaborações e portanto surgir diversas soluções de projeto possíveis. A aplicação dos Axiomas 1 e 2 e teoremas relacionados pode ajudar a determinar qual dessas soluções deve ser adotada. O Axioma 1 e teoremas relacionados devem ser aplicados para identificar soluções de projeto não aceitáveis (acopladas), identificando-se a existência de acoplamentos devido ao fato da matriz de projeto não ser quadrada ou de existirem casos de uso acoplados.

As melhores soluções de projeto, segundo o Axioma 1, são as desacopladas, entretanto, as mais comuns são as semi-acopladas. No caso de existirem apenas soluções semi-acopladas (as mais comuns), pode-se identificar a melhor solução através da reangularidade (ver Seção 3.2.2). Como exemplo, a Figura 22 apresenta uma solução para o sistema de biblioteca, com a correspondente matriz de projeto apresentada na Figura 24. A Figura 25 apresenta uma solução alternativa para o mesmo problema com a respectiva matriz de projeto apresentada na Figura 26. Ao calcular a reangularidade, usando a Equação (9), para as duas soluções obtém-se o valor 0.0817 para a primeira solução e 0.0257 para a segunda solução. Quanto maior o valor da reangularidade, maior a independência funcional. Então a primeira solução (ver Figura 22) é melhor que a segunda solução (ver Figura 25).

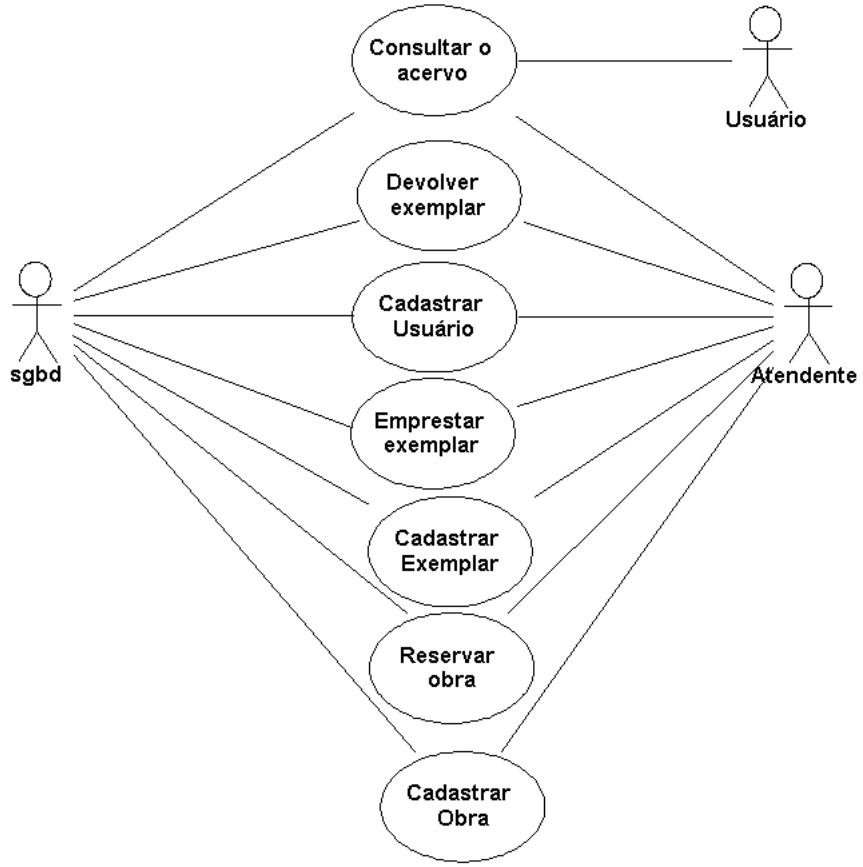


Figura 25 - Solução alternativa para a identificação dos casos de uso

FRs \ DPs	DP 1 colaboração Cadastrar Obra	DP 2 colaboração Cadastrar Exemplar	DP 3 Colaboração Cadastrar Usuário	DP 4 colaboração Emprestar Exemplar	DP 5 colaboração Devolver Exemplar	DP 6 colaboração Consultar Acervo	DP 7 colaboração Reservar Obra
FR 1 Cadastrar Obra	X						
FR 2 Cadastrar exemplar	X	X					
FR 3 Cadastrar Usuário	X	X	X				
FR 4 Emprestar Exemplar	X	X	X	X			
FR 5 Devolver Exemplar	X	X	X	X	X		
FR 6 Consultar Acervo	X	X		X		X	
FR 7 Reservar Obra	X		X	X			X

Figura 26 - Matriz de projeto referente a solução alternativa

4.5.2 Decomposição Funcional na Etapa de Modelagem de Subcasos de Uso Independentes de Características Técnicas

A segunda etapa da abordagem proposta nesta tese prevê a identificação e decomposição dos subcasos de uso independentes de características técnicas a partir dos casos de uso. Neste nível de abstração, os requisitos funcionais são representados pelos subcasos de uso independentes de características técnicas e os parâmetros de projeto são representados pelas colaborações e seus papéis. Nesta etapa são realizadas as seguintes atividades: decomposição dos subcasos de uso independentes de características técnicas, criação das colaborações que realizarão os subcasos de uso independentes, mapeamento dos subcasos de uso independentes e das colaborações na matriz de projeto e aplicação dos Axiomas 1 e 2 e dos teoremas relacionados.

A partir de um determinado nível de abstração, não é mais interessante decompor os casos de uso, pois o conceito de caso de uso se perde. Neste momento é mais interessante o refinamento de cada caso de uso em termos dos seus subcasos de uso. Os subcasos representam parte da funcionalidade de um caso de uso. Estes subcasos podem ser independentes ou dependentes de características técnicas. As características técnicas são características da solução relacionadas com as tecnologias adotadas, com restrições, requisitos não funcionais ou com requisitos de interface com outros sistemas.

Por exemplo, o subcaso de uso independente chamado “FR 3.1 Emprestar exemplar” do sistema de biblioteca é definido sem levar em consideração as características técnicas da solução, como por exemplo o tipo de banco de dados usado. Agora, um subcaso chamado “FR 3.2.1 Iniciar leitor de código de barras de mesa” é dependente de características técnicas, pois alguns tipos de leitores de códigos de barras, como por exemplo os manuais, necessitam de acionamento manual pelo operador.

Durante o projeto do sistema, nas etapas iniciais os requisitos funcionais identificados possuem um nível de abstração mais alto, ocultando detalhes e, portanto, não levam em consideração as características técnicas. Os casos de uso e os subcasos de uso independentes de características técnicas representam os requisitos funcionais nestas fases iniciais. Já nas etapas posteriores, os requisitos funcionais requerem maior detalhamento e, portanto, devem levar em consideração essas características técnicas.

Nesta etapa, a decomposição dos subcasos de uso independentes pode seguir uma seqüência de passos similar a que é comumente usada por vários autores (JACOBSON; BOOCH; RUMBAUGH, 1999); (LEFFINGWELL; WIDRIG, 2003); (JACOBSON; NG,

2004) e (BITNER; SPENCE, 2003). Após a identificação do conjunto adequado de casos de uso (utilizações completas do sistema), na primeira etapa, este conjunto de casos de uso pode ser refinado ou decomposto, criando subcasos de uso independentes para representar um maior detalhamento.

Primeiro tenta-se identificar subcasos de uso independentes que representam os casos de uso incluídos, de extensão, de utilidade ou de infra-estrutura (JACOBSON; NG, 2004). Após esta identificação, os subcasos são decompostos (especificados) em subcasos que usando as representações de fluxos de eventos, atividades, mudanças de estados, cenários, pré-condições e pós-condições ou outra forma de especificação desejada. Não está previsto um número fixo, ou mesmo um número mínimo, de decomposições para a segunda etapa. É a necessidade de representação do projetista que vai ditar este número.

Estes subcasos de uso derivados do refinamento de um caso de uso neste nível de decomposição não representam uma utilização completa do sistema Os casos de uso podem ser decompostos ou refinados em subcasos de uso, normalmente pelos mecanismos de inclusão ou extensão, como exemplificado em (LEFFINGWELL; WIDRIG, 2003). Em outra abordagem, o refinamento de casos de uso pode se dar, também, através da identificação de funcionalidades importantes do sistema que não estão diretamente relacionadas a um ator, representadas pelos (sub) casos de uso de utilidade (JACOBSON; NG, 2004).

Alguns autores como (BITNER; SPENCE, 2003) e (LEFFINGWELL; WIDRIG, 2003), sugerem que os requisitos não funcionais devem ser representados como parte do documento de especificação do caso de uso relacionado. Já mais recentemente, um tipo especial de caso de uso foi definido para representar requisitos não funcionais. São os chamados subcasos de uso de infra-estrutura (JACOBSON; NG, 2004).

De acordo com (LEFFINGWELL; WIDRIG, 2003); (JACOBSON; NG, 2004); (BOOCH; RUMBAUGH; JACOBSON, 2005) e (FOWLER, 2003), quando os casos de uso são decompostos em subcasos de uso de inclusão, de extensão, de utilidade ou de infra-estrutura, o diagrama de casos de uso resultante mostra apenas as relações entre os casos de uso atuais e os novos subcasos identificados a partir da decomposição. É necessário representar na matriz de projeto que os novos subcasos não representam a totalidade da funcionalidade original.

Por isso existe uma parte da funcionalidade referente ao caso de uso original que deve estar representada na decomposição. Isto pode ser feito adicionando-se um ou mais novos subcasos de uso independentes que irão representar esta funcionalidade e que terá um nome semelhante ao caso de uso original. A Figura 27 apresenta a identificação de subcasos

de uso incluídos e de extensão para o refinamento dos casos de uso apresentados na Figura 22 para o sistema de biblioteca.

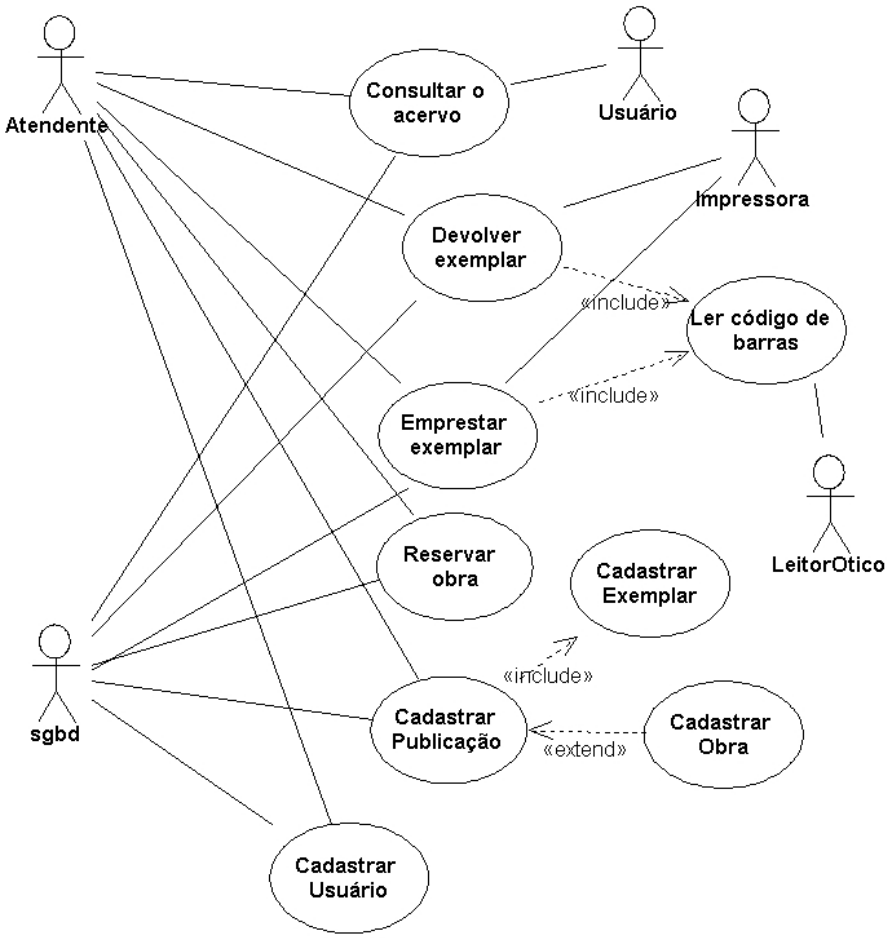


Figura 27 - Subcasos de uso incluídos e de extensão para o sistema de biblioteca

Por exemplo, para a decomposição do caso de uso "FR 1 Cadastrar Publicação" são encontrados os subcasos de uso "FR 1.2 Cadastrar Obra" e "FR 1.3 Cadastrar Exemplar". O conjunto das funcionalidades descritas por estes subcasos de uso não representam totalmente a funcionalidade descrita pelo caso de uso "FR 1 Cadastrar Publicação". É necessário adicionar um outro subcaso de uso, "FR 1.1 Controlar Cadastro de Publicação" para complementar a funcionalidade do caso de uso original. Este subcaso adicionado está representado na matriz de projeto (ver Figura 28). Normalmente este subcaso de uso adicionado representa as chamadas para execução dos outros subcasos de uso.

FRs \ DPs		DP 1 colaboração Cadastrar Publicação			DP 2 colaboração Cadastrar Usuário	DP 3 colaboração Emprestar Exemplar		DP 4 colaboração Devolver Exemplar	DP 5 colaboração Consultar Acervo	DP 6 colaboração Reservar Obra
		DP 1.1 colaboração Controlar Cadastro de Publicação	DP 1.2 colaboração Cadastrar Obra	DP 1.3 colaboração Cadastrar Exemplar	DP 2 colaboração Cadastrar Usuário	DP 3.1 colaboração Emprestar Exemplar	DP 3.1 colaboração Ler Código de Barras	DP 4 colaboração Devolver Exemplar	DP 5 colaboração Consultar Acervo	DP 6 colaboração Reservar Obra
FR 1 Cadastrar Publicação	FR 1.1 Controlar Cadastro de Publicação	X								
	FR 1.2 Cadastrar Obra	X	X							
	FR 1.3 Cadastrar Exemplar	X	X	X						
FR 2 Cadastrar Usuário	FR 2 Cadastrar Usuário	X			X					
FR 3 Emprestar Exemplar	FR 3.1 Emprestar Exemplar	X			X	X				
	FR 3.2 Ler Código de Barras						X			
FR 4 Devolver Exemplar	FR 4.1 Devolver Exemplar	X			X	X	X	X		
FR 5 Consultar Acervo	FR 5.1 Consultar Acervo	X				X			X	
FR 6 Reservar Obra	FR 6.1 Reservar Obra	X			X	X				X

Figura 28 - Matriz de projeto da decomposição dos casos de uso do sistema de biblioteca

A complexidade de um caso de uso depende do número de cenários que ele representa (JACOBSON; NG, 2004). Neste caso, torna-se necessária a representação desta complexidade. A especificação do comportamento de casos de uso através dos fluxos de eventos é ilustrada em (BOOCH; RUMBAUGH; JACOBSON, 2005). Jacobson e NG (2004) propuseram uma extensão da UML para representar os fluxos de eventos em fluxos básicos, alternativos e subfluxos usando as etiquetas *{basic}*, *{alt}* e *{sub}* respectivamente, além de propor sua representação nos diagramas através de um compartimento de um caso de uso (JACOBSON; NG, 2004). Neste trabalho, os fluxos de eventos são representados através de subcasos de uso. De forma similar, pode-se representar os subcasos em um compartimento de um caso de uso (ver Figura 29).

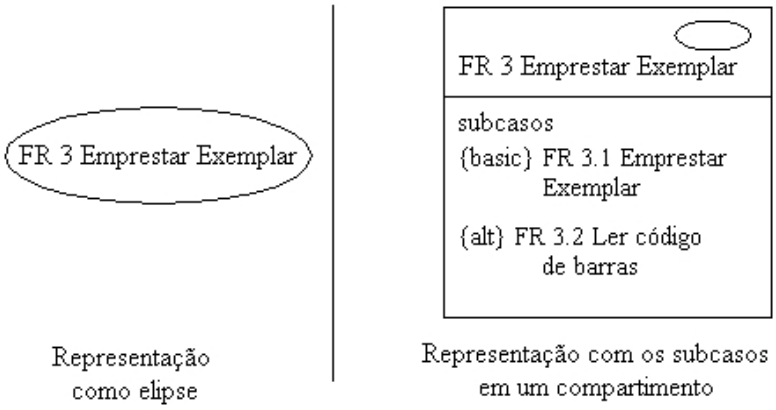


Figura 29 - Representação de subcasos de uso e casos de uso

Como exemplo de decomposição de subcasos de uso independentes tem-se o caso de uso "FR 3 Emprestar Exemplar" que é decomposto nos subcasos de uso independentes "FR 3.1 Emprestar Exemplar" e "FR 3.2 Ler Código de Barras", apresentados na Figura 28. O subcaso de uso independente "FR 3.1 Emprestar Exemplar", por sua vez, pode ser decomposto em outros subcasos de uso independentes como: "FR 3.1.1 Escolher Opção", "FR 3.1.2 Receber Id do Exemplar", "FR 3.1.3 Receber id do Usuário", "FR 3.1.4 Confirmar Empréstimo" e "FR 3.1.5 Gravar Empréstimo". Um exemplo da representação de subcasos de uso usando um diagrama de atividades é apresentado na Figura 30, onde é apresentada a decomposição do subcaso de uso independente "FR 3.1 Emprestar Exemplar".

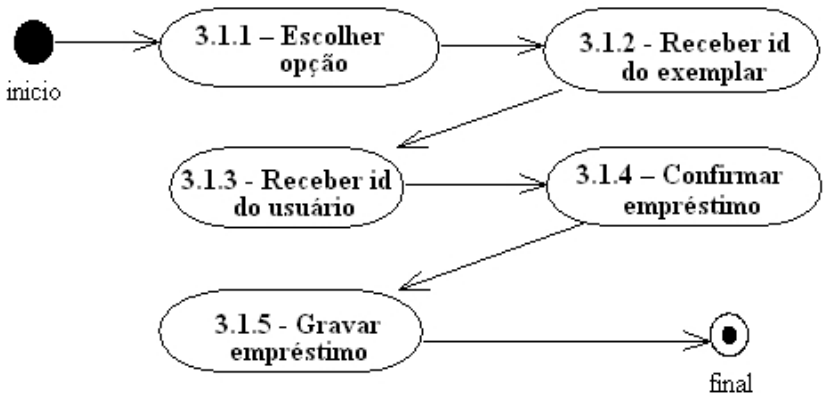


Figura 30 - Atividades da decomposição do subcaso "FR 3.1 Emprestar exemplar"

Para cada subcaso de uso independente de características técnicas identificado, são criadas colaborações que irão realizá-lo. Nesta etapa, torna-se fundamental que os papéis das

colaborações sejam identificados e representados com maior detalhamento, pois nesta etapa é desejável que muitas das possíveis classes de análise do modelo de análise para o sistema já sejam identificadas.

Após a identificação dos subcasos de uso independentes de características técnicas (FRs) e da criação das colaborações correspondentes (DPs), estes são colocados na matriz de projeto e as relações entre subcasos e colaborações são representadas nos elementos da matriz. Um elemento não nulo da matriz de projeto, neste nível, pode representar que uma colaboração é usada totalmente para realizar um subcaso de uso independente ou que apenas um dos papéis de uma determinada colaboração é usado nesta realização.

Neste nível de abstração, a aplicação do Axioma 1 vai ajudar, também, na identificação de soluções de projeto acopladas devido ao fato da matriz de projeto possuir mais linhas que colunas, como definido no Teorema 1 (ver Anexo 1). É possível a identificação de subcasos de uso independentes que são realizados pela mesma colaboração. Neste caso, os dois subcasos de uso independentes devem ser agrupados para a matriz voltar a ser quadrada e satisfazer o Teorema 4 (ver Anexo 1).

4.5.3 Decomposição Funcional na Etapa de Modelagem de Subcasos de Uso Dependentes de Características Técnicas

A terceira etapa da abordagem proposta nesta tese prevê a identificação e decomposição dos subcasos de uso dependentes de características técnicas. Neste nível de abstração, os requisitos funcionais (FRs) são representados pelos subcasos de uso dependentes de características técnicas e os parâmetros de projeto (DPs) são representados pelas colaborações e seus papéis. Nesta etapa, são realizadas as seguintes atividades: decomposição dos subcasos de uso dependentes de características técnicas, criação das colaborações que irão realizar os subcasos de uso dependentes, mapeamento dos subcasos dependentes e das colaborações na matriz de projeto e aplicação dos Axiomas 1 e 2 e dos teoremas relacionados.

A principal diferença entre os subcasos de uso independentes de características técnicas e os subcasos dependentes é o fato de considerar ou não características técnicas da solução. Além disso, os subcasos dependentes normalmente possuem maior detalhamento que os subcasos independentes. Do ponto de vista de notação e representação, nesta etapa podem ser usadas as mesmas formas de representação da etapa anterior.

Cada subcaso de uso dependente ou independente de característica técnicas é composto de seqüências de interações entre o sistema e os atores, representando um conjunto

específico de cenários na execução de um caso de uso. Muitas vezes um subcaso de uso é composto de subcasos de uso, que por sua vez, também podem ser compostos por subcasos de uso dependentes ou independentes de características técnicas. Durante a realização deste processo freqüentemente chega-se em um nível de abstração em que são obtidos as chamadas Operações de Sistema (*System Operations*) (LARMAN, 1997).

Uma operação do sistema é uma operação requerida do sistema para tratar um único evento de interface provocado pelo usuário (LARMAN, 1997). Em outra palavras, é a representação de um único passo de interação entre o usuário e o sistema. Este processo de decomposição continua e só termina quando for necessário representar a funcionalidade requerida de uma classe ou objeto (serviço técnico). Do ponto de vista prático, nem sempre são realizadas decomposições nesta etapa devido ao fato de que uma boa parte dos sistemas de informação usam soluções tecnológicas já estabelecidas. Neste caso, o projetista considera que não existem características técnicas que exijam maiores considerações e muitas vezes esta etapa é suprimida.

Após a decomposição dos subcasos de uso dependentes de características técnicas, para cada subcaso, são criadas colaborações que irão realizá-lo. Nesta etapa, mais que na etapa anterior, é fundamental que os papéis das colaborações sejam identificados e representados com o maior detalhamento possível, pois nesta etapa é desejável que quase todas as classes de análise já estejam identificadas. Estas classes serão usadas na quarta etapa da abordagem proposta neste trabalho.

Após a identificação dos subcasos de uso dependentes de características técnicas (FRs) e da criação das colaborações correspondentes (DPs), eles são colocados na matriz de projeto e suas dependências são representadas nos elementos da matriz. Um elemento não nulo da matriz de projeto, neste nível, pode representar que uma colaboração é usada totalmente para realizar um subcaso ou que apenas um dos papéis de uma determinada colaboração é usado nesta realização.

Neste nível de abstração, a aplicação do Axioma 1 vai ajudar principalmente na identificação de soluções de projeto acopladas devido ao fato da matriz de projeto possuir mais linhas que colunas, como definido no Teorema 1 (ver Anexo 1). Neste nível é comum a identificação de subcasos de uso dependentes que são realizados pela mesma colaboração.

4.5.4 Decomposição Funcional na Etapa de Modelagem de Serviços Técnicos

A quarta etapa da abordagem proposta nesta tese se inicia logo após a identificação de todos os subcasos de uso dependentes de características técnicas (FR) e respectivas colaborações (DP), necessários para satisfazer as necessidades do projetista para a terceira etapa. Na quarta etapa são encontrados e decompostos os serviços técnicos que representam os requisitos funcionais (FRs) e criadas classes e objetos que representam os parâmetros de projeto (DPs) para satisfazê-los. Nesta etapa são realizadas as seguintes atividades: decompor os serviços técnicos, criar classes ou objetos para satisfazer estes serviços técnicos, mapear os serviços técnicos e classes ou objetos na matriz de projeto e aplicar os Axiomas 1 e 2 para identificar a melhor solução de projeto criada.

Um requisito funcional pode ser decomposto em sub-requisitos de vários tipos. Segundo Hintersteiner (1999), eles podem ser classificados em funções de processo, de comando e controle, e de suporte e integração (HINTERSTEINER, 1999). Esta classificação é similar aos conceitos da arquitetura Modelo-Visão-Controlador (MVC) (KRASNER; POPE, 1988) que pode ser usado como base para a decomposição neste nível.

No caso de sistemas de *software*, são encontrados três tipos de funcionalidades. Em uma atividade são comumente encontradas funcionalidades para trocar informações entre o sistema e atores (humanos ou não), funcionalidades para manipular os dados do sistema e funcionalidades para controlar o processo realizado na atividade. Devido a este fato, pode-se representar cada uma destas funcionalidades como sendo um serviço técnico que, a rigor, é um requisito funcional de um nível hierárquico mais baixo. Neste trabalho, os serviços técnicos serão classificados nas seguintes categorias:

- serviços técnicos de fronteira
- serviços técnicos de entidade
- serviços técnicos de controle

Serviços técnicos de fronteira são serviços relacionados com a interface do sistema com os atores. Serviços técnicos de entidade são serviços para manipular as informações do sistema. Serviços técnicos de controle são serviços para controlar o processo realizado na atividade.

Na abordagem de projeto apresentada nesta tese, a decomposição funcional de um subcaso de uso dependente de características técnicas é feita identificando-se os serviços

técnicos de fronteira, controle e entidade que compõem a atividade. Para cada serviço técnico identificado, serão escolhidos os parâmetros de projeto que serão os objetos que participam da colaboração identificada no nível anterior. Com o mapeamento FR x DP, cada serviço técnico será mapeado em um objeto. Um objeto que realiza um serviço de fronteira recebe o estereótipo de <<*fronteira*>>, os que realizam serviços de entidade recebem o estereótipo <<*entidade*>> e os que executam serviços de controle recebem o estereótipo <<*controle*>>.

FRs \ DPs	DP 1.1.5.1 interface banco de dados	DP 3.1.1.1 tela de escolha de opção	DP 3.1.1.2 variável opção	DP 3.1.2.1 tela receber id exemplar	DP 3.1.2.2 variável da classe exemplar	DP 3.1.3.1 tela receber id usuário	DP 3.1.3.3 variável da classe usuário	DP 3.1.4.1 consulta à tabela usuário	DP 3.1.4.2 variável da classe usuário	DP 3.1.4.3 consulta à tabela empréstimos	DP 3.1.4.4 variável número de atrasos	DP 3.1.4.5 consulta à tabela exemplar	DP 3.1.4.6 variável da classe exemplar	DP 3.1.4.7 consulta à tabela obra	DP 3.1.4.8 variável da classe obra	DP 3.1.4.9 tela confirmação empréstimo	DP 3.1.4.10 variável confirmação	DP 3.1.5.1 variável da classe empréstimo	DP 3.1.5.2 variável de consulta à tabela empréstimo	DP 3.1.5.3 variável interface com a impressora
FR 1.1.5.1 conectar com a base de dados	X																			
FR 3.1.1.1 mostrar tela escolha		X																		
FR 3.1.1.2 receber opção			X																	
FR 3.1.2.1 mostrar tela receber id exemplar				X																
FR 3.1.2.2 receber id exemplar					X															
FR 3.1.3.1 mostrar tela receber id usuário						X														
FR 3.1.3.2 receber id usuário							X													
FR 3.1.4.1 buscar dados Usuário	X							X												
FR 3.1.4.2 receber busca dados Usuário	X						X		X											
FR 3.1.4.3 verificar na base empréstimos atrasados	X									X										
FR 3.1.4.4 receber busca empréstimos atrasados	X										X									
FR 3.1.4.5 buscar dados exemplar	X											X								
FR 3.1.4.6 receber busca dados exemplar	X				R								X							
FR 3.1.4.7 buscar dados obra	X													X						
FR 3.1.4.8 receber busca dados obra	X														X					
FR 3.1.4.9 mostrar tela confirmação					R		R		X		X		X		X	X				
FR 3.1.4.10 receber confirmação																	X			
FR 3.1.5.1 criar novo empréstimo																		X		
FR 3.1.5.2 gravar dados de empréstimo na base	X																	X	X	
FR 3.1.5.3 imprimir comprovante empréstimo																				X

Figura 31 - Matriz de projeto parcial de 4o. Nível

Após a identificação dos serviços técnicos e das classes ou objetos que irão realizá-los, as dependências entre eles é mapeada na matriz de projeto. A matriz de projeto resultante terá representado nas linhas os serviços técnicos e nas colunas as classes ou objetos que irão realizá-los. Como exemplo, na matriz de projeto apresentada na Figura 31, o serviço técnico "receber id usuário" (FR 3.1.3.2) é mapeado no parâmetro de projeto "variável usuário" (DP

3.1.3.2). Nesta forma de mapeamento, quando um objeto é usado em mais de um requisito funcional (FR), esta relação é marcada com um "X". Quando não é o mesmo objeto, mas uma outra instância da mesma classe aparece em outro requisito funcional (FR), esta relação pode ser marcada com um "R". Então, das relações "X" e "R" pode-se concluir em quais requisitos funcionais (FRs) aparecem objetos de uma classe. Com isso, pode-se identificar as principais utilizações de cada objeto e seus métodos principais. Pode-se também representar apenas as classes que são usadas para realizar o serviço técnico. Neste caso, os serviços técnicos que são realizados por classes que já foram colocadas na matriz devem ser agrupados para que a matriz continue quadrada.

Na célula da matriz de projeto que representa o mapeamento FR x DP é representado não apenas o método chamado, mas uma composição das visões interativa, estrutural e dinâmica do projeto. Estas visões estão de acordo com a divisão da modelagem em estrutural e comportamental, proposta em (BOOCH; RUMBAUGH; JACOBSON, 2005). Cada elemento não nulo da matriz de projeto representa a interação em que o envio da mensagem é realizado, para qual objeto é feito, a criação de um novo método ou o uso de um já existente e a mudança de estados do objeto que este envio ocasiona.

A representação destas visões na forma de uma célula resultante do mapeamento FR x DP permite que seja feito o rastreamento do local de utilização de cada método e, seja visualizado qual o efeito de cada chamada deste método. Além disso, permite que todos os métodos sejam rastreados segundo os requisitos funcionais que eles satisfazem. Este tipo de rastreamento se torna importante em projetos grandes e de risco.

Após o mapeamento do serviço técnico e da classe ou objeto correspondente na matriz de projeto, as diferentes visões para o elemento não nulo da matriz serão representados nos respectivos diagramas. O envio da mensagem para o objeto correspondente será representado em um diagrama de seqüência. A definição do método, correspondente ao envio dessa mensagem, será representada no diagrama de classes. O envio da mensagem para o objeto ocasionará a mudança de estado do objeto. Esta mudança de estado será representada no diagrama de estados para o objeto.

Para ilustrar esta representação será apresentada como exemplo a decomposição do subcaso de uso "FR 3.1.5 Gravar empréstimo" referente a um sistema de gerenciamento de bibliotecas. Este subcaso de uso é resultante da decomposição do subcaso de uso "FR 3.1 Emprestar exemplar" referente ao caso de uso "FR 3 Emprestar exemplar".

FRs \ DPs	DP 3.1.5.1 - variável empréstimo	DP 3.1.5.2 - variável de consulta à tabela empréstimo	DP 3.1.5.3 – variável interface com a impressora
FR 3.1.5.1 – criar novo empréstimo	X		
FR 3.1.5.2 – gravar dados do empréstimo na base	X	X	
FR 3.1.5.3 – imprimir comprovante empréstimo			X

Figura 32 - Submatriz de projeto referente à decomposição de confirmar empréstimo

Como exemplo de decomposição dos serviços técnicos é apresentada a decomposição relativa ao caso de uso “Emprestar Exemplar”. O caso de uso “FR 3 Emprestar Exemplar” tem como um dos subcasos de uso derivados “FR 3.1 Emprestar Exemplar”, apresentado na Figura 28. Este subcaso de uso tem como um dos seus componentes o subcaso de uso “FR 3.1.5 Gravar Empréstimo”, que por sua vez é decomposto nos serviços técnicos “criar novo empréstimo” (FR 3.1.5.1), “gravar dados do empréstimo na base” (FR 3.1.5.2) e “imprimir comprovante empréstimo” (FR 3.1.5.3). Estes serviços técnicos serão mapeados nos seus respectivos parâmetros de projeto que são: variável “empréstimo” (DP 3.1.5.1), variável “consulta à tabela empréstimo” (DP 3.1.5.2) e variável “interface com a impressora” (DP 3.1.5.3). Este mapeamento está representado em uma matriz de projeto parcial, ilustrada na Figura 32.

O serviço técnico “gravar dados do empréstimo na base” (FR 3.1.5.2) utiliza o parâmetro de projeto variável “consulta à tabela empréstimo” (DP 3.1.5.2). Para representar os envios de mensagens referentes à colaboração confirmar empréstimo (DP 3.1.5) é feito um diagrama de seqüência para esta colaboração. Neste diagrama é representado o envio da mensagem “gravarEmprestimo(emprestimo)” referente ao mapeamento do serviço técnico “gravar dados do empréstimo na base”, ilustrado no diagrama de seqüência da Figura 33.

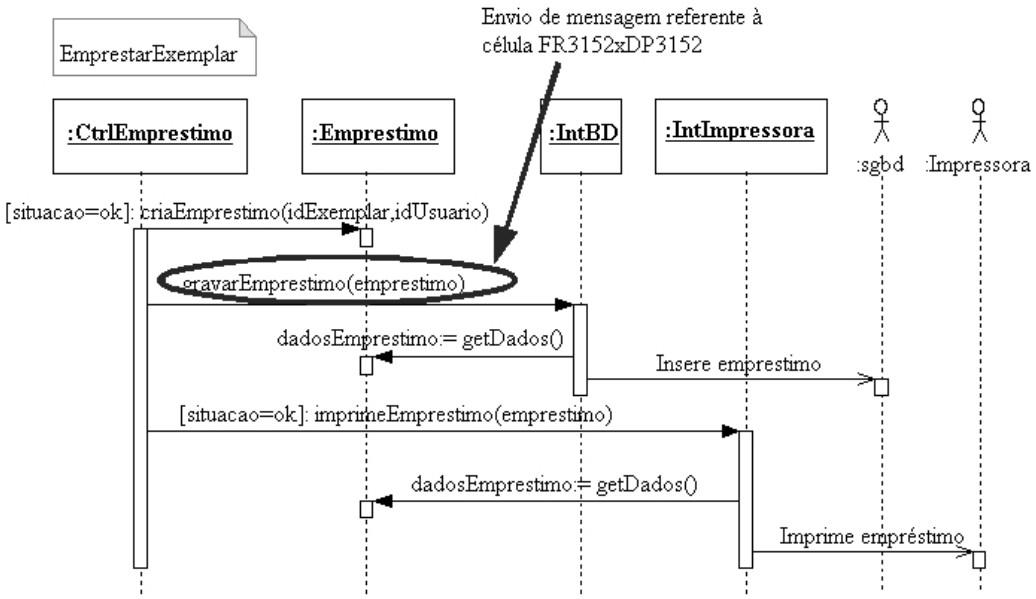


Figura 33 - Envio de mensagem referente A FR3152 x DP3152

As relações entre os serviços técnicos e os objetos na matriz de projeto representam os objetos que serão usados na realização dos requisitos funcionais. Com isso pode-se identificar, por exemplo, os objetos que tomarão parte em um diagrama de seqüência. Se forem considerados apenas os serviços técnicos referentes à decomposição do subcaso de uso "FR 3.1.5 Gravar Empréstimo" tem-se que os parâmetros de projeto utilizados são: "variável "empréstimo" (DP 3.1.5.1), variável "consulta à tabela empréstimo" (DP 3.1.5.2) e variável "interface com a impressora" (DP 3.1.5.3) Estes objetos são usados no diagrama de seqüência que representa as trocas de mensagens entre os objetos referentes a este subcaso de uso, como representado da Figura 33.

O mapeamento do serviço técnico "gravar dados do empréstimo na base", além do envio da mensagem "gravarEmprestimo(empréstimo)" deve produzir a descrição de um método novo no diagrama de classes. Esta descrição está representada no diagrama da Figura 34. Mesmo no caso da utilização de um método que já existe, o elemento não nulo da matriz fica vinculado ao método representando que este método está sendo usado para satisfazer o requisito funcional "gravar dados do empréstimo na base" (FR 3.1.5.2).

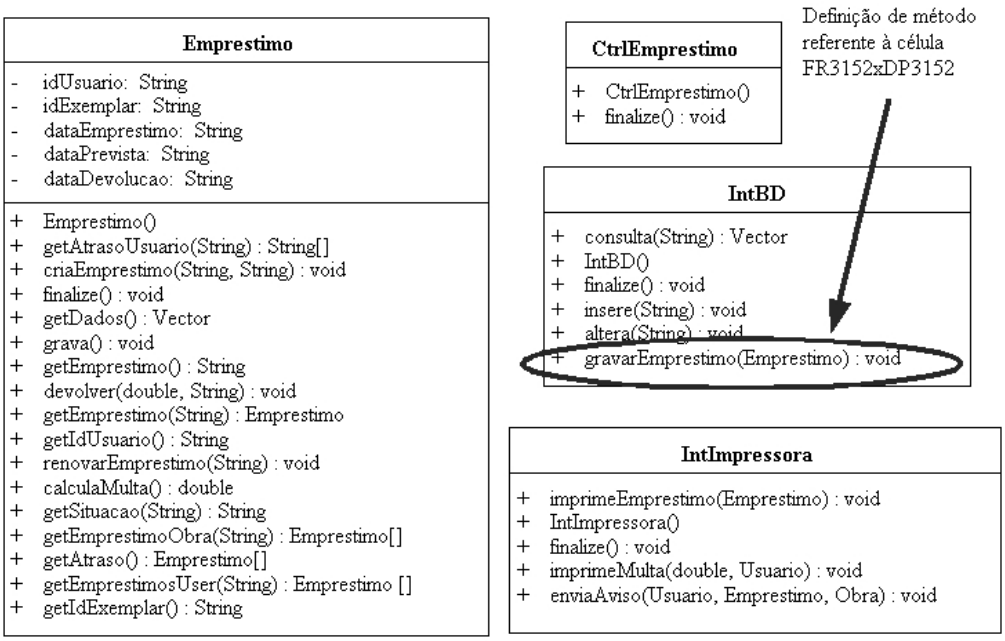


Figura 34 - Definição de Método referente A FR3152 x DP3152

Também é representada a mudança de estado que o envio da mensagem "gravarEmprestimo (empréstimo)" causa no objeto "IntBD". Esta mudança de estados é representada no diagrama de estados da classe "IntBD" apresentado na Figura 35.

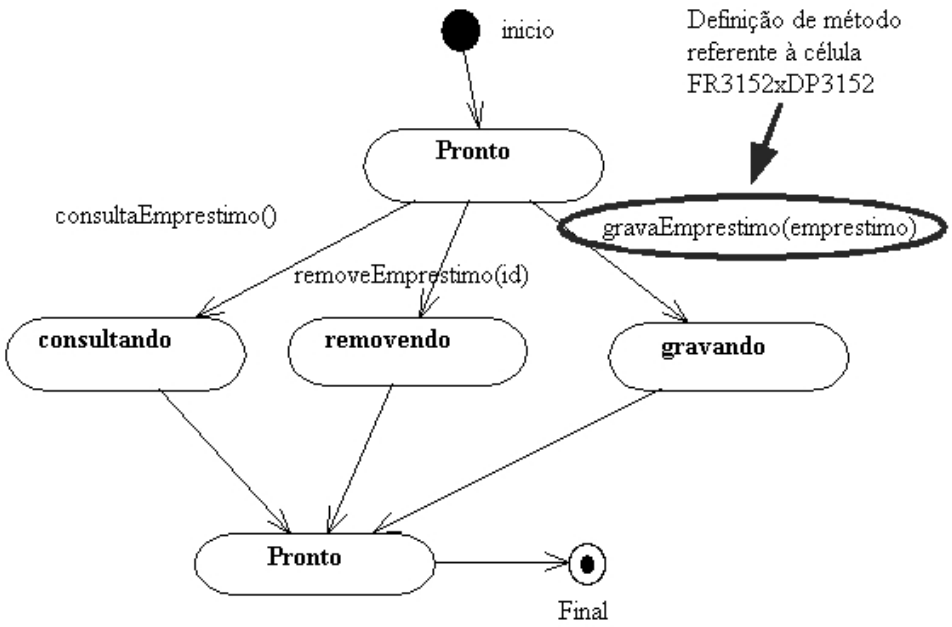


Figura 35 - Mudança de estado referente à célula FR3152 x DP3152

No nível de abstração relativo à quarta etapa, o detalhamento do projeto é bem

completo. Neste nível de decomposição, a aplicação do Axioma 1 tem como objetivos manter a matriz de projeto quadrada e analisar a reangularidade das matrizes de projeto para identificar quais classes ou objetos realizam melhor os serviços técnicos identificados. Neste nível se torna muito importante a aplicação do Axioma 2 ou Axioma da Informação. Este axioma tem como critério de avaliação o conteúdo de informação que tem como uma de suas possíveis definições a dificuldade para que um parâmetro de projeto (classe ou objeto) satisfaça um requisito funcional (serviço técnico) (SUH, 1990). Para o cálculo do conteúdo de informação para as classes identificadas nesta etapa, foi criado um arcabouço baseado em métricas de complexidade de projeto de *software* orientado a objetos.

4.6 Conclusões

Este Capítulo apresentou a abordagem de projeto de *software* orientado a objetos, baseada na Teoria de Projeto Axiomático, que pode ser aplicada em conjunto com um processo de desenvolvimento como o Processo Unificado, que é o objetivo principal desta tese. A aplicação em conjunto, dessas duas metodologias, pode trazer vantagens ao desenvolvimento de *software* orientado a objetos como por exemplo: critérios para a escolha de um conjunto adequado da requisitos funcionais, critérios para a escolha da melhor solução de projeto para satisfazer um conjunto adequado de requisitos funcionais, permitir ao projetista um maior detalhamento dos requisitos funcionais com a decomposição funcional e o *ziguezagueamento,* prover mecanismos de rastreabilidade dos componentes do sistema em relação aos requisitos funcionais de alto e baixo nível de abstração.

Foram definidas as correspondências entre o Projeto Axiomático e o Processo Unificado. Existe uma correspondência importante entre os domínios do Projeto Axiomático e as fases do Processo Unificado e entre os conceitos básicos do Projeto Axiomático e os conceito básicos do Processo Unificado que possibilita a aplicação em conjunto destas duas metodologias. A abordagem proposta nesta tese se insere nas atividades de requisitos, análise e projeto do Processo Unificado sendo aplicada entre as fases de concepção, elaboração e o início da fase de construção.

A abordagem proposta possui 4 etapas de acordo com os níveis de abstração propostos para os requisitos funcionais e a cada etapa são realizadas atividades como decomposição e aplicação dos Axiomas. A aplicação do Axioma da Independência em

projetos de *software* é analisada e discutida. Os processos de decomposição funcional e *ziguezagueamento* do Projeto Axiomático são adaptados para a aplicação em conjunto com o Processo Unificado sendo estabelecido um arcabouço para a decomposição funcional específico para o desenvolvimento orientado a objetos.

5 Axioma da Informação para Sistemas de S*oftware*

O Axioma 2 ou Axioma da Informação é um importante critério para auxiliar na tomada de decisões de projeto e é usado em conjunto com o Axioma da Independência. O Axioma 2 é baseado no conceito de conteúdo de informação. Parâmetros para a aplicação do Axioma da Informação em projetos de *software* são definidos. Este capítulo define um arcabouço para o cálculo do conteúdo de informação para projeto de *software* orientado a objetos. Para isso, o conteúdo de informação é definido para sistemas computacionais. Definem-se, métricas para calcular o conteúdo de informação para cada uma das etapas da abordagem proposta. Por fim, a aplicação do arcabouço proposto é discutida e ilustrada através do cálculo do conteúdo de informação em um exemplo.

5.1 Conteúdo de Informação de Sistemas Computacionais

O processo de projeto tem como resultado um conjunto de informações que serão usadas na manufatura (ou construção) do produto e outras atividades (SUH, 1990). Quanto mais complexa for a informação necessária para a construção de um produto, mais difícil se torna o processo de construção e aumenta a probabilidade dos requisitos funcionais não serem satisfeitos. Em processos de manufatura de produtos essa informação pode ser associada a vários tipos de atributos como: comprimento, dureza, nível de impurezas e custo, entre outras (SUH, 1990). Em se tratando de projetos de *software*, a informação necessária associada à implementação pode ser de vários tipos como: definições das classes, objetos, interações, mudanças de estados, planos de testes, restrições temporais, estruturas de dados, tarefas, servidores, plataformas, entre outros. Genericamente, pode-se definir "conteúdo de informação" como a medida do conhecimento para se satisfazer um requisito funcional (SUH, 1990). Na Seção 3.3, são apresentadas as formas de se medir o conteúdo de informação de

projetos, definidas em (SUH, 1990), tomando como base para o cálculo a probabilidade de se atingir um requisito funcional (ver equações (12) e (13)) ou a razão entre a faixa de valores do sistema e a tolerância (ver Equação (14)).

O conteúdo de informação é definido como sendo a probabilidade do projeto não satisfazer aos requisitos funcionais. Segundo o Axioma 2 (SUH, 1990) e alguns teoremas e corolários relacionados, quanto menor o conteúdo de informação melhor será o projeto (SUH, 1990). Um exemplo do cálculo do conteúdo de informação é apresentado na seção 3.3. O conteúdo de informação normalmente é calculado quando se está fazendo o mapeamento entre o domínio físico e o de processo, mas ele pode ser calculado a qualquer etapa do desenvolvimento. A utilidade do cálculo do conteúdo de informação é permitir que seja possível identificar, entre soluções possíveis para o problema proposto, que tenham independência funcional, qual delas é a melhor solução. Por exemplo, quando se está fazendo uma decomposição em um determinado nível e existem duas decomposições possíveis, pode-se decidir qual delas adotar. Apesar da fórmula para o cálculo do conteúdo de informação ser independente do domínio do problema (SUH, 1990), seus parâmetros devem ser adaptados para o projeto de *software*.

O conteúdo de informação é calculado a cada decomposição dos requisitos funcionais (FRs). Isto se deve ao fato de que é necessário verificar se os sub-requisitos funcionais (sub-FRs) encontrados e os parâmetros de projeto (DPs) correspondentes satisfazem os Axiomas 1 e 2. Além disso, o cálculo do conteúdo de informação mostra a vantagem da escolha por um conjunto de sub-requisitos funcionais (sub-FRs) ao invés de um outro conjunto através do Axioma 2 (SUH, 1990). No caso, se os dois conjuntos de sub-requisitos funcionais (sub-FR) tiverem independência funcional, o melhor conjunto é o que possuir o menor conteúdo de informação.

Medir a quantidade de informação de um sistema de *software* nem sempre é fácil devido à diversidade de tipos e características que ela pode assumir. Na literatura existem poucos trabalhos que discutem como calcular o conteúdo de informação para sistemas de *software*. Em (SUH, 2001), o conteúdo de informação é associado ao número de linhas de código, argumentando que se o número de linhas de código for minimizado, a probabilidade do programa executar as funções desejadas corretamente aumenta. O uso da Equação (14) para o cálculo do conteúdo de informação em projetos de *software* é limitado pelas características próprias do desenvolvimento de *software*. Esta medida de conteúdo de informação leva em consideração faixas de valores que as variáveis de projeto podem atingir ou a probabilidade de que estas faixas de valores estejam dentro de uma tolerância. Em

termos de projeto de *software*, as necessidades são diferentes, a faixa de valores das variáveis de projeto não tem uma importância tão grande como a complexidade destas variáveis.

Os requisitos funcionais de *software* representam um comportamento do sistema do ponto de vista da funcionalidade provida ao usuário (LEFFINGWELL; WIDRIG, 2003). Assim, a satisfação de um requisito funcional é representada por dois valores apenas: o *software* executou a função corretamente ou não. A obtenção de uma definição de faixas de valores e de tolerância só é possível no caso de requisitos não funcionais como disponibilidade ou tempo de resposta.

Para solucionar esse problema pode-se adotar uma forma indireta de calcular o conteúdo de informação. Quanto mais informação for necessária para se construir um produto ou *software*, mais complexa será essa construção e menor será a probabilidade de se atingir os requisitos funcionais. Neste caso, pode-se calcular o conteúdo de informação através da complexidade do *software*. Quanto maior e mais complexo for o *software*, maior a possibilidade de ocorrerem erros e, portanto, dos requisitos funcionais não serem satisfeitos. Portanto, serão estabelecidas medidas de conteúdo de informação baseadas em medidas de complexidade de componentes de sistemas de *software* que possam ser aplicadas nos diversos níveis de decomposição de um projeto.

O Axioma 2 é usado para comparar soluções alternativas para um mesmo projeto. Quando o problema para qual o *software* será criado, é inerentemente complexo, o cálculo do conteúdo de informação pode ajudar a selecionar a solução mais adequada que satisfaça os requisitos funcionais, mesmo que ela seja complexa.

Neste Capítulo são definidos os parâmetros para o cálculo do conteúdo de informação para projetos de *software*. Para esta definição serão usadas métricas de projeto de *software*. Estas métricas refletem a complexidade referente ao projeto e a complexidade referente à implementação do sistema computacional. O foco desta tese é o desenvolvimento de *software* orientado a objetos usando o Processo Unificado, o qual é dito dirigido a casos de uso e centrado na arquitetura (KRUCHTEN, 2003). Portanto, as métricas adotadas devem considerar características envolvendo o modelo de casos de uso e a arquitetura da solução (classes, objetos, interações e relacionamentos) como: número e complexidade dos atores, casos de uso e relacionamentos entre eles; número e complexidade das classes envolvidas e número e complexidade das interações e relacionamentos entre as classes. Dentre as métricas de *software* existentes (ver Seção 2.1.2) as que mais se adequam aos propósitos deste trabalho são as métricas de complexidade orientada a objetos como as propostas por Lorenz e Kidd (LORENZ; KIDD, 1994) e por Chidamber e Kemerer (CHIDAMBER; KEMERER,

1994), além das métricas de complexidade das funcionalidades de *software* como Pontos por Função (VAZQUEZ; SIMÕES; ALBERT, 2003) e os pontos por caso de uso (*use case points*) (ANDA et al., 2001).

O cálculo do conteúdo de informação para sistemas computacionais será feito com base no tamanho e na complexidade do sistema. Os sistemas computacionais, em sua grande maioria, ainda são projetados e construídos manualmente. Não é uma máquina ou sistema computacional que construirá o *software*. Portanto, o fator humano deve ser levado em consideração quando o inverso da probabilidade de sucesso é calculado. Por esta razão, neste trabalho, o valor da tolerância será calculado através de uma estimativa de complexidade baseada em registros da complexidade de sistemas construídos anteriormente pela organização ou registros usados como referência.

A Equação (14) calcula o conteúdo de informação através do logaritmo da razão entre a faixa de valores do sistema e a tolerância para esses valores. Analogamente, dada uma métrica de complexidade de *software*, M_i, pode-se calcular o conteúdo de informação pelo logaritmo da razão entre o valor obtido pela métrica de complexidade e o valor estimado para esta métrica com base no histórico da organização ou valor de referência.

$$I_{M_i} = \log\left(\frac{valor\ M_i\ obtido}{valor\ M_i\ estimado}\right) \tag{16}$$

Com a Equação (16) consegue-se calcular a razão fundamental do conteúdo de informação que é a probabilidade de que o parâmetro de projeto (DP) identificado satisfaça o respectivo requisito funcional (FR). No caso de projetos de *software*, organizações que desenvolvem projetos freqüentemente possuem registros históricos dos valores obtidos para métricas de *software*. Através desse registro histórico é possível fazer uma estimativa para o valor da métrica M_i. Se o valor obtido para a métrica M_i para a realização do requisito funcional, através do parâmetro de projeto, for maior que a estimativa, o conteúdo de informação será positivo e a probabilidade total de sucesso será menor. Se for menor que a estimativa, I será negativo e a probabilidade total de insucesso diminuirá. A Figura 36 mostra a comparação entre o valor do conteúdo de informação de uma métrica M_i, dado pela Equação (16) e a função de distribuição da métrica M_i.

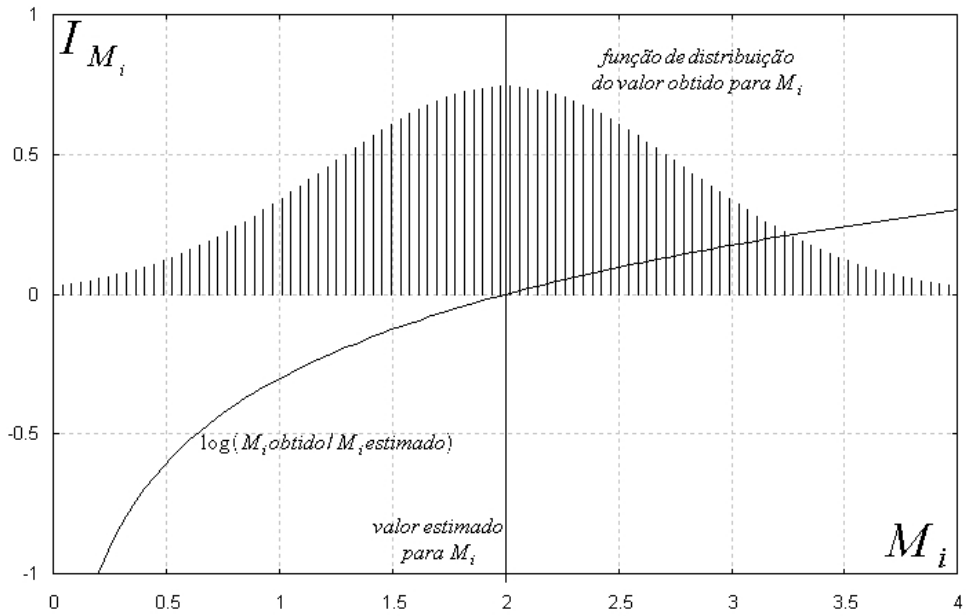


Figura 36 - Conteúdo de informação de M_i x Função de distribuição de M_i

O valor obtido para a métrica M_i do requisito funcional (FR) menor que a estimativa feita com base na média histórica de contagens também significa que este requisito funcional (FR) tem uma complexidade menor que a complexidade média dos projetos produzidos na empresa. Em outras palavras, este requisito funcional, teoricamente, é mais fácil de ser realizado. No caso contrário, ele seria mais difícil de ser construído pela empresa. Além disso, pela Equação (15), o conteúdo total de informação é dado pela soma dos conteúdos parciais. Portanto, um valor negativo para uma medida de conteúdo de informação parcial irá diminuir o valor total do conteúdo de informação para o sistema.

5.2 Conteúdo de Informação nas Etapas da Abordagem

O conteúdo de informação pode ser medido a cada decomposição dos requisitos funcionais (FRs). Segundo a metodologia de desenvolvimento adotada, as etapas de decomposição são:

- 1ª etapa - identificação dos casos de uso
- 2ª etapa - identificação dos subcasos de uso independentes de características técnicas
- 3ª etapa - identificação dos subcasos de uso dependentes de características técnicas
- 4ª etapa - identificação dos serviços técnicos

A cada etapa da abordagem proposta (ver capítulo 4) podem ser realizadas várias decomposições, enquanto houver necessidade de detalhamento para este nível de abstração do projeto. Para cada etapa ou nível de decomposição é proposta a utilização de métricas de complexidade de *software* bem estabelecidas para servir como base para o cálculo do conteúdo de informação do projeto. As métricas de complexidade de *software* adotadas nesta tese são: pontos por caso de uso (ANDA et al., 2001) para a primeira e segunda etapa, pontos por função (VAZQUEZ; SIMÕES; ALBERT, 2003) para a terceira etapa e o conjunto de métricas orientadas a objetos criada por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994) para a quarta etapa. A Tabela 2 apresenta as relações entre as etapas da abordagem proposta, os tipos de requisitos funcionais e as métricas de complexidade usadas em cada uma das etapas.

Tabela 2 - Métricas de complexidade e tipos de requisitos funcionais

Etapa da Abordagem	Tipo de Requisito funcional	Métrica de complexidade
1ª Etapa	Casos de uso	Pontos por caso de uso
2ª Etapa	Subcasos independentes	Pontos por caso de uso
3ª Etapa	Subcasos dependentes	Pontos por função
4ª Etapa	Serviços técnicos	Conjunto de Métricas C K (CHIDAMBER; KEMERER, 1994)

Estas métricas foram adotadas por duas razões principais. A primeira razão é que estas métricas de complexidade de *software* são bem conhecidas e, portanto, têm uma quantidade grande de referências na literatura. A segunda razão é que tanto pontos por caso de uso, pontos por função, quanto o conjunto de métricas CK possuem formas bem estabelecidas para serem computadas, facilitando o cálculo. O conjunto de métricas CK, inclusive, pode ser calculada quase automaticamente, como sugerido em (VERVERS; van DALEN; van KATWIJK, 1996).

5.2.1 Conteúdo de Informação para a 1ª e 2ª Etapas da Abordagem

Na primeira e segunda etapas da abordagem (identificação dos casos de uso e subcasos de uso independentes de características técnicas), o conteúdo de informação é calculado com base nos pontos por caso de uso (ANDA et al., 2001). Os pontos por caso de uso são originados a partir de uma métrica de sistemas computacionais amplamente utilizada que é a de pontos por função (*Function Points*) (VAZQUEZ; SIMÕES; ALBERT, 2003). Tal

como os pontos por função, os permitem que a complexidade do sistema seja mensurada a partir dos seus requisitos funcionais. Além disso, podem ser medidas as complexidades de partes do sistema, e em diferentes níveis de decomposição funcional e de abstração.

Os pontos por caso de uso são calculados da seguinte maneira. A complexidade de cada ator é avaliada e é atribuído um peso. Estes pesos são somados e é obtido o peso não ajustado dos atores. A seguir, a complexidade de cada caso de uso é avaliada e é atribuído um peso. Estes pesos também são somados e é obtido o peso não ajustado dos casos de uso. Estes valores são somados e é obtido o valor para os pontos por caso de uso não ajustados. A este valor são aplicados os fatores de ajuste técnico e ambiental, obtendo-se assim o valor para os pontos por caso de uso. Os fatores de ajuste técnico e de ambiente levam em consideração fatores como usabilidade, instalabilidade, reusabilidade, complexidade de processamento, concorrência de processos, experiência da equipe, motivação, entre outros.

Adaptando a Equação (14), o *system range* é dado pela contagem dos pontos por caso de uso de cada caso de uso e a tolerância é dada pela estimativa baseada na média histórica das contagens de pontos por caso de uso da organização.

$$I_{FR_i} = \log\left(\frac{ucpc_i}{eucp_i}\right) \tag{17}$$

A Equação (17) representa o cálculo do conteúdo de informação do requisito funcional FR_i, onde $ucpc_i$ representa os pontos por caso de uso contados para o requisito funcional FR_i e $eucp_i$ a estimativa baseada nos pontos por caso de uso médios da organização para requisitos funcionais (FR) da mesma complexidade.

5.2.2 Conteúdo de Informação para a 3ª Etapa

Na 3a etapa da abordagem, são definidos subcasos de uso dependentes de características técnicas (ver seção 4.3.2). Nesta etapa, a complexidade relativa às atividades não pode ser calculada através dos pontos por caso de uso. Pelo fato dos requisitos funcionais representarem funcionalidades muito pequenas. Neste nível de decomposição ainda não estão identificadas características das classes suficientes para se calcular o conteúdo de informação através de métricas orientadas a objetos. Neste caso, pode-se calcular o conteúdo de informação através das funcionalidades do sistema como um todo. A métrica que se mostra mais adequada é a de pontos por função (VAZQUEZ; SIMÕES; ALBERT, 2003).

A métrica de pontos por função (*Function Points*), descrita em detalhes em

(VAZQUEZ; SIMÕES; ALBERT, 2003), é amplamente utilizada para a estimativa e medida de tamanho de sistemas computacionais. A contagem de pontos por função permite que o sistema seja mensurado a partir dos seus requisitos funcionais. Além disso, podem ser medidas partes do sistema, e em diferentes níveis de decomposição funcional e de abstração. Então, se para *system range* for adotado a contagem de pontos por função do requisito funcional (FR) e para tolerância for adotada a estimativa feita com base na média histórica para requisitos funcionais (FRs) de mesma complexidade, obtém-se a seguinte fórmula.

$$I = \log\left(\frac{pfc}{epf}\right) \tag{18}$$

Na Equação (18), é representado o cálculo do conteúdo de informação do conjunto de requisitos funcionais (FR) de um sistema específico na 3ª etapa, onde *pfc* representa os pontos por função contados para o sistema e *epf*, a estimativa dos pontos por função com base na média histórica da organização para um conjunto de requisitos funcionais (FRs) de complexidade semelhante.

5.2.3 Conteúdo de Informação para a 4ª Etapa

Para a 4ª etapa da abordagem, o cálculo de pontos por função não é mais satisfatório. Neste nível precisam ser medidas as complexidades de classes, métodos, interações entre objetos, hierarquia de herança, entre outros. Para satisfazer essas necessidades será adotado um conjunto de métricas de complexidade mais voltado para esses aspectos, como a definida em (CHIDAMBER; KEMERER, 1994).

Segundo Pressman (PRESSMAN, 2005), a mais amplamente referenciada dentre as métricas orientadas a objetos é o conjunto de métricas criado por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994). Este conjunto tem como vantagem medir as características de classes individuais como número e tamanho dos métodos, interações com outras classes, herança e encapsulamento, entre outras. Como os requisitos funcionais neste nível são serviços técnicos e os parâmetros de projeto são objetos ou variáveis é necessário usar uma métrica que contemple características de classes, atributos e métodos. Nesta métrica são definidas seis grandezas que serão medidas (CHIDAMBER; KEMERER, 1994):

1. Métodos ponderados por classe
2. Profundidade da árvore de herança
3. Número de subclasses

4. Acoplamento entre objetos
5. Respostas para a classe
6. Falta de coesão nos métodos

5.2.3.1 Métodos Ponderados por Classe

A grandeza "métodos ponderados por classe" (WMC, do inglês *weighted methods per class*) representa o total das complexidades dos métodos de uma classe. O valor é dado pela soma das complexidades de cada método. Uma forma simplificada de calcular o WMC é considerar os métodos com o mesmo valor para complexidade, sendo este valor igual a 1. Neste caso, o valor do WMC é igual ao número de métodos da classe. No caso de métodos maiores, pode-se calcular a complexidade do método usando-se métricas de tamanho de programa. A mais fácil de ser computada é o número de linhas de código (LOC[15]). Pode-se, também, calcular a complexidade de cada método usando a complexidade ciclomática (McCABE, 1976) como realizado em (VERVERS; van DALEN; van KATWIJK, 1996)(VERVERS; van DALEN; van KATWIJK, 1996).

O número de métodos de uma classe e a complexidade destes ajuda a estimar o tempo e o esforço para desenvolver e manter a classe (CHIDAMBER; KEMERER, 1994). Classes com um número grande de métodos são mais específicas para determinadas aplicações e limitam a possibilidade de reuso (CHIDAMBER; KEMERER, 1994). Sendo c_i a complexidade do método *i* de uma classe, WMC pode ser calculado pela Equação (19).

$$WMC = \sum_{n}^{i=1} c_i \tag{19}$$

5.2.3.2 Profundidade da Árvore de Herança

A profundidade da árvore de herança (DIT, do inglês *depth of the inheritance tree*), para uma classe, é definida como sendo o comprimento máximo do nó que representa a classe até a raiz da árvore (classes mais abstratas). Árvores de herança muito profundas geram muita complexidade no projeto pelo fato de mais classes e métodos estarem envolvidos (CHIDAMBER; KEMERER, 1994). A herança, ou generalização, pode aumentar a complexidade de uma classe pois o projetista deve conhecer, além dos métodos da própria classe, todos os métodos e atributos relacionados que esta classe herda. Neste caso, quanto

[15] do inglês, *Lines of Code*

mais profunda for a árvore de herança, maior o número de métodos a ser considerado. A Figura 37 ilustra uma árvore de herança. Neste exemplo, o valor de DIT para a classe "AlunoGraduacao" é igual a 2.

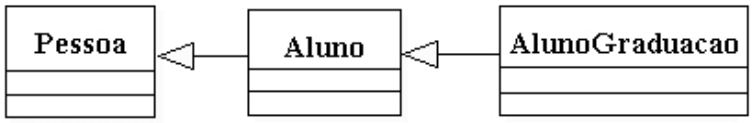


Figura 37 - Profundidade da árvore de herança (DIT)

5.2.3.3 Número de Subclasses

O número de subclasses (NOC, do inglês *number of children*) é o número de subclasses diretas de uma classe. Segundo CHIDAMBER e KEMERER (1994), quanto maior o número de subclasses diretas de uma classe, maior a possibilidade de que a herança tenha sido usada incorretamente para esta classe, em outras palavras, é provável que estejam faltando níveis da árvore de herança (CHIDAMBER; KEMERER, 1994). Uma classe com muitas subclasses diretas possui um potencial muito grande de propagação dos efeitos da mudança em um dos seus métodos, exigindo mais testes. A Figura 38 ilustra uma árvore de hierarquia. O valor para NOC para a classe "Obra" é igual a 4, pois ela tem 4 subclasses diretas.

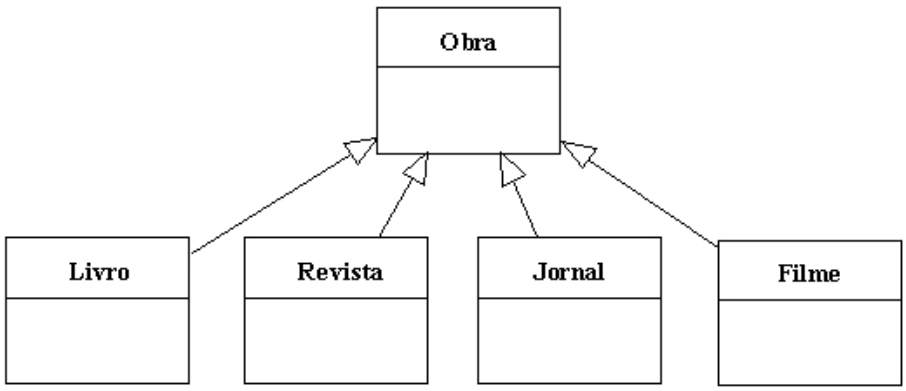


Figura 38 - Número de subclasses (NOC)

5.2.3.4 Acoplamento entre Objetos

A grandeza "acoplamento entre objetos" (CBO, do inglês *coupling between objects*) de uma classe é definida como sendo o número de outras classes com as quais esta classe está acoplada (relacionada através de uma associação). Dois objetos estão acoplados quando os métodos de um objeto usam métodos ou variáveis de instância do outro (CHIDAMBER; KEMERER, 1994). Em outras palavras, o número de classes das quais esta classe usa métodos ou variáveis de instância. Uma maneira prática para determinar o CBO é usando os cartões classe-responsabilidade-colaborador (CRC) definidos em (BECK; CUNNINGHAM, 1989). Um colaborador é uma classe que presta serviços para que uma outra classe consiga realizar suas responsabilidades.

Acoplamento excessivo entre objetos de um sistema prejudica o desenvolvimento modular, além de dificultar o reuso. Mesmo que esse acoplamento seja através de chamadas de métodos, ele faz com que o projetista precise se concentrar em várias outras classes além daquela que está projetando. Encapsulamento é uma forma de prevenir o acoplamento. Outra forma é projetar as classes de forma que o projeto satisfaça o Axioma da Independência.

5.2.3.5 Resposta para uma Classe

A grandeza "resposta para uma classe" (RFC, do inglês *response for a class*) é definida como a cardinalidade do conjunto de respostas de uma classe, como na Equação (20).

$$RFC = |RS| \tag{20}$$

O conjunto de respostas de uma classe é o conjunto de métodos de uma classe que podem potencialmente ser executados em resposta a uma mensagem recebida (CHIDAMBER; KEMERER, 1994). Quanto maior o conjunto de métodos que podem ser chamados de uma classe, maior a complexidade da classe (CHIDAMBER; KEMERER, 1994). Se um grande número de métodos de uma classe pode ser chamado por outras classes, as operações de teste e correção tornam-se mais complexas, exigindo maior experiência por parte do desenvolvedor (CHIDAMBER; KEMERER, 1994). Sendo *RS*, o conjunto de resposta da classe, $\{R_i\}$, o conjunto dos métodos da classe chamados pelo método *i* e $\{M\}$, o conjunto de todos os métodos da classe. O conjunto de resposta de uma classe, *RS,* pode ser definido pela Equação (21). O conjunto de resposta para cada método M_i corresponde ao método M_i e todos os métodos da classe chamados por M_i $\{R_i\}$. O conjunto de resposta de uma

classe, *RS*, é calculado pela união dos conjuntos de respostas de todos os métodos da classe.

$$RS = \{M\} \cup_{all\,i} \{R_i\} \tag{21}$$

5.2.3.6 Falta de Coesão nos Métodos

Considerando uma classe com um conjunto de métodos M_1, M_2, M_3, ..., M_n e $\{I_j\}$ o conjunto de variáveis de instância usados pelo método M_j. O conjunto P é definido como sendo o conjunto de pares de métodos (M_i, M_j) tal que $\{I_i\} \cap \{I_j\} = \emptyset$. O conjunto Q é definido como sendo o conjunto de pares de métodos (M_i, M_j) tal que $\{I_i\} \cap \{I_j\} \neq \emptyset$. A grandeza "falta de coesão nos métodos" (LCOM, do inglês *lack of cohesion of methods*) é definida pela Equação (22) (CHIDAMBER; KEMERER, 1994).

$$\begin{aligned} LCOM &= |P| - |Q|, se\, |P| > |Q| \\ LCOM &= 0, caso\ contrário \end{aligned} \tag{22}$$

A grandeza LCOM é a diferença entre o número de pares de métodos de uma classe que não compartilham um mesmo conjunto de variáveis de instância (atributos) e o número de pares que compartilham. LCOM mede a coesão entre os métodos de uma classe. Se LCOM é alto, isto pode significar que a classe pode ser dividida em duas ou mais sub-classes (CHIDAMBER; KEMERER, 1994). Como visto no capítulo 2, a coesão de um módulo de *software* mede o grau com que as tarefas executadas por este módulo se relacionam entre si (IEEE, 1990). Quando dois métodos, de uma mesma classe, não compartilham atributos entre si, pode-se dizer que o grau de relacionamento entre eles é baixo e, portanto, a coesão entre eles é baixa.

O valor do conteúdo de informação do 4º nível de decomposição é obtido pela soma dos valores do conteúdo de informação para cada classe. O valor do conteúdo de informação de cada classe é calculado pela soma dos conteúdos de informação relativos a cada umas das grandezas definidas em (CHIDAMBER; KEMERER, 1994), como apresentado na Equação (23).

$$I_{classe_i} = I_{WMC_i} + I_{DIT_i} + I_{NOC_i} + I_{CBO_i} + I_{RFC_i} + I_{LCOM_i} \tag{23}$$

Este cálculo pode ser feito, também, usando-se fatores de ponderação para cada

grandeza, como apresentado na Equação (24). Nesta equação, α, β, δ, γ, θ e ω são os fatores de ponderação. Estes fatores são definidos pelo projetista com base na importância do conteúdo de informação de cada grandeza.

$$I_{classe_i} = \alpha I_{WMC_i} + \beta I_{DIT_i} + \delta I_{NOC_i} + \gamma I_{CBO_i} + \theta I_{RFC_i} + \omega I_{LCOM_i} \quad (24)$$

Os valores dos conteúdos de informação são calculados pelas relações entre os valores encontrados para cada classe, para cada grandeza, apresentados nas Equações (25), (26), (27), (28), (29) e (30).

$$I_{WMC_i} = \log\left(\frac{WMC_i}{EWMC}\right) \quad (25)$$

Na Equação (25), I_{WMC_i} é o conteúdo de informação relativo à grandeza "métodos ponderados por classe" (WMC), para a classe *i*. WMC_i é o valor obtido da grandeza para a classe em questão e *EWMC* é o valor estimado de *WMC* para a classe com base na média histórica da organização.

$$I_{DIT_i} = \log\left(\frac{DIT_i}{EDIT}\right) \quad (26)$$

Na Equação (26), I_{DIT_i} é o conteúdo de informação relativo à grandeza "profundidade da árvore de herança" (DIT), para a classe *i*. DIT_i é o valor obtido da grandeza para a classe em questão e *EDIT* é o valor estimado de *DIT* para a classe com base na média histórica da organização.

$$I_{NOC_i} = \log\left(\frac{NOC_i}{ENOC}\right) \quad (27)$$

Na Equação (27), I_{NOC_i} é o conteúdo de informação relativo à grandeza "número de subclasses" (NOC), para a classe *i*. NOC_i é o valor obtido da grandeza para a classe em questão e *ENOC* é o valor estimado de *NOC* para a classe com base na média histórica da organização.

$$I_{CBO_i} = \log\left(\frac{CBO_i}{ECBO}\right) \quad (28)$$

Na Equação (28), I_{CBO_i} é o conteúdo de informação relativo à grandeza

"acoplamento entre objetos" (CBO), para a classe *i*. CBO_i é o valor obtido da grandeza para a classe em questão e *ECBO* é o valor estimado de *CBO* para a classe com base na média histórica da organização.

$$I_{RFC_i} = \log\left(\frac{RFC_i}{ERFC}\right) \tag{29}$$

Na Equação (29), I_{RFC_i} é o conteúdo de informação relativo à grandeza "resposta para a classe" (RFC), para a classe *i*. RFC_i é o valor obtido da grandeza para a classe em questão e *ERFC* é o valor estimado de *RFC* para a classe com base na média histórica.

$$I_{LCOM_i} = \log\left(\frac{LCOM_i}{ELCOM}\right) \tag{30}$$

Na Equação (30), I_{LCOM_i} é o conteúdo de informação relativo a "falta de coesão nos métodos" (LCOM), para a classe *i*. $LCOM_i$ é o valor obtido da grandeza para a classe em questão e *ELCOM* é o valor estimado de *LCOM* para a classe com base na média histórica.

Como o cálculo do conteúdo de informação usa logaritmos, ele possui algumas restrições quando os valores das métricas são iguais a zero. Para que os cálculos possam ser efetuados é necessário que sejam feitas algumas aproximações. Quando o valor obtido para uma métrica for igual a zero, o conteúdo de informação para esta métrica é igual a zero. Por outro lado, o valor estimado para uma métrica não pode ser zero. Como este valor é dado pelo projetista, pode-se aproximar este valor, atribuindo-se um valor estimado entre 0 e a média histórica da organização para a métrica.

Como exemplo de valores obtidos para as métricas de Chidamber e Kemerer, são apresentados dois estudos realizados. O primeiro estudo foi realizado por Chidamber e Kemerer e compara os valores obtidos para as métricas em duas organizações diferentes (CHIDAMBER; KEMERER, 1994). A primeira organização, chamada site A, desenvolve *software* para vendas e possui uma coleção de bibliotecas de classes C++ (STROUSTRUP, 1997). O estudo nesta organização contempla 634 classes de duas bibliotecas usadas para desenvolver interfaces gráficas para usuários (GUI). A segunda organização deste estudo é um fabricante de semi-condutores que usa Smalltalk (GOLDBERG; ROBSON, 1989) para desenvolver sistemas de controle flexíveis de manufatura. Para esta organização foram analisadas 1459 classes.

Tabela 3 - Valores comparativos para as métricas CK entre classes em C++ e Smalltalk

	WMC, $c_i = 1$	DIT	NOC	CBO	RFC	LCOM
Site A - C++ 634 classes	5	1	0	0	6	0
Site B - Smalltalk 1459 classes	10	3	0	9	29	2
Valores relativos às medianas de cada amostra.						

Na Tabela 3, são mostrados os valores para as métricas de Chindamber e Kemerer para o estudo realizado em (CHIDAMBER; KEMERER, 1994). É necessário considerar as diferenças entre as linguagens de programação e principalmente as diferenças entre os tipos e propósitos das classes analisadas. Outro detalhe importante a ser considerado é que neste estudo os valores para a métrica "métodos ponderados por classe" (WMC) levam em consideração apenas o número de métodos de cada classe ($c_i = 1$).

Num outro estudo, Verves, van Dalen e van Katwijk obtiveram valores para algumas métricas orientadas a objetos, entre elas 5 das 6 métricas de Chidamber e Kemerer (VERVERS; van DALEN; van KATWIJK, 1996). Foram medidos valores para 958 classes escritas na linguagem Objective C (COX, 1991). O objetivo deste estudo foi avaliar as diferenças entre as principais métricas orientadas a objetos, como as descritas em (LORENZ; KIDD, 1994) e (CHIDAMBER; KEMERER, 1994).

Tabela 4 - Valores das métricas CK no estudo de Ververs, vDalen e vKatwijk (1996)

	WMC ciclomático	Linhas p/ método	Complexidade ciclomática	DIT	NOC	CBO	LCOM
Média	59,27	9,44	2,75	3,4	1	5,45	286,48
Desvio padrão	76,22	6,38	1,52	2	8,21	7,22	1279,81
Mediana	37	7,95	2,39	3	0	3	11

Os valores obtidos para a métrica WMC, apresentados na Tabela 4, levam em consideração a complexidade de cada método. Para obter esta complexidade foi usada a métrica da complexidade ciclomática descrita em (McCABE, 1976). Além disso foram obtidos os valores médios para número de linhas de código por método, que é uma medida de complexidade mais simples de ser obtida.

5.3 Aplicação do Conteúdo de Informação na Metodologia

O conteúdo de informação é um parâmetro importante para se escolher entre as diferentes soluções possíveis para um projeto. A metodologia apresentada neste trabalho propõe o cálculo do conteúdo de informação a cada decomposição feita para, no caso de surgir mais de uma possibilidade para a decomposição, facilitar a decisão de qual adotar.

Os pontos por caso de uso, usados para calcular o conteúdo de informação para o primeiro e o segundo nível de decomposição, podem ser contados atribuindo-se pesos não ajustados, para cada ator, de acordo com sua complexidade e para cada caso de uso, de acordo com o número de transações efetuadas. Estes pesos são somados e ajustados de acordo com fatores de complexidade técnica e ambiental, para dar a contagem final dos pontos por caso de uso (ANDA et al., 2001). Com a ajuda de ferramentas CASE (*Computer Aided Software Engineering*) como o *Enterprise Architect*[16], o projetista atribui os pesos não ajustados para os atores e casos de uso do sistema, além dos fatores de ajuste, e a ferramenta CASE efetua o cálculo dos pontos por caso de uso.

No terceiro nível de decomposição, o conteúdo de informação é calculado através dos pontos por função. Para isto, também existem várias opções de ferramentas de *software*. Mas para se calcular o conteúdo de informação é necessário que seja possível estimar os valores de referência com base em registros históricos da empresa que vai realizar o projeto. Então, é necessário que a empresa mantenha controle e registros dos projetos realizados, o que é uma característica de maturidade de processo de *software* (PAULK et al., 1991).

No quarto nível de decomposição, onde são obtidos os serviços técnicos, classes, objetos e interações, as matrizes de projeto costumam ser muito grandes. Freqüentemente, as matrizes das soluções alternativas apresentam dimensões muito diferentes umas das outras . Devido a este fato, medidas de independência funcional como a reangularidade apresentam distorções prejudicando a escolha da melhor solução de projeto. Portanto, neste nível de decomposição, é mais adequado usar o conteúdo de informação como critério para decidir qual a melhor solução entre soluções que satisfazem o Axioma 1.

Para o quarto nível de decomposição será usado o conjunto de métricas definido em (CHIDAMBER; KEMERER, 1994). A grande vantagem deste conjunto de métricas é a facilidade em se calcular seus valores. Estes valores podem ser calculados durante o projeto, usando-se os diagramas de classe e seqüência da UML 2.0 (OBJECT MANAGEMENT

[16] O *software Enterprise Arquitect* é marca registrada da *Sparx Systems*

GROUP, 2005), ou diretamente do código através de uma análise, que pode ser feita através de uma ferramenta automatizada, como a CKJM[17]. Isto permite que projetos anteriores da organização sejam analisados e seus valores obtidos, auxiliando a obtenção da estimativa a ser usada como referência.

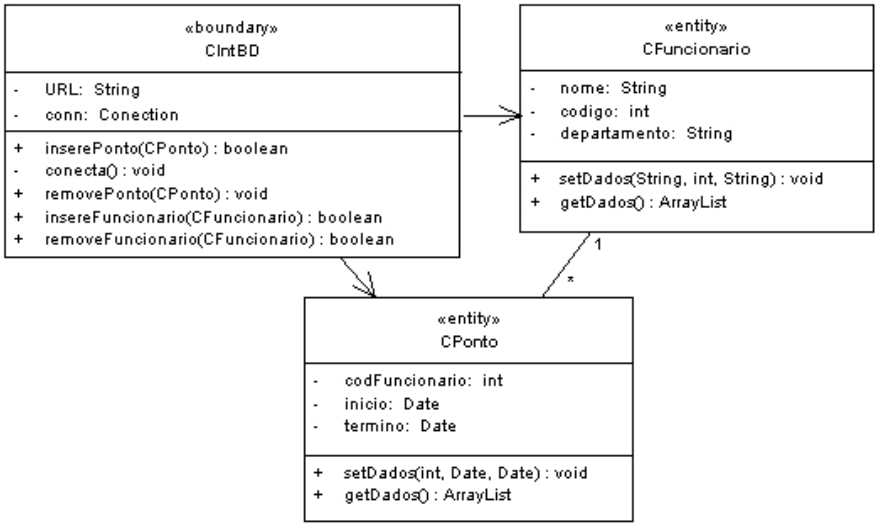


Figura 39 - Classes para o cálculo do conteúdo de informação

Para ilustrar como este cálculo do conteúdo de informação é realizado, será apresentado um exemplo a seguir. No diagrama de classes da Figura 39, são apresentadas as classes "CIntBD", "CPonto" e CFuncionario". Será apresentado o cálculo do conteúdo de informação para as classes "CIntBD", "CFuncionario" e "CPonto".

Tabela 5 - Conteúdo de informação das classes "CIntBD", "CFuncionario" e "CPonto"

CLASSE	*WMC*	*EWMC*	*DIT*	*EDIT*	*NOC*	*ENOC*	*CBO*	*ECBO*	*RFC*	*ERFC*	*LCOM*	*ELCOM*	I_{classe}
CIntBD	5	7	0	0	0	0	2	2	5	4	0	0	0,08230
I_M	-0,0146128		0		0		0		0,09691		0		
CFuncionario	2	2	0	0	0	0	0	0	2	2	0	0	0
I_M	0		0		0		0		0		0		
CPonto	2	2	0	0	0	0	0	0	2	2	0	0	0
I_M	0		0		0		0		0		0		
total													0,08230

Na Tabela 5, são apresentados os valores obtidos e de referência para cada métrica, o conteúdo de informação para cada métrica e para cada uma das classes "CIntBD",

[17] A ferramenta CKJM foi desenvolvida por Diomidis Spinellis da Universidade de Economia e Negócios de Atenas - Grécia. Disponível em http://www.spinellis.gr/sw/ckjm/doc/indexw.html

"CFuncionario" e "CPonto". A primeira coluna apresenta o nome da classe, as colunas 2 a 12, os valores obtidos e de referência para cada métrica e a última coluna, o valor do conteúdo de informação para a classe. Abaixo de cada conjunto de valores obtidos e de referência para cada métrica, é apresentado o valor do conteúdo de informação para a métrica.

O conteúdo de informação de cada classe é calculado pela soma dos conteúdos de informação de cada grandeza, apresentados na Tabela 5. O valor obtido é $I = 0,08230$. Este valor é um valor baixo para o conteúdo de informação. Isto pode indicar que estas classes podem representar uma boa solução. Mas, só é possível ter certeza de que esta solução é melhor que outras com base no conteúdo de informação total, isto é, quando todas as classes já tiverem sido projetadas e avaliadas.

Supondo os mesmos requisitos funcionais (FRs) do exemplo anterior, será considerado que o projetista criou apenas uma classe, chamada "CIntBDNova", para receber dados de ponto e funcionário, retornar dados de ponto e empregado e gravar e remover esses dados. Esta classe reune todos os métodos e atributos das classes "CIntBD", "CFuncionario" e "CPonto". A classe "CIntBDNova" é apresentada na Figura 40. Esta classe irá substituir as classes "CIntBD", "CFuncionario" e "CPonto" no sistema.

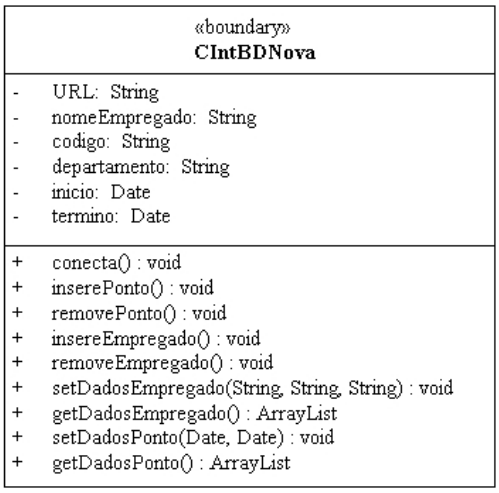


Figura 40 - Classe CIntBDNova

O conteúdo de informação da classe "CIntBDNova" é calculado pela soma dos conteúdos de informação de cada grandeza, apresentados na Tabela 6. O valor obtido é $I_{CIntBDNova} = 0,82930$. A classe "CIntBDNova" possui um conteúdo de informação maior que as classes "CIntBD", "CFuncionario" e "CPonto" combinadas. O fator que mais pesou neste resultado foi o valor obtido para a grandeza "falta de coesão nos métodos" (LCOM), o que

indica que a classe "CIntBDNova" tem um grau de coesão baixo. Neste caso, pelo Axioma 2, as classes "CIntBD", "CFuncionario" e "CPonto", separadas, podem ser consideradas uma solução melhor que se fossem reunidas em uma só classe, como na classe "CIntBDNova".

Tabela 6 - Conteúdo de informação da classe "CIntBDNova"

CLASSE	*WMC*	*EWMC*	*DIT*	*EDIT*	*NOC*	*ENOC*	*CBO*	*ECBO*	*RFC*	*ERFC*	*LCOM*	*ELCOM*	I_{classe}
CIntBDNova	9	9	0	0	0	0	0	0	9	8	12	2	0,82930
I_M	0		0		0		0		0,051152		0,7781510		

5.4 Conclusões

Neste capítulo foi discutido a aplicação do Axioma da Informação em projetos de *software* orientados a objetos, principalmente, na abordagem proposta. O Axioma 2, o Axioma da Informação, é baseado no conceito de conteúdo de informação. Este capítulo apresentou a definição de informação para projetos de *software* orientado a objetos. Baseado nessa definição, foi proposto um arcabouço para calcular o conteúdo de informação para projeto de *software* orientado a objetos. Este arcabouço é baseado na probabilidade de uma organização desenvolver o sistema que satisfaça seus requisitos funcionais, usando para isto métricas de complexidade de *software* bem estabelecidas na literatura como pontos por caso de uso, pontos por função e o conjunto de métricas orientadas a objetos de Chidamber e Kemerer. Estas métricas foram adotadas porque são bem conhecidas na literatura e porque têm métodos bem estabelecidos para serem computadas, podendo inclusive ser calculadas com o auxílio de ferramentas computacionais. Foi apresentado um exemplo da cálculo de conteúdo de informação de classes, aplicando este valor para escolher o conjunto de classes mais adequado para a implementação entre duas alternativas.

6 Estudo de Caso

Este capítulo apresenta um estudo de caso para exemplificar e avaliar a abordagem proposta nesta tese. Neste estudo de caso é apresentado o projeto de um *software* que serve como base para avaliar se o uso da abordagem proposta permite aplicar os princípios da Teoria de Projeto Axiomático a um projeto de *software* que segue o Processo Unificado. Além disso, são avaliadas as principais vantagens e desvantagens da aplicação da abordagem proposta em um projeto de *software* orientado a objetos.

Neste Capítulo, são apresentados os objetivos do estudo de caso, a descrição das características do sistema a ser projetado, a descrição da plataforma de *hardware* e *software* na qual o sistema está inserido e o ambiente de desenvolvimento usado. É apresentada, também, a descrição da aplicação da abordagem proposta em cada nível de decomposição, descrevendo-se as decisões de projeto tomadas com base nos axiomas.

6.1 Objetivos do Estudo de Caso

O objetivo geral deste estudo de caso é aplicar a abordagem proposta a um projeto de *software* orientado a objetos, passando por todas as etapas e avaliar a aplicação da abordagem de projeto proposta nesta tese. Como objetivos específicos deste estudo de caso tem-se:

1. Avaliar se as correspondências estabelecidas e as etapas da abordagem proposta permitem a aplicação do Projeto Axiomático a um projeto de *software* orientado a objetos que segue o Processo Unificado.
2. Verificar se a aplicação dos Axiomas 1 e 2 permite identificar a melhor solução de projeto entre algumas soluções alternativas e permite garantir a qualidade da solução de projeto, identificando e possibilitando descartar soluções de projeto não adequadas.
3. Avaliar quais vantagens a aplicação da abordagem proposta traz ao

desenvolvedor do ponto de vista de tomada de decisões de projeto.

6.2 Descrição do Sistema

O sistema projetado neste estudo de caso é um sistema embarcado de tempo real para um *hardware* específico, uma placa de avaliação *eSysTech eAT55*[18]. O objeto deste estudo de caso foi escolhido por possuir características de sistemas de tempo real e de sistemas embarcados. O propósito do sistema é receber dados de operação, como temperatura e pressão de uma máquina ou outro tipo de equipamento conectado à placa eAT55 através da porta serial e mostrar esses dados e outros eventos da máquina em um *display*. O sistema será capaz também de receber comandos do usuário através de um teclado para selecionar a informação a ser visualizada.

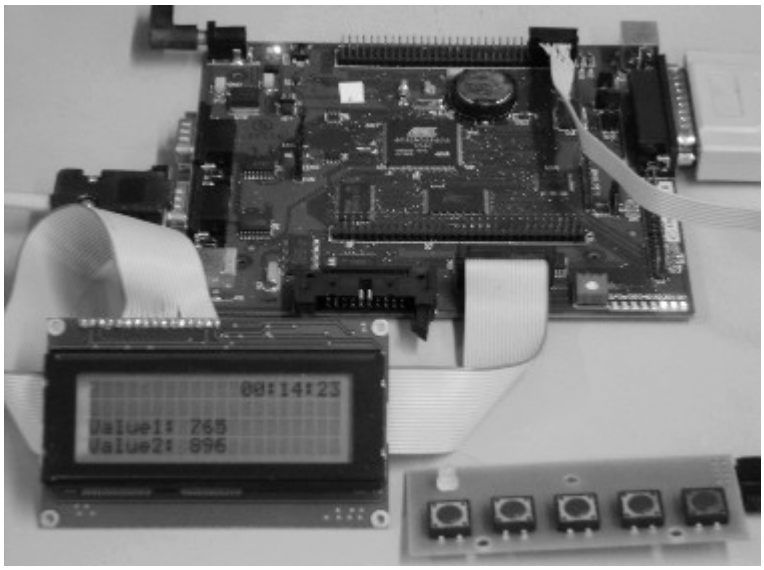


Figura 41 - Foto do sistema desenvolvido em execução na placa de avaliação eAT55

A eAT55, ilustrada na Figura 41, é uma placa de avaliação produzida pela eSysTech[19] para a arquitetura ARM[20]. Placas de avaliação como a eAT55 são usadas como plataforma para experimentação, aprendizado, avaliação e treinamento no desenvolvimento de sistemas embarcados. Este tipo de *hardware* é usado em departamentos de engenharia, laboratórios de pesquisa, universidades, escolas, centros de treinamento, entre outros

[18] eSysTech eAT55's *evaluation board* é marca registrada da eSysTech
[19] Http://www.esystech.com.br
[20] ARM (Advanced RISC Machines) Holdings, http://www.arm.com/

(ESYSTECH, 2005).

A eAT55 usa um processador AT91M5580 da Atmel[21] com núcleo ARM7TDMI de 32 bits, baseado no modelo de arquitetura *Reduced Instruction Set Computer* (RISC). O núcleo ARM é o mais usado por fabricantes de processadores para sistemas embarcados como Intel, Motorola, Atmel, Samsung, Sharp, entre outros. A placa eAT55 inclui também *Static Random Access Memory* (SRAM), memória *Flash*, *Analog to Digital Converter* (ADC), *Digital to Analog Converter* (DAC), *Real-time Clock* (RTC), interfaces (serial, *universal serial bus* (USB), PS/2 e *liquid cristal display* (LCD)), além de disponibilizar o barramento do processador para expansão (ESYSTECH, 2005). O sistema alvo deste estudo de caso usará, além da placa eAT55, um *display* LCD com 4 linhas e 20 colunas, um teclado com 5 teclas e um dispositivo (botão) para reiniciar o sistema pela entrada DAC.

A aplicação desenvolvida no estudo de caso é executada sobre um núcleo de tempo-real desenvolvido pela eSysTech chamado *X Real-Time Kernel*. O *X Kernel* é uma camada de *software* que oferece serviços compartilhados de gerenciamento de tarefas, de sincronização, de temporização e de troca de mensagens. Além disso, ele disponibiliza formas de interagir com periféricos e/ou recursos de *hardware* através de *drivers* de dispositivos. Com isso o *X Kernel* permite que o desenvolvedor se concentre na lógica da aplicação propriamente dita, abstraindo detalhes sobre o gerenciamento de tarefas, sua sincronização, temporização e troca de mensagens. Isto simplifica o desenvolvimento do sistema embarcado (ESYSTECH, 2007).

O desenvolvimento da aplicação deste estudo de caso foi feito usando-se o ambiente de desenvolvimento IAR *Embbedded Workbench* para ARM[22]. Trata-se de um ambiente integrado de desenvolvimento, possuindo funcionalidades de edição, compilação, montagem, depuração e conexão com o *hardware* usado. A implementação deste estudo de caso foi feita em linguagem C++.

6.3 Descrição do Estudo de Caso

Este estudo de caso foi desenvolvido seguindo as etapas da abordagem proposta. Inicialmente, foi realizada a definição do escopo de sistema. A partir desta definição, foi iniciada a aplicação da abordagem passando por todos os níveis de decomposição. Segundo a abordagem, durante as etapas foi realizada a aplicação dos Axiomas 1 e 2. Nas etapas 1, 2 e 3,

[21] Atmel corporation, http://www.atmel.com/
[22] Marca registrada da IAR *Systems*, http://www.iar.com/

foi aplicado o Axioma 1, que permitiu identificar a melhor solução entre duas soluções aceitáveis. A aplicação do Axioma 1 permitiu, também, descartar soluções não aceitáveis, principalmente soluções com matrizes de projeto não quadradas. Na quarta etapa que corresponde ao quarto nível de decomposição foi aplicado o Axioma 2 para identificar a melhor solução, já que, para matrizes maiores, o cálculo da reangularidade (ver Equação (9)) não se mostra adequado. No quarto nível, juntamente com a matriz de projeto, foram criados diagramas da UML, previstos pelo Processo Unificado. A seguir, foi realizada a implementação do sistema de acordo com o projeto realizado.

6.3.1 Etapa de Modelagem de Casos de Uso

O projeto se inicia com o primeiro nível de decomposição, que corresponde à primeira etapa da abordagem proposta, a modelagem de casos de uso. Na primeira atividade desta etapa, os casos de uso são identificados e descritos. Os casos de uso identificados, neste estudo, são mostrados na Figura 42.

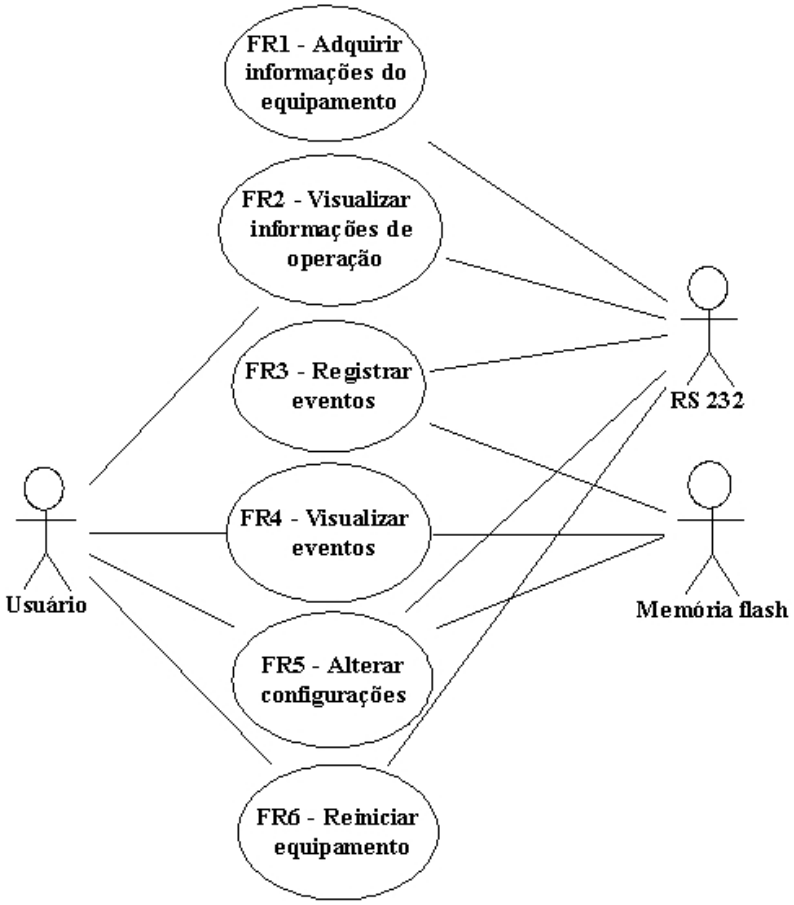


Figura 42 - Casos de uso para o sistema do estudo de caso

No caso de uso “FR1 - Adquirir informações do equipamento”, o sistema verifica, a cada intervalo de tempo de 50 ms, se o equipamento conectado à porta serial enviou alguma informação, seja ela informação de operação ou evento. Caso afirmativo, o sistema coloca esta informação à disposição dos outros módulos do sistema. O caso de uso “FR2 - Visualizar informações de operação” permite que o usuário visualize, através do *display*, as informações referentes à operação do equipamento conectado ao sistema. Este caso de uso permite, também, que o usuário escolha, usando o teclado, a informação que ele deseja ver.

No caso de uso “FR3 - Registrar eventos”, o sistema grava os eventos oriundos do equipamento conectado, na memória *flash*. O caso de uso “FR4 - Visualizar eventos” permite que o usuário visualize os eventos ocorridos com o equipamento conectado ao sistema. Estes eventos estão armazenados na memória *flash*. Este caso de uso permite que o usuário selecione o evento que ele deseja ver. O caso de uso “FR5 - Alterar configurações” permite que o usuário altere configurações do sistema. Estas configurações estão armazenadas na memória *flash*. O caso de uso “FR6 - Reiniciar equipamento” permite que o usuário reinicie o equipamento conectado ao sistema.

O ator “Usuário” representa o papel do usuário humano que interage com o sistema através do teclado, do *display* e do botão para reiniciar o equipamento conectado. O ator “RS232” representa o dispositivo “porta serial” que serve para comunicação com o equipamento conectado. O ator “Memória Flash” representa o dispositivo de armazenamento de informações do sistema.

A segunda atividade da primeira etapa é a criação das colaborações, que correspondem aos parâmetros de projeto, para satisfazer os casos de uso. Para cada caso de uso identificado foi criada uma colaboração. As colaborações “DP1 - Adquirir informações do equipamento”, “DP2 - Visualizar informações de operação”, “DP3 - Registrar eventos”, “DP4 - Visualizar eventos”, “DP5 - Alterar configurações” e “DP6 - Reiniciar equipamento” foram criadas para satisfazer os casos de uso “FR1 - Adquirir informações do equipamento”, “FR2 - Visualizar informações de operação”, “FR3 - Registrar eventos”, “FR4 - Visualizar eventos”, “FR5 - Alterar configurações” e “FR6 - Reiniciar equipamento”, respectivamente. As colaborações criadas estão ilustradas na Figura 43.

As colaborações representam relacionamentos entre papéis (instâncias) para implementar alguma funcionalidade. As colaborações podem ser expressas em vários níveis de abstração, podendo ser refinadas e expandidas em uma ou mais colaborações (RUMBAUGH; JACOBSON; BOOCH, 2004). No nível de abstração correspondente à primeira etapa da abordagem, as colaborações devem ser criadas, mas não existe a

necessidade da identificação dos papéis que as compõem.

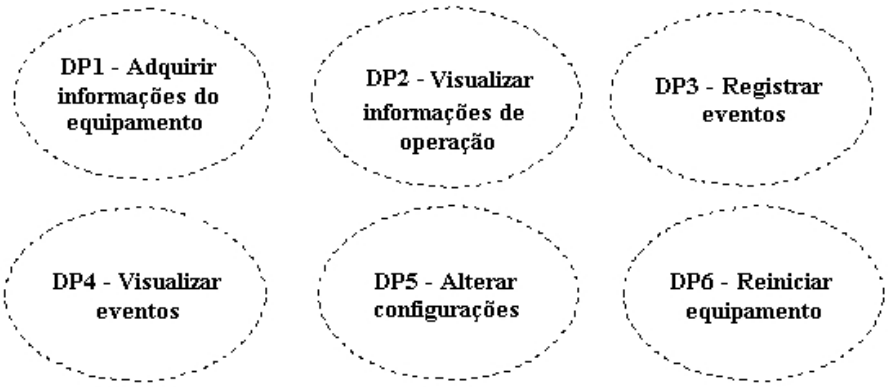


Figura 43 - Colaborações para o 1o. nível de decomposição

De acordo com a abordagem proposta, a próxima atividade corresponde a mapear as dependências entre os requisitos funcionais, representados pelos casos de uso, e os correspondentes parâmetros de projeto, representados pelas colaborações, na matriz de projeto. A matriz de projeto, mostrada na Figura 44, representa nas linhas os casos de uso e as respectivas colaborações nas colunas.

FRs \ DPs	DP1 - Adquirir informações do equipamento	DP2 - Visualizar informações de operação	DP3 - Registrar eventos	DP4 - Visualizar eventos	DP5 - Alterar configurações	DP6 - Reiniciar equipamento
FR1 - Adquirir informações do equipamento	X					
FR2 - Visualizar informações de operação		X				
FR3 - Registrar eventos			X			
FR4 - Visualizar eventos				X		
FR5 - Alterar configurações					X	
FR6 - Reiniciar equipamento						X

Figura 44 - Matriz de projeto de 1o. nível para o sistema

A matriz de projeto representa as relações entre os casos de uso e as colaborações para o sistema. Por exemplo, o “X” na célula que relaciona o caso de uso “FR2 - Visualizar informações de operação” e a colaboração “DP2 - Visualizar informação de operação” indica

que esta colaboração participa da realização do caso de uso em questão. Neste estudo de caso, quando as colaborações foram criadas, para o nível de decomposição da primeira etapa, não foram criados os papéis das instâncias participantes em cada colaboração. Isto será feito na segunda etapa, modelagem dos subcasos independentes de características técnicas.

A atividade seguinte é a aplicação do Axioma 1. A solução de projeto satisfaz o Axioma 1. Ela é uma solução desacoplada, o que significa que ela é uma solução de projeto satisfatória. Se for calculada a reangularidade para esta solução, será obtido o valor 1. Neste estudo de caso a solução de projeto se mostrou adequada e continuará a ser desenvolvida nas próximas etapas.

6.3.2 Etapa de Modelagem de Subcasos de Uso Independentes de Características Técnicas

Na segunda etapa da abordagem, a primeira atividade é a decomposição dos casos de uso identificados na primeira etapa em subcasos de uso independentes de características técnicas. A seguir, na segunda atividade desta etapa, são criadas colaborações para satisfazer os subcasos de uso independentes, tomando como base, as colaborações criadas na primeira etapa. Na terceira atividade desta etapa, as relações de dependência entre os subcasos e as colaborações são mapeadas na matriz de projeto e, na quarta atividade desta etapa, o Axioma 1 é aplicado para validar as soluções de projeto encontradas e auxiliar na escolha da solução mais adequada.

6.3.2.1 Decomposição de Subcasos de Uso Independentes de Características Técnicas

Na primeira atividade da segunda etapa, que corresponde à decomposição dos subcasos de uso independentes de características técnicas, foram identificadas algumas decomposições. Os subcasos de uso independentes de características técnicas identificados estão ilustrados na Figura 45. Os casos de uso “FR1 - Adquirir informações do equipamento”, “FR3 - Registrar eventos” e “FR6 - Reiniciar equipamento” não foram decompostos, mas são representados neste nível para que a solução desta etapa contemple toda a funcionalidade do sistema.

O caso de uso “FR2 - Visualizar informações de operação” foi decomposto nos subcasos de uso independentes de características técnicas: “FR2.1 - Mostrar dados de operação” e “FR2.2 - Selecionar visualização de dados”. O subcaso “FR2.1 - Mostrar dados de operação” exibe os dados de operação do equipamento para o usuário e o subcaso “FR2.2 -

Selecionar visualização de dados", permite ao usuário selecionar a informação que ele deseja visualizar. O caso de uso "FR4 - Visualizar eventos" foi decomposto nos subcasos de uso independentes: "FR4.1 - Mostrar eventos" e "FR4.2 - Selecionar visualização de eventos". O subcaso "FR4.1 - Mostrar eventos" exibe os eventos gerados na operação do equipamento para o usuário e o subcaso "FR4.2 - Selecionar visualização de eventos", permite ao usuário escolher o evento que ele deseja visualizar. O caso de uso "FR5 - Alterar configurações" foi decomposto nos subcasos: "FR5.1 - Alterar configurações" e "FR5.2 - Selecionar configurações". O subcaso "FR5.1 - Alterar configurações" realiza a alteração da configuração e o subcaso "FR5.2 - Selecionar configurações", permite ao usuário escolher a configuração desejada.

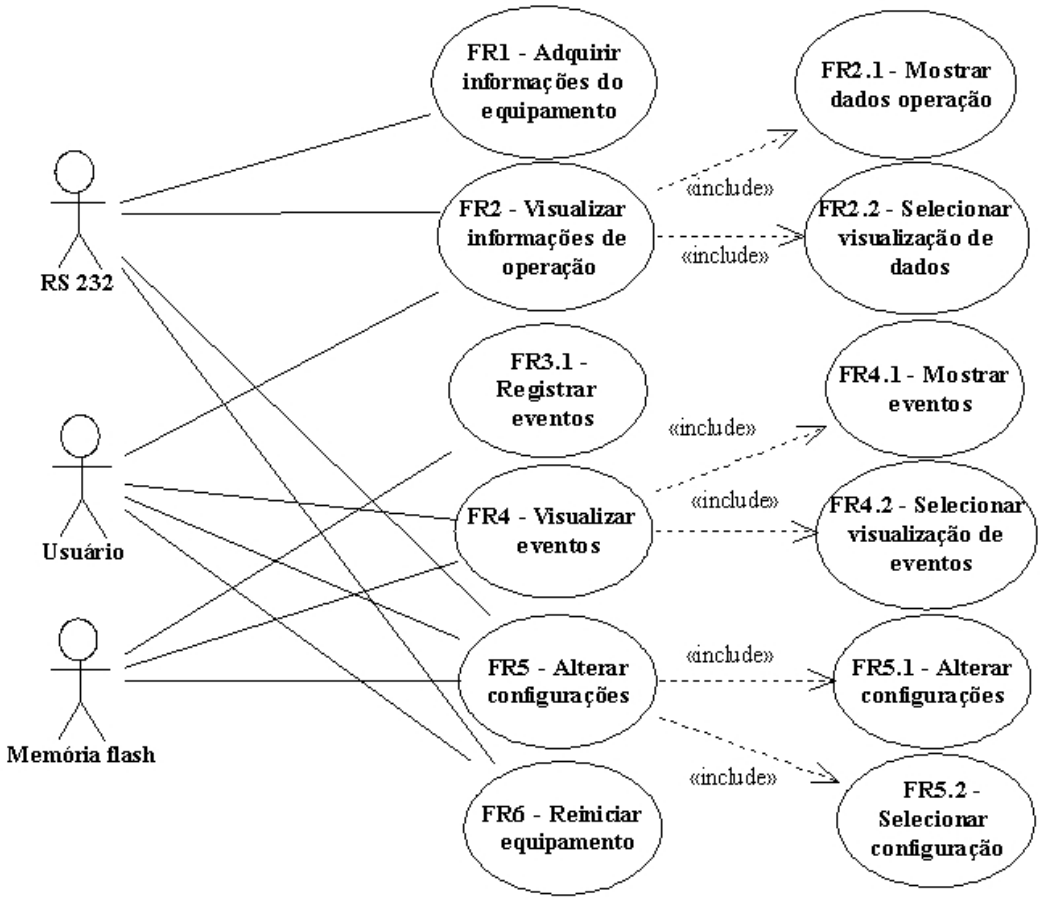


Figura 45 - Subcasos de uso independentes do 2o. nível de decomposição

6.3.2.2 Criação das Colaborações para Satisfazer os Subcasos de Uso Independentes

A segunda atividade da segunda etapa é a criação das colaborações para satisfazer

os subcasos de uso independentes de características técnicas. As colaborações “DP2.1 - Mostrar dados de operação”, “DP2.2 - Selecionar visualização de dados”, “DP4.1 - Mostrar eventos”, “DP4.2 - Selecionar visualização de eventos”, “DP5.1 - Alterar configurações” e “DP5.2 - Selecionar configuração” foram criadas para satisfazer os subcasos de uso independentes de características técnicas “FR2.1 - Mostrar dados de operação”, “FR2.2 - Selecionar visualização de dados”, “FR4.1 - Mostrar eventos”, “FR4.2 - Selecionar visualização de eventos”, “FR5.1 - Alterar configurações” e “FR5.2 - Selecionar configuração”, respectivamente. As colaborações criadas estão ilustradas na Figura 46.

Para as colaborações “DP1 - Adquirir informações de equipamento”, “DP3 - Registrar eventos” e “DP6 - Reiniciar equipamento”, foram identificadas as instâncias participantes.

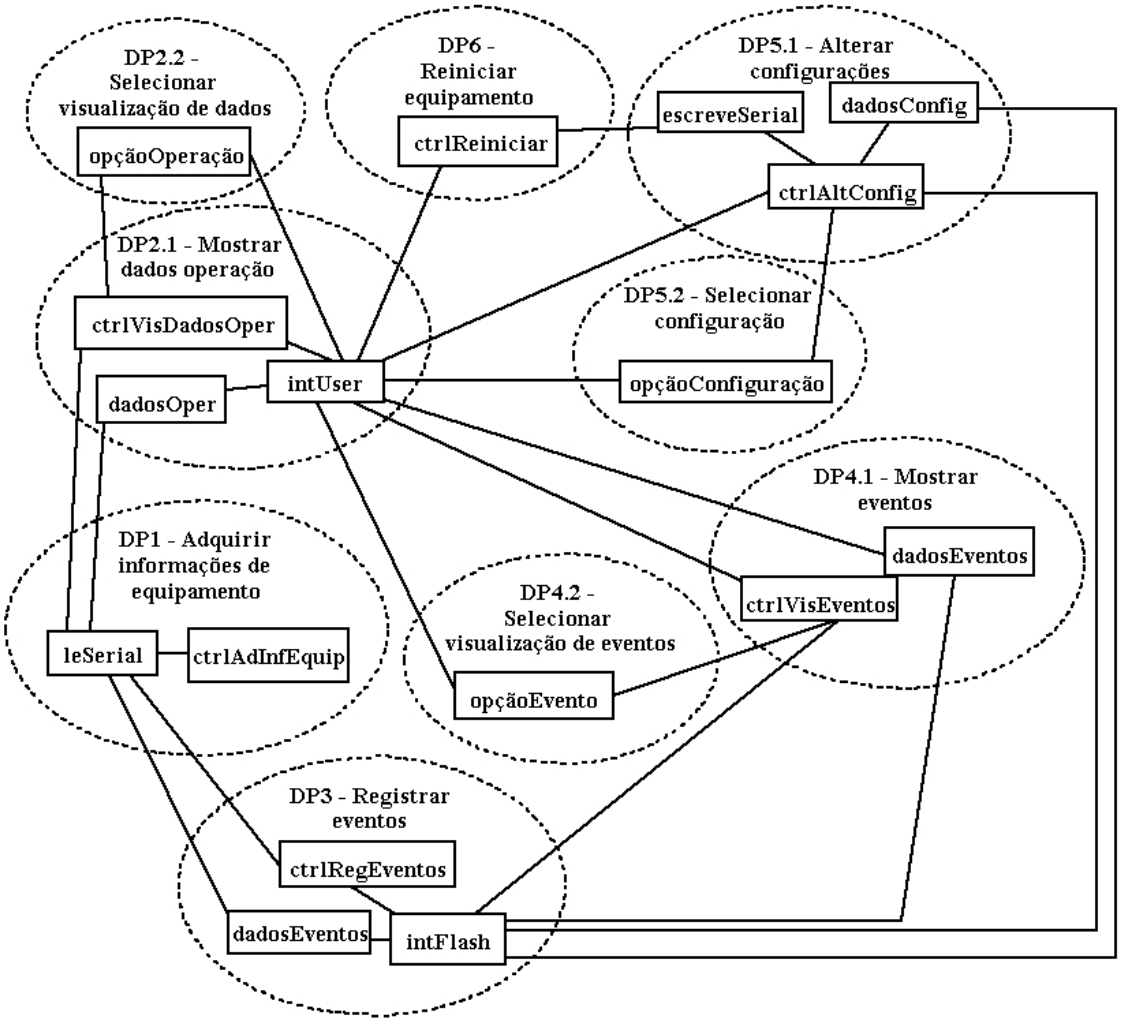


Figura 46 - Colaborações e instâncias para a solução na 2a. etapa

Além disso, para cada colaboração criada, foram identificadas as instâncias que

participam na colaboração. Nesta segunda etapa da abordagem, as instâncias são identificadas através dos papéis que representam nas colaborações, sem a preocupação de identificar suas classes. Uma instância pode participar de mais de uma colaboração, o que estabelece uma dependência entre essas colaborações. O objeto deste estudo de caso é um sistema embarcado com características de tempo real. Por isso, irão existir tarefas que devem ser executadas concorrentemente. Estas tarefas são representadas por objetos ativos (GOMAA, 2000).

6.3.2.3 Mapeamento das Dependências na Matriz de Projeto

De acordo com a abordagem proposta, a próxima atividade corresponde a mapear as dependências entre os subcasos de uso independentes de características técnicas e as colaborações respectivas na matriz de projeto. As colaborações representam interações entre objetos (instâncias) para realizar um requisito funcional, neste caso um subcaso de uso. O fato de uma colaboração usar uma instância que já participa em outra colaboração cria uma dependência funcional entre os dois subcasos de uso correspondentes.

FRs \ DPs		DP1 - Adquirir informações do equipamento	DP2		DP3 - Registrar eventos	DP4		DP5		DP6 - Reiniciar equipamento
			DP2.1 - Mostrar dados de operação	DP2.2 - Selecionar visualização de dados		DP4.1 -Mostrar eventos	DP4.2 - Selecionar visualização de eventos	DP5.1 - Alterar configurações	DP5.2 - Selecionar configuração	
FR1 - Adquirir informações do equipamento		X								
FR2	FR2.1 - Mostrar dados de operação	X	X							
	FR2.2 - Selecionar visualização de dados		X	X						
FR3 - Registrar eventos		X			X					
FR4	FR4.1 - Visualizar eventos		X		X	X				
	FR4.2 - Selecionar visualização de eventos		X		X	X	X			
FR5	FR5.1 - Alterar configurações		X		X	X		X		
	FR5.2 - Selecionar configuração		X		X			X	X	
FR6 - Reiniciar equipamento			X					X		X

Figura 47 - Matriz de projeto do 2o. nível de decomposição

Por exemplo, a colaboração “DP2.1 - Mostrar dados de operação” usa a instância “leSerial” que já participa da colaboração “DP1 - Adquirir informações de equipamento”. Isto significa que a realização do subcaso “FR2.1 - Mostrar dados de operação” depende da

colaboração “DP2.1 - Mostrar dados de operação”, mas também da colaboração “DP1 - Adquirir informações de equipamento”. Esta dependência funcional extra é representada na matriz de projeto com um “X” na célula “ FR2.1 x DP1”, como ilustrado na matriz de projeto da Figura 47.

A matriz de projeto para a solução correspondente a esta segunda etapa está ilustrada na Figura 47. A matriz de projeto para esta solução é uma matriz semi-acoplada. Isto indica que esta solução é uma solução aceitável para o projeto.

6.3.2.4 Solução de Projeto Alternativa

Nesta segunda etapa, foi identificada uma solução alternativa para a decomposição dos casos de uso, ilustrados na Figura 42, em subcasos de uso independentes de características técnicas. Esta solução alternativa originou-se de uma forma diferente de se particionar a funcionalidade de um caso de uso.

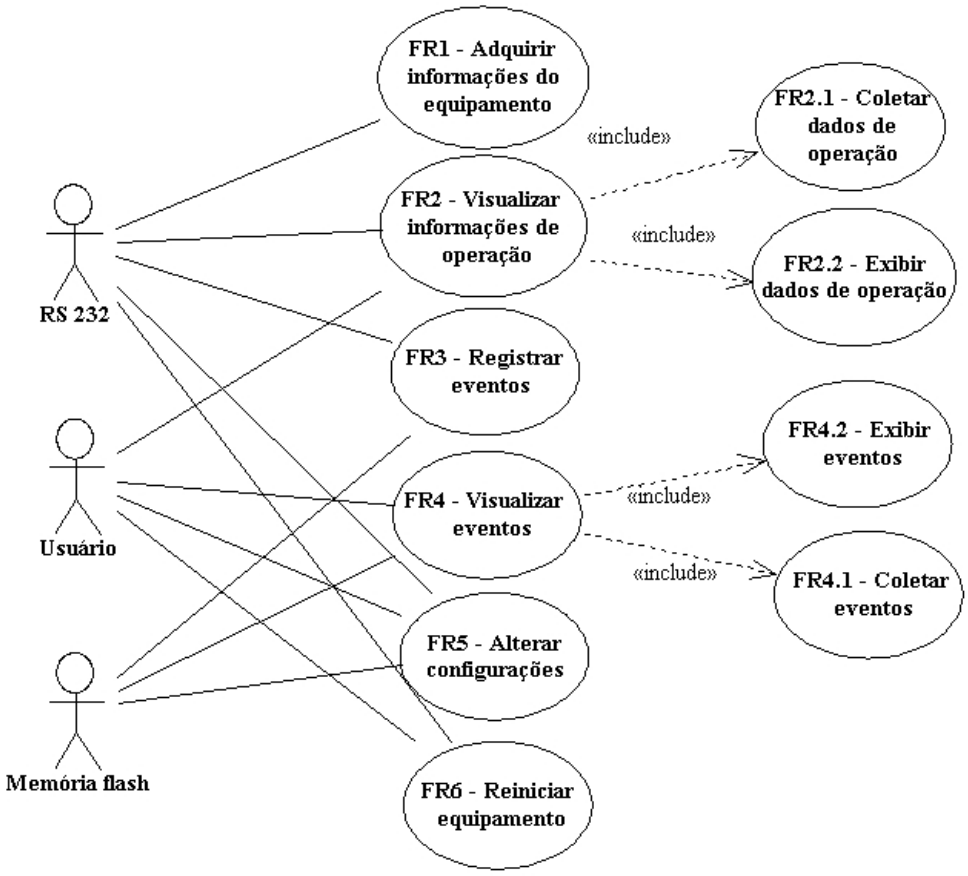


Figura 48 - Subcasos de uso para o 2o. nível da solução alternativa

Na primeira solução, a decomposição dos casos de uso foi feita em função de seus

fluxos de eventos básicos e alternativos. Nesta solução, a funcionalidade do caso de uso foi decomposta em suas partes principais. Um diagrama com os subcasos de uso independentes para esta solução alternativa é apresentado na Figura 48.

Os casos de uso “FR1 - Adquirir informações do equipamento”, “FR3 - Registrar eventos”, “FR5 - Alterar configurações” e “FR6 - Reiniciar equipamento” não foram decompostos nesta solução alternativa. O caso de uso foi decomposto nos subcasos de uso independentes de características técnicas: “FR2.1 - Coletar dados de operação” e “FR2.2 - Exibir dados de operação”. O subcaso “FR2.1 - Coletar dados de operação” captura os dados de operação do equipamento deixados disponíveis no sistema e o subcaso “FR2.2 - Exibir dados de operação”, exibe estes dados para o usuário. O caso de uso “FR4 - Visualizar eventos” foi decomposto nos subcasos de uso independentes: “FR4.1 - Coletar eventos” e “FR4.2 - Exibir eventos”. O subcaso “FR4.1 - Coletar eventos” captura os evento gerados pelo equipamento deixados disponíveis no sistema e o subcaso “FR4.2 - Selecionar visualização de eventos”, exibe estes eventos. Os subcasos de uso “FR2.1 - Coletar dados de operação” e “FR4.1 - Coletar eventos” são parecidos, pois ambos coletam dados (operação e eventos) que estão no sistema. Deixar estes subcasos separados e independentes traz vantagens à manutenibilidade do sistema como evitar a propagação de erros e diminuir o efeito de mudanças nos requisitos.

Para esta solução alternativa foram criadas colaborações e suas instâncias que estão ilustradas na Figura 49. As colaborações “DP2.1 - Coletar dados de operação”, “DP2.2 - Exibir dados de operação”, “DP4.1 - Coletar eventos” e “DP4.2 - Exibir eventos” foram criadas para satisfazer os subcasos de uso independentes de características técnicas “FR2.1 - Coletar dados de operação”, “FR2.2 - Exibir dados de operação”, “FR4.1 - Coletar eventos” e “FR4.2 - Exibir eventos”, respectivamente. Para as colaborações “DP1 - Adquirir informações de equipamento”, “DP3 - Registrar eventos”, “DP5 - Alterar configurações” e “DP6 - Reiniciar equipamento”, foram identificadas as instâncias participantes. As relações entre os subcasos de uso independentes e as colaborações correspondentes para esta solução alternativa são mapeadas na matriz de projeto. Esta matriz de projeto está representada na Figura 50. Tal como a matriz da solução original, esta é uma matriz semi-acoplada, o que indica que esta solução de projeto é uma solução aceitável.

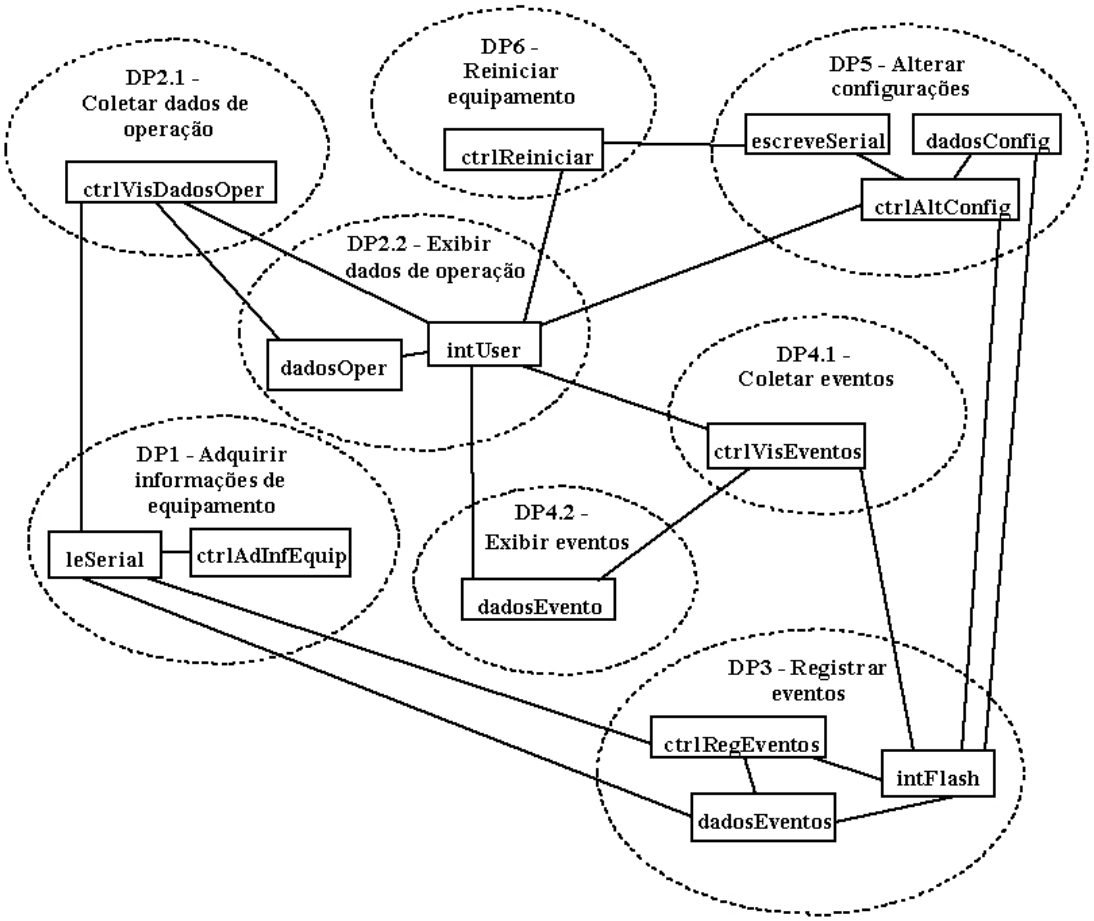


Figura 49 - Colaborações e instâncias para a solução alternativa

FRs \ DPs		DP1 - Adquirir informações do equipamento	DP2		DP3 - Registrar eventos	DP4		DP5 - Alterar configurações	DP6 Reiniciar equipamento
			DP2.1 - Coletar dados de operação	DP2.2 - Exibir dados de operação		DP4.1 - Coletar eventos	DP4.2 - Exibir eventos		
FR1 - Adquirir informações do equipamento		X							
FR2	FR2.1 - Coletar dados de operação	X	X						
	FR2.2 - Exibir dados de operação		X	X					
FR3 - Registrar eventos		X			X				
FR4	FR4.1 - Coletar eventos				X	X			
	FR4.2 - Exibir eventos			X		X	X		
FR5 - Alterar configurações				X	X			X	
FR6 - Reiniciar equipamento				X				X	X

Figura 50 - Matriz de projeto para o 2o. nível da solução alternativa

6.3.2.5 Aplicação do Axioma da Independência na Escolha da Melhor Solução

Um projeto semi-acoplado pode ser considerado satisfatório. No caso de haver duas soluções satisfatórias, existe um critério para auxiliar na escolha da melhor solução de projeto. O Axioma da independência pode ser enunciado em termos da independência funcional como: "...de duas soluções de projeto aceitáveis, a solução com maior independência funcional e superior" (SUH, 1990). A avaliação de qual solução de projeto é melhor pode ser feita com a aplicação do cálculo da reangularidade das soluções possíveis.

As duas soluções de projeto satisfazem o Axioma 1. Ambas são semi-acopladas, o que significa que elas são soluções de projeto satisfatórias. Foi calculada a reangularidade das duas matrizes de projeto, usando-se a Equação (9). O valor obtido para a reangularidade da primeira solução de projeto foi igual a 0,08156427, enquanto que o valor obtido para a solução alternativa foi igual a 0,15932426. Isto significa que o valor da reangularidade foi melhor para a solução alternativa. Um valor maior para a reangularidade significa, de acordo com o Axioma 1, uma solução de projeto menos acoplada e portanto, melhor. A partir desta etapa, a solução alternativa será desenvolvida e a solução original será descartada.

Do ponto de vista de projeto de *software*, o valor menor para a reangularidade da matriz da primeira solução, mostrada na Figura 47, se deve principalmente ao fato que vários subcasos dependem das colaborações "DP2.1 - Mostrar dados de operação" e "DP3 - Registrar Eventos", mostradas na Figura 49. Esses subcasos usam as instâncias "intUser" e "intFlash", definidas nessas colaborações. Este fato cria dependências entre os subcasos e as colaborações "DP2.1 - Mostrar dados de operação" e "DP3 - Registrar Eventos", tornando a primeira solução de projeto mais acoplada que a solução alternativa.

6.3.3 Etapa de Modelagem de Subcasos de Uso Dependentes de Características Técnicas

No terceiro nível de decomposição, os subcasos de uso independentes de características técnicas são decompostos em subcasos de uso dependentes de características técnicas. Cada subcaso de uso dependente de características técnicas representa uma funcionalidade do sistema que está intimamente relacionada com características tecnológicas adotadas na solução.

O subcaso de uso independente de características técnicas "FR2.2 - Exibir dados de operação" foi decomposto nos subcasos de uso dependentes de características técnicas "FR2.2.1 - Exibir dados de operação" e "FR2.2.2 - Selecionar dados de operação". A

decomposição do subcaso de uso independente “FR4.2 - Exibir eventos” resultou nos subcasos de uso dependentes “FR4.2.1 - Exibir eventos” e “FR4.2.2 - Selecionar eventos”. De forma análoga, o subcaso “FR5 - Alterar configurações” foi decomposto nos subcasos de uso dependentes “FR5.1 - Alterar configurações” e “FR5.2 - Selecionar configurações”.

Os subcasos de uso independentes de características técnicas “FR1 - Adquirir informações de equipamento”, “FR2.1 - Coletar dados de operação”, “FR3 - Registrar eventos”, “FR4.1 - Coletar eventos” e “FR6 - Reiniciar equipamento”, não foram decompostos.

Para cada subcaso de uso dependente, uma colaboração entre objetos é criada representando um parâmetro de projeto. As colaborações “DP2.2.1 - Exibir dados de operação”, “DP2.2.2 - Selecionar dados de operação”, “DP4.2.1 - Exibir eventos”, “DP4.2.2 - Selecionar eventos”, “DP5.1 - Alterar configurações” e “DP5.2 - Selecionar configurações” foram criadas para satisfazer os subcasos de uso dependentes de características técnicas “FR2.2.1 - Exibir dados de operação”, “FR2.2.2 - Selecionar dados de operação”, “FR4.2.1 - Exibir eventos”, “FR4.2.2 - Selecionar eventos”, “FR5.1 - Alterar configurações” e “FR5.2 - Selecionar configurações”, respectivamente.

FRs \ DPs			DP1 - Adquirir informações do equipamento	DP2			DP3 - Registrar eventos	DP4			DP5		DP6 Reiniciar equipamento
				DP2.1 - Coletar dados de operação	DP2.2			DP4.1 - Coletar eventos	DP4.2		DP5.1 - Alterar configurações	FR5.2 - Selecionar configuração	
					DP2.2.1 - Exibir dados de operação	DP2.2.2 - Selecionar dados de operação			DP4.2.1 - Exibir eventos	DP4.2.2 - Selecionar eventos			
FR1 - Adquirir informações do equipamento			X										
FR2	FR2.1 - Coletar dados de operação		X	X									
	FR2.2	FR2.2.1 - Exibir dados de operação		X	X								
		FR2.2.2 - Selecionar dados de operação		X	X	X							
FR3 - Registrar eventos			X				X						
FR4	FR4.1 - Coletar eventos						X	X					
	FR4.2	FR4.2.1 - Exibir eventos			X			X	X				
		FR4.2.2 - Selecionar eventos			X			X	X	X			
FR5	FR5.1 - Alterar configurações				X		X				X		
	FR5.2 - Selecionar configuração				X		X				X	X	
FR6 - Reiniciar equipamento					X						X		X

Figura 51 - Matriz de projeto do 3o nível

As relações entre os subcasos de uso dependentes de características técnicas e as colaborações correspondentes são mapeadas na matriz de projeto. A matriz da solução de projeto para este nível de decomposição é apresentada na Figura 51.

Uma solução alternativa para este nível de decomposição foi obtida e é apresentada na Figura 52. Nesta solução alternativa, para satisfazer os subcasos de uso dependentes "FR2.2.1 - Exibir dados de operação" e "FR4.2.1 - Exibir eventos" foi criada uma colaboração mais genérica chamada "DP2.2.1 - Exibir dados" e para satisfazer os subcasos "FR2.2.2 - Selecionar dados de operação" e "FR4.2.2 - Selecionar eventos" foi criada uma outra colaboração, também genérica, chamada "DP2.2.2 - Selecionar dados". A colaboração "DP2.2.1 - Exibir dados" foi criada para mostrar tanto informações de operação quanto eventos e a colaboração "DP2.2.2 - Selecionar dados" foi criada para selecionar tanto informações de operação, eventos ou configurações.

FRs \ DPs			DP1 - Adquirir informações do equipamento	DP2			DP3 - Registrar eventos	DP4.1.1 - Coletar eventos	DP5		DP6 Reiniciar equipamento
				DP2.1.1 - Coletar dados de operação	DP2.2				DP5.1 - Alterar configurações	FR5.2 - Selecionar configuração	
					DP2.2.1 - Exibir dados	DP2.2.2 - Selecionar dados					
FR1 - Adquirir informações do equipamento			X								
FR2	FR2.1 - Coletar dados de operação		X	X							
FR2	FR2.2	FR2.2.1 - Exibir dados de operação		X	X						
FR2	FR2.2	FR2.2.2 - Selecionar dados de operação		X	X	X					
FR3 - Registrar eventos			X				X				
FR4	FR4.1 - Coletar eventos						X	X			
FR4	FR4.2	FR4.2.1 - Exibir eventos			X			X			
FR4	FR4.2	FR4.2.2 - Selecionar eventos			X	X		X			
FR5	FR5.1 - Alterar configurações				X		X		X		
FR5	FR5.2 - Selecionar configuração				X	X	X		X	X	
FR6 - Reiniciar equipamento					X				X		X

Figura 52 - Matriz de projeto do 3o nível para solução alternativa

Esta solução alternativa não se mostra uma solução de projeto adequada. Em primeiro lugar a matriz de projeto não é quadrada, possuindo mais requisitos funcionais (FRs)

que parâmetros de projeto (DPs), o que pelo Teorema 1 (ver Anexo 1), constitui um projeto acoplado.

Neste terceiro nível de decomposição, a aplicação, principalmente do Axioma da Independência e teoremas relacionados, tem o objetivo de evitar a criação de parâmetros de projeto (DPs) que venham a produzir um acoplamento e também fazer com que a matriz de projeto permaneça quadrada. Com isso a matriz resultante atenderia o Axioma da Independência, mesmo sendo semi-acoplada, mantendo a solução de projeto aceitável.

6.3.4 Etapa de Modelagem de Serviços Técnicos

Os subcasos de uso dependentes de características técnicas são decompostos em serviços técnicos na quarta etapa da abordagem. Nesta etapa, os requisitos funcionais (serviços técnicos) representam serviços requeridos de um objeto ou classe por outro objeto ou classe e estão intimamente relacionados com a solução do problema. Os parâmetros de projeto para esta etapa são representados por objetos. Para cada serviço técnico, um objeto é criado e esta relação é mapeada na matriz de projeto. Também são mapeadas nesta matriz, relações entre cada serviço técnico e objetos que foram criados para realizar outros serviços técnicos. Portanto, cada linha da matriz de projeto representa um serviço técnico e os objetos que tomam parte na sua realização. A matriz de projeto para esta etapa do estudo de caso é mostrada na Figura 53.

O *software* desenvolvido para este estudo de caso possui restrições de tempo real, que requerem que algumas atividades sejam executadas de forma concorrente. Estas atividades concorrentes, chamadas de tarefas, são uma característica comum em sistemas embarcados e são implementadas em alguns sistemas operacionais como o *X Kernel*, através de *threads*. No projeto de *software* orientado a objetos, as *threads* são representadas por objetos ativos (GOMAA, 2000). Existem critérios para estruturar tarefas (objetos ativos), definidos em (GOMAA, 2000), como a estruturação das tarefas de interface com dispositivos e a estruturação das tarefas internas. Para a solução de projeto para este estudo de caso foram criados os seguintes objetos ativos para interfaceamento com os dispositivos: um objeto da classe “CIntRS232” responsável por ler dados de operação do equipamento através da porta serial, um outro objeto da classe “CIntRS232” responsável por enviar comandos para o equipamento através da porta serial, um objeto da classe “CIntDisplay” responsável por atualizar o *display* LCD, um objeto da classe “CIntKeyboard” responsável por receber entradas do teclado e um objeto da classe “CIntDigInput” responsável por receber comandos

do dispositivo conectado à entrada digital da placa. Além disso, foi criado um objeto ativo para tarefas internas, da classe "CBuffer" responsável por organizar o *buffer* interno com os dados de operação do equipamento conectado ao sistema.

FRs \ DPs	DP1.1 - Objeto da classe CIntRS232	DP1.2 - Objeto da classe CBuffer	DP1.3 - Objeto da classe CCtdReadSerialPort	DP2.1.1 - Objeto da classe COpData	DP2.1.1.1 - Objeto da classe CIntDisplay	DP2.2.1.2 - Variável errorMsg	DP2.2.2.1 - Objeto da classe CIntKeyboard	DP2.1.2 - Objeto da classe CCtdVizOpData	DP3.1 - Objeto da classe COpEvent	DP3.2 - Objeto da classe CIntFlash	DP3.3 - Objeto da classe CCtdRegEvents	DP4.1.1 - Objeto da classe COpEvent	DP4.2.1.2 - Objeto da classe COpEvent	DP4.2.2.1 - Variável optionEvent	DP4.1.2 - Objeto da classe CCtdVizEvents	DP5.1.2 - Objeto da classe Cconfiguration	DP5.1.3 - Objeto da classe CIntRS232	DP5.2.1 - Variável optionConfiguration	DP5.1.1 - Objeto da classe CCtdChangeConf	DP6.1 - Objeto da classe CIntDigInput	DP6.2 - Objeto da classe CCtdReset
FR1.1 - Ler porta	X																				
FR1.2 - Receber informações da porta	X	X																			
FR1.3 - Controlar leitura da porta	X	X	X																		
FR2.1.1 - Receber de dados de operação				X																	
FR2.2.1.1 - Mostrar dados de operação				X	X																
FR2.2.1.2 - Mostrar erro					X	X															
FR2.2.2.1 - Selecionar dados de operação					X		X														
FR2.1.2 - Controlar exibição de dados de operação		X		X	X	X	X	X													
FR3.1 - Receber eventos									X												
FR3.2 - Gravar eventos									X	X											
FR3.3 - Controlar o registro de eventos		X							X	X	X										
FR4.1.1 - Coletar eventos										X		X									
FR4.2.1.1 - Mostrar eventos					X					X		X	X								
FR4.2.2.1 - Selecionar eventos					X		X			X			X	X							
FR4.1.2 - Controlar exibição de eventos					X					X		X	X	X	X						
FR5.1.2 - Receber nova configuração					X					X						X					
FR5.1.3 - Iniciar nova configuração																X	X				
FR5.2.1 - Selecionar configuração					X					X						X		X			
FR5.1.1 - Controlar alteração de configuração					X											X	X	X	X		
FR6.1 - Ler entrada digital																				X	
FR6.2 - Controlar reinício do equipamento					X												X			X	X

Figura 53 - Matriz de projeto da 4a etapa

A definição e a representação das interações entre objetos é um artefato importante em uma solução de projeto de *software* orientado a objetos. A identificação destas interações, normalmente, é realizada através da construção dos diagramas de seqüência ou diagramas de comunicação (OBJECT MANAGEMENT GROUP, 2005). Para realizar o caso de uso "FR1 - Adquirir informações do equipamento" foram identificados os serviços técnicos, "FR1.1 - Ler porta", "FR1.2 - Receber informações da porta" e "FR1.3 - Controlar leitura da porta". Para

satisfazer estes serviços técnicos, foram criadas as interações entre os objetos das classes "CCtrlReadSerialPort", "IntRS232" e "CBuffer" através da construção do diagrama de seqüência ilustrado na Figura 54. A construção deste diagrama serviu de base para estabelecer as relações entre os serviços técnicos (FRs) "FR1.1 - Ler porta", "FR1.2 - Receber informações da porta" e "FR1.3 - Controlar leitura da porta" e os objetos (DPs) "CCtrlReadSerialPort", "IntRS232" e "CBuffer", que os satisfazem. Estas relações foram mapeadas matriz de projeto da Figura 53.

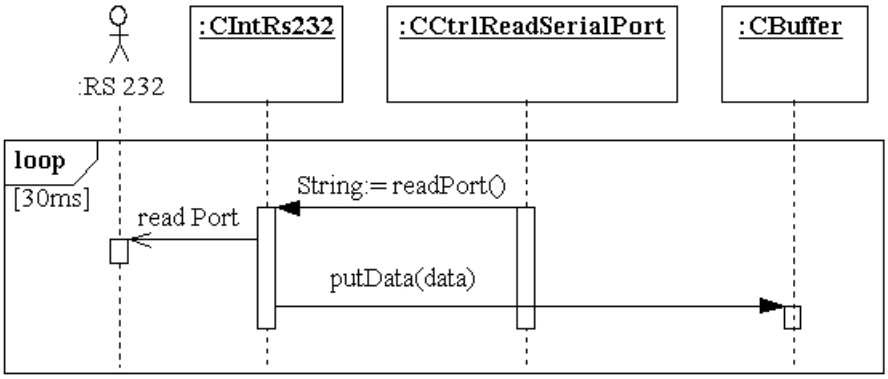


Figura 54 - Diagrama de seqüência para "FR1 - Adquirir informações do equipamento"

A matriz de projeto é uma ferramenta útil para identificar, criar e selecionar os parâmetros de projeto adequados e em garantir a independência funcional dos requisitos. Além desta utilidade, a matriz de projeto pode ajudar o projetista na tarefa de criar diagramas UML como diagramas de seqüência.

A escolha da solução de projeto mais adequada nesta etapa pode ser realizada através da aplicação do Axioma 1, verificando se a matriz de projeto referente a solução é quadrada, e neste caso, se é acoplada ou não. As soluções de projeto acopladas são consideradas não adequadas pelo Axioma 1. Soluções de projeto com matrizes de projeto não quadradas são consideradas não adequadas, pois, pelos Teoremas 1 e 4 (ver Anexo 1) elas ou são acopladas ou redundantes (SUH, 1990).

Na quarta etapa da abordagem, as matrizes de projeto normalmente possuem um tamanho grande pois representam o projeto em um nível de abstração mais próximo ao código fonte. Além disso, neste nível, as matrizes de projeto para diferentes soluções possuem grandes diferenças de tamanho. Por estes motivos quando se tem várias soluções de projeto aceitáveis, a aplicação da reangularidade (ver Equação (9)) é prejudicada pois esta medida de independência funcional é bastante sensível no caso de matrizes muito grandes ou de

tamanhos muito diferentes. Portanto, neste nível de decomposição, para escolher a solução de projeto mais adequada entre soluções possíveis (não acopladas) é mais interessante a aplicação do Axioma 2, através do cálculo do conteúdo de informação de cada solução de projeto. A solução que apresentar o menor conteúdo de informação é a mais adequada.

Para este estudo de caso, o conteúdo de informação foi calculado para exemplificar e avaliar a aplicação do Axioma 2. Este conteúdo de informação foi calculado para a solução de projeto da quarta etapa da abordagem. Para calcular o conteúdo de informação das classes do sistema foi usado o arcabouço apresentado no Capítulo 5, baseado nas métricas de complexidade de *software* orientadas a objetos propostas por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994). Foram estimados os valores de projeto e calculados os valores para cada métrica, para cada classe do sistema. Estes valores são apresentados na Tabela 7. As classes criadas para o sistema deste estudo de caso são mostradas na Figura 55.

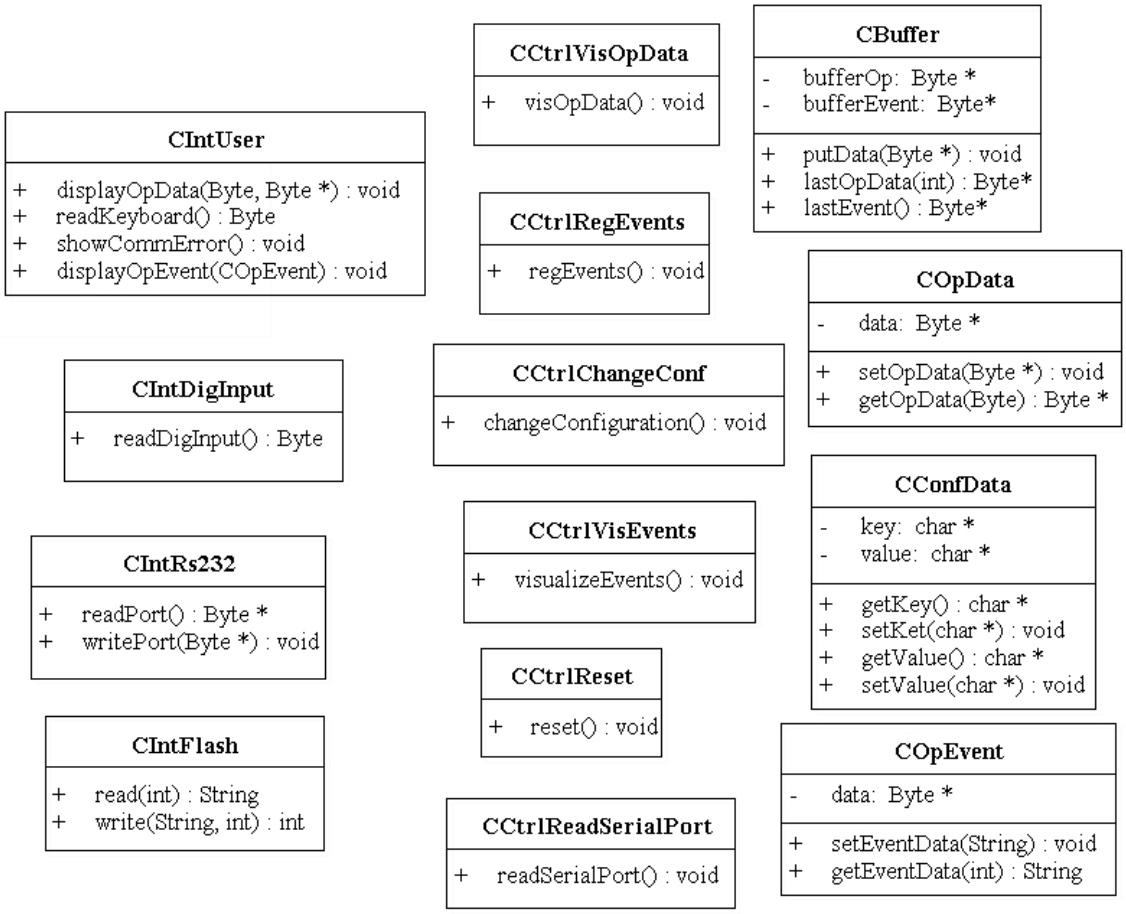


Figura 55 - Classes do sistema objeto deste estudo de caso

Tabela 7 - Valores das métricas CK para as classes do sistema

CLASSE	WMC	EWMC	DIT	EDIT	NOC	ENOC	CBO	ECBO	RFC	ERFC	LCOM	ELCOM
CCtrlVisOpData	1	1	0	0	0	0	2	2	1	1	0	0
COpData	2	2	0	0	0	0	0	0	2	2	0	0
CBuffer	3	2	0	0	0	0	0	0	3	3	0	0
CIntDisplay	4	2	0	0	0	0	3	3	4	2	0	0
CIntKeyboard	1	1	0	0	0	0	2	2	1	1	0	0
CCtrlVisEvents	1	1	0	0	0	0	3	3	1	1	0	0
CIntFlash	2	2	0	0	0	0	0	0	2	2	0	0
COpEvent	2	2	0	0	0	0	0	0	2	2	0	0
CCtrlChangeConf	1	1	0	0	0	0	4	4	1	1	0	0
CConfiguration	4	2	0	0	0	0	0	0	4	4	0	0
CCtrlReadSerialPort	1	1	0	0	0	0	3	3	1	1	0	0
CIntRS232	2	2	0	0	0	0	1	1	2	2	0	0
CCtrlRegEvent	1	1	0	0	0	0	3	3	1	1	0	0
CCtrlReset	1	1	0	0	0	0	2	2	1	1	0	0
CIntDigInput	1	1	0	0	0	0	0	0	1	1	0	0

O valor do conteúdo de informação do sistema como um todo é a soma dos conteúdos de informação de cada classe. Para cada classe, o conteúdo de informação é a soma dos conteúdos relativos a cada métrica e é calculado pelas equações (19) ou (20). Os conteúdos de informação relativos à cada métrica são calculados pelas equações (21), (22), (23), (24), (25) e (26). O conteúdo de informação para a solução de projeto cuja matriz de projeto é apresentada na Figura 53 e suas classes na Figura 55 foi calculado e resultou *I=1,556*.

Uma solução alternativa para este mesmo projeto foi criada substituindo-se as classes “COpData”, “COpEvent” e “CBuffer” pelas classes “COpDataAlt” e “COpEventAlt”, mostradas na Figura 56. As classes “COpDataAlt” e “COpEventAlt” interagem com o restante das classes do sistema de maneira similar às classes “COpData”, “COpEvent” e “CBuffer”.

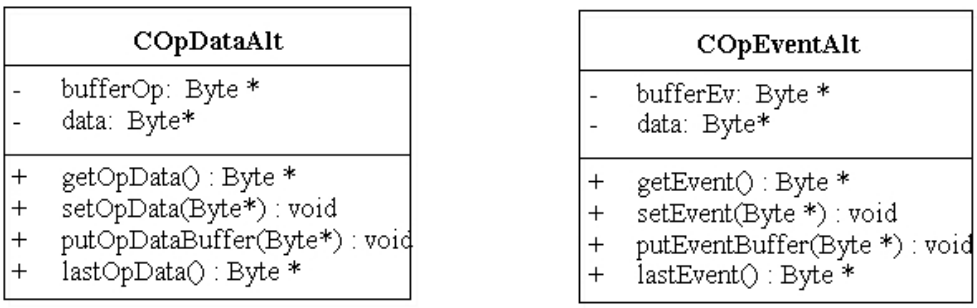


Figura 56 - Novas classes para a solução alternativa

Tabela 8 - Valores das métricas CK para as classes da solução alternativa

CLASSE	WMC	EWMC	DIT	EDIT	NOC	ENOC	CBO	ECBO	RFC	ERFC	LCOM	ELCOM
COpDataAlt	4	2	0	0	0	0	0	0	4	2	2	1
COpEventAlt	4	2	0	0	0	0	0	0	4	2	2	1

Os valores de cada uma das métricas CK, tanto o valor obtido como o estimado estão apresentados na Tabela 8. A criação destas novas classes criou uma nova solução de projeto. Apesar de diferente, a matriz de projeto desta nova solução mantém compatibilidade com a matriz de projeto do terceiro nível de decomposição. O conteúdo de informação para a solução de projeto alternativa, com as classes "COpDataAlt"e "COpEventAlt", foi calculado e resultou $I_{nova}= 3,006$. Este resultado indica que a solução original possui um conteúdo de informação menor, que a solução alternativa. Pelo Axioma da Informação a solução original pode ser considerada uma solução melhor.

6.4 Conclusões

Este Capítulo apresentou um estudo de caso onde foi criada uma solução de projeto para um sistema embarcado com restrições de tempo real. Foram apresentados os objetivos deste estudo de caso, a descrição do sistema alvo e as características do ambiente de *software* e *hardware* usados. O estudo de caso foi realizado seguindo a abordagem proposta nesta tese. A aplicação da abordagem seguiu a etapas e níveis de decomposição funcional definidos.

Na segunda etapa da abordagem duas soluções de projeto foram criadas e avaliadas de acordo com o Axioma da Independência. Uma das soluções foi considerada melhor de acordo com a abordagem proposta. Na terceira etapa, a aplicação da abordagem, principalmente do Axioma da Independência, auxilia na escolha de soluções de projeto cuja matriz de projeto é coerente com as de níveis de decomposição anteriores e soluções com matriz de projeto quadrada. As soluções de projeto com matrizes não quadradas são consideradas acopladas ou redundantes e, portanto, não adequadas.

O estudo de caso, também, ilustrou que na quarta etapa da abordagem, a criação da matriz de projeto está relacionada com a criação de diagramas UML para o projeto, tais como os diagramas de seqüência. Além disso, foi usado o cálculo do conteúdo de informação para a aplicação do Axioma da Informação como critério para a escolha da melhor solução projeto. O conteúdo de informação foi calculado usando-se o arcabouço proposto no Capítulo 5, com

base em métricas de complexidade de *software* orientado a objetos.

A aplicação da abordagem, neste estudo de caso, trouxe vantagens ao desenvolvimento da solução de projeto. Esta aplicação possibilitou manter a qualidade da solução de projeto, ao longo de todo o processo. O Axioma 1 se mostrou um critério importante na garantia da qualidade da solução de projeto, nas primeiras três etapas, enquanto que o Axioma 2 se mostrou um critério importante na quarta etapa. Outra vantagem é que a manutenção da qualidade da solução de projeto ao longo do processo tornou mais difícil o aparecimento de soluções ruins nos níveis de decomposição mais baixos. Por exemplo, no quarto nível, o aparecimento de uma solução ruim para o projeto, que fosse compatível com o terceiro nível, foi dificultado.

O estudo de caso explorou e exemplificou os principais aspectos da abordagem proposta. Outros estudos são necessários para que se possa avaliar os diferentes cenários relacionados com o desenvolvimento de *software*.

7 Conclusões e Trabalhos Futuros

Esta tese apresenta uma abordagem de projeto de *software* orientado a objetos, baseada na Teoria de Projeto Axiomático, que pode ser aplicada em conjunto com um processo de desenvolvimento de *software* como o Processo Unificado. A abordagem proposta nesta tese provê critérios para auxiliar o projetista na tomada de decisões de projeto. Estes critérios são baseados em princípios gerais de bons projetos que formam o arcabouço da Teoria de Projeto Axiomático.

A abordagem proposta nesta tese traz vantagens ao projeto de *software* orientado a objetos, como:

- Auxiliar na escolha da melhor solução de projeto (colaborações, objetos e classes) entre possíveis soluções, para satisfazer o conjunto de requisitos funcionais.
- Auxiliar a manter o conjunto de requisitos funcionais do projeto adequado ao nível de abstração desejado, durante o projeto.
- Auxiliar a manter alta a probabilidade de que o sistema produzido satisfaça os requisitos funcionais.
- Permitir que o nível de abstração da solução de projeto se mantenha adequado às necessidades do projetista.
- Auxiliar a manter a rastreabilidade entre os artefatos que implementam o sistema e os requisitos funcionais de alto nível.

A finalidade básica de um projeto é traduzir as necessidades percebidas em termos de soluções tecnológicas que satisfaçam essas necessidades, de forma a obter um produto com qualidade. A identificação dos requisitos, a síntese de soluções e as decisões de projeto adequadas, como a escolha da melhor entre as possíveis soluções tecnológicas criadas, contribuem para a realização de um bom projeto. A qualidade do projeto é fundamental para a realização de um produto com qualidade, seja ele uma edificação, um automóvel, um

computador ou um sistema de *software*.

Para poder aplicar a Teoria de Projeto Axiomático ao projeto de *software* orientado a objetos, a abordagem proposta nesta tese define analogias entre os conceitos do Projeto Axiomático e os de projeto de *software* orientado a objetos, de forma que os axiomas, teoremas e corolários possam ser aplicados ao projeto de *software*. Neste sentido, foi estabelecido um modelo de hierarquia, baseada em casos de uso, para a decomposição dos requisitos funcionais e dos parâmetros de projeto. Foram estabelecidas relações entre o conceito de requisito funcional e casos de uso, além de terem sido criados novos conceitos para representar os requisitos funcionais em níveis de decomposição mais baixos, como: subcasos de uso independentes de tecnologia, subcasos de uso dependentes de tecnologia e serviços técnicos. O conceito de parâmetro de projeto foi relacionado com colaborações, objetos e classes, de acordo com o nível de decomposição.

Foi criado um processo de *ziguezagueamento* que estende o processo de *ziguezagueamento* original e que pode ser aplicado a um ciclo de vida iterativo e incremental. Este processo é feito em 4 níveis de decomposição, permitindo várias decomposições a cada nível e permitindo que a decomposição possa ser feita em profundidade desde que a matriz de projeto seja consistente com a dos níveis anteriores. A abordagem proposta possui 4 etapas em correspondência com os níveis de decomposição propostos e pode ser inserida nas atividades de requisitos, análise e projeto do Processo Unificado sendo aplicada entre as fases de concepção, elaboração e o início da fase de construção.

A matriz de projeto é uma importante ferramenta para a aplicação do Axioma da Independência. A abordagem de projeto proposta estabelece formas de montar a matriz de projeto para *software* orientado a objetos, de forma que a aplicação do Axioma da Independência ajude a garantir a qualidade da solução de projeto. As relações entre requisitos funcionais e colaborações, objetos e classes, representadas na matriz auxiliam o projetista na elaboração dos diagramas de projeto da UML usados no Processo Unificado. A matriz de projeto em conjunto com a hierarquia funcional facilita a rastreabilidade dos parâmetros de projeto com relação aos requisitos funcionais, o que permite relacionar um componente do *software* ao requisito funcional de alto nível que ele satisfaz.

Também foi estabelecido um arcabouço para o cálculo do conteúdo de informação para projeto de *software* orientado a objetos que permite a aplicação do Axioma da Informação. Este arcabouço é baseado na probabilidade de uma organização desenvolver o sistema que satisfaça seus requisitos funcionais, usando para isto métricas de complexidade de *software* bem estabelecidas na literatura como *use case points*, pontos por função e o conjunto

de métricas orientadas a objetos de Chidamber e Kemerer. Estas métricas foram adotadas porque são bem conhecidas na literatura e porque têm métodos bem estabelecidos para serem computadas, podendo inclusive ser calculadas com o auxílio de ferramentas computacionais.

Com o intuito de avaliar a abordagem proposta, foi apresentado um estudo de caso em que foi criada uma solução de projeto para um sistema embarcado com restrições de tempo. A aplicação da abordagem seguiu a etapas e níveis de decomposição funcional definidos. Este estudo de caso ilustra como a decomposição funcional e o *ziguezagueamento* podem ser aplicados a um projeto de *software* orientado a objetos para a identificação de novas soluções de projeto. Também foi ilustrado como os Axiomas da Independência e da Informação podem ser aplicados para escolher a melhor solução de projeto a cada nível de decomposição garantindo a qualidade da solução de projeto.

Além disso, o autor também observou que:

- Uma das contribuições deste trabalho foi propor uma abordagem para decomposição dos requisitos funcionais, criando uma hierarquia que relaciona os requisitos funcionais de alto nível com os de baixo nível.
- Como efeito da aplicação desta abordagem de decomposição, se obtém uma qualidade maior que as abordagens tradicionais na definição dos papéis e responsabilidades dos componentes de *software*. Isto facilita a avaliação das dependências funcionais entre estes componentes permitindo aplicar os Axiomas da Independência e da Informação ao desenvolvimento de *software*. A aplicação destes Axiomas, juntamente com os teoremas e corolários relacionados, fornece um critério preciso para avaliar o acoplamento e coesão dos módulos do *software*.
- Esta tese apresentou um novo paradigma para análise da qualidade do projeto. A qualidade do projeto pode ser analisada com a aplicação dos Axiomas, passo a passo ao longo do desenvolvimento e não como uma medida da qualidade realizada ao final, para ser usada em estatísticas estimativas futuras. Para auxiliar este processo, o uso das métricas de *software* foi proposto para orientar a tomada de decisões de projeto durante o desenvolvimento.
- Este trabalho estabelece um fundamento teórico para projeto de *software* orientado a objetos permitindo que novas teorias (axiomas, teoremas e corolários) venham a ser desenvolvidas e aplicadas.

7.1 Trabalhos Futuros

A abordagem proposta nesta tese aplica a Teoria de Projeto Axiomático à atividade de projeto de *software*. O projeto é apenas uma das atividades do desenvolvimento. Como uma das perspectivas de trabalhos futuros, considera-se que a aplicação do Projeto Axiomático ao restante do processo de desenvolvimento é uma perspectiva promissora. Uma das linhas interessantes é o projeto dos testes de *software*, pois, no Processo Unificado ele também é guiado pelos casos de uso. O gerenciamento de um projeto de *software*, de uma forma geral, pode se valer da Teoria de Projeto Axiomático na hora de projetar as atividades a serem realizadas, seus prazos e quem irá realizá-las.

Uma outra proposta promissora de trabalhos futuros é a evolução da fundamentação teórica estabelecida neste trabalho com o desenvolvimento e aplicação de novas teorias (axiomas, teoremas e corolários) representando princípios de bons projetos mais específicos para *software*.

Outra perspectiva promissora para trabalhos futuros é o desenvolvimento de uma abordagem que permita aplicar a Teoria de Projeto Axiomático no desenvolvimento de *software* orientado a aspectos. A hierarquia funcional e a aplicação dos axiomas e teoremas do Projeto Axiomático podem trazer vantagens na identificação dos interesses transversais e na criação dos aspectos.

Uma proposta de trabalhos futuros é vislumbrado no desenvolvimento de uma abordagem que aplica a Teoria de Projeto Axiomático para o projeto de *software* para aviônica, usando o arcabouço proposto nesta tese. Estes sistemas devem atender à norma DO - 178 B que exige decomposição dos requisitos em níveis de abstração e a rastreabilidade entre os requisitos do sistema, requisitos de alto nível, de baixo nível e o código fonte.

Seguindo em outra linha, o desenvolvimento de software orientado a componentes visa aumentar a reutilização de software, encapsulando o código em componentes, permitindo a utilização tanto de componentes novos como dos já existentes. O desenvolvimento de uma abordagem que permita aplicar a Teoria de Projeto Axiomático no desenvolvimento de *software* orientado a componentes é uma perspectiva promissora pois pode prover critérios para a seleção dos componentes adequados para satisfazer os requisitos funcionais do sistema a ser desenvolvido.

Referências

ABNT. *Tecnologia de informação - avaliação de produto de software - características de qualidade e diretrizes para o seu uso - NBR 13.596*. Associação Brasileira de normas Técnicas: Rio de Janeiro, Brasil, 1996

ADOLPH, S.; BRAMBLE, P.. *Patterns for effective use cases*. Reading, Massachusetts, EUA: Addison-Wesley, 2003.

AKAO, Y. (ed.). *Quality Function Deployment: Integrating customer requirements into product Design*. Cambridge, Massachusets, EUA: Productivity Press, 1990.

ANDA, B.; DREIEM, D.; SJOERG, D.; JORGENSEN, M.. Estimating software development effort based on use cases - experiences from industry. *Anais do UML 2001 - The Unified Modeling Language. 4th International Conference*. Toronto, Canadá: 2001.

ANGIULO, M.. *Design and development of personal computer software : a case study*. Seatle, EUA, 1997. Dissertação de mestrado - Department of Mechanical Engineering, University of Washington.

ARCIDIACONO, G.. Keys to success for Six Sigma. *Anais do ICAD2006 - the 4th International Conference on Axiomatic Design*. Florença, Itália: 2006.

ARCIDIACONO, G.; CITTI, P.; FONTANA, V.; MARTELLI, T.. Reliability improvement of car sliding door using axiomatic approach. *Anais do ICAD2004 - the 3rd International Conference on Axiomatic Design*. Seul, Coreia: 2004.

ASTEL, D.; MILLER, G.; NOVAK, M.. *Extreme programming: guia prático*. Rio de Janeiro, Brasil: Campus, 2002.

BECK, K; CUNNINGHAM, W.. A laboratory for teaching Object-Oriented thinking. *Anais do International Conference on Object-Oriented Programming, Systems, Languages and Applications OOPLSA89*. New Orleans, EUA: 1989.

BELADY, L.. *Prefácio de Software design: methods and Techniques (PETERS, L. J. autor)*. New Jersey, EUA: Yourdon Press, 1981.

BEZZERRA, E.. *Princípios de análise e projeto de sistemas com UML*. Rio de Janeiro, Brasil: Campus, 2002.

BIÇER, V.; TOGAY, C.; DOGRU, A. H.. Service-Oriented e-learning systems with axiomatic design. *Anais do 9th World Conference on Integrated Design and Process Technology - IDPT 2006*. San Diego, EUA: 2006.

BIEMAN, J. M.; OTT, L. M.. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, Nova York, EUA, v. SE -20, n. 8, p. 308-320, ago. 1994.

BIRTWISTLE, G. M.; DAHL, O-J.; MYHRHAUG, B.; NYGAARD, K.. *Simula Begin*. New

York, EUA: Petrocelli/Charter, 1975.

BITNER, K.; SPENCE, I.. *Use Case Modeling*. Boston, EUA: Addison-Wesley, 2003.

BOEHM, B. W.. *Software risk management*. Piscataway, NJ, EUA: IEEE Computer Society Press, 1989.

BOOCH, G.. *Object-Oriented Analysis and Design with Applications*. San Francisco, EUA: Benjamin/Cummings, 1994.

BOOCH, G.. The accidental architecture. *IEEE Software*, New York, EUA, n. 3, 2006.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I.. *The Unified Modeling Language user guide*. 2a. ed.Westford, Massachusets, EUA: Addison-Wesley, 2005.

BRAHA, D.; MAIMON, O.. *A Mathematical Theory of Design:Foundations, Algorithms, and Applications*. Heidelberg, Alemanha: Springer, 1998.

BROOKS Jr., F. P.. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, New York, EUA, v. 20, n. 4, abr. 1987.

CALDENFORS, D.. *Top-Down Reasoning Syntesis and Evaluation*. Linköping, Suécia, 1998. Tese de Doutorado- Linköping Studies in Science and Technology, Linköping University.

CAPPETTI, N.; NADEO, A; PELLEGRINO, A.. Design decoupling method based on para-complete logics. *Anais do ICAD2004 - the 3rd International Conference on Axiomatic Design*. Seul, Coreia: 2004.

CARMEL, E.. *Global software teams - Collaborating across borders and time-zones*. New Jersey, EUA: Prentice-Hall, 1999.

CARVALHO, M. A.; BLACK, N.. Uso dos conceitos fundamentais da TRIZ e do método dos princípios inventivos no desenvolvimento de produtos. *Anais do 3o. Congresso Brasileiro de Gestão de Desenvolvimento de Produto*. Florianópolis, Brasil: 2001.

CASANOVA, M. A.; GIORNO, F. A; FURTADO, A. L.. *Programação em Lógica e a linguagem Prolog*. Rio de Janeiro, Brasil: Edgard Blücher, 1987.

CAVANO, J. P.; McCALL, J. A.. A framework for the measurement of software quality. *ACM SIGMETRICS Performance Evaluation Review*, Portland, EUA, v. 7, n. 3-4, p. 133-139, nov. 1978.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, New York, EUA, v. SE-20, n. 6, p. 476-493, jun. 1994.

CLAPIS, P. J.; HINTERSTEINER, J. D.. Enhancing Object-oriented software development through axiomatic design. *Anais do ICAD2000 - the 1st International Conference on Axiomatic Design*. Cambridge, EUA: 2000.

CLAUSING, D. P.. *Total quality development*. New York, EUA: ASME press, 1994.

COLEMAN, D.; ARNOLD, P.;BODOFF, S.; DOLLIN, C.; GILCHRIST, H.; HAYES, F..

Desenvolvimento Orientado a Objetos: O método Fusion. Rio de Janeiro, Brasil: Campus, 1996.

COX, B.. *Object-oriented programming: an evolutionary approach*. New Jersey, EUA: Addison-Wesley, 1991.

DEO, H. V.; SUH, N. P.. Axiomatic design of customizable automotive suspension. *Anais do ICAD2004 - the 3rd International Conference on Axiomatic Design*. Seul, Coreia: 2004.

DHAMA, H.. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, Orlando, EUA, v. 29, n. 4, abr. 1995.

DO, S. H.. Software product lifecycle management using axiomatic approach. *Anais do ICAD2004 - the 3rd International Conference on Axiomatic Design*. Seul, Coreia: 2004.

DO, S. H.; PARK, G. J.. Application of Design Axioms for Glass-Bulb design and Software Development for Design Automation. *Anais do CIRP Workshop on Design and Inteligent Manufacturing Systems*. Tokyo, Japão: 1996.

DO, S. H.; SUH, N. P.. Systematic O O programming with axiomatic design. *IEEE Computer*, New York, EUA, v. 32, n. 10, p. 121-124 1999.

ESTEREL. *Efficient development of avionic software with DO-178B objectives using Scade suite*. Esterel Technologies: Elancourt, França, 2006

ESYSTECH. *eAT55 ARM evaluation board: manual do usuário*. ESYSTECH - Embedded Systems Technologies: Curitiba, Brasil, 2005

ESYSTECH. *X Real Time Kernel*. ESYSTECH - Embedded Systems Technologies. 2007. Disponível em: <http://www.esystech.com.br/produtos/XKernel/XKernel.htm>. Acessado em 12/02/2007.

FERGUSON, D. F.; STOCKTON, M. L.. Service-Oriented Architecture: Programming model and product architecture. *IBM Systems Journal*, White Plains, EUA, v. 44, n. 4 2005.

FOWLER, M.. *Patterns for enterprise application architecture*. Reading, Massachusetts, EUA: Addison-Wesley, 2003.

GAMMA, E.; HELM, R.; RALPH, J.; VLISSIDES, J.. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Porto Alegre, Brasil: Bookman, 2000.

GANE, C.; SARSON, T.. *Análise estruturada de sistemas*. 20a. ed.Rio de Janeiro, Brasil: LTC, 1995.

GIMENES, I. M. S.; HUZITA, E. H. M.. *Desenvolvimento baseado em componentes: conceitos e técnicas*. Rio de Janeiro, EUA: Ciência Moderna, 2005.

GOLDBERG, A.; ROBSON, D.. *Smalltalk 80: the language*. New Jersey, EUA: Addison-Wesley, 1989.

GOMAA, H.. *Designing concurrent, distributed, and real-time applications with UML*. Indianapolis, EUA: Addison-Wesley, 2000.

GOMAA, H.; SCOTT, D. B. H.. Prototyping as a tool in the specification of user requirements. *Anais do 5th International Conference of Software Engineering*. 1981.

GRADY, R. B.. *Pratical Software Metrics for Project Management and Process Improvement*. Reading, EUA: Prentice-Hall, 1992.

GUMMUS, B.. *Axiomatic product development lifecycle*. Lubbock, Texas, EUA, 2005. Tese de Doutorado- , Texas Tech University.

HAAG, H.; RAJA, M. K.; SCHKADE, L. L.. Quality Function Deployment usage in software development. *Communications of the ACM*, New York, EUA, v. 39, n. 1, p. 41-49, jan. 1996.

HAREL, D.. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, North, Holanda, n. 8, p. 231-274 1987.

HAUSER, J. R.; CLAUSING, D.. The house of quality. *Harvard Business Review*, Harvard, EUA, p. 63-73, mai./jun. 1988.

HILSDALE, E.; KERSTEN, M.. Aspect-Oriented Programming with AspectJ (tutorial 24). *Anais do Proceedings of OOPSLA2004 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Vancouver, Canadá: 2004.

HINTERSTEINER, J. D.. A fractal representation for systems. *Anais do 1999 International CIRP Seminar*. Enschede, Holanda: 1999.

HINTERSTEINER, J. D.; NAIN, A. S.. Integrating software into systems: an axiomatic design approach. *Anais do 3rd international Conference on Engineering Design and Automation*. Vancouver, Canadá: 1999.

HU, M.; YANG, K.; TAGUCHI, S. Enhancing Robust Design with the Aid of TRIZ and Axiomatic Design. *The TRIZ Journal*, , out. 2000.

IEEE. *610.12-1990 IEEE standard glossary of software engineering terminology*. Institute of Electrical and Electronics Engineers: Nova York, EUA, 1990

IEEE. *IEEE standard for developing software life cycle processes*. Institute of Electrical and Electronics Engineers: New York, EUA, 1997

JACOBSON, I.. Object-Oriented Development in an Industrial Environment. *Anais do OOPSLA 87*. 1987.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J.. *The Unified Software Development Process*. Reading, Massachusetts, EUA: Addison-Wesley, 1999.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; ÖVERGAARD, G.. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, Massachusetts, EUA: Addison-Wesley, 1992.

JACOBSON, I.; NG, P. W.. *Aspect-oriented software development with use cases*. Reading, Massachusetts, EUA: Addison-Wesley, 2004.

KAPOR, M.. A Software Design Manifesto. *Dr. Dobb's Journal*, Bolder, EUA, v. 16, n. 1, jan. 1990.

KIM, S. J.; SUH, N. P.; KIM, S. K.. Design of software systems based on axiomatic design. *Anais do CIRP General Assembly 1991*. : 1992.

KRASNER, G.; POPE, S.. A cookbook for using the Model-View-Controler user interface paradigm in Smalltalk 80. *Journal of Object-Oriented Programming*, Framingham, EUA, v. 1, n. 3, p. 26-49 1988.

KROGSTIE, J.. Using Quality Function Deployment in Software Requirements Specification. *Anais do REFSQ'99*. Heidelberg, Alemanha: 1999.

KRUCHTEN, P.. *Introdução ao RUP - Rational Unified Process*. Rio de Janeiro, Brasil: Ciência Moderna, 2003.

KUMAR, V.; CAMPION, M.. A Web-based Course for Practicing Engineers on Axiomatic Design Principles. *Anais do ICAD2004 - the 3rd InternationalConference on Axiomatic Design*. Seul, Coreia: 2004.

LARMAN, C.. *Applying UML and paterns: an introduction to object-oriented analysis and design*. New Jersey, EUA: Prentice Hall, 1997.

LARMAN, C.. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3a. ed. New Jersey, EUA: Addison-Wesley Professional, 2004.

LEE, T.. *Complexity Theory in Axiomatic Design*. Cambridge, EUA, 2003. Tese de Doutorado - Department of mechanical engineering, Massachusets Institute of Technology.

LEE, T.. Optimal Strategy. *Anais do 9th World Conference on Integrated Design and Process Technology - IDPT 2006*. San Diego, EUA: 2006.

LEFFINGWELL, D.; WIDRIG, D.. *Managing software requirements*. 2a. ed. Reading, Massachusets, EUA: Addison-Wesley, 2003.

LORENZ, M.; KIDD, J.. *Object-Oriented Software Metrics*. New Jersey, EUA: Prentice-Hall, 1994.

McCABE, T. J.. A complexity measure. *IEEE Transactions on Software Engineering*, New York, EUA, v. 30, n. 4, p. 308-320 1976.

McMENAMIM, S.; PALMER, J. F.. *Análise Essencial de Sistemas*. São Paulo, Brasil: Makron, 1991.

MOORE, G.. Cramming more components onto integrated circuits. *Electronics Magazine*, Nova York, EUA,, v. 38, n. 8, abr. 1965.

NAKAZAWA, H.. *Information integration method*. Tokio, Japão: Corona, 1987.

NORLUND, M.. *An Information Framework for Engineering Desing*. Estocolmo, Suécia,

1996. Tese de Doutorado - Department of Manufacturing Systems, Royal Institute of Technology (KTH).

OLEWNIK, A. T.; LEWIS, K. E.. On validating design decision methodologies. *Anais do Design Theory and Methodology Conference - DETC 2003*. Chicago, EUA: 2003.

OMG. *Unified Modeling Language - Superstructure specification Versão 2.0*. OBJECT MANAGEMENT GROUP: Nedham, Massachusets, EUA, 2005

PADKE, M. S.. *Quality engineering using robust Design*. New Jersey, EUA: Prentice-Hall, 1989.

PAULK, M.C.; CURTIS, B.; CHRISSIS, M.; WEBERr, C.V.Capability Maturity Model for Software (Ver 1.1)1991

PETERS, J. F.; PEDRYCZ, W.. *Engenharia de Software: teoria e prática*. Rio de Janeiro, Brasil: Campus, 2001.

PIMENTEL, A. R.; STADZISZ, P. C.. Object-Oriented software design using the axiomatic design theory. *Anais do Anais do II Congresso Brasileiro de Tecnologia*. Recife, Brasil: 2005.

PIMENTEL, A. R.; STADZISZ, P. C.. The aplication of the independence axiom on the design of object-oriented software using the axiomatic design theory. *Journal of Integrated Design and Process Science*, EUA, v. 10, n. 1, p. 57-70, mar. 2006.

PIMENTEL, A. R.; STADZISZ, P. C.. A use case based object-oriented software design approach using the axiomatic design theory. *Anais do ICAD2006 - the 4th International Conference on Axiomatic Design*. Florença, Itália: 2006.

PRASAD, B.. *Concurrent Engineering Fundamentals - Volume 1*. New Jersey, EUA: Prentice-Hall, 1996.

PRESSMAN, R. S.. *Engenharia de software*. 3a. ed.São Paulo, Brasil: Makron Books, 1995.

PRESSMAN, R. S.. *Software Engineering: a practitioner's approach*. 6a. ed .New York, EUA: McGraw-Hill, 2005.

QUINAIA, M. A.. *Contribuição a uma metodologia para identificação e especificação de padrões arquiteturais de software*. Curitiba, Brasil, 2005. Tese de Doutorado - CPGEI, Universidade Tecnológica Federal do Paraná.

RAWLINSON, G.. TRIZ and software. *Anais do TRIZCON 2001, the Altshuler Institute Conference*. 2001.

REA, K. C.. TRIZ and software 40 principles analogies part 2. *The TRIZ Journal*, nov. 2001.

REA, K. C.. TRIZ and software 40 principles analogies part 1. *The TRIZ Journal*, set. 2001.

REA, K. C.. Appliyng TRIZ to software problems, creatively bridging academia and practice in computing. *Anais do TRIZCON 2002, the Altshuler Institute Conference*. 2002.

ROYCE, W. W.. Managing the Development of Large Software Systems: Concepts and

Techniques. *Anais do IEEE Western Conference (WESCON)*. 1970.

RTCA. *Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics: Washington, EUA, 1999

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.. *Object-Oriented Modeling and Design*. New Jersey, EUA: Prentice Hall, 1991.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G.. *The unified modeling language reference manual*. 2a. ed. Reading, Massachusetts, EUA: Addison-Wesley, 2004.

SALUSTRI, F. A.; VENTER, R. D.. An Axiomatic Theory of Engineering Design Information. *Engineering with Computers*, Londres, Inglaterra, v. 8, n. 4, p. 197-211 1992.

SAVRANSKI, S. D.. *Engineering of Creativity*. Boca Raton, EUA: CRC Press, 2000.

SCHREYER, M.; TSENG, M. M.. Hierarquichical state decomposition for the design of PLC software by applying axiomatic design. *Anais do ICAD2000 - the 1st International Conference on Axiomatic Design*. Cambridge, EUA: 2000.

SHIN, G. S.; YI, J. W.; YI, S. I.; KWON, Y. D.; PARK, G. J.. Calculation of infornation content in axiomatic design. *Anais do ICAD2004 - the 3rd International Conference on Axiomatic Design*. Seul, Coreia: 2004.

SHLAER, S.; MELLOR, S.. *Object-Oriented Systems Analysis: Modeling the World in Data*. Indianapolis, EUA: Prentice Hall, 1988.

STADZISZ, P. C.. *Contribuition à une Mèthodologie de Conception Intégrée des Famillies deProduits pour L'Assemblage*. Besançon, França, 1997. Tese de Doutorado, L'Universite de Franche-Comte.

STEVENS, W.; MEYERS, G.; CONSTANTINE, L.. Structured design. *IBM Systems Journal*, New York, EUA, v. 13, n. 2, p. 115-139 1974.

STROUSTRUP, B.. *The C++ programming language*. 3a. ed.Indianapolis, EUA: Pearson Education, 1997.

SUH, N. P.. *The Principles of Design*. Oxford, EUA: Oxford University Press, 1990.

SUH, N. P.. *Axiomatic Design: advances and applications*. New York, EUA: Oxford University Press, 2001.

SUH, N. P.; BELL, A. C.; GOSSARD, D. C.. On an Axiomatic Approach to Manufactoring and Manufactoring Systems. *Journal of Engineering for Industry*, New York, EUA,, v. 100, n. 2, p. 127-130 1978.

SUH, N. P.; DO, S. H.. Axiomatic design of software systems. *Anais do CIRP annals*. Sidney, Australia: 2000.

SULLIVAN, L. P.. Quality Function Deployment. *Quality Progress*, Milwaukee, EUA, p. 39-50, jun. 1986.

TATE, D.. *A roadmap for decomposition: activities, theories and tools for system design*. Cambridge, EUA, 1999. Tese de doutorado - Department of mechanical engineering, Massachusets Institute of Technology.

TAYLOR, E. S. *An interim report on engineering design*. Massachusets Institute of Technology, Cambridge, EUA, 1959.

TOGAY, C.; AKTUNC, O.; TANIK, M. M.; DOGRU, A. H.,. Measurement of component congruity for composition based on axiomatic design. *Anais do 9th World Conference on Integrated Design and Process Technology - IDPT 2006*. San Diego, EUA: 2006.

TOGAY, C.; DOGRU, A. H.; TANIK, U. J.; GRIMES, G. J.. Component oriented simulation development with axiomatic design. *Anais do 9th World Conference on Integrated Design and Process Technology - IDPT 2006*. San Diego, EUA: 2006.

TOMIYAMA, T.; KIRIYAMA, T.; TAKEDA, H.; XUE, D.; YOSHIKAWA, H.. Metamodel: a key to inteligent CAD systems. *Research in Engineering Design*, Heidelberg, Alemanha, n. 1, p. 19-34 1989.

ULRICH, K. T.; EPPINGER, S. D.. *Product design and development*. New York, EUA: McGraw-Hill, 2003.

VALCKENAERS, P.. *Flexibility for integrated production automation*. Leuven, Bélgica, 1993. Tese de Doutorado, Katholiek Universiteit Leuven.

VAZQUEZ, C. E.; SIMÕES, G. S.; ALBERT, R. M.. *Análise de Pontos de Função: medição, estimativas e gerenciamento de projetos de software*. São Paulo, Brasil: Érica, 2003.

VERVERS, F; van DALEN, P; van KATWIJK, J. A case study in the application of object oriented metrics. *Anais do 9th World Conference on Integrated Design and Process Technology*. Austin, EUA: 1996.

WHITCOMB, C. A.; SZATKOWSKI, J. J.. Concept level naval surface combatant design in the axiomatic approach todesign framework. *Anais do ICAD2000 - the 1st International Conference on Axiomatic Design*. Cambridge, EUA: 2000.

WINCK, D. V.; GOETTEN Jr., V.. *AspectJ: programação Orientada a Aspectos com Java*. São Paulo, Brasil: NovaTec, 2006.

WISSE, P.. *MetaPattern*. Massachusets, EUA: Addison-Wesley, 2001.

WYNS, J.. *Reference architecture for holonic manufacturing systems: the key to support evolution and reconfiguration*. Leuven, Bélgica, 1999. Tese de Doutorado, Katholiek Universiteit Leuven.

YOSHIKAWA, H.. General design theory and a CAD system. *Anais do IFIP WG2.5/2.6 working conference, Man-Machine Communications in CAD/CAM*. 1981.

ZADEH, L. A.; FU, K. S.; TANAKA, K.; SHIMURA, M.. *Fuzzy sets and their applications to cognitive and decision processes*. New York, EUA: Academic Press, 1975.

ZENG, Y.. Axiomatic Theory of Design Modeling. *Anais do IDPT - 2003 - World*

Conference on Integrated Design and Process Technology. 2003.

ZULTNER, R. E.. TQM for technical teams. *Communications of the ACM*, New York, EUA, v. 36, n. 10, p. 79-91, out. 1993.

Anexo A - Teoremas e Corolários Relacionados com os Axiomas

Neste anexo são apresentados corolários e teoremas relacionados com os axiomas 1 e 2. Serão apresentados os teoremas sobre projeto em geral e os teoremas relacionados com projeto de *software*. Os corolários e teoremas apresentados a seguir foram traduzidos de (SUH, 2001).

A.1 Corolários

Corolário 1 (Desacoplamento de projetos acoplados) *Desacople ou separe as partes ou aspectos de uma solução se os requisitos funcionais estiverem acoplados ou se tornarem interdependentes no projeto proposto.*

Corolário 2 (Minimização dos requisitos funcionais) *Minimize o número de requisitos funcionais e restrições.*

Corolário 3 (Integração das partes físicas) *Integre características de projeto em uma única parte física se os requisitos funcionais puderem ser satisfeitos independentemente na solução proposta.*

Corolário 4 (Uso de padronização) *Use partes padronizadas ou intercambiáveis se o uso destas partes estiver consistente com os requisitos funcionais e restrições.*

Corolário 5 (Uso de simetria) *Use formas e componentes simétricos se eles estiverem consistentes com os requisitos funcionais e restrições.*

Corolário 6 (Maior faixa de variação de projeto) *Especifique a maior faixa de variação de projeto permitido quando estiver estabelecendo os requisitos funcionais.*

Corolário 7 (Projeto desacoplado com menos informação) *Procure um projeto desacoplado que requer menos informação que projetos acoplados em satisfazer um conjunto de requisitos funcionais.*

Corolário 8 (Reangularidade efetiva de um escalar) *A reangularidade efetiva R para uma matriz de acoplamento escalar ou elemento de matriz é 1.*

A.2 Teoremas Gerais de Projeto

Teorema 1 (Acoplamento devido a um número insuficiente de parâmetros de projeto (DPs)) *Quando o número de parâmetros de projeto (DPs) é menor que o número de requisitos funcionais (FRs), ou ele resulta em um projeto acoplado ou os requisitos funcionais (FRs) não poderão ser satisfeitos.*

Teorema 2 (Redução do acoplamento de um projeto acoplado) *Quando um projeto é acoplado devido a um número de requisitos funcionais (FRs) maior que o número de parâmetros de projeto (DPs) (m > n), seu acoplamento pode ser reduzido pela adição de novos parâmetros de projeto (DPs) de forma a tornar o número de requisitos funcionais (FRs) igual ao número de parâmetros de projeto (DPs) se um subconjunto da matriz de projeto contendo n x n elementos constitui uma matriz triangular.*

Teorema 3 (Projeto redundante) *Quando existem mais parâmetros de projeto (DPs) que requisitos funcionais (FRs), o projeto ou é um projeto redundante ou é um projeto acoplado.*

Teorema 4 (Projeto ideal) *Em um projeto ideal, o número de parâmetros de projeto (DPs) é igual ao número de requisitos funcionais (FRs) e os requisitos funcionais são sempre mantidos independentes uns dos outros.*

Teorema 5 (Necessidade de um novo projeto) *Quando um conjunto de requisitos funcionais (FRs) é alterado pela adição de um novo requisito funcional (FR), pela substituição de um dos requisitos funcionais (FRs) por um novo ou pela seleção de um conjunto completamente diferente de requisitos funcionais (FRs), a solução do projeto dada pelos parâmetros de projeto (DPs) originais não pode satisfazer o novo conjunto de requisitos funcionais (FRs). Consequentemente, uma nova solução de projeto deve ser buscada.*

Teorema 6 (Independência de caminho do projeto desacoplado) *O conteúdo de informação de um projeto desacoplado é independente da seqüência pela qual os parâmetros de projeto (DPs) são mudados para satisfazer um dado conjunto de requisitos funcionais (FRs).*

Teorema 7 (Dependência de caminho de projeto acoplados e semi-acoplados) *O conteúdo de informação de um projeto acoplado ou semi-acoplado depende da seqüência pela qual os parâmetros de projeto (DPs) são mudados e dos caminhos específicos de mudança destes parâmetros de projeto (DPs).*

Teorema 8 (Independência e a faixa de variação de projeto) *Um projeto é um projeto desacoplado quando a faixa de variação especificada pelo projetista é maior que*

$$\left(\sum_{i \neq j,\ j=1}^{n} \frac{\partial FR_i}{\partial DP_j} \Delta DP_j \right) \tag{31}$$

Neste caso, os elementos fora da diagonal da matriz de projeto podem ser desconsiderados

do projeto.

Teorema 9 (Projeto para "manufaturabilidade") *Para um produto ser manufaturável com confiabilidade e robustez, a matriz de projeto para o produto [A], (que relaciona o vetor de requisitos funcionais (FRs) para o produto com o vetor dos parâmetros de projeto (DPs) do produto), multiplicada pela matriz de projeto do processo de manufatura [B] (que relaciona o vetor de requisitos funcionais (FRs) para o processo de manufatura com o vetor dos parâmetros de projeto (DPs) do processo), deve resultar ou em uma matriz diagonal ou em uma matriz triangular. Conseqüentemente, quando tanto [A] quanto [B] representam um projeto acoplado, a independência dos requisitos funcionais (FRs) e um projeto robusto não podem ser alcançados. Quando as matrizes forem matrizes triangulares completas, ambas devem ser triangulares superiores ou ambas devem ser triangulares inferiores para que o processo de manufatura satisfaça a independência dos requisitos funcionais (FRs).*

Teorema 10 (Modularidade das medidas de independência) *Supondo que uma matriz de projeto [DM] possa ser particionada em submatrizes quadradas que somente possuam elementos diferentes de zero na diagonal principal, então, a reangularidade e a semangularidade para [DM] são iguais ao produto das suas medidas correspondentes para cada submatriz.*

Teorema 11 (Invariância) *A reangularidade e a semangularidade para uma matriz de projeto [DM] não varia com a mudança da ordem dos requisitos funcionais (FRs) e dos parâmetros de projeto (DPs) enquanto forem preservadas as associações entre cada requisito funcional (FR) e seus correspondentes parâmetros de projeto (DPs).*

Teorema 12 (Soma da informação) *A soma da informação para um conjunto de eventos também é informação, desde que probabilidades condicionais apropriadas sejam usadas quando os eventos não forem estatisticamente independentes.*

Teorema 13 (Conteúdo de informação de todo o sistema) *Se cada parâmetro de projeto (DP) for probabilisticamente independente dos outros parâmetros de projeto (DPS), o conteúdo de informação total do sistema é a soma da informação de todos os eventos individuais associados com o conjunto de requisitos funcionais (FRs) que devem ser satisfeitos.*

Teorema 14 (Conteúdo de informação de projetos acoplados *versus* desacoplados) *Quando o estado dos requisitos funcionais (FRs) é mudado de um estado para o outro no domínio funcional, a informação requerida para a mudança é maior para projetos acoplados que para projetos desacoplados.*

Teorema 15 (Interface projeto-manufatura) *Quando um sistema de manufatura compromete a independência dos requisitos funcionais (FRs) do produto, ou o projeto do produto deve ser modificado ou um novo processo de manufatura deve ser projetado e/ou utilizado para manter a independência dos requisitos funcionais (FRs) do produto.*

Teorema 16 (Igualdade do conteúdo de informação) *Todos os conteúdos de informação que são relevantes para a tarefa de projetar são igualmente importantes, não importando sua origem física, e nenhum fator de ponderação deverá ser aplicado a eles.*

Teorema 17 (Projeto na ausência de informação completa) *O projeto pode ser feito mesmo na ausência de informação completa apenas no caso de um projeto semi-acoplado se a informação faltante está relacionada com elementos fora da diagonal.*

Teorema 18 (Existência de um projeto desacoplado ou semi-acoplado) *Sempre irá existir um projeto desacoplado ou semi-acoplado que possua menor conteúdo de informação que um projeto acoplado.*

Teorema 19 (Robustez do projeto) *Um projeto desacoplado e um projeto semi-acoplado são mais robustos que um projeto acoplado no sentido de que é mais fácil reduzir o conteúdo de informação de projetos que satisfazem o axioma da independência.*

Teorema 20 (Faixa de variação de projeto e acoplamento) *Se as faixas de variação de projeto para projetos desacoplados ou semi-acoplados forem limitadas, os projetos podem se tornar projetos acoplados. No sentido contrário, se as faixas de variação de projeto para projetos acoplados forem relaxadas, eles podem se tornar projetos desacoplados ou semi-acoplados.*

Teorema 21 (Projeto robusto quando o sistema possui uma função de distribuição de probabilidade não uniforme) *Se a função de distribuição de probabilidade do requisito funcional (FR) na faixa de variação de projeto não é uniforme, a probabilidade de sucesso é igual a 1 quando a faixa de variação do sistema está dentro da faixa de variação de projeto.*

Teorema 22 (Robustez comparativa de um projeto semi-acoplado) *Dadas as faixas de variação de projeto máximas para um dado conjunto de requisitos funcionais (FRs), projetos semi-acoplados não podem ser tão robustos quanto projetos desacoplados em que as tolerâncias permitidas para parâmetros de projeto (DPs) de um projeto semi-acoplado são menores que aqueles para um projeto desacoplado.*

Teorema 23 (Diminuindo a robustez de um projeto semi-acoplado) *A tolerância permitida e, portanto, a robustez de um projeto semi-acoplado com uma matriz triangular completa diminui com um aumento no número de requisitos funcionais (FR).*

Teorema 24 (Agendamento ótimo) *Antes que um agendamento para a movimentação de um robô ou agendamento de manufatura possa ser otimizado, o projeto das tarefas deve ser feito de maneira a satisfazer o axioma de independência pela adição de desacopladores para eliminar o acoplamento. Os desacopladores podem ser na forma de um hardware isolado ou buffer.*

A.3 Teoremas Relacionados com Projeto de *software*

Teorema *Soft* 1 (Conhecimento requerido para operar um sistema desacoplado) *Sistemas de software ou hardware desacoplados podem ser operados sem um conhecimento preciso sobre elementos do projeto (módulos) se o projeto é verdadeiramente um projeto desacoplado e se as saídas dos requisitos funcionais (FRs) puderem ser monitoradas para permitir um controle dos requisitos funcionais (FRs).*

Teorema *Soft* 2 (Tomada de decisões corretas na ausência do conhecimento completo para um projeto semi-acoplado com controle) *Quando o sistema de software é um projeto semi-acoplado, os requisitos funcionais (FRs) podem ser satisfeitos pela mudança dos parâmetros de projeto (DPs) se a matriz de projeto é conhecida para a extensão que o conhecimento a respeito da seqüência apropriada é dada, mesmo que não haja um conhecimento preciso a respeito dos elementos do projeto.*

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo