UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Estudo Experimental Comparativo de Modelos de Componentes para o Desenvolvimento de Software Sob o Aspecto de Evolutibilidade

Nádia Milena da Silva Barbosa

Campina Grande, Paraíba, Brasil
Maio de 2007

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Estudo Experimental Comparativo de Modelos de Componentes para o Desenvolvimento de Software Sob o Aspecto de Evolutibilidade

Nádia Milena da Silva Barbosa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação (MSc).

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Angelo Perkusich
Orientador

Campina Grande, Paraíba, Brasil
Maio de 2007

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

B238e Barbosa, Nádia Milena Da Silva
2007 Estudo experimental comparativo de modelos de componentes para o desenvolvimento de software sob o aspecto de evolutibilidade/ Nádia Milena da Silva Barbosa. – Campina Grande, 2007
84fs.: il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências

Orientadores: Angelo Perkusich

1. Desenvolvimento de Software Baseado em componentes. 2. Engenharia de Software Experimental. 3. Modelos de Componentes. 4. Qualidade. 7. Métricas de Software. I. Título.

CDU 004.415.2

Resumo

A utilização do desenvolvimento de software baseado em componentes (DBC) oferece vantagens com relação a tempo de produção, custos e maior facilidade de manutenção. Com isso, o DBC apresenta-se como uma abordagem viável para atender as exigências dos sistemas de softwares atuais, tais como: dinamicidade, robustez e flexibilidade para contemplar mudanças de requisitos. No entanto, não se pode garantir, apenas utilizando componentes, que o software poderá atender essas exigências e evoluir mantendo-se consistente. Atributos como evolutibilidade, que está diretamente relacionado com as arquiteturas de sistemas, ainda não são muito explorados nos estudos relativos a DBC. Neste contexto, um estudo experimental comparativo dos modelos de componentes EJB, COMPOR e CCM é realizado neste trabalho, tendo como foco a evolução de software. Nesse estudo experimental utiliza-se um arcabouço de medição composto por um modelo de qualidade e um conjunto de métricas de software para capturar informações sobre o sistema em termos dos atributos fundamentais do software. O estudo está dividido em duas fases: a fase de construção e a fase de evolução. Na fase de evolução é possível observar o comportamento de cada um dos modelos de componentes diante de cenários evolutivos. Por meio deste estudo experimental é formada uma base de conhecimento sobre os modelos de componentes analisados, tornando possível uma escolha entre os modelos de componentes de forma adequada aos requisitos do sistema que se pretende desenvolver. Além disso, tem-se um conjunto de métricas reusáveis que podem servir como parâmetro de medição em outros estudos sobre evolução em DBC.

Palavras-chave

Desenvolvimento de software baseado em componentes, engenharia de software experimental, modelos de componentes, evolução de software, arcabouço de medição, modelo de qualidade, métricas de software.

Abstract

The use of component-based software development (CBD) offers advantages as production time, cost and maintainability. Therefore, CBD presents itself as a viable approach to attend the requirements of current systems, such as: dynamism, robustness, and requisite change possibility. However, one cannot be sure that only by using components, the software will be able to attend to these requirements and evolve maintaining, at the same time, its consistency. Attributes such as evolutionability, directly related to systems' architecture, have not been sufficiently explored on studies involving CBD. In this context, an experimental study comparing the component model EJB, COMPOR and CORBA is performed in the present work, focusing on software evolution. In this study, it is used a measurement framework composed by one quality model and one set of the software metrics for capturing information about the system in terms of the software basic attributes. The study is divided into two parts: the construction phase and the evolution phase. In the evolution phase it is possible to observe the behavior of each component model in face of evolution scenarios. By means of this experimental study, it will be created a knowledge base on analyzed component models, making possible to correctly choose between the component models that best fits the requirements of the system to be developed. Moreover, a set of reusable metrics that can serve as a measurement parameter on other studies about evolution in CBD is created.

key words

Component based software development, experimental software engineering, components models, software evolutions, measurement framework, quality model, metrics of software

Agradecimentos

Agradeço aos meus pais, irmãs, amigos, professores, orientadores e a todos que contribuíram para a realização deste trabalho. E, agradeço a Deus, pois, sozinha eu não teria conseguido.

Conteúdo

1 Introdução **1**

1.1 Descrição do problema 2

1.2 Objetivo da dissertação 3

1.3 Relevância do tema 3

1.4 Organização do trabalho 4

2 Modelo de Componentes: definições e conceitos relacionados **5**

2.1 Componentes 5

2.2 Modelo e arcabouço de componentes 6

2.2.1 Arcabouço de componentes 7

2.2.2 Modelo de componentes 7

3 Medição de Software **11**

3.1 Modelos de qualidade de software 13

3.1.1 Modelo de McCall 13

3.1.2 Modelo de Boehm 15

3.2 A Abordagem Goal-Question-Metric 16

4 Arcabouço de Medição **19**

4.1 O Modelo de Qualidade 20

4.1.1 Objetivo - Atributo Externo 21

4.1.2 Fatores 22

4.1.3 Atributos Internos 22

4.1.4 Métricas 23

4.2 Métricas de software 24

4.2.1 Métrica de Acoplamento 25
4.2.2 Métrica de Coesão 27
4.2.3 Métricas de Tamanho 27

5 Estudo Experimental 29

5.1 A aplicação 29
5.1.1 Organização do experimento 30
5.2 Resultados obtidos 31
5.2.1 Fase de construção 32
5.2.2 Resultados obtidos na fase de evolução 35
5.2.3 Discussões sobre os Resultados do Estudo Experimental 43

6 Trabalhos Relacionados 45

6.1 Manutenibilidade e Reusabilidade de Software Orientado a Aspectos 45
6.2 Medição em Componentes Caixa-preta 46
6.3 *Component Quality Model* (CQM) 46
6.4 *Interface Complexity Metric* (ICM) 47
6.5 Validação Cruzada de Métricas 47
6.6 Métricas de Reusabilidade 47
6.7 Discussões sobre os trabalhos relacionados 48

7 Conclusões e Trabalhos Futuros 50

7.1 Conclusões 50
7.2 Trabalhos futuros 51

A Modelos de Componentes 57

Lista de Figuras

2.1 Relacionamento entre componentes, modelo e arcabouço de componentes . 9

3.1 Modelo de MacCall de Qualidade de Software 14
3.2 Modelo de Boehn de Qualidade de Software 15
3.3 O processo principal da abordagem GQM 17
3.4 Estrutura hierárquica da abordagem GQM 18

4.1 Modelo de qualidade 21
4.2 Objetivo e perguntas geradas com o uso da abordagem GQM 24

5.1 Diagrama de casos de uso da biblioteca particular 30
5.2 Diagrama de classes da biblioteca particular 31
5.3 Diagrama de componentes da biblioteca particular 32
5.4 Gráficos da aplicação das métrica de acoplamento e coesão 33
5.5 Gráficos da aplicação das métrica de tamanho 34
5.6 Gráficos da aplicação das métrica de acoplamento e coesão para o cenário 1 36
5.7 Gráficos da aplicação das métricas de tamanho para o cenário 1 38
5.8 Gráficos da aplicação das métrica de acoplamento e coe são para o cenário 2 39
5.9 Gráficos da aplicação das métricas de tamanho para o cenário 2 40
5.10 Gráficos da aplicação das métrica de acoplamento e coesão para o cenário 3 41
5.11 Gráficos da aplicação das métricas de tamanho para o cenário 3 42

A.1 Interação baseada em serviços 59
A.2 Interação baseada em eventos 60
A.3 Estrutura da arquitetura do MCC 68

Lista de Tabelas

4.1 Atributos, Métricas e Descrição 26

Capítulo 1

Introdução

Um componente de software pode ser definido como uma unidade de composição com interfaces bem definidas e dependências de contexto explícitas [Szy99]. Sistemas desenvolvidos utilizando o paradigma de Desenvolvimento Baseado em Componentes (DBC), enfatizam o reúso e apresentam vantagens no sentido de possibilitar o aumento da produtividade e da qualidade do software [Crn01]. As razões para o interesse por DBC, segundo [BW98], vêm da maturidade das tecnologias que permitem a construção de componentes e a combinação destes para o desenvolvimento de aplicações, bem como o atual contexto organizacional e de negócios que apresenta mudanças em como as aplicações são desenvolvidas, utilizadas e mantidas.

No contexto de DBC, a arquitetura de software assume um importante papel, pois define o sistema em termos de seus componentes e os relacionamentos entre eles, sendo possível especificar como se dá a interconexão entre componentes [Wer00]. As abstrações arquiteturais expressam regras de projeto que impõem um modelo padrão de coordenação. Estas regras de projeto têm a forma de um modelo de componentes.

Um modelo de componentes estabelece a separação entre a definição do componente (suas interfaces, serviços e dependências) e sua implementação. Componentes são manipulados como caixas-pretas, ou seja, são manipulados com base exclusivamente na sua definição. A definição de um componente especifica um conjunto de conectores através dos quais é possível acessar os serviços do componente e fornecer os recursos esperados pelo componente, definidos como suas dependências.

Há diversos modelos de componentes, tais como EJB [Mic99], COMPOR [APP+06] e

CCM [OMG05], presentes neste trabalho. Cada modelo de componentes define os tipos de conectores que os componentes de um software podem fornecer. Na investigação relacionada com componentes, tem-se enfatizado, sobretudo, a definição dos seus aspectos funcionais e a composição de componentes na construção de sistemas [HC01]. Atributos externos como manutenibilidade e evolutibilidade têm sido pouco explorados, bem como o impacto que a evolução de um software pode causar na arquitetura do sistema.

A maior dificuldade de avaliar os atributos externos dos sistemas é que eles, em geral, só podem ser analisados quando o produto final já está pronto: por exemplo, o desempenho do sistema, flexibilidade, dificuldade de navegação entre as telas ou a quantidade de memória utilizada para realizar uma tarefa. Com isso, o impacto que uma mudança pode causar aos sistemas e em suas arquiteturas nem sempre pode ser previsto. A conseqüência mais direta disso é que os custos para evoluir um sistema são proporcionais ao grau de dificuldade desta evolução ocorrer. Assim, quanto mais difícil for para evoluir um sistema, mais caro será para mantê-lo.

Os sistemas podem passar por um processo de evolução acompanhando as mudanças no ambiente. Os principais tipos de evolução de software são citados a seguir.

- Reparar falhas em softwares, isto é, evoluir o sistema para corrigir deficiências para que esteja de acordo com os requisitos.
- Evolução do software para adaptá-lo a um ambiente operacional diferente da sua implementação original (sistema operacional e hardware, por exemplo).
- Evolução do software para adicionar ou modificar funcionalidades ou para satisfazer novos requisitos que não faziam parte do sistema inicialmente.

1.1 Descrição do problema

Os modelo de componentes constroem Sistemas Baseados em Componentes (SBC) seguindo especificações próprias. A maneira como um software evolui é dependente da arquitetura de software utilizada durante a construção. Assim, aplicações desenvolvidas utilizando modelos de componentes diferentes podem apresentar diferentes graus de dificuldade quando se deseja adicionar, atualizar ou remover um componente do sistema.

Estudos experimentais realizados em SBC têm enfatizado atributos externos como reusabilidade [Gon06], testabilidade e portabilidade [MABC+03]. Atualmente, não existem estudos que mostrem evidências sobre a evolutibilidade dos softwares. São necessários também, estudos que façam uma comparação entre o comportamento das aplicações que utilizam modelos de componentes distintos. Assim, será possível definir qual modelos melhor se adequa a uma aplicação que tenha a evolutibilidade como um dos seus requisitos.

1.2 Objetivo da dissertação

Neste trabalho propõe-se realizar um estudo experimental comparativo de modelos de componentes sob o aspecto da evolutibilidade de software. Os modelos de componentes estudados são: EJB, COMPOR e CCM. Um arcabouço de medição é apresentado para avaliar o requisito não funcional de evolutibilidade em sistemas baseados em componentes. O arcabouço é formado por um conjunto de métricas de software e por um modelo de qualidade. Os elementos do arcabouço são baseados em princípios conhecidos da Engenharia de Software e métricas já existentes e validadas em projetos orientados a objetos [SGC+03]. Tais métricas são adaptadas para serem aplicadas no projeto e no código de software baseado em componentes.

1.3 Relevância do tema

O estudo é realizado em duas fases: fase de construção e fase de evolução de software. O sistema escolhido para viabilizar a aplicação das métricas do arcabouço de medição, uma biblioteca pessoal, foi implementado em várias versões sob as mesmas condições de projeto e seguindo o mesmo modelo conceitual. Inicialmente, na fase de construção, foram construídas três versões, uma para cada modelo de componente presente no experimento. Na fase seguinte, a fase de evolução do sistema, três cenários de evolução foram definidos: adição de componente, atualização de componente e exclusão de componente.

As métricas foram aplicadas sobre o código em cada uma das versões construídas. Os dados coletados são apresentados em gráficos, nos quais é possível observar as diferenças entre os valores obtidos nas aplicações implementadas utilizando cada modelo de componentes,

assim como, o comportamento dos modelos diante de cenários de evolução de software.

Com este estudo experimental, são apresentadas evidências sobre o comportamento dos modelos de componentes EJB, COMPOR e CCM diante de cenários evolutivos. Assim, para construir uma aplicação baseada em componentes que tenha como requisito a facilidade de evolução, a análise tem grande importância, pois, através dos dados coletados será possível escolher um dos modelos de acordo com as necessidades do sistema.

Além do objetivo já citado, o arcabouço de medição também pode ser usado em avaliações periódicas durante a reestruturação de um software baseado em componentes. Numa dessas avaliações, pode-se verificar se uma determinada métrica obteve melhores resultados em relação ao valor da avaliação anterior. Isso pode, então, indicar que a estratégia de reestruturação usada está surtindo efeito e deve ser mantida. As métricas do arcabouço de medição foram definidas de maneira que venham a ser reutilizadas em outros estudos que envolvam o aspecto de evolutibilidade.

1.4 Organização do trabalho

Este trabalho está estruturado da seguinte maneira.

- No Capítulo 2 é apresentada a definição de modelo de componentes e alguns conceitos relacionados como o de componentes e desenvolvimento baseado em componentes.
- No Capítulo 3, apresenta-se uma visão sobre medição de software, modelos de qualidade e abordagens utilizadas.
- No Capítulo 4 é apresentado o arcabouço de medição e seus dois elementos básicos: o conjunto de métricas e o modelo de qualidade definidos para o trabalho.
- O estudo experimental realizado utilizando o arcabouço de medição é descrito no Capítulo 5.
- No Capítulo 6 são apresentados alguns trabalhos relacionados.
- E finalmente, no Capítulo 7, são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Modelo de Componentes: definições e conceitos relacionados

2.1 Componentes

Várias definições de componentes são encontradas na literatura. Em [Sam97], define-se componentes como alguma parte do sistema de software que é identificável e reusável. Considera-se que componentes de software reusáveis são artefatos autocontidos, facilmente identificáveis, que descrevem e executam funções específicas e têm interfaces claras, documentação apropriada e uma condição de reúso definida.

Outra definição a ser considerada é apresentada por [BW98], na qual um componente é caracterizado como um conjunto independente de serviços reutilizáveis que provê habilidades acessíveis a outros componentes. Essa característica indica a necessidade da existência de uma especificação que apresente o que o componente faz e como se comporta quando os serviços são usados. O segundo elemento da definição - independente, indica a ausência de vínculo do componente com o contexto em que ele pode ser usado. A expectativa de que os componentes cooperem entre si para completar uma solução não deve estar vinculada à existência de dependências entre eles, pois os componentes não devem ser desenvolvidos com dependências fixas entre si.

Em [Szy99], define-se componente de software como uma unidade de composição com interfaces contratualmente especificadas e apenas explícitas dependências de contexto. Componente de software pode ser usado de forma independente e combinado com outras partes.

A propriedade de ser uma unidade de atuação independente exige que o componente seja separado do ambiente e dos demais componentes. Também não deve existir a perspectiva de se ter acesso a detalhes de construção do componente. Logo, o componente precisa encapsular sua implementação e interagir com o ambiente através de interfaces bem definidas. Para que o componente possa ser combinado com outros componentes ele precisa ser suficientemente autocontido. É necessário que o componente apresente uma especificação indicando o que é exigido para seu reúso e o que ele provê.

As três definições apresentadas [BW98; Szy99; Sam97], em conjunto com suas explicações, abordam diferentes aspectos e características comuns dos componentes. Neste trabalho, define-se um componente como uma unidade de software com função específica e interface bem definida. Para abordagem de desenvolvimento baseado em componentes utilizada no trabalho, os componentes são considerados apenas artefatos da fase de implementação.

2.2 Modelo e arcabouço de componentes

Um componente não pode ser visto de forma completamente independente dos outros componentes com o qual se relaciona. Desta forma, a arquitetura de software assume um importante papel por ser a partir dela que é possível especificar de forma mais detalhada como ocorre a interconexão entre componentes [Wer00].

Conforme [Bac00], os componentes podem ser vistos segundo duas distintas perspectivas: implementações e abstrações arquiteturais. Vistos como implementações, os componentes representam artefatos de software que podem ser disponibilizados e usados para compor grandes sistemas. Por outro lado, vistos como abstrações arquiteturais, os componentes expressam regras de projeto que impõem um modelo padrão de coordenação para todos os componentes. Estas regras de projeto têm a forma de um modelo de componentes, ou de um conjunto de padrões e convenções com as quais os componentes devem estar em conformidade.

Existe um relativo consenso na literatura quanto ao uso do termo modelo de componentes para identificar o conjunto de padrões e convenções com as quais os componentes devem estar em conformidade. Entretanto, a infra-estrutura que dá suporte a estes modelos

é encontrada na literatura tanto identificada como arcabouço de componentes [Bac00] como infra-estrutura de componentes [Bro95].

2.2.1 Arcabouço de componentes

Segundo [Szy99], um arcabouço de componentes é uma entidade de software que provê suporte a componentes que seguem um determinando modelo e possibilita que instâncias destes componentes sejam conectados no arcabouço de componentes. Ele estabelece as condições necessárias para um componente ser executado e regula a interação entre as instâncias destes componentes. Um arcabouço de componentes pode ser único na aplicação, criando uma ilha de componentes ao seu redor, ou pode cooperar com outros componentes ou arcabouços de componentes.

Componentes de software, como já foi dito, podem ser definidos como unidades independentes, que encapsulam dentro de si seu projeto e implementação e oferecem serviços para o meio externo através de interfaces bem definidas. Arcabouços de componentes também fornecem interfaces bem definidas, ou pontos de extensão, que as aplicações devem estender. Contudo, enquanto um arcabouço possui necessariamente pontos de extensão, um componente totalmente autocontido não possui interfaces requeridas. Além disso, um componente deve ser conectado a uma interface requerida de outro componente enquanto que os pontos de extensão dos arcabouços são menos exigentes, possibilitando que sejam conectados componentes, classes ou qualquer artefato que realiza o contrato definido.

O arcabouço de componentes representa a base sobre a qual os padrões e convenções do modelo de componentes são empregados. Desta forma, arcabouço e modelo de componentes podem ser considerados dois modelos complementares e fortemente relacionados. As definições estabelecidas pelo modelo de componentes devem ser suportadas pelo arcabouço, bem como o arcabouço deve respeitar e regular as definições estabelecidas pelo modelo de componentes [Bac00].

2.2.2 Modelo de componentes

Um modelo de componentes representa um elemento da arquitetura do sistema na qual são definidos os padrões e convenções impostas aos componentes do sistema, de modo a descre-

ver as funções de cada um e como eles interagem entre si. De acordo com [Bac00] espera-se através de um modelo de componentes definir os seguintes padrões e convenções:

- Tipos de componentes: definidos em termos das interfaces que implementam, onde cada interface de um componente corresponde a um tipo. Caso um componente implemente as interfaces A e B, então ele é do tipo A e B, o que lhe garante uma capacidade polimórfica em relação a estes tipos. Isto permite que estes diferentes tipos de componentes desempenhem diferentes papéis no sistema, bem como participem de diferentes formas de interação.

- Formas de interação: definição da forma de interação entre componentes e entre componentes e o arcabouço de componentes, através da especificação de como os componentes são localizados, o protocolo de comunicação usado e como a qualidade dos serviços é alcançada. A classe de interação entre componentes compreende restrições quanto aos tipos de componentes que podem ser clientes de outros tipos, o número de possíveis clientes simultâneos e outras restrições topológicas. A classe de interação entre componentes e arcabouço inclui definições relacionadas a gerenciamento de recursos, como o ciclo de vida de um componente (ativação, desativação), formas de gerência, persistência e assim por diante. As formas de interação podem dizer respeito a todos os tipos de componentes ou apenas a tipos particulares.

- Definição de recursos: a composição dos componentes é realizada pela ligação dos componentes a um ou mais recursos, onde um recurso pode ser tanto produzido por um arcabouço de componentes quanto por algum componente utilizado no arcabouço. O modelo de componentes descreve quais recursos estão disponíveis a cada componente, como e quando eles estão associados a estes recursos. Em contrapartida, o arcabouço vê os componentes como recursos a serem gerenciados.

O conjunto de convenções definidas por um modelo de componentes em [HC01] é apresentado a seguir:

- Interfaces - especificação do comportamento e propriedades.

- Nomeação - nomes globais únicos para a interfaces e componentes.

- Metadados - informações sobre os componentes, interfaces e seus relacionamentos.
- Interoperabilidade - comunicação e troca de dados entre componentes de diferentes origens, implementados em diferentes linguagens.
- Customização - interfaces que possibilitam a customização dos componentes.
- Composição - interfaces e regras para combinar componentes no desenvolvimento de aplicações e para substituir e adicionar componentes às aplicações já existentes.
- Suporte à evolução - regras e serviços para substituir componentes ou interfaces por versões mais novas.
- Empacotamento e utilização - empacotar implementações e recursos necessários para instalar e configurar componentes.

O relacionamento entre componentes, modelo e arcabouço de componentes pode ser identificado na Figura 2.1:

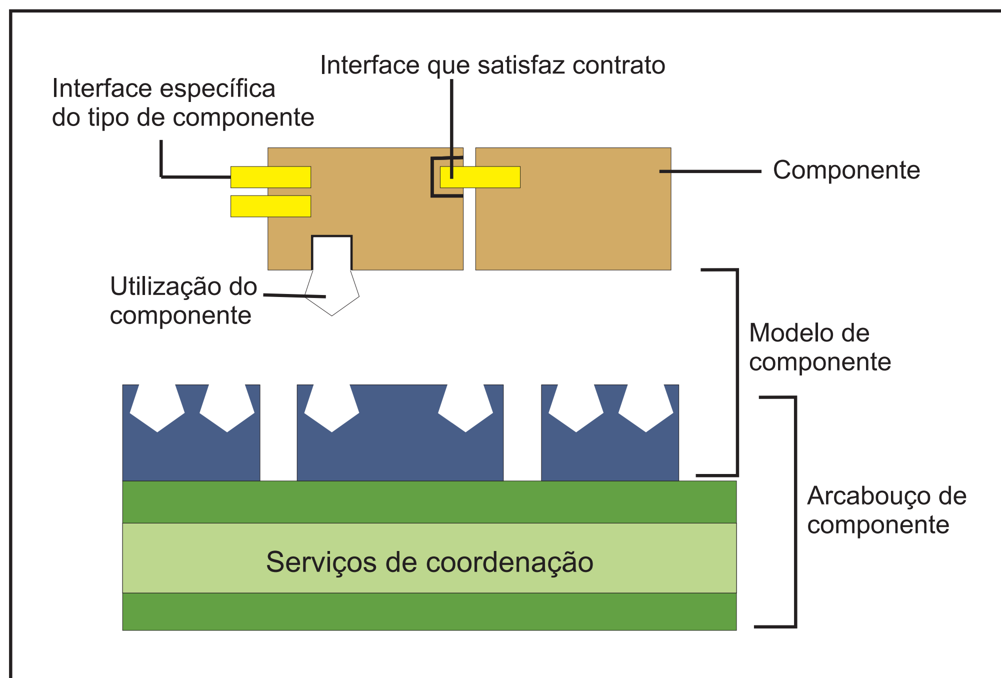


Figura 2.1: Relacionamento entre componentes, modelo e arcabouço de componentes

É possível identificar o arcabouço de componentes como uma infra-estrutura de suporte à comunicação e ligação dos componentes, através do fornecimento de serviços de coordenação. Já o modelo de componentes identifica as definições quanto a tipos de componentes,

formas de interação e recursos necessários. Os componentes definidos para executar determinadas funcionalidades devem estar em conformidade com as definições do modelo de componentes, pertencendo a um dos possíveis tipos e respeitando as formas de interação definidas, além de utilizarem os serviços disponibilizados. Por fim, a utilização dos componentes caracteriza a ligação entre modelo e arcabouço de componentes.

Capítulo 3

Medição de Software

Processos de medição se tornaram uma parte tão importante quanto necessária nas organizações que desenvolvem software. Pois, para competir em um ambiente caracterizado por rápidas e constantes mudanças, é fundamental trabalhar de maneira produtiva, eficiente e com alto nível de qualidade. A maioria dos profissionais da área de desenvolvimento de software compreende a necessidade de se realizar medições, mas, infelizmente, a implementação de um processo que venha a se tornar comum e integrado aos ciclos de vida de desenvolvimento e manutenção de software de uma forma geral ainda é um grande problema. As principais razões para o fracasso de programas de medição não são problemas técnicos, e sim organizacionais [Rub92], tais como: não-alinhamento aos objetivos de negócio, resistência cultural, motivação errônea e falta de liderança [Sch04].

Um programa de medição de sucesso é mais que simplesmente uma coletânea de dados. Os benefícios e valores agregados obtidos através da medição dizem respeito às decisões e ações tomadas a partir da análise dos dados obtidos, e não da coleção de dados em si. Portanto, a melhor abordagem para definição e implementação de um processo de medição é a que define, antes de tudo, o que a organização deseja ou precisa saber. Somente então a escolha das medidas apropriadas é realizada. Uma vez que as medidas estejam definidas, o passo seguinte é encontrar uma coleção de dados específica que possa apoiar a obtenção destas medidas. Especificamente, um processo deste tipo envolve os seguintes passos [Sch04]:

1. definir os objetivos e iniciativas;

2. definir as medidas que apoiarão estes objetivos e iniciativas;

3. definir os dados que serão necessários para produzir estas medidas;
4. definir como analisar e comunicar os resultados das medidas;
5. implementar o processo.

Em termos gerais, medição é o processo pelo qual números ou símbolos são designados a atributos de entidades do mundo real de forma a descrevê-los de acordo com regras claramente definidas. Portanto, a medição captura informações sobre atributos de entidades. Uma entidade é um objeto (como uma pessoa ou uma sala) ou um evento (como uma viagem ou o projeto de desenvolvimento de um software). Um atributo é uma característica ou propriedade de uma entidade. Exemplos de atributos são a área de uma sala, o tempo de uma viagem ou o custo de um projeto de desenvolvimento de um software [SGC+03].

Toda medição deve iniciar-se com a identificação das entidades que se deseja medir. Existem, na Engenharia de Software, três classes de entidades: processos, produtos e recursos [SGC+03]. Processos são coleções de atividades relacionadas ao desenvolvimento de software. Produtos são quaisquer artefatos que resultam de uma atividade do processo. Recursos são entidades requeridas para realizar uma atividade do processo.

A medição de software é caracterizada em termos de atributos internos e externos. Um atributo interno pode ser medido em termos do objeto em si, esses atributos geralmente são das medidas diretas. Por exemplo, especificações podem ser avaliadas em termos de seu tamanho e grau de reutilização. O projeto detalhado e o código de um software podem ser avaliados por esses mesmos atributos e mais alguns outros como acoplamento e coesão.

Um atributo externo pode ser medido somente a respeito dos atributos de outros objetos. Os atributos externos estão relacionados às medidas indiretas e devem ser derivados dos atributos internos. Existem muitos atributos externos de produtos de software, tais como: confiabilidade, manutenibilidade, usabilidade, eficiência, reusabilidade e portabilidade. Esses atributos não estão relacionados apenas ao código, mas também a outros documentos e artefatos que dão apoio ao desenvolvimento de software. Neste trabalho, os atributos internos de tamanho, acoplamento, coesão e independência foram utilizados para avaliar o atributo externo de evolução.

3.1 Modelos de qualidade de software

A literatura propõe vários modelos de qualidade de software, de forma a determinar exatamente quais características devem ser analisadas para a obtenção de um padrão de qualidade. Os modelos de [McC77] e [Boe73], citados em [Pri00] são os mais abrangentes e citados na literatura.

3.1.1 Modelo de McCall

Este modelo é destinado aos projetistas de software, a fim de ser utilizado durante o processo de construção do sistema. Ele identifica três áreas de trabalho de software:

1. Operação do produto: requer que o software seja aprendido facilmente, operado eficientemente e que os resultados sejam aqueles que o utilizador espera.

2. Revisão do produto: está relacionada com a correção dos erros e com a adaptação do sistema.

3. Transição do produto: pode não ser importante em todas as aplicações. Entretanto, a transição para sistemas distribuídos e o rápido aumento na troca de hardware fazem com que esta área tenha sua importância cada vez mais enfatizada.

Na Figura 3.1, ilustra-se o modelo de qualidade de software de McCall. Os critérios de qualidade definidos por [McC77] são descritos a seguir.

- Usabilidade é a facilidade de utilização do software.

- Integridade é a proteção do programa contra o acesso não autorizado.

- Eficiência está relacionada com a utilização de recursos. Pode ser subdividida em eficiência na execução e eficiência no armazenamento.

- Corretude é a garantia de que o programa atende a sua especificação.

- Credibilidade é a habilidade do programa de não ter falhas.

- Manutenibilidade é o esforço necessário para localizar e reparar uma falha no programa no seu ambiente operacional.

- Flexibilidade é a facilidade de realizar modificações necessárias no ambiente operacional.
- Testabilidade é a facilidade de testar o programa para assegurar que ele não contenha erros e satisfaça a sua especificação.
- Portabilidade é o esforço necessário para transferir um programa de um ambiente para o outro.
- Reusabilidade é a facilidade de reutilizar o software em um contexto diferente.
- Interoperabilidade é o esforço necessário para juntar um sistema ao outro.

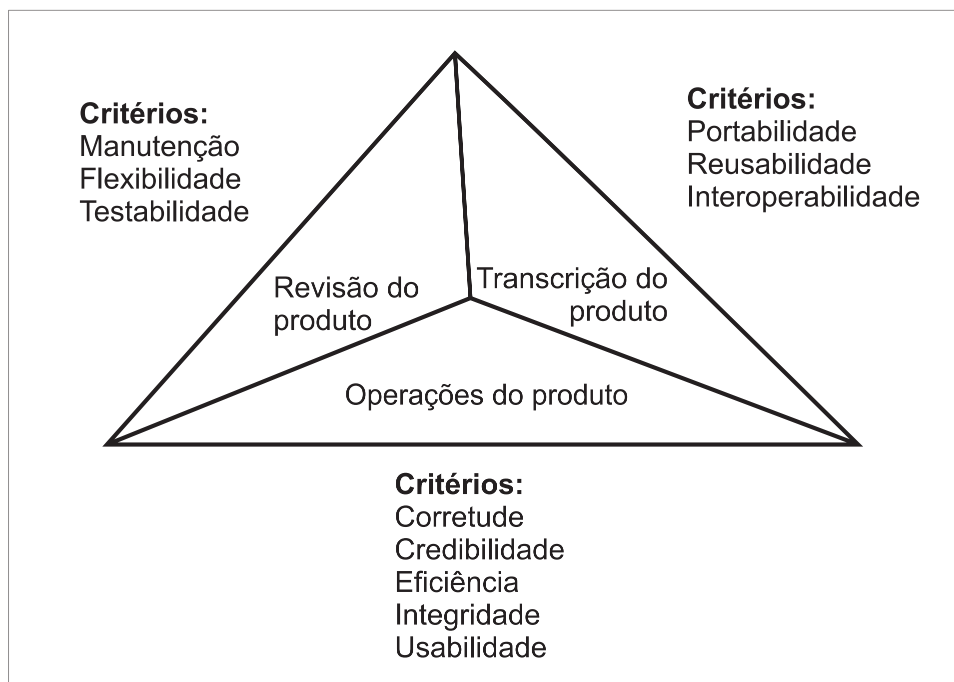


Figura 3.1: Modelo de MacCall de Qualidade de Software

A divisão de vários aspectos de qualidade de software em áreas, como no modelo de MacCall, possibilita a concentração de esforços de forma direcionada e particularizada. Neste trabalho, enfatiza-se características relacionadas com a revisão do produto. Ou seja, apesar de outras características serem bastante importantes no desenvolvimento do software, pretende-se enfatizar características de qualidade que levem à construção de um produto testável, fácil de manter e flexível [Pri00].

3.1.2 Modelo de Boehm

O modelo de qualidade proposto por Boehm foi definido com o objetivo de fornecer um conjunto de características de qualidade de software bem definidas e diferenciadas. Na Figura 3.2, ilustra-se o modelo de qualidade de Boehm [Boe73]. O modelo é hierárquico e os critérios de qualidade são subdivididos, a fim de identificar áreas a serem consideradas, de forma semelhante ao modelo de McCall. O modelo é apresentado de forma a enfatizar as escalas hierárquicas definidas, que têm por objetivo agrupar características com pontos em comum, que não poderiam ser reunidas em um único critério, devido à sua individualidade intrínseca.

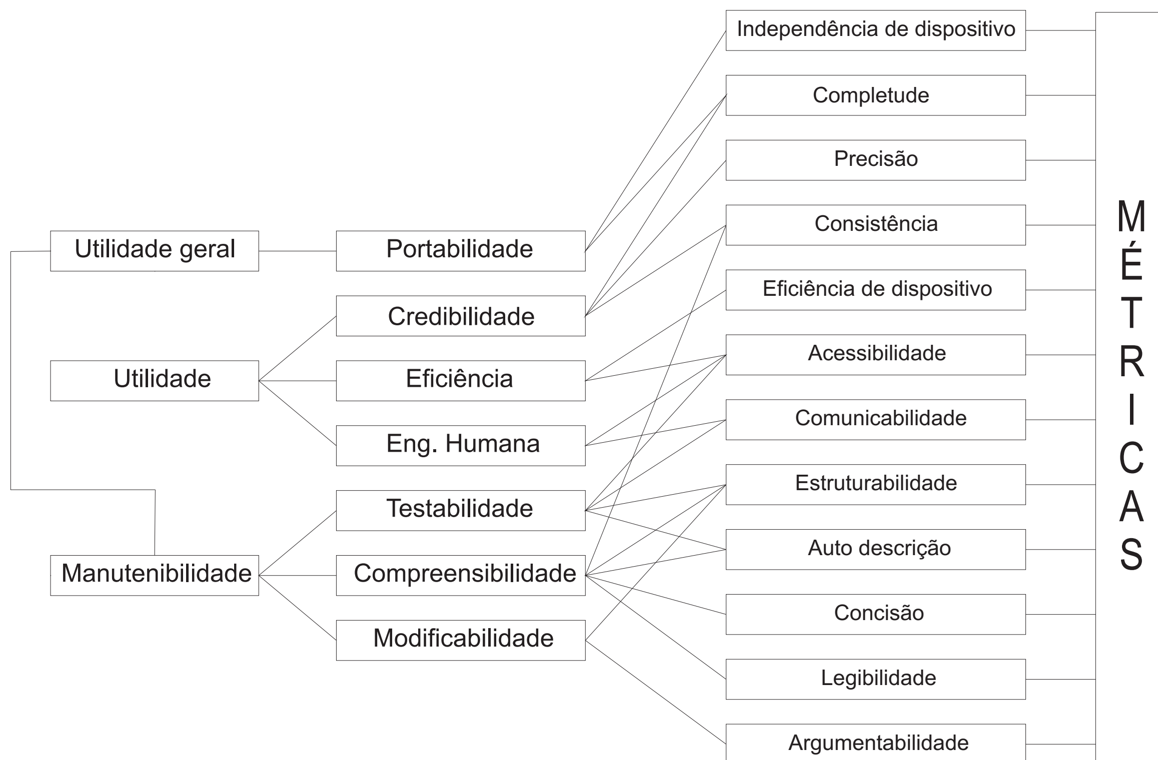


Figura 3.2: Modelo de Boehn de Qualidade de Software

A partir do modelo, atribui-se pesos para cada uma das características, de acordo com os objetivos da medição de qualidade em determinada aplicação. Por exemplo, a medição de qualidade considerada neste trabalho está relacionada diretamente com a manutenibilidade, e como tal, devem ser consideradas as características primitivas associadas.

Os modelos de MacCall e Boehm são modelos focados no produto final, e identificam atributos-chave das expectativas do utilizador. Estes atributos-chave são chamados de fatores

de qualidade, como por exemplo, credibilidade, usabilidade, manutenibilidade, testabilidade e eficiência. Cada um dos modelos assume que os fatores de qualidade são ainda muito altos para serem medidos diretamente e, desta forma, são decompostos em atributos de níveis mais baixos, chamados critérios de qualidade. Por exemplo, no modelo de Boehm, o fator credibilidade é composto pelo critério completude, precisão e consistência.

Para medir o critério de qualidade, outro nível de decomposição é necessário, no qual os critérios são associados com um conjunto de níveis mais baixos de atributos, diretamente mensuráveis. Estas são as chamadas métricas de qualidade.

De acordo com [GSF+05], modelos de qualidade podem ser usados de duas maneiras:

- um modelo já existente é escolhido e os relacionamentos entre os atributos externos, os atributos internos e as métricas são aceitos exatamente como propostos pelo autor do modelo;
- a filosofia geral de que um atributo externo é influenciado por vários atributos internos é aceita, mas não é adotado um modelo de qualidade já desenvolvido. Neste caso, um modelo de qualidade próprio é definido, baseado em modelos de qualidade e teorias já existentes;

Neste trabalho, a abordagem do modelo próprio foi utilizada. Um modelo de qualidade focado no atributo externo de evolução foi definido e será apresentado posteriormente.

3.2 A Abordagem Goal-Question-Metric

O paradigma GQM [BCR02] é uma abordagem orientada a objetivos para a medição de produtos e processos de software, que apóia a definição *top-down* do processo de medição e a análise *bottom-up* dos dados resultantes. Ela tem uma série de vantagens: ajuda na identificação de métricas úteis e relevantes; apóia a análise e interpretação dos dados coletados; permite uma avaliação da validade das conclusões tiradas; e diminui a resistência das pessoas contra processos de medição. Na Figura 3.3, ilustra-se o processo principal da abordagem GQM.

A abordagem GQM pode apoiar a medição de qualquer tipo de produto ou processo, cujo propósito seja qualquer um, desde a caracterização, até o controle e aperfeiçoamento, cujo

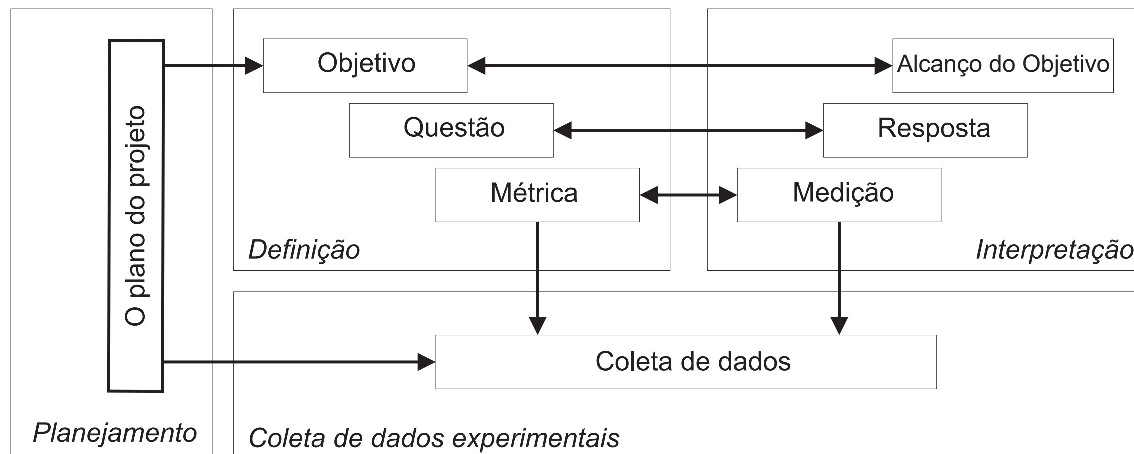


Figura 3.3: O processo principal da abordagem GQM

foco seja qualquer atributo de qualidade, definido sob qualquer perspectiva e em qualquer ambiente. A abordagem GQM é composta de quatro fases descritas a seguir.

- Fase de planejamento - quando o projeto da medição está selecionado, definido, caracterizado e planejado. Resultando em um plano de projeto.
- Fase de definição - quando o programa da medição está conceitualmente preparado, ou seja, os objetivos, as questões, as métricas e as hipóteses são estabelecidos.
- Fase de coleta de dados - quando a coleta de dados experimentais é efetivamente realizada resultando em um conjunto de dados prontos para interpretação.
- Fase de interpretação - quando os dados são processados a respeito das métricas, questões e objetivos definidos.

Uma parte essencial da abordagem GQM é a análise "custo-benefício" que é utilizada para avaliar se o benefício estimado supera o custo total. A abordagem GQM provê um arcabouço que envolve três passos, os quais são descritos a seguir.

1. Listar os principais objetivos do processo de medição.
2. Derivar de cada objetivo as perguntas que devem ser respondidas para determinar se os objetivos foram atingidos.
3. Decidir o que precisa ser medido para ser capaz de responder as perguntas adequadamente (definição das métricas).

Na Figura 3.4, ilustra-se a estrutura hierárquica gerada pela utilização da abordagem GQM. Neste trabalho a abordagem GQM foi usada para organizar o processo de medição e definir as métricas do framework de avaliação.

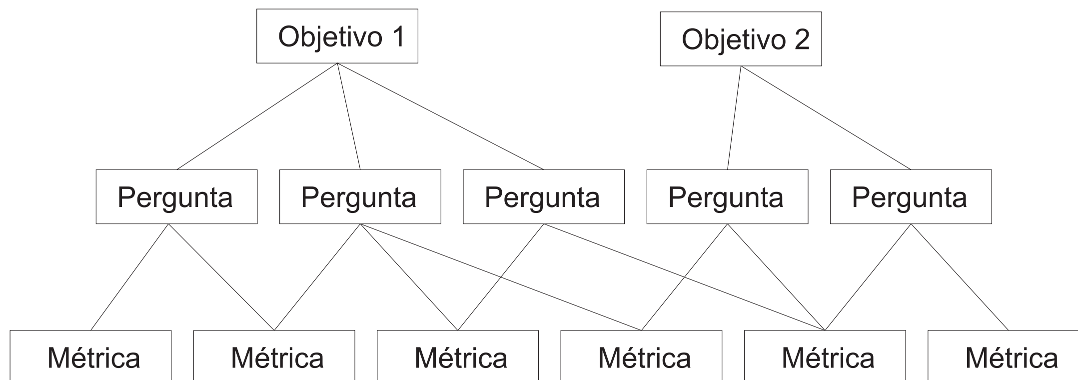


Figura 3.4: Estrutura hierárquica da abordagem GQM

Capítulo 4

Arcabouço de Medição

Os modelos de componentes especificam padrões e convenções que regem o desenvolvimento de componentes e a interação entre tipos de componentes. Cada modelo define especificações apropriadas a um domínio de aplicação. Sendo assim, os sistemas que são construídos utilizando os diversos modelos de componentes existentes apresentam características diferentes. As diferenças podem ser observadas nos atributos de produto. Este trabalho está focado no atributo externo de evolução de cada uma das aplicações construídas a partir de modelos de componentes distintos.

Para se obter evidências do entendimento das diferentes dimensões de complexidade de software, as métricas de software são o meio mais eficaz encontrado atualmente. Elas são mais efetivas quando associadas a algum arcabouço de medição, que auxilia os engenheiros de software no entendimento do significado dos dados coletados [SGC+03]. Métricas avaliam o uso de abstrações durante o desenvolvimento de software em termos de atributos de qualidade.

Neste trabalho, propõe-se um arcabouço de medição para capturar o entendimento dos atributos acoplamento, coesão e tamanho em termos de sua utilidade como meios para predizer o atributo externo de evolutibilidade de software. O objetivo do arcabouço é dar apoio à medição do projeto e do código das aplicações baseadas em componentes construídas utilizando os modelos de componentes COMPOR, CORBA e EJB, com o foco em evolução.

Os elementos do arcabouço ajudam na organização do processo de medição e na coleta e interpretação dos dados. Os elementos básicos do arcabouço são o conjunto de métricas e o modelo de qualidade. O conjunto de métricas é formado por métricas de tamanho, aco-

plamento e coesão. O modelo de qualidade descreve os relacionamentos entre os atributos externos, os atributos internos e as métricas. O usuário do arcabouço poderá reutilizar não só as métricas, como também todas as suposições e pontos de vista que justificam o uso de acoplamento, tamanho e coesão como atributos internos de produto que influenciam a evolutibilidade do software.

O principal contexto em que o arcabouço de medição pode ser usado, como já foi dito, é o contexto em que se compara o projeto de software construindo usando modelos de componentes distintos. No caso, CCM, COMPOR e EJB. Porém, o arcabouço também pode ser usado em avaliações periódicas durante a reestruturação de um software baseado em componentes. Numa dessas avaliações, pode-se verificar se uma determinada métrica obteve melhores resultados em relação ao valor da avaliação anterior. Isso pode, então, indicar que a estratégia de reestruturação usada, por exemplo, está surtindo efeito e deve ser mantida.

4.1 O Modelo de Qualidade

O modelo de qualidade proposto foi definido com base em um conjunto de suposições sobre os atributos internos de produto de software e métricas que influenciam a evolução. Esse conjunto de suposições e, conseqüentemente, a definição do modelo de qualidade foram baseados na revisão de modelos de qualidade existentes [Crn01] e definições clássicas de atributos de qualidade [Crn01; Szy99].

O modelo de qualidade apresentado é composto de quatro elementos: qualidades (atributos externos), fatores, atributos internos e métricas. As qualidades são os atributos externos que são o foco do arcabouço de avaliação (o atributo externo de evolução). Os fatores são atributos de qualidade secundários que influenciam as qualidades, mas que ainda não podem ser medidos diretamente nos artefatos de software. Os atributos internos, por sua vez, referem-se a propriedades internas de sistemas de software que influenciam os fatores e, conseqüentemente, as qualidades que se deseja avaliar. Os atributos internos são mais fáceis de medir que as qualidades e fatores, por isso as métricas são ligadas a esses atributos.

Também são descritos os elementos do modelo de qualidade proposto. Bem como, são explicadas as suposições que levaram à sua definição e apresentadas a definição de cada atributo usado. Na Figura 4.1, ilustra-se o modelo de qualidade proposto.

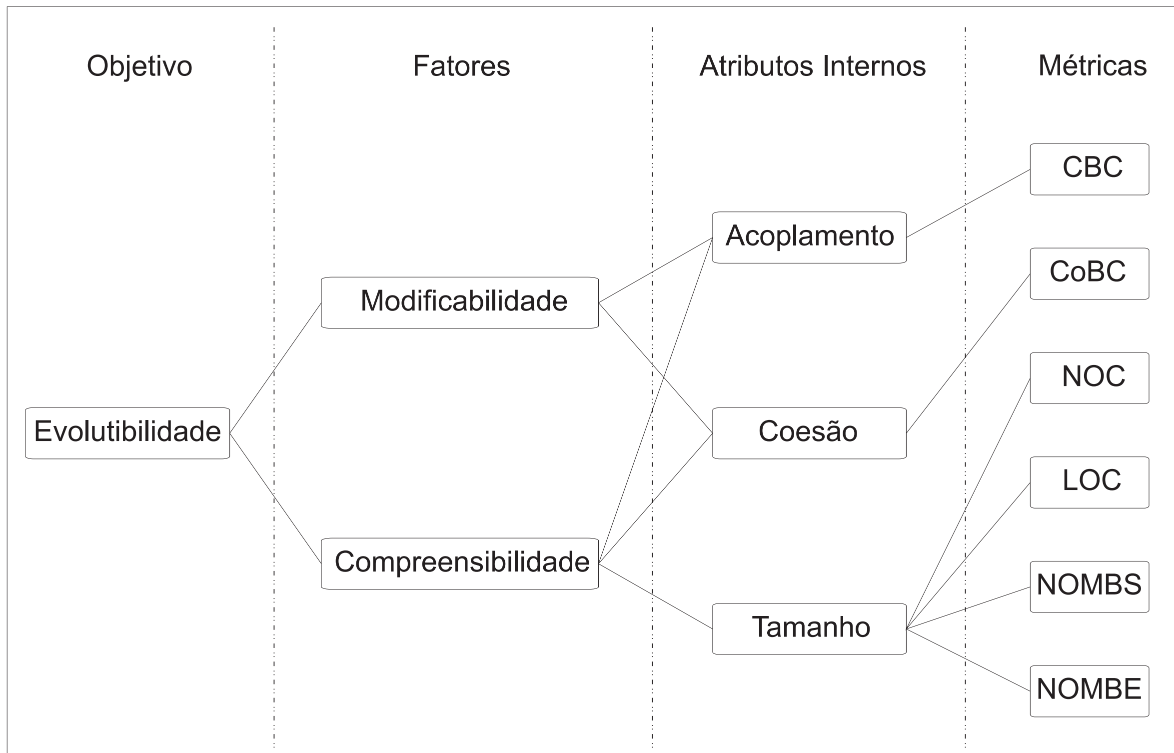


Figura 4.1: Modelo de qualidade

4.1.1 Objetivo - Atributo Externo

O atributo externo evolução é o foco do arcabouço de medição proposto neste trabalho. O arcabouço de medição apóia a avaliação da evolução de elementos do projeto e do código de sistemas baseados em componentes. Desta forma, é possível observar e comparar o comportamento de sistemas construídos utilizando modelos de componentes distintos como COMPOR, CORBA e EJB, presentes neste experimento sob o aspecto de evolutibilidade.

Para a realização do experimento na fase de evolução do sistema, foram definidos como parâmetros de evolução os três cenários.

1. Adicionar um componente ao sistema.
2. Remover um componente do sistema.
3. Alterar um componente no sistema.

A idéia do experimento é submeter as aplicações desenvolvidas utilizando cada um dos modelos de componentes aos cenários citados. Aplicando as métricas derivadas do objetivo

principal do estudo - o atributo de evolução, espera-se obter evidências sobre o comportamento de cada um dos modelos.

4.1.2 Fatores

O modelo de qualidade propõe fatores similares que exercem influência na evolução, tais como modificabilidade e compreensibilidade. A similaridade ocorre quando os ítens definidos como fatores derivam, em termos de compreensão, do objetivo estabelecido, no caso o atributo externo. No modelo, capacidade de mudanças e facilidade de compreensão são os fatores centrais que influenciam na evolução. Autores como Somerville [Som01], ao falar sobre qualidade de projeto de software, dizem que a facilidade de compreensão de um projeto é importante, pois, qualquer pessoa, para realizar mudanças em um projeto, precisa entendê-lo primeiro. Já o fator capacidade de mudança é justificado pela necessidade de se ter o software preparado para evoluir.

Compreensibilidade indica o nível de dificuldade para estudar e entender o projeto e o código de um sistema [KNMB02]. Projeto e código fáceis de entender contribuem para melhorar a evolução do sistema, pois a maioria das atividades de evolução requer que os engenheiros de software entendam primeiramente os componentes do sistema antes de modificá-los, estendê-los ou reutilizá-los.

Modificabilidade indica o nível de dificuldade para fazer mudanças no componente de um sistema sem afetar o resto do projeto [KNMB02]. Este fator se refere à questão da propagação, para outros componentes, do efeito de uma mudança em um componente. Para adicionar ou remover alguma funcionalidade do sistema durante as atividades de evolução em algum componente, é necessário realizar modificações no sistema. Portanto, quanto mais flexível for o projeto ou código de um sistema, melhor será sua evolução, pois o efeito das modificações se propagará por menos componentes do sistema [GSF+05].

4.1.3 Atributos Internos

Os atributos internos que compõem o modelo de qualidade proposto são tamanho, acoplamento e coesão. Acoplamento é uma indicação da interconexão entre os componentes de um sistema [Szy99]. A coesão de um componente é a medida da proximidade do relaciona-

mento entre seus componentes internos [Szy99]. O atributo tamanho mede quão extenso é o projeto e o código do sistema de software [Fen97].

No modelo proposto, o fator compreensibilidade está relacionado aos seguintes atributos internos: acoplamento, coesão e tamanho. Acoplamento e coesão afetam a facilidade de compreensão, porque um componente do sistema não pode ser completamente entendido sem a referência para os outros componentes com os quais ele está relacionado. O tamanho do projeto ou código pode indicar a quantidade de esforço necessária para entender os componentes do software.

O fator modificabilidade é influenciado, mais diretamente, pelos atributos internos de acoplamento e coesão. Alta coesão, baixo acoplamento e independência entre componentes são características desejadas, porque elas significam que um componente representa uma parte única do sistema e os componentes do sistema são independentes ou quase independentes. O ideal é que se for necessário adicionar, remover ou reutilizar alguma funcionalidade, ela esteja localizada em um único componente ou em uma única parte do sistema.

4.1.4 Métricas

A abordagem Goal-Question-Metric [Szy99] foi usada para definir o objetivo do arcabouço de avaliação e para derivar dele as perguntas que devem ser respondidas no intuito de determinar se o objetivo foi atingido. O processo de medição deve ser realizado de acordo com um objetivo específico e um conjunto de perguntas que represente a definição desse objetivo.

As perguntas também associam o modelo de qualidade a uma semântica mais precisa para as qualidades, fatores e atributos internos. Além disso, as perguntas podem ajudar os engenheiros de software na interpretação dos dados obtidos durante o processo de medição.

Na Figura 4.2, apresenta-se o objetivo e as perguntas geradas. A primeira pergunta, refere-se ao objetivo estabelecido, portanto, ao atributo externo evolutibilidade. Esta pergunta é refinada por perguntas sobre os atributos derivados do objetivo que são modificabilidade e compreensibilidade. Essas duas perguntas são novamente refinadas por perguntas sobre os atributos internos de tamanho, acoplamento e coesão.

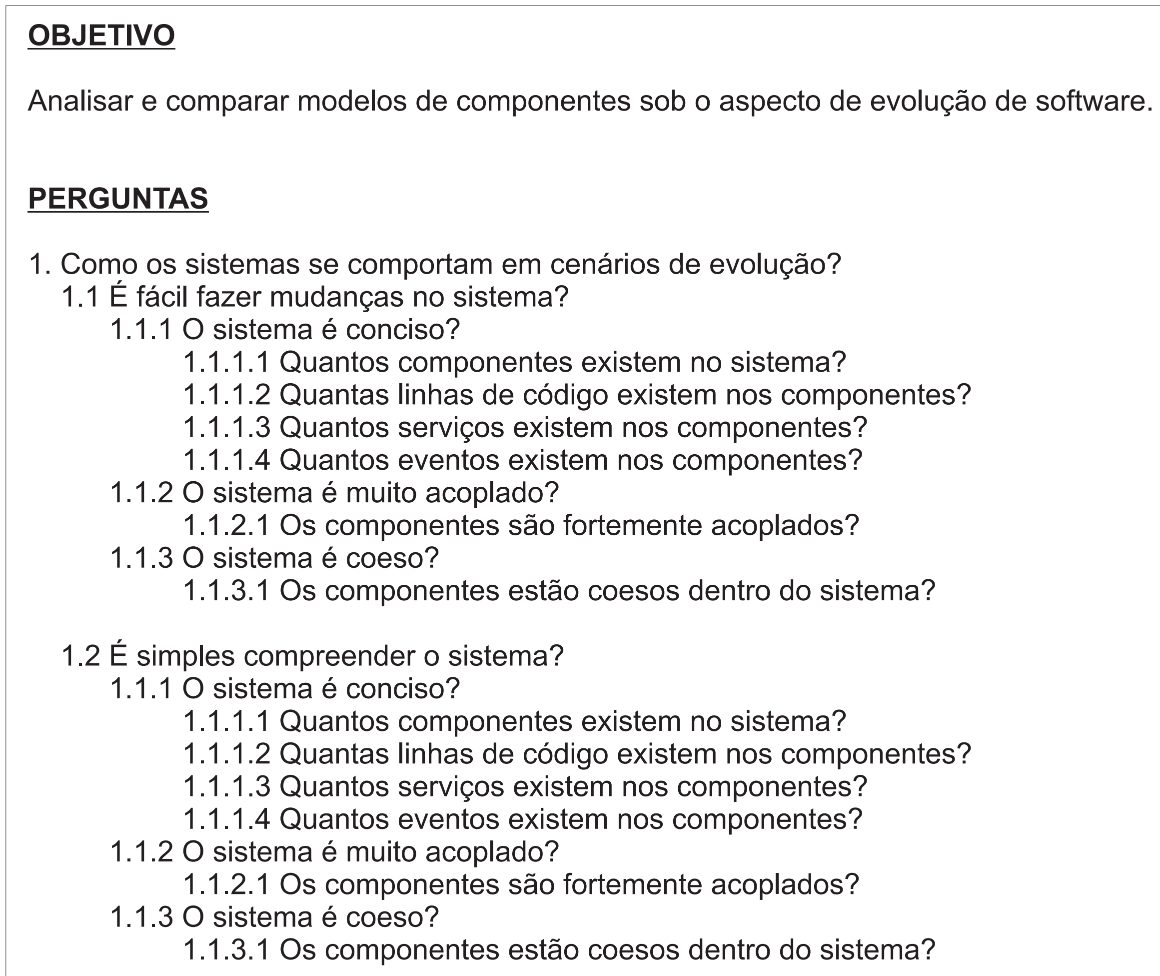


Figura 4.2: Objetivo e perguntas geradas com o uso da abordagem GQM

4.2 Métricas de software

Um dos intuitos de medir software é entender e aperfeiçoar o processo de desenvolvimento, identificar as melhores práticas de desenvolvimento de software, avaliar o impacto da variação de um ou mais atributos do produto ou do processo na qualidade e/ou produtividade, além de melhorar a exatidão das estimativas oferecendo dados qualitativos e quantitativos ao gerenciamento de desenvolvimento de software, de forma a realizar melhorias em todo o processo [Boe01].

De forma geral as métricas devem conter alguns ítens básicos para que sejam utilizadas na medição de software. A métrica deve ser válida, ou seja, quantificar o que queremos medir; confiável, para produzir os mesmos resultados dadas as mesmas condições; e prática, sendo fácil de computar e fácil de interpretar [Boe01]. As métricas propostas neste trabalho

têm por objetivo capturar informações sobre código de software baseado em componentes em termos de atributos de software fundamentais como acoplamento, coesão e tamanho. As métricas escolhidas foram reutilizadas e refinadas por serem métricas clássicas no mundo da orientação a objetos.

Os desenvolvedores, em geral, querem medir atributos externos, pois, estes são afetados pelo comportamento do sistema. No entanto, os atributos externos são mais difíceis de medir que os atributos internos e, normalmente, só podem ser medidos nas etapas finais do processo de desenvolvimento. Desenvolveu-se, então, os modelos de qualidade que definem atributos externos em termos de atributos internos para controlar os produtos durante o desenvolvimento. Como os atributos internos são mais fáceis de medir, métricas são propostas para eles. As métricas de software têm sido apontadas como indicadores de qualidade para projetos de software, além de serem o meio mais eficaz para se obter evidências experimentais que podem melhorar o entendimento da complexidade do software.

A maioria das métricas existentes até então são focadas em software funcional e orientado a objetos. Sendo assim, este trabalho adaptou métricas orientadas a objetos de forma que elas pudessem ser aplicadas a componentes. Cada definição de métrica foi estendida para ser aplicada de maneira independente do paradigma, apoiando a geração de resultados comparáveis. O critério para seleção das métricas usadas no arcabouço levou em consideração demandas teóricas e práticas. Por exemplo, as métricas de Chidamber e Kemerer [CK94] são baseadas na teoria de medição e têm sido amplamente usadas e validadas experimentalmente.

Na Tabela 4.1, as métricas são apresentadas em termos de sua relevância em relação ao atributo evolutibilidade. Apresenta-se também a seguir, de forma esquemática, a relação da métrica com os atributos internos e os autores que embasaram cada uma delas.

4.2.1 Métrica de Acoplamento

Acoplamento entre Componentes (CBC) é a métrica de acoplamento que faz parte do arcabouço de medição. A métrica CBC representa o número de outros componentes com os quais um dado componente está acoplado. Essa métrica é derivada da métrica de acoplamento entre objetos (CBO) que foi adaptada para ser capaz de atender um código baseado em componentes [CK94].

Atributo	**Métrica**	**Definição**
Acoplamento	Acoplamento entre componentes (CBC)	Mede o nível de acoplamento entre os componentes
Coesão	Coesão entre componentes (CoBC)	Mede o nível de coesão entre os componentes
Tamanho	Quantidade de componentes (NOC)	Mostra se existe variação na quantidade de componentes durante a evolução do sistema
Tamanho	Linhas de código (LOC)	Mostra se existe variação na quantidade de linhas de código durante a evolução do sistema
Tamanho	Número de métodos por serviço (NOMBS)	Mostra o número de métodos utilizados na implementação de um serviço
Tamanho	Número de métodos por evento (NOMBE)	Mostra o número de métodos utilizados na implementação de um evento

Tabela 4.1: Atributos, Métricas e Descrição

A escolha dessa métrica deve-se ao fato de que a compreensão de um componente envolve o entendimento dos componentes aos quais ele está acoplado. Então, quanto mais forte for o acoplamento entre componentes, mais difícil será para um componente isolado ser compreendido. Ou seja, quanto maior o número de acoplamentos, maior é a sensibilidade a mudanças em outras partes do projeto e, portanto, a evolução será mais difícil. Além disso, quanto menos acoplado for um componente, mais facilmente ele poderá ser reutilizado em outra aplicação.

Para os cenários de evolução estabelecidos neste trabalho, a análise do código a partir da métrica de acoplamento é bastante relevante. Considerando que o grau de acoplamento entre componentes é diretamente proporcional ao grau de dificuldade para que as ações de acréscimo, remoção e atualização de componentes em um sistema sejam implementadas.

4.2.2 Métrica de Coesão

O arcabouço de medição possui uma única métrica de coesão que é descrita como falta de coesão entre componentes (CoBC) [CK94]. A métrica CoBC mede a falta de coesão entre componentes dos sistema. Essa métrica estende a métrica LCOM de Chidamber [CK94] para objetos. A escolha desta métrica deve-se ao fato de que a baixa coesão entre as operações dos componentes sugere um projeto inadequado, pois isso significa o encapsulamento de entidades de programa não relacionadas entre si e que não deveriam estar juntas. Quanto maior for o grau com o qual diferentes ações executadas por um componente contribuam para funcionalidades distintas, mais difícil será para manter e reutilizar o componente ou uma de suas funcionalidades.

Para os cenários evolutivos de adição, exclusão e atualização de componentes num sistema, o grau de coesão entre os componentes pode ser determinante, considerando que quanto menos coesa for a relação entre componentes mais difícil será para evoluir o software.

4.2.3 Métricas de Tamanho

As métricas de tamanho tratam de diferentes aspectos do tamanho do sistema. O conjunto de métricas do arcabouço de avaliação é composto das seguintes métricas de tamanho: Número de componentes (NOC), Linhas de Código (LOC), Número de métodos por Serviço (NOMBS) e Número de métodos por Evento (NOMBE).

Número de componentes(NOC)

A métrica NOC mostra o número total de componentes do sistema. Essa métrica deriva da métrica tamanho do vocabulário que mede o número de classes, interfaces e aspectos que constituem o sistema [Pap01]. Assim sendo, para um sistema de componentes, cada nome de componente é contado como parte do vocabulário do sistema.

A escolha dessa métrica deve-se ao fato de que quanto maior for o tamanho do vocabulário, mais difícil será para entender o sistema. Conseqüentemente, mais difícil será para encontrar os componentes que precisam ser modificados durante atividades de evolução.

Linhas de Código (LOC)

A métrica LOC contabiliza o número de linhas de código do sistema. É uma medida de tamanho tradicional. Linhas em branco e comentários referentes ao código não são contadas. Para evitar problemas com os diferentes estilos de programação que poderiam influenciar os resultados obtidos o mesmo estilo de programação foi assegurado em todos os sistemas avaliados neste trabalho [Fen97].

A escolha dessa métrica deve-se ao fato de que o número de linhas de código é diretamente proporcional à dificuldade para entender o sistema. Quanto maior for o número de linhas de código, mais difícil será para encontrar as linhas que precisam ser alteradas durante atividades de evolução, e mais difícil será para entender a implementação das funcionalidades.

Número de métodos por Serviço (NOMBS)

Essa métrica mede a quantidade de métodos utilizados na implementação de um serviço presente no sistema de componentes. O somatório do número de métodos de todos os serviços disponibilizados numa aplicação, será o valor total de métodos usado como parâmetro de comparação entre os modelos de componentes.

A escolha dessa métrica deve-se ao fato de que quanto maior for o número e a complexidade das operações por componente, mais difícil será para entender o sistema, e conseqüentemente, de evoluir o sistema.

Número de métodos por Evento (NOMBE)

Essa métrica mede a quantidade de métodos utilizados na implementação de um evento presente no sistema de componentes. O somatório do número de métodos de todos os eventos disponibilizados numa aplicação, será o valor total de métodos usado como parâmetro de comparação entre os modelos de componentes.

A escolha dessa métrica deve-se ao fato de que quanto maior for o número e a complexidade das operações por componente, mais difícil será para entender o sistema, e conseqüentemente, de evoluir o sistema.

Capítulo 5

Estudo Experimental

Segundo Travassos [Tra06], os experimentos podem explorar os fatores críticos para que as teorias sejam formuladas e corrigidas. Neste trabalho o principal contexto em que o arcabouço pode ser usado é na comparação de projetos de software construídos usando os modelos de componentes EJB 2.0, COMPOR JCF e CCM 3.0. Desta forma, verifica-se o Comportamento de cada um dos modelos diante de cenários de evolução.

5.1 A aplicação

Para que o experimento fosse realizado de forma que os modelos estivessem em igualdade de condições foi escolhida uma aplicação simples, porém com recursos necessários para a aplicação das métricas de medição. Como as métricas são independentes da aplicação, qualquer outro sistema poderia servir como base para o estudo, desde que este contivesse os requisitos necessário para a realização do mesmo.

Neste trabalho a aplicação escolhida foi um sistema para controle de uma biblioteca particular. Os elementos básicos são:

- usuários, que são o proprietário da biblioteca e o gerente;
- material, que pode ser uma revista, um livro ou um artigo;
- empréstimo de material e devolução;
- reservas de material;

- eventos em que um material foi publicado.

Na Figura 5.1, ilustra-se de forma simplificada, o diagrama de casos de uso da biblioteca pessoal. Na Figura 5.2, apresenta-se um subconjunto do diagrama de classes gerado para um melhor entendimento da aplicação. É possível identificar as classes que formam a estrutura do sistema e as relações entre as mesmas. Na seção seguinte é apresentado o diagrama de componentes, que descreve o sistema em termo dos componentes que o compõem.

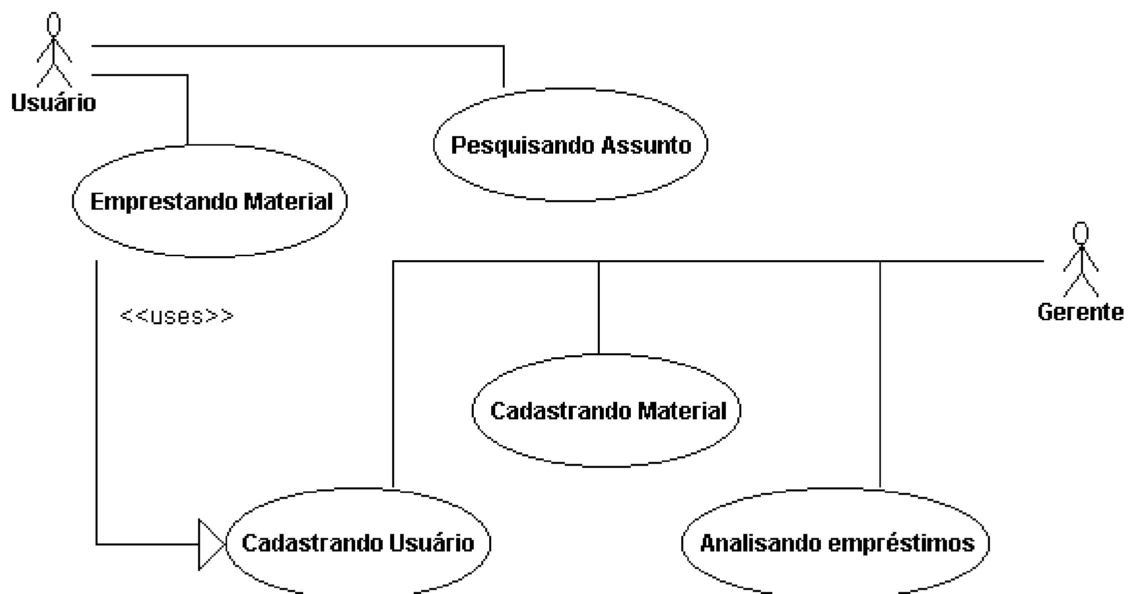


Figura 5.1: Diagrama de casos de uso da biblioteca particular

5.1.1 Organização do experimento

Foram desenvolvidas 3 versões do sistema baseado em componentes utilizando os modelos de componentes COMPOR, EJB e CCM. O desenvolvimento de cada uma das versões esteve sob as mesmas condições em termos de projeto, ou seja, seguiram a mesma modelagem. O diagrama conceitual de componentes pode ser visto na Figura 5.3.

Para que os modelos de componentes pudessem ser comparados e o Comportamento dos sistemas desenvolvidos utilizando cada um dos modelos analisado em termos de evolução de software foi necessário dividir o estudo em duas fases: a fase de construção e a de evolutibilidade do sistema. Para garantir que o entendimento das funcionalidades implementadas e

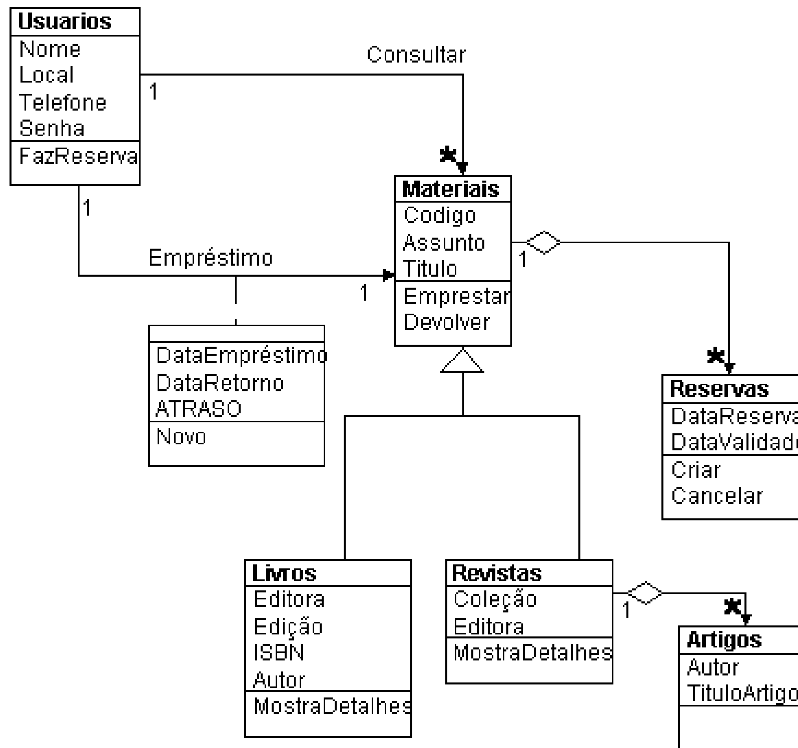


Figura 5.2: Diagrama de classes da biblioteca particular

evitar problemas relacionados à codificação, foi estabelecida uma padronização de projeto.

Na fase de construção a aplicação escolhida foi implementada, primeiramente, utilizando o modelo de componentes COMPOR, em seguida CCM e por último utilizando o EJB. As métricas foram especificadas no arcabouço de avaliação foram aplicadas sobre o código gerado em cada uma das versões. Em seguida, na fase de evolução foram feitas mudanças seguindo o cenário evolutivo. As métricas foram novamente aplicadas sobre o código de cada uma das versões. Os cenários evolutivos escolhidos são ações comuns na evolução baseada em componentes de softwares: adicionar componentes, excluir componentes e atualizar componentes.

5.2 Resultados obtidos

As métricas foram aplicadas sobre o códigos das três versões implementadas na fase de construção e na fase de evolução. A partir das informações coletadas, foi realizada a comparação

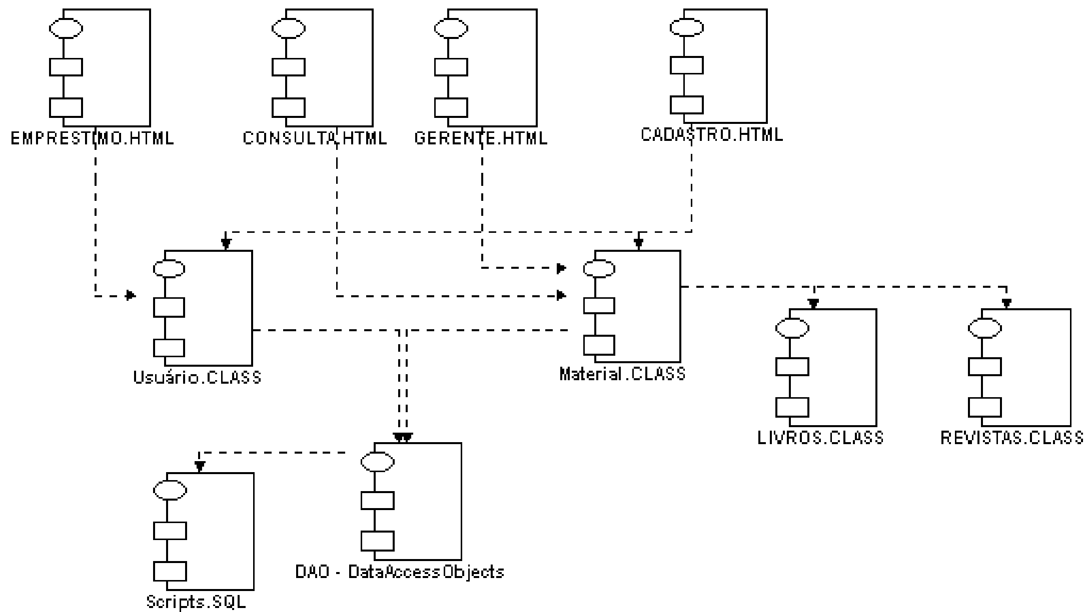


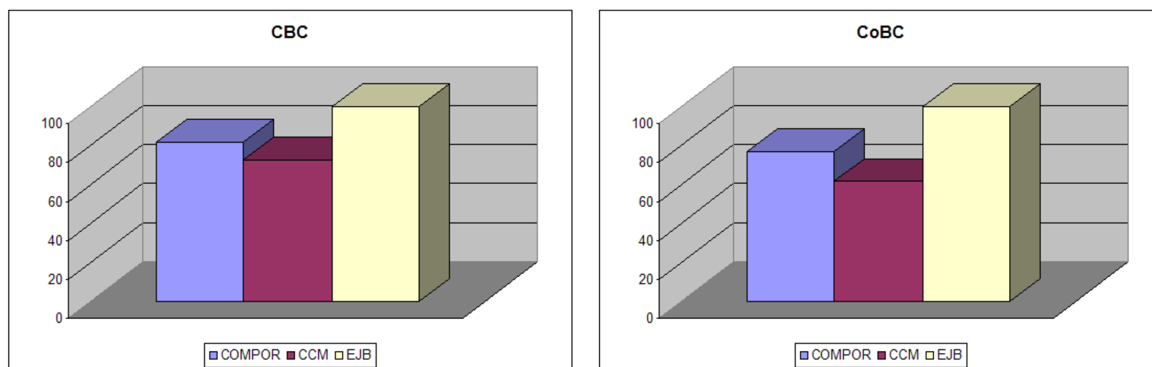
Figura 5.3: Diagrama de componentes da biblioteca particular

entre os modelos de componentes em termos de projeto e evolução de software. Nas seções seguintes são mostrados resultados obtidos das fases de construção e evolução.

5.2.1 Fase de construção

Nessa seção, são apresentados dados coletados na fase de construção. Versões da aplicação foram implementadas utilizando os modelos de componentes COMPOR, CCM e EJB. Como cada modelo de componentes possui uma especificação própria para a construção de componentes, os dados obtidos com a aplicação das métricas do arcabouço de avaliação no projeto e código da aplicação foram colhidos manualmente. Pois, ainda não existe ferramenta específica que dê suporte a aplicação de métricas em componentes.

Na Figura 5.4, são apresentados gráficos com os resultados das métricas de acoplamento, coesão e tamanho. São resultados gerais das versões geradas inicialmente e que ainda não tinham sido submetidas ao processo de evolução.



(a) Resultados da métrica CBC

(b) Resultados da métrica CoBC

Figura 5.4: Gráficos da aplicação das métrica de acoplamento e coesão

Resultado da métrica de acoplamento

A aplicação da métrica de acoplamento mostrou que a versão da aplicação construída utilizando o modelo EJB mostra-se mais fortemente acoplada que as versões utilizando COMPOR e CCM. Estes apresentam a aplicação 18% e 27% menos acopladas, respectivamente, como pode ser visto em 5.4. A justificativa para esse resultado deve-se ao fato de que para construir cada componente de uma aplicação EJB um número maior de interfaces são construídas mantendo o código mais interligado e aumentando, assim, o resultado de acoplamento.

Resultado da métrica de coesão

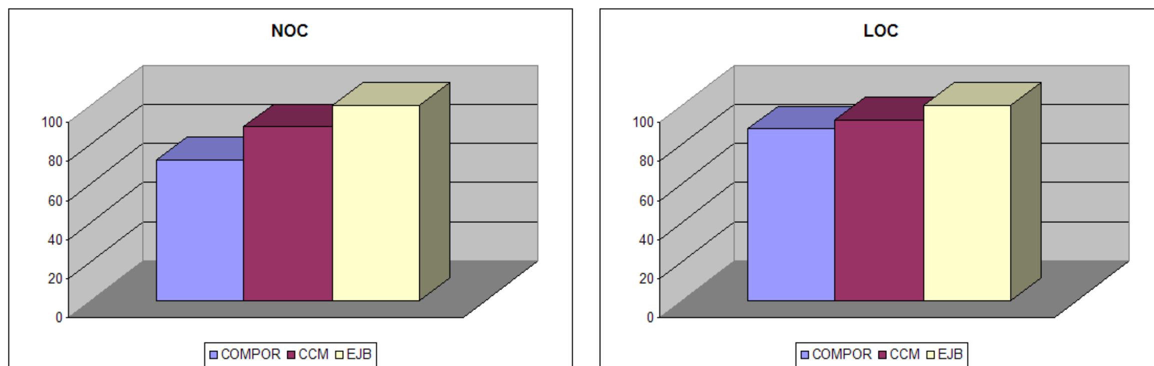
Ainda segundo o gráfico mostrado na Figura 5.4 o melhor resultado apresentado foi da aplicação que utilizou o CCM, 38% melhor que a aplicação do EJB. Já o modelo COMPOR mostrou um resultado 23% melhor que o do EJB.

Observa-se com a aplicação dessa métrica que a versão da aplicação que utilizou EJB é menos coesa que as dos outros modelos. Os resultados apresentados são gerais, ou seja, uma média do ocorrido durante a aplicação da métrica.

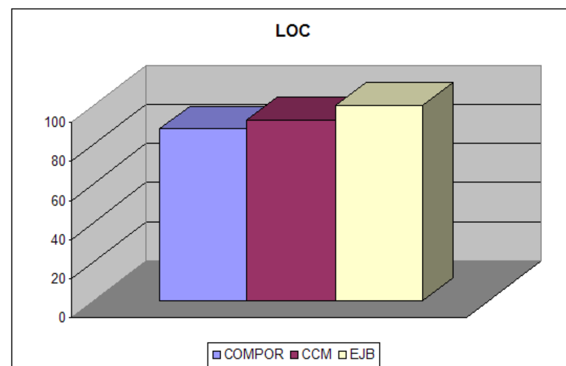
Resultados da métricas de tamanho

Os gráficos da Figura 5.5 apresentam os resultados da aplicação das métricas de tamanho no projeto e código da aplicação da biblioteca pessoal. As métricas de tamanho foram aplicadas

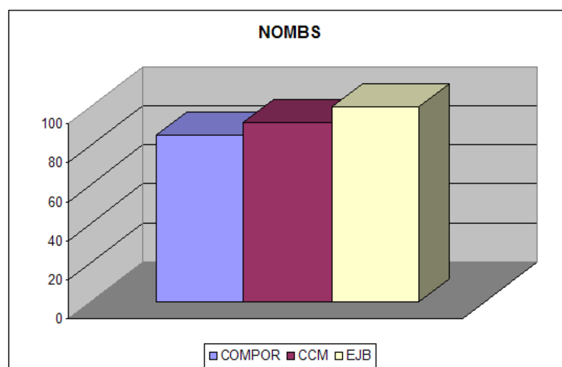
nas três versões da aplicação desenvolvida utilizando os modelos de componentes COMPOR, CCM e EJB.



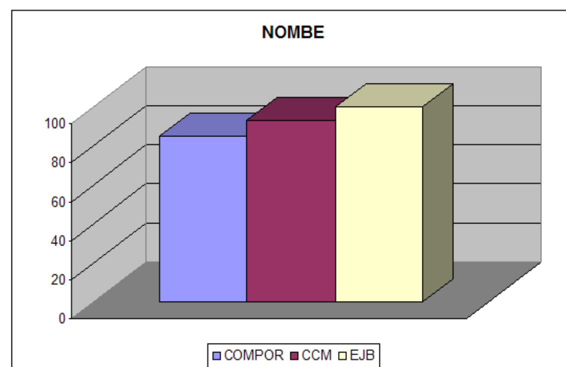
(a) Resultados da métrica NOC



(b) Resultados da métrica LOC



(c) Resultados da métrica NOMBS



(d) Resultados da métrica NOMBE

Figura 5.5: Gráficos da aplicação das métrica de tamanho

Número de componentes - NOC. A aplicação desenvolvida utilizando o modelo de componentes COMPOR apresentou melhores resultados com relação a métrica NOC, necessitando de 28% menos componentes que a aplicação que utilizou o EJB. Para esta métrica EJB obteve os resultados mais altos que os outros modelos. O CCM apresentou um resultado 12% melhor que o EJB.

É provável que estes resultados se devam ao tipo de arquitetura utilizada pelos modelos de componentes estudados. Os resultados mostram que CCM e EJB precisam de um número maior de interfaces e objetos para construir um componentes que o modelo COMPOR.

Número de linhas de código - LOC. A métrica de tamanho LOC foi aplicada sobre as três versões e os melhores resultados obtidos em termos de linhas de código foi da aplicação desenvolvida utilizando o modelo de componentes COMPOR. O segundo melhor resultado

foi do CCM seguido do modelo EJB. COMPOR e CCM tiveram resultados 12% e 9% melhores que EJB.

Número de métodos por serviço - NOMBS. A aplicação da métrica de tamanho NOMBS que mede o número de métodos criados para um serviço oferecido no sistema, apresentou melhores resultados para a aplicação desenvolvida utilizando o COMPOR, 15% melhor que o EJB. A aplicação que utilizou o modelo EJB obteve, por sua vez, um resultado mais próximo da que utilizou o modelo CCM que apresentou resultado 8% melhor que o modelo EJB.

Número de métodos por eventos - NOMBE. A aplicação da métrica de tamanho NOMBE que mede o número de métodos criados para um evento anunciado no sistema, apresentou melhores resultados para a aplicação desenvolvida utilizando CCM e o COMPOR, sendo 7% e 15% melhor que o EJB, respectivamente.

5.2.2 Resultados obtidos na fase de evolução

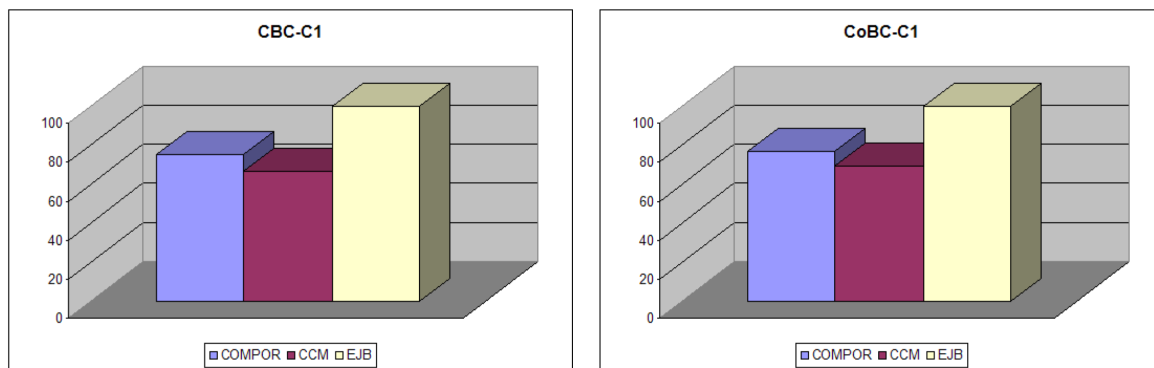
Nesta seção, discute-se cada cenário de evolução e os respectivos resultados das versões da aplicação implementada utilizando os modelos de componentes do estudo: COMPOR, CCM e EJB. A evolução do sistema foi definida seguindo as formas mais comuns de mudanças nos sistemas que são: adicionar, atualizar e excluir componentes do sistema. Essas possíveis maneiras de evoluir foram organizadas em três cenários que são apresentados nas subseções que seguem.

Para manter a organização do experimento, as mesmas condições de evolução foram fornecidas para cada um dos modelos de componentes. Sendo assim, as tarefas realizadas dentro de cada cenário de evolução foram as mesmas nas três versões implementadas na fase anterior, de construção. Em seguida, as métricas definidas no arcabouço de medição foram novamente aplicadas sobre o código e observado o Comportamento da aplicação, e conseqüentemente, dos modelos utilizados para implementá-las. Novos dados foram gerados e os modelos de componentes, comparados entre si e com os dados obtidos na fase de construção. As seções seguintes apresentam em detalhes como foi feita a evolução do sistema e os dados gerados.

Cenário 1 - Adicionar um componente ao sistema

O cenário 1 criado para evoluir o sistema é caracterizado pela adição de um novo com-

ponente à aplicação em cada uma das versões. O o modelo conceitual de componentes foi tomando como base para as mudanças. Inicialmente, foi adicionado um novo tipo de Material ao sistema da biblioteca pessoal. Depois foi acrescentado um novo ator, o Administrador, ao sistema. Os resultados obtidos são apresentados nos gráficos que seguem na Figura 5.6. Para coletar esses resultados os experimentos são repetidos e é feita uma média dos valores até chegar aos apresentados nos gráficos.



(a) Resultados da métrica CBC (b) Resultados da métrica CoBC

Figura 5.6: Gráficos da aplicação das métrica de acoplamento e coesão para o cenário 1

Resultado da métrica de acoplamento

A aplicação da métrica de CBC sobre as três versões do sistema da biblioteca após o processo de evolução definido para o cenário 1, apresentou poucas diferenças com relação ao acoplamento, em comparação com os dados gerados na fase de construção.

Comparando-se a implementação da fase de construção, apenas a aplicação desenvolvida usando o modelo EJB aumentou um pouco o valor atribuído a esta métrica, os valores para COMPOR e CCM foram os mesmo da fase anterior no cenário 1. Pode-se, então, considerar que os valores se mantiveram em relação aos colhidos na fase anterior. A diferença dos valores coletados na fase de evolução dos modelos foram: COMPOR e CCM tiveram resultados 25% e 33% melhores que modelo EJB, em relação a acoplamento.

Este fato confirma a viabilidade da métrica de acoplamento entre componentes para cenários evolutivos de software. Pois, foi mantida a proporção entre os valores obtidos na fase de construção e na fase de evolução.

Resultado da métrica de coesão

A métrica de coesão foi aplicada novamente sobre a aplicação desenvolvidas utilizando cada um dos modelos. Para o cenário 1 de evolução a diferença foi discreta. Apenas a aplicação que utilizou CCM apresentou uma alteração negativa no resultado. A aplicação que utilizou COMPOR e EJB manteve os mesmos valores da fase de construção. Os valores coletados na fase de evolução dos modelos foram: COMPOR 23% mais coeso que EJB e CCM 31% mais coeso que o modelo EJB. Ou seja, mesmo mantendo os mesmos valores colhidos na fase de construção, a aplicação que utilizou o modelo de componentes EJB ainda mostra uma maior falta de coesão entre seus componentes, quando comparado aos outros dois modelos.

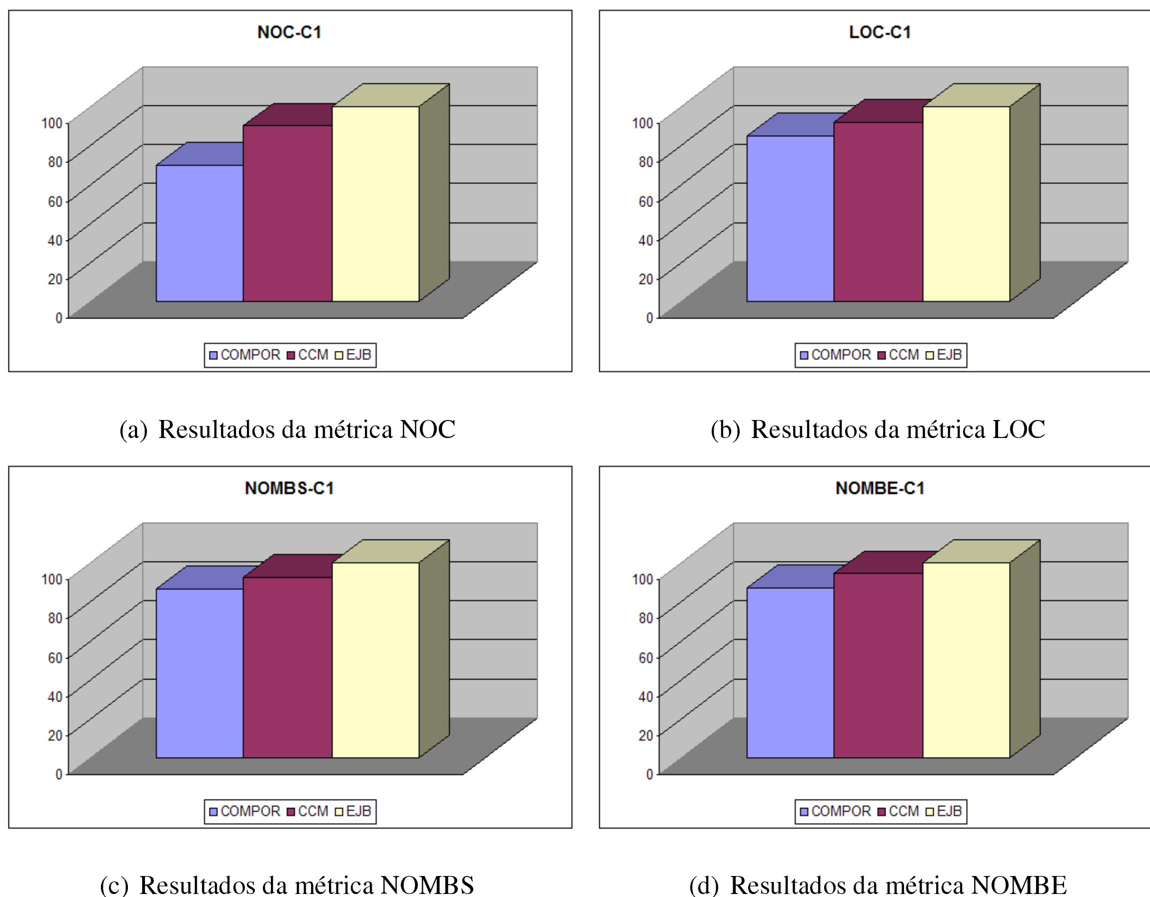
Resultados da métricas de tamanho

Os gráficos da Figura 5.7 apresentam os resultados da aplicação das métricas de tamanho no código da aplicação da biblioteca pessoal após o processo de evolução se software nas três versões implementadas.

Número de componentes - NOC. A três versões da aplicação apresentaram aumento nos valores coletado para a métrica de NOC após a evolução definida no cenário 1. A proporção entre os valores se manteve com resultados semelhantes aos apresentados na fase de construção, apesar do aumento nos valores absolutos. A aplicação desenvolvida utilizando o modelo de componentes COMPOR apresentou melhores resultados com relação à métrica NOC, necessitando de 30% menos componentes que a aplicação que utilizou o EJB. Para esta métrica EJB obteve os resultados mais altos que os outros modelos. O CCM apresentou um resultado 10% melhor que o EJB.

Número de linhas de código - LOC. A métrica de tamanho LOC foi aplicada sobre as três versões e todas mostraram aumento no número de linhas de código no cenário 1 de evolução. O resultado apresentado foi: COMPOR e CCM tiveram resultados 15% e 8% melhores que EJB, respectivamente.

Número de métodos por serviço - NOMBS. Os resultados para o cenário 1 de evolução foi muito próximo do já observado na fase de construção, para a métrica NOMBS. A aplicação desenvolvida utilizando o COMPOR foi 13% melhor que o EJB . A aplicação que utilizou o modelo EJB obteve um resultado mais próximo da que utilizou o modelo CCM



(a) Resultados da métrica NOC (b) Resultados da métrica LOC

(c) Resultados da métrica NOMBS (d) Resultados da métrica NOMBE

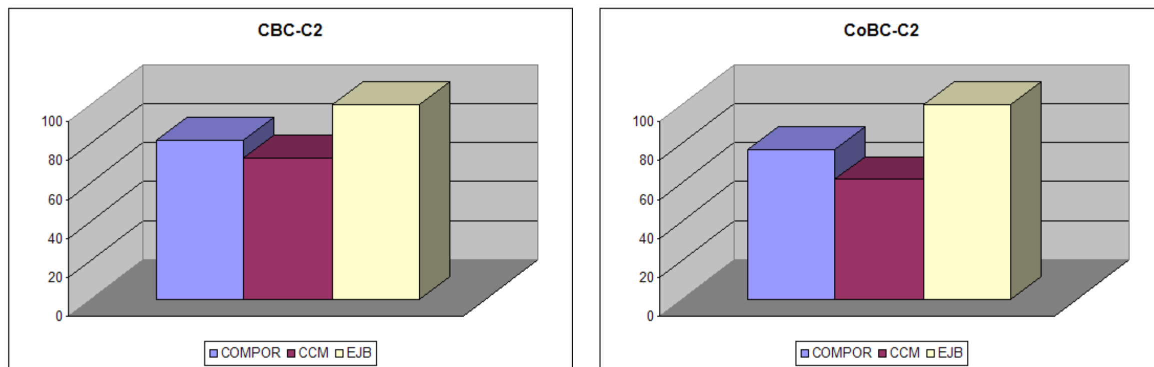
Figura 5.7: Gráficos da aplicação das métricas de tamanho para o cenário 1

que apresentou resultado 7% melhor que o modelo EJB.

Número de métodos por evento - NOMBE. Para o cenário 1 de evolução os resultados apresentaram uma pequena variação, comparados aos resultados obtidos na fase de construção. A aplicação que utilizou o modelo COMPOR e CCM teve resultados 15% e 7% menores que a que utilizou o modelo EJB.

Cenário 2 - Atualizar um componente do sistema

O cenário 2 foi criado para evoluir o sistema e é caracterizado pela atualização de um componente da aplicação em cada uma das versões implementadas. A primeira atualização definida foi mudar as funções atribuídas ao ator Gerente. Depois foi efetuada uma mudança na atividade Empréstimo do sistema. Os resultados obtidos são apresentados nos gráficos que seguem na Figura 5.8. Para coletar esses resultados, os experimentos são repetidos e é feita uma média dos valores até chegar aos resultados apresentados nos gráficos.



(a) Resultados da métrica CBC

(b) Resultados da métrica CoBC

Figura 5.8: Gráficos da aplicação das métrica de acoplamento e coe são para o cenário 2

Resultado da métrica de acoplamento

A aplicação da métrica de CBC sobre as versões do sistema após o processo de evolução definido para o cenário 2 não apresentou diferenças consideráveis para a métrica de acoplamento. Os valores para a aplicação desenvolvida utilizando os 3 modelos se manteve constante com relação aos resultados obtidos na fase de construção. A proporção entre os valores também foi mantida nesta fase: COMPOR e CCM apresentaram números 12% e 27% menores que o modelo EJB.

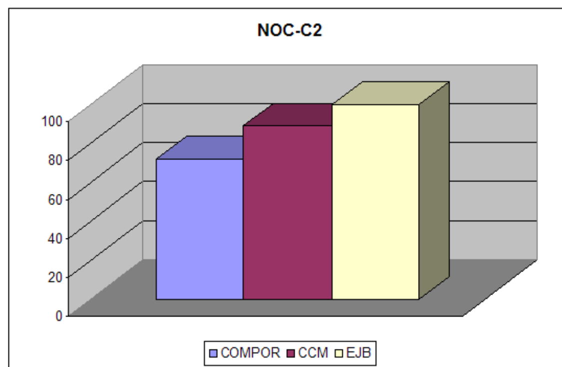
Resultado da métrica de coesão

A aplicação da métrica de CoBC sobre as versões do sistema após o processo de evolução definido para o cenário 2 apresentou valores constantes comparado a fase anterior, de construção. Os resultados apresentados na fase de evolução para o cenário 2 foram: a aplicação que utilizou o CCM mostrou-se 38% melhor que a aplicação do EJB. O modelo COMPOR mostrou um resultado 23% melhor que EJB.

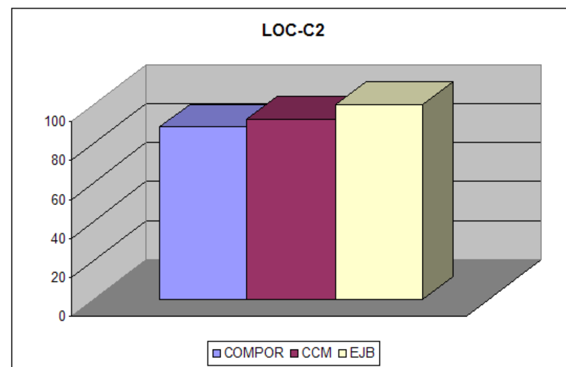
Resultados da métricas de tamanho

Os gráficos da Figura 5.9 apresentam os resultados da aplicação das métricas de tamanho no código da aplicação da biblioteca pessoal após o processo de evolução se software nas três versões implementadas.

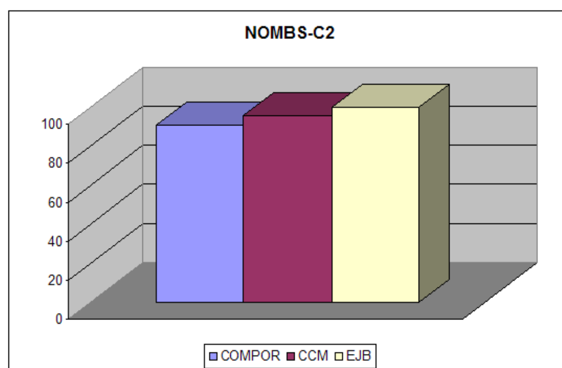
Número de componentes - NOC. Para o cenário 2 de evolução a aplicação da métrica NOC praticamente não apresentou mudanças. Os valores e proporções foram semelhantes



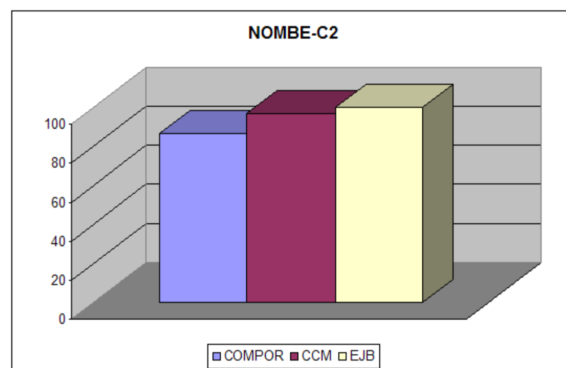
(a) Resultados da métrica NOC



(b) Resultados da métrica LOC



(c) Resultados da métrica NOMBS



(d) Resultados da métrica NOMBE

Figura 5.9: Gráficos da aplicação das métricas de tamanho para o cenário 2

aos coletados na fase de construção. A aplicação que utilizou os modelos de componentes COMPOR e CCM apresentou valores 28% e 12% menores que a aplicação que utilizou o modelo EJB.

Número de linhas de código - LOC. A métrica de tamanho LOC foi aplicada e resultados obtidos mostraram uma diminuição no número de linhas de código para o cenário 2. Mesmo com a alteração númerica os valores ainda foram compatíveis com os coletados na fase de construção. A aplicação que utilizou o modelo COMPOR e CCM tiveram resultados 12% e 8% melhores que EJB, respectivamente.

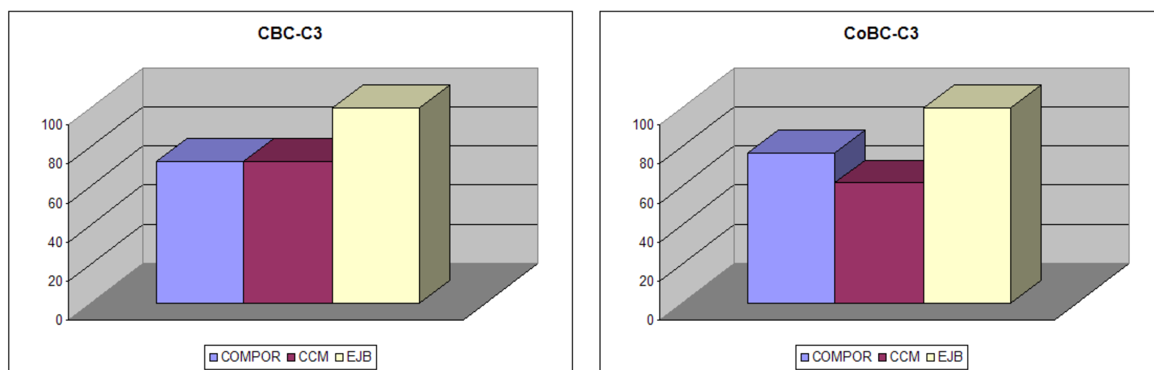
Número de métodos por serviço - NOMBS. No cenário 2 de evolução, a métrica NOMBS obteve resultados proporcionalmente semelhantes aos da fase de construção. A aplicação que utilizou o CCM, obteve resoltados 5% menores que a que utilizou o modelo EJB. A que utilizou o modelo COMPOR foi 9% menores.

Número de métodos por evento - NOMBE. Resultados semelhantes também foram

obtidos para a métrica NOMBE. A aplicação que utilizou o modelo EJB tive um número de eventos um pouco maior que a que utilizou os outros modelos, sendo valores 13% e 3% maiores que COMPOR e CCM.

Cenário 3 - Excluir um componente do sistema

O cenário 3 criado para evoluir o sistema é caracterizado pela exclusão de um componente da aplicação em cada uma das versões implementadas. O experimento foi realizado excluindo componentes do sistema e observando os resultados obtidos, em seguida uma média dos resultados é extraída e comparados os resultados de cada modelo. Um componente excluído do sistema foi o Livros que está ligado ao outro componente Material. Depois foi efetuada mais uma mudança, excluindo o componente Gerente. Os resultados obtidos são apresentados nos gráficos ilustrados na Figura 5.10.



(a) Resultados da métrica CBC (b) Resultados da métrica CoBC

Figura 5.10: Gráficos da aplicação das métrica de acoplamento e coesão para o cenário 3

Resultado da métrica de acoplamento

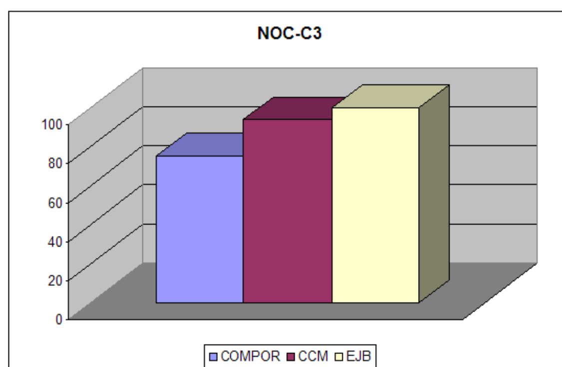
A aplicação da métrica de CBC sobre as versões do sistema após o processo de evolução definido para o cenário 3 apresentou diferenças muito pequenas comparado com os dados gerados na fase de construção. Pode-se, assim, considerar que os valores obtidos foram semelhantes aos anteriores. A aplicação que utilizou o modelo COMPOR e CCM mostraram um resultado 27% menor que a aplicação que utilizaram o modelo EJB. A diferença entre os modelos na fase de evolução se manteve proporcional a da fase de construção.

Resultado da métrica de coesão

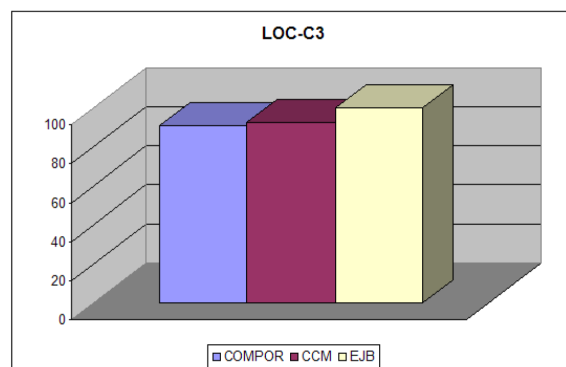
Assim como no cenário 2, a aplicação da métrica de CoBC sobre as versões do sistema após o processo de evolução definido para o cenário 3 apresentou valores constantes comparados à fase de construção. Os resultados apresentados na fase de evolução para o cenário 3 foram: a aplicação que utilizou o CCM mostrou-se 38% melhor que a aplicação do EJB. O modelo COMPOR mostrou um resultado 23% melhor que o do EJB.

Resultados da métricas de tamanho

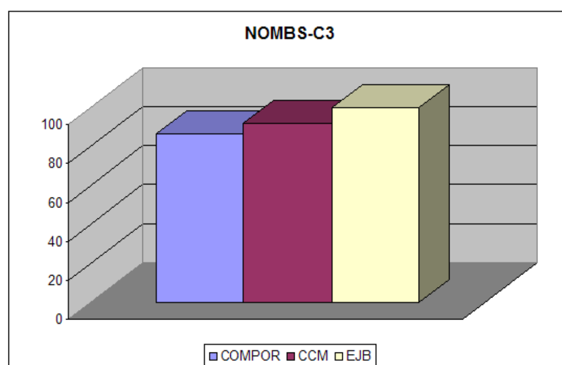
Os gráficos da Figura 5.11 apresentam os resultados da aplicação das métricas de tamanho no código da aplicação da biblioteca pessoal após o processo de evolução de software nas três versões implementadas.



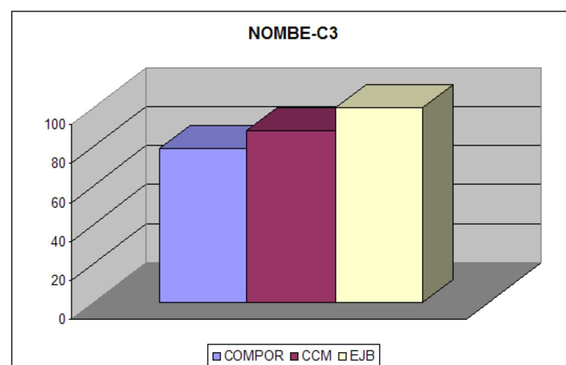
(a) Resultados da métrica NOC



(b) Resultados da métrica LOC



(c) Resultados da métrica NOMBS



(d) Resultados da métrica NOMBE

Figura 5.11: Gráficos da aplicação das métricas de tamanho para o cenário 3

Número de componentes - NOC. Para o cenário 3 de evolução, a aplicação da métrica NOC apresentou mudanças significativas. A aplicação que utilizou os modelos de com-

ponentes COMPOR e CCM apresentaram valores 25% e 6% menores que a aplicação que utilizou o modelo EJB.

Número de linhas de código - LOC. A métrica de tamanho LOC foi aplicada sobre as três versões e para o cenário 3 observou-se que houve uma diminuição significativa no número de linhas de código nas três versões implementadas. Notou-se também, que os valores percentuais se aproximaram quando comparados aos da fase de construção. O resultado apresentado foi: a aplicação que utiliza o modelo COMPOR e CCM tiveram resultados 9% e 8% melhores que EJB, respectivamente.

Número de métodos por serviço - NOMBS. No cenário 3 de evolução houve alteração dos valores para a métrica de NOMBS, porém a proporção dos valores entre os modelos se manteve. A aplicação que utilizou o CCM obteve resultados 13% e a que utilizou o COMPOR, 8% menores que a que utilizou o modelo EJB.

Número de métodos por evento - NOMBE. Houve mudanças consideráveis no resultado da métrica NOMBE para o cenário 3 de evolução. A proporção entre os resultados da fase de evolução e de construção se manteve constante. A aplicação que utilizou o CCM obteve resultados 21% e a que utilizou o COMPOR, 12% menores que a que utilizou o modelo EJB.

5.2.3 Discussões sobre os Resultados do Estudo Experimental

Com os resultados da fase de construção, observou-se que para as métricas de tamanho, a aplicação que utilizou o modelo COMPOR obteve resultados melhores. A maior diferença foi observada na métrica de número de componentes, onde a aplicação que utiliza o modelo de componentes EJB obteve valores mais elevados. A métrica de "número de métodos por serviço" mostrou que, apesar das diferenças encontradas nas outras métricas de tamanho, os valores obtidos, para a aplicação que utilizou os três modelos de componentes, foram muito próximos. A métrica "número de métodos por eventos" também apresentou uma diferença mínima entre os resultados.

Ainda na fase de construção, as métricas de acoplamento e coesão apresentaram melhores resultados para a aplicação desenvolvida utilizando o CCM. A aplicação que utilizou o modelo COMPOR teve resultados próximos aos da que utilizou o modelo CCM. A maior diferença foi do modelo EJB, com resultados mais elevados.

Os resultados da fase de evolução confirmaram os já conhecidos da fase de construção. No cenário 1 de evolução, em que um componente foi adicionado ao sistema, o comportamento dos modelos se manteve constante. Para as métricas de acoplamento e coesão, a aplicação que utilizou o CCM obtive melhores resultados, já para as métricas de tamanho a aplicação que utilizou o modelo COMPOR continuou com resultados mais favoráveis. Observou-se que a aplicação que utilizou o modelo EJB, mesmo com um aumento nos valores devido ao cenário estabelecido, teve um bom resultado mostrando uma variação menor entre os valores colhidos na fase de construção e na fase de evolução, se comparado aos outros modelos.

O cenário 2, em que foram feitas atualizações em componentes do sistema também apresentou resultados muito semelhantes aos observados na fase de construção. Os gráficos gerados tiveram algumas alterações, porém não houve grande mudança na fase de evolução se comparada à fase de construção. O cenário 3, em que componentes do sistema foram excluídos, mostrou resultados semelhantes nas fases de evolução e construção. Existe uma margem de diferença percentual para os valores colhidos em todo o experimento e, assim, considera-se que os resultados da fase de evolução se mantiveram próximos aos já observados na fase de construção.

Capítulo 6

Trabalhos Relacionados

A maioria dos estudos sobre software baseados em componentes se concentraram na análise qualitativa da construção do componente e de suas funcionalidades. Menos explorados, no entanto, têm sido os estudos sobre os requisitos não funcionais de software, tais como: usabilidade, flexibilidade e modificabilidade. O principal motivo para isso é a dificuldade de se medir tais requisitos. A seguir, são apresentados alguns trabalhos relacionados.

6.1 Manutenibilidade e Reusabilidade de Software Orientado a Aspectos

Em [San04], foi apresentado um arcabouço cujo objetivo principal é avaliar manutenibilidade e reusabilidade de software orientado a aspectos. As métricas são aplicadas no projeto e no código de software orientado a aspectos. Elas medem quatro atributos de qualidade: separação de concerns, acoplamento, coesão e tamanho.

O arcabouço de avaliação foi utilizado no contexto de dois estudos experimentais de domínios distintos, com características, níveis de controle e níveis de complexidade diferentes. Esses estudos serviram como avaliações iniciais da utilidade e usabilidade do conjunto de métricas e do modelo de qualidade. O primeiro estudo experimental comparou uma abordagem orientada a objetos com uma abordagem orientada a aspectos para o projeto e implementação de um sistema multi-agentes. O segundo estudo envolveu a aplicação do framework proposto para avaliar as implementações orientadas a aspectos e orientadas a objetos

de alguns padrões de projeto de software.

6.2 Medição em Componentes Caixa-preta

Em [WYF02], Washizaki deu ênfase aos componentes do tipo caixa-preta. Foram apresentadas métricas para a medição da reusabilidade de componentes caixa-preta. Estas métricas são baseadas em informações limitadas que podem ser obtidas a partir dos resultados de estudos experimentais sobre os componentes.

Um conjunto de métricas foi definido para a medição da compreensibilidade, adaptabilidade e portabilidade dos componentes. As métricas indicam uma classificação de reúso em que estão os componentes em um produto de software. As métricas somente podem ser utilizadas nas situações em que o objetivo do componente, sobre o qual serão aplicadas, já tenha sido definido. Como resultado dos experimentos, observou-se que estas métricas podem identificar componentes caixa-preta com alto grau de reusabilidade.

6.3 *Component Quality Model* (CQM)

Em [MABC+03], métricas também são usadas, porém, para avaliar a qualidade dos componentes e até de sistemas, através de um modelo chamado *CQM (Component Quality Model)*. Com isso, pretende-se ajudar na escolha entre os diversos componentes existentes em um repositório.

Para compor um sistema baseado em componentes contidos em um repositório, os usuários selecionam um ou mais componentes do repositório e integram esses componentes aos seus sistemas. Essa abordagem possibilita uma economia de tempo e custo para a construção dos sistemas. Entretanto, para construir componentes ou utilizar os componentes de um repositório é necessário um prévio conhecimento das características e da qualidade do que se pretende desenvolver ou utilizar.

Métricas que possam medir a qualidade dos componentes de um repositório adquirem grande importância para a construção de sistemas. O modelo de qualidade de componente CQM foi apresentado para proporcionar uma classificação de diversas métricas para componentes de sistema. Assim, a escolha dos componentes contidos em um repositório para a

construção de um sistema poderá ter como critério a classificação, em termo de qualidade dos componentes, fornecida pelo modelo apresentado.

6.4 *Interface Complexity Metric* (ICM)

Com a necessidade de melhoria da qualidade e da produtividade dentro das organizações que adotam o desenvolvimento baseado e componentes, em [GG04] foi apresentada uma técnica para medir a complexidade de um componente de software com base no modelo de interface do componente. Isto inclui assinatura de interface, restrições de interface, empacotamento e configuração.

Com base nos valores destas métricas, a complexidade de componentes do software pode ser mantida em limites razoáveis. Assim, os componentes de software podem ter uma melhora na qualidade e produtividade dos sistemas.

6.5 Validação Cruzada de Métricas

Em [Gou05], foi apresentada uma técnica que facilita a replicação de experiências de validação cruzada de métricas para o desenvolvimento baseado em componentes. A aplicação da técnica é ilustrada com uma experiência de validação cruzada de um conjunto de métricas publicadas na literatura que se destinam a aferir a facilidade de reutilização em sistemas baseados em componentes.

As métricas a validar foram originalmente propostas usando uma notação parcialmente informal, que combinava fórmulas matemáticas com descrições informais dos elementos nelas usadas. Esta técnica permite obter uma definição formal, portável e executável das métricas, que pode ser usada em experiências de validação cruzada das mesmas.

6.6 Métricas de Reusabilidade

Em [Gon06], é realizada uma avaliação de aplicações existentes, com a finalidade de identificar componentes de software reusáveis por meio da aplicação de métricas de reusabilidade do modelo de reúso e das métricas que quantificam os atributos deste modelo. Levando em

consideração as características deste modelo e a interpretação dos resultados da coleta, os componentes que apresentaram um alto grau de abstração, um baixo nível de instabilidade e uma distância próxima da seqüência principal podem ser considerados fortes candidatos para o reúso.

Componentes de software menos complexos tornam-se mais reusáveis, segundo o modelo adotado. Porém, os resultados das métricas para medição da complexidade dos componentes ficou muito abaixo dos valores de referência definidos para interpretação. Isto porque eles foram definidos considerando todo o componente, e identificou-se que as métricas, quando coletadas, melhor se aplicariam para análise mais específica de métodos e classes isoladamente, e não para todo o componente.

6.7 Discussões sobre os trabalhos relacionados

Neste trabalho, a proposta foi utilizar métricas para a medição do projeto e código de sistemas baseados em componentes sob o aspecto de evolutibilidade. Para viabilizar o estudo, um arcabouço de medição foi proposto. Este arcabouço de medição é composto por dois elementos básicos: um modelo de qualidade e um conjunto de métricas de softwares.

As principais diferenças para os trabalhos relacionados existentes é que os estudos experimentais se basearam em análises de cenários evolutivos, apoiadas pela aplicação de métricas de projeto e código. O conjunto de métricas proposto se baseou em outras métricas já existentes, tais como acoplamento e coesão entre componentes e a métrica de tamanho.

Portanto, as principais características do trabalho apresentado nesta dissertação são listadas a seguir.

- Os estudos experimentais apresentados aqui foram baseados em análises quantitativas, apoiadas pela aplicação de métricas de projeto e código.

- As métricas propostas medem atributos de software bem conhecidos da engenharia de software e o conjunto de métricas proposto se baseou em outras métricas já existentes, usadas e validadas.

- O arcabouço de medição é composto por elementos que separadamente podem ser reusados em outros projetos. O modelo de qualidade que tem como objetivo o atributo

de evolutibilidade, pode ser usado em outros trabalhos em que se precise saber sobre este atributo. E, o conjunto de métricas também pode ser usado em outros estudos.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Conclusões

A aplicação e uso de tecnologias de componentes para o desenvolvimento de software estão diretamente relacionados à visão de componentes como unidades de código. Por ser uma visão que recebe maior ênfase atualmente, ela é considerada, neste trabalho, como sendo o aspecto mais maduro do DBC. Neste sentido as definições de [BW98; Szy99] ressaltam este aspecto, onde se preocupam com componentes como sendo a abstração seguinte a funções, módulos e classes.

Os componentes devem ser vistos como unidades de software que estão interligados, e assim, constroem sistemas completos. A forma como essas interconexões acontecem é definida pela arquitetura de software. A visão de componentes segundo a perspectiva de abstrações arquiteturais, expressa regras de projeto que têm a forma de um modelo de componente, ou de um conjunto de padrões e convenções com as quais os componentes devem estar em conformidade. A qualidade do software baseado em componentes está vinculada a essas regras definidas pelo modelo de componentes utilizado.

Na busca pela qualidade nos sistemas, muitos estudos sobre componentes de software têm sido realizados. A maioria analisa aspectos funcionais dos sistemas e a qualidade do componente. Porém, os atributos não funcionais como evolução, tem sido pouco explorados. Um motivo para isso é que, em geral, esses atributos só podem ser analisados no final do processo de desenvolvimento.

A medição de software tornou-se uma parte tão importante quanto necessária no processo

de desenvolvimento de software. Modelos de qualidade como o Modelo de McCall [McC77] e o Modelo de Boehm [Boe73] foram propostos de forma a determinar exatamente quais características deveriam ser medidas para a obtenção de um padrão de qualidade. Para auxiliar a medição de qualquer tipo de produto ou processo, cujo foco seja um atributo de qualidade, a abordagem GQM é bastante apropriada.

Nesse contexto, neste trabalho apresentou-se um estudo experimental comparativo dos modelos de componentes EJB, CCM e CORBA sob o aspecto de evolução de software. O arcabouço de medição se baseou em modelos de qualidade existentes e na abordagem GQM. Foi realizada uma medição do atributo evolução através de atributos internos como acoplamento, coesão e tamanho que estão relacionados com o atributo de evolução. Para viabilizar essa medição, métricas de software relacionadas com os atributos internos foram definidas, tornando possível comparar o projeto e implementação de aplicações desenvolvidas utilizando os modelos de componentes citados.

Objetivo principal dos experimentos foi entender as diferenças encontradas entre as aplicações e, conseqüentemente, entre os modelos de componentes estudados. Os cenários de evolução criados, tornaram reais as possibilidades de mudanças que podem ocorrer no processo de desenvolvimento de um software. Foi possível observar o comportamento de cada uma das versões que utilizaram os modelos de componentes diante de cenários de evolução como: adição, atualização e exclusão de componentes do sistema. Além do objetivo já citado, a realização desse estudo experimental serviu como uma primeira avaliação da utilidade do conjunto de métricas e do modelo de qualidade proposto.

Os dados gerais mostraram que, para métricas de acoplamento e coesão, as aplicações desenvolvidas utilizando o CCM obtiveram resultados mais favoráveis. Já para as métricas de tamanho, o modelo de componentes COMPOR apresentou melhores resultados que os outros modelos estudados.

7.2 Trabalhos futuros

O trabalho apresentado nesta dissertação pode ser estendido com a realização de algumas atividades descritas a seguir.

- Realizar outros experimentos utilizando o arcabouço de avaliação proposto.

- Desenvolvimento de uma ferramenta para automatizar a aplicação das métricas propostas.
- Realizar estudos de como as métricas podem ser usadas em outras partes do projeto, como a verificação da qualidade de projetos baseados em componentes, e também para apoiar a realização de refatoramento de sistemas baseados em componentes.

Bibliografia

[APP^{+}06] Hyggo Oliveira Almeida, Angelo Perkusich, Rodrigo Barros Paes, Evandro Barros Costa, Glauber Vinícius Ventura de Melo Ferreira, Emerson Cavalcante Loureiro Filho e Loreno Oliveira. A Component Based Infrastructure to Develop Software Supporting Dynamic Unanticipated Evolution. *Anais do XX Simpósio Brasileiro de Engenharia de Software*, pages 145–160, Florianópolis, SC, Brasil, 2006.

[Bac00] Bass L. Buhman C. Comella-Dorda S. Long F. Robert J. Seacord R. Wallnau K. Bachmann, F. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. 2000.

[BCR02] Victor R. Basili, Gianluigi Caldiera e H. Dieter Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1(2):578–583, 2002.

[Boe73] J. R. Brown H. Kaspar M. Lipow G. J. MacLeod M. J. Merritt Boehm, B. W. Characteristics of Software Quality. Technical report, TRW Series of Software Technology, North Holland, Dezembro 1973.

[Boe01] Basili V. Boehm, B. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[Bro95] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[BW98] Alan W. Brown and Curt C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[CK94] S. Chidamber e C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Crn01] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. *Software Focus*, volume 4, pages 127–133. Wiley, Dezembro 2001.

[Fen97] Pfleeger S. Fenton, N. *Software Metrics: A Rigorous and Practical Approach*. London: PWS, 1997.

[GG04] Nasib S. Gill e P. S. Grover. Few Important Considerations for Deriving Interface Complexity Metric for Component-based Systems. *SIGSOFT Softw. Eng. Notes*, 29(2):4, 2004.

[Gon06] Oliveira T. Oliveira K. Gonçalves, J. Métricas de Reusabilidade para Componentes de Softwares. *VI Workshop de Sistemas Basesdos em Componentes*, 2006.

[Gou05] Brito F. Abreu Goulão, M. Cross-validation of metrics for software components. *Revista IEEE América Latina*, volume 3, 2005.

[GSF+05] Alessandro Garcia, Cláudio Santanna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena e Arndt Von Staa. Modularizing Design Patterns With Aspects: A Quantitative Study. *LNCS Transactions on Aspect-Oriented Software Development (TAOSD'05)*, pages 36–74, USA, 2005.

[HC01] G. T. Heineman e W. T. Councill. *Component-Based Software Engineering - Putting the Pieces Together*. AddisonWesley, 2001.

[KNMB02] Gunter Kniesel, Joost Noppen, Tom Mens e Jim Buckley. First International Workshop on Unanticipated Software Evolution. *ECOOP2002 Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag, 2002.

[MABC+03] J. Martín-Albo, M.F. Bertoa, C. Calero, A. Vallecillo, A. Cechich e M. Piattini. CQM: A Software Component Metric Classification Model. *7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2003.

[McC77] Richards P. Walters G. McCALL, J. *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality*, volume 1. Springfield, 1977.

[Mic99] Sun Microsystems. Enterprise JavaBeans Specification. urlhttp://java.sun.com/products/ejb, 1999. Acessado em maio de 2007.

[OMG05] OMG. CORBA Component Model Joint Revised Submission. urlhttp://www.omg.org, 2005. Acessado em maio de 2007.

[Pap01] Edwards J. Papaioannou, T. Building Agile Systems with Mobile Code. *Journal of Autonomous Agents and Multi-Agent Systems*, pages 293–310, 2001.

[Pri00] Herbert J. Price, A. Técnicas de Teste de Software Orientado a Objetos. *Revista de Informática Teórica e Aplicada do Instituto de Informática da Universidade Federal do Rio Grande do Sul*, 6(1), 2000.

[Rub92] H. Rubin. The Making Measurement Happen Workshop. *Proceedings of the 3rd International Conference on Applications of Software Measurements*, pages 15–19, 1992.

[Sam97] Jazayeri M. Klosch R. Trausmuth G. Sametinger, J. Software Engineering with Reusable Components. *Springer Verlag*, page 275, New York, 1997.

[San04] Cláudio Nogueira Santanna. Manutenibilidade e Reusabilidade de Software Orientado a Aspectos: Um Framework de Avaliação. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, 2004.

[Sch04] Santos G. Montoni M. Rocha A. Schneider, L. Uma Abordagem para Medição e análise em Projetos de Desenvolvimento de Software. *III Simpósio Brasileiro de Qualidade de Software*, pages 343–353, Brasília, Brasil, 2004.

[SGC+03] Cláudio Santanna, Alessandro Garcia, Christina Chavez, Carlos Lucena e Arndt Von Staa. On The Reuse and Maintenance Of Aspect-Oriented Software: An Assessment Framework. *Simpósio Brasileiro de Engenharia de Software*, pages 10–11, Manaus, AM, Brasil, Outubro 2003.

[Som01] I. Sommerville. *Software Engineering*. Addison-Wesley, England, 2001.

[Szy99] Clemens Szypersky. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[Tho98] Anne Thomas. Enterprise JavaBeans Technology. urlhttp://java.sun.com/products/ejb/white_paper.html, 1998. Acessado em maio de 2007.

[Tra06] Araujo M. Travassos, G. Estudos Experimentais em Manutenção de Software: Observando Decaimento de Software Através de Modelos Dinâmicos. *SBQS/WMSWM - Workshop Manutencao de Software Moderna*, Vila Velha, Brasil, 2006.

[Wer00] C. Werner. Desenvolvimento Baseado em Componentes. *Simpósio Brasileiro de Engenharia de Software - Tutoriais*, volume 14, pages 297–329, João Pessoa, PB, Brasil, 2000.

[WYF02] H. Washizaki, H. Yamamoto, and Y. Fukazawa. Software Component Metrics and It's Experimental Evaluation. *International Symposium on Empirical Software Engineering*, volume 2, pages 19–20, 2002.

Apêndice A

Modelos de Componentes

COMPOR

COMPOR é uma infra-estrutura de engenharia para o desenvolvimento de software para sistemas complexos, abertos e dinâmicos, através de uma abordagem multiagentes. Trata-se de um conjunto de diretrizes e ferramentas para o desenvolvimento de software com foco na obtenção de flexibilidade e adaptabilidade. Estas diretrizes devem guiar o desenvolvedor desde a análise do problema até a implementação do software, com o auxílio das ferramentas computacionais desenvolvidas no contexto do COMPOR [APP+06].

A especificação do modelo de composição de componentes (Component Model Specification - CMS) tem base no princípio de que a inexistência de referências explícitas, ou diretas, entre provedores de funcionalidades tem como conseqüência uma maior flexibilidade na inserção, remoção e alteração destes provedores, inclusive em tempo de execução.

O modelo de componentes especificado na CMS possui três tipos de entidades definidos em sua arquitetura: contêineres, componentes funcionais e adaptadores. Os componentes funcionais implementam as funcionalidades do sistema, disponibilizando-as em forma de serviços. Além disso, não são compostos por outros componentes, ou seja, não possuem componentes-filhos. Os contêineres, por sua vez, não implementam funcionalidades, apenas gerenciam o acesso aos serviços dos seus componentes-filhos. Sendo assim, os contêineres servem como portas de acesso às funcionalidades dos componentes neles contidos. Por fim, os adaptadores, os quais não possuem funcionalidades próprias implementadas, apenas adaptam funcionalidades de componentes ou contêineres.

Os componentes funcionais são disponibilizados através dos contêineres. É necessário que haja ao menos um contêiner (contêiner-raiz) para que seja possível adicionar os componentes funcionais. Cada contêiner possui uma lista dos serviços providos e eventos de interesse de cada um dos seus componentes-filhos. Após a inserção de um componente, a lista de serviços e eventos de cada contêiner até a raiz da hierarquia é atualizada [APP^{+}06]. Isto é necessário para que os serviços providos pelo componente recém-adicionado estejam disponíveis para qualquer outro componente funcional do sistema, assim como para permitir que ele seja notificado caso algum evento de seu interesse seja disparado por outro componente.

Em CMS são definidos dois tipos de interação entre componentes: baseada em serviços e baseada em eventos. Na interação baseada em eventos o foco está sobre o anúncio da mudança de estado de um determinado componente, localizado em um determinado contêiner, aos componentes interessados, mesmo que estes estejam localizados em contêineres diferentes. A interação baseada em serviços permite a invocação dos serviços de um determinado componente a partir de qualquer outro componente do sistema, ainda que pertençam a contêineres diferentes [APP^{+}06]. Em ambos os tipos de interação não há referência explícita entre os componentes, como mostrado a seguir.

Modelos de interação

No modelo de componentes especificado na CMS são definidos dois tipos de interação entre componentes: baseada em serviços e baseada em eventos. Na interação baseada em eventos o foco está sobre o anúncio da mudança de estado de um determinado componente, localizado em um determinado contêiner, aos componentes interessados, mesmo que estes estejam localizados em contêineres diferentes. A interação baseada em serviços permite a invocação dos serviços de um determinado componente a partir de qualquer outro componente do sistema, ainda que pertençam a contêineres diferentes. Em ambos os casos não há referência explícita entre os componentes.

Modelo de interação baseada em serviços

Após a inserção de um componente em um determinado contêiner, seus serviços tornam-se disponíveis a qualquer outro componente do sistema. Sendo assim, supondo a existência do serviço "salvar" implementado pelo componente "K", pode-se solicitar a execução deste

serviço a partir de um componente “X”, sem fazer referência a “K”. Este processo é apresentado na Figura A.1, onde pode-se verificar que não há referência alguma entre o componente solicitante do serviço (“X”) e o componente provedor do mesmo (“K”). Desta forma, é possível alterar o componente que provê o serviço “salvar” sem modificar o restante da estrutura [APP+06].

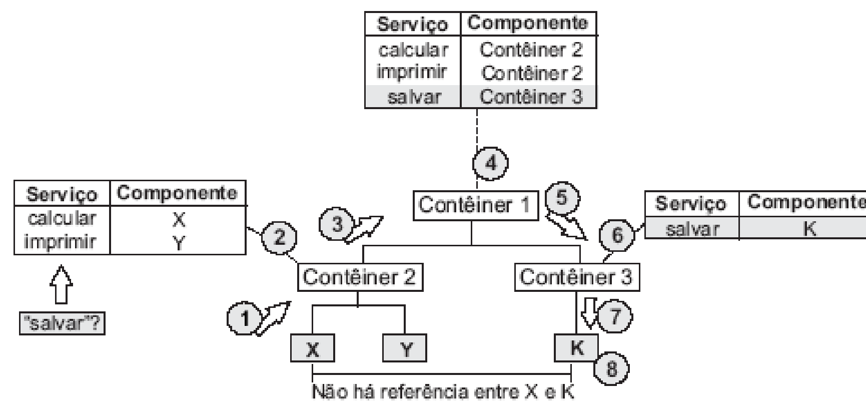


Figura A.1: Interação baseada em serviços

1. O componente “X” solicita a execução do serviço “salvar” ao seu “contêinerpai”.
2. O Contêiner 2 verifica, de acordo com sua tabela de serviços, que nenhum dos seus “componentes-filhos” implementa o serviço “salvar”.
3. O Contêiner 2 então encaminha a solicitação ao seu “contêiner-pai” (Contêiner 1).
4. O Contêiner 1 verifica, de acordo com sua tabela de serviços, que um de seus “componentes-filhos” implementa o serviço “salvar”(Contêiner 3). Para o Contêiner 1, o Contêiner 3 é visto como um componente que implementa o serviço.
5. O Contêiner 1 então encaminha a solicitação de serviço para o Contêiner 3.
6. O Contêiner 3 não implementa o serviço mas possui em sua tabela uma referência ao real implementador do serviço - componente “K”.
7. O Contêiner 3 então encaminha a solicitação de serviço para o componente funcional “K”.

8. O componente “K” executa o serviço “salvar” e retorna, caso exista, o resultado da execução, o qual segue o caminho inverso da solicitação.

Interação baseada em eventos

Quando um evento é disparado por um determinado componente funcional, toda a hierarquia de componentes do sistema deve ser verificada para que todos os interessados no evento sejam avisados. A interação baseada em eventos também é realizada pelos contêineres, não havendo referência direta entre os componentes funcionais. Um exemplo pode ser visto na Figura A.2, onde o componente “X” anuncia um evento denominado “EventoA”. Neste caso também não há referência alguma entre o componente anunciador do evento (“X”) e os interessados no evento (“Y” e “K”). Desta forma, o componente que anuncia o evento poderia ser alterado sem modificar o restante da estrutura [APP+06].

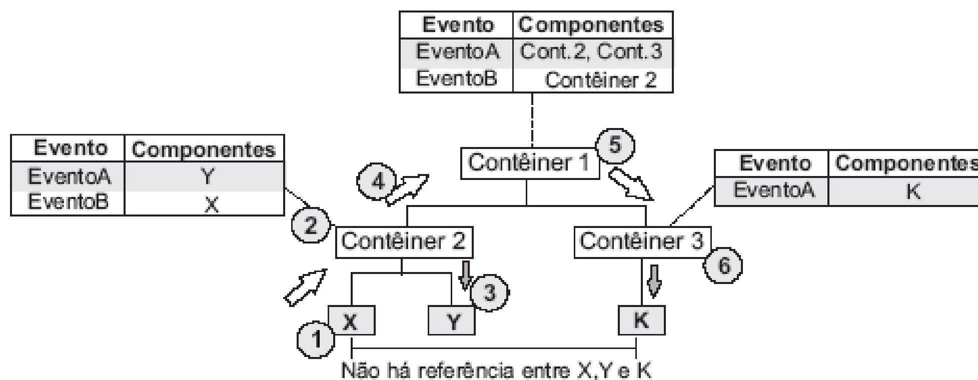


Figura A.2: Interação baseada em eventos

1. O componente “X” anuncia um evento denominado “EventoA”.

2. O anúncio é recebido, diretamente, apenas pelo seu contêiner-pai (Contêiner 2) que verifica em sua tabela de eventos de interesse dos “componentes-filhos” se algum deles está interessado no evento.

3. O Contêiner 2 encaminha o evento ao único interessado no evento de acordo com a sua tabela - o componente “Y”.

4. Além disso, o Contêiner 2 também encaminha o evento ao seu Contêiner-pai (Contêiner 1).

5. O Contêiner 1, de acordo com sua tabela de eventos, repassa o evento aos interessados, exceto para o originador do anúncio do evento (Contêiner 2). Sendo assim, encaminha o evento ao Contêiner 3. Por representar a raiz da hierarquia, ele não possui um contêiner-pai para o qual também repassaria o evento.

6. O Contêiner 3 repassa o evento ao componente interessado de acordo com a sua tabela de eventos. No exemplo, o componente "K" é o único interessado.

O mecanismo de "herança caixa-preta"

Devido aos modelos de disponibilização e interação mediados por contêineres, o modelo de componentes provê mecanismos para sobrescrever serviços através de "herança caixa-preta". Em outras palavras, é possível reutilizar serviços de componentes sem estender a classe que implementa o componente, em tempo de execução. Isto ocorre porque um componente pode ter como serviço requerido um serviço que ele mesmo provê. Uma vez que os serviços providos por componentes funcionais são acessados apenas via contêiner, é necessário apenas publicar funcionalidades internas do componente como serviços externos e acessá-los via contêiner.

Redefinindo serviços e eventos através de adaptadores

Quando um componente precisa ser utilizado em um contexto diferente daquele para o qual foi desenvolvido, ou seja, precisa ser reutilizado, sua interface pode precisar ser adaptada à aplicação. A utilização de mecanismos de reutilização de serviços por herança, como descrito anteriormente, demanda esforço de desenvolvimento para a construção de um novo componente. Para casos em que apenas parâmetros, retorno, exceções, restrições, dentre outras características da interface do componente precisam ser adaptados, pode-se definir um adaptador para o componente.

O conceito de adaptadores é bem estabelecido em paradigmas como orientação a objetos e componentes, com definições de padrões de projeto e implementação. A utilização de adaptadores no contexto do modelo de componentes especificado na CMS é similar à solução descrita no padrão Adapter. Um Adaptador é uma entidade que possui referência direta para um componente ou contêiner, mas que implementa a mesma interface de um componente.

Sendo assim, um contêiner não faz distinção entre os filhos que são componentes e os que são adaptadores. Da mesma forma que componentes funcionais, adaptadores são acessados apenas via contêiner. Sendo assim, a interface de um componente pode ser adaptada em tempo de execução.

Definindo requisitos não-funcionais como aspectos

A programação orientada a aspectos tem como objetivo principal separar a implementação dos requisitos funcionais do software (núcleo) da implementação dos requisitos não funcionais. Desta forma, o código funcional se torna mais claro, mais fácil de manter e gerenciar mudanças. No caso do modelo de componentes especificado na CMS, uma vez que os provedores de funcionalidades são os componentes funcionais, os aspectos são programados nestas entidades. Além disso, como os componentes são disponibilizados de forma independente da aplicação e são componentes de prateleira (*Commercial Off The Shelf* - COTS), ainda que um determinado aspecto atravesse mais de um componente, ele tem que ser tratado de forma isolada no momento em que o componente for implementado.

Portanto, a integração de aspectos à CMS ocorre a nível de componentes funcionais, utilizando visões para disponibilizar a execução dos serviços com e sem aspectos. Por exemplo, pode-se disponibilizar uma visão do componente com o aspecto de Log e outra visão sem este aspecto entrelaçado no código. Desta forma, o desenvolvedor pode usar uma visão durante o desenvolvimento do sistema e outra quando uma versão do software for liberada.

Enterprise JavaBeans

Enteprise JavaBeans (EJB) é uma tecnologia da Sun Microsystems que possibilita o desenvolvimento de componentes distribuídos para a camada de negócios. Servidores de aplicações compatíveis com a especificação J2EE trazem frameworks de componentes que implementam este modelo. EJB também pode ser classificado como um arcabouço de integração de *middleware*, pois provê suporte à comunicação remota entre componentes através de RMI-IIOP e também como arcabouço de infra-estrutura, pois trata aspectos como persistência, segurança e controle de transações, típicos de infra-estrutura.

A tecnologia Enterprise JavaBeans (EJB) define um modelo para o desenvolvimento e a disponibilização de componentes reutilizáveis para Java. EJB é uma extensão do sistema de componentes de JavaBeans para oferecer um suporte mais adequado para componentes servidores [Mic99]. De acordo como a terminologia de EJB, componentes servidores são partes de uma aplicação que executam em um servidor de aplicações. A tecnologia EJB apresenta-se como um ambiente Java robusto para o suporte a aplicações com rigorosos requisitos de escalabilidade, distribuição e disponibilidade [Mic99]. A tecnologia EJB oferece suporte ao desenvolvimento de aplicações baseadas em uma arquitetura de objetos distribuídos em várias camadas (*multitier*), na qual grande parte da lógica da aplicação é movida do cliente para o servidor. A lógica da aplicação é particionada em um ou mais objetos que são disponibilizados em um servidor de aplicações.

Um componente EJB deve não apenas poder executar em qualquer plataforma, como também ser completamente portável entre diferentes implementações de servidores de aplicações compatíveis com EJB [Tho98]. Para isso, o ambiente EJB deve mapear automaticamente um componente para uma determinada infra-estrutura de serviços. A infra-estrutura de EJB define uma série de interfaces padronizadas para que uma aplicação possa ter acesso a serviços de chamadas remotas de métodos (RMI), de nomes e diretórios (JNDI), de integração com CORBA (Java IDL), de criação dinâmica de páginas HTML (*Servlets* e JSP), de mensagens (JMS), de transações (JTS), e de acesso a bancos de dados (JDBC).

Há 3 tipos de componentes EJB, os beans de sessão (*Session Beans*), que representam processos síncronos de negócios, os beans de entidade (*Entity Beans*) que representam entidades persistentes e os beans orientados a mensagem (*Message Driven Beans*) que repre-

sentam processos de negócios assíncronos. Componentes EJB podem ser distribuídos em várias máquinas virtuais Java. Neste caso, diz-se que os componentes têm interfaces remotas, ou podem estar contidos na mesma máquina virtual, diz-se que têm interfaces locais. Opcionalmente um componente oferece os dois tipos de interfaces.

Os Beans de entidade usam persistência gerenciada pelo bean (bean managed persistency - BMP), onde o próprio desenvolvedor do componente programa o código que persiste os dados, ou persistência gerenciada pelo contêiner (*container managed persistency* - CMP), onde o contêiner gera o código que persiste os dados. Beans de entidade podem usar relacionamentos gerenciados pelo contêiner (*container managed relationship* - CMR) onde o contêiner mantém a integridade referencial dos relacionamentos entre beans de entidade. De forma similar, Enterprise JavaBeans oferece suporte às transações gerenciadas pelo bean (*bean managed transaction* - BMT) e às transações gerenciadas pelo contêiner (*container managed transaction* -CMT). Com BMT, não aplicável a beans de entidade, o desenvolvedor delimita programaticamente o início e o fim de cada transação. Já com CMT o desenvolvedor declara em um arquivo descritor o escopo das transações.

Além destes aspectos, EJB também trata questões relacionadas à segurança. Através do uso da API de segurança do Enterprise JavaBeans, o acesso a um componente ou a um determinado conjunto de operações de um componente pode ser restrito a um grupo de usuários. Este controle é feito de forma declarativa ou de forma programática.

Por fim, o contêiner Enterprise JavaBeans é responsável por gerenciar o ciclo de vida das instâncias dos EJBs. Utilizando *pool* de objetos, o contêiner é capaz de reutilizar instâncias de beans, reduzindo a necessidade de criar novas instâncias e aumentando assim o desempenho da aplicação. O contêiner também controla o acesso aos componentes, evitando problemas decorrentes do acesso simultâneo de várias *threads* a uma mesma instância de componente.

O EJB oferece uma série de características que oferecem suporte aos aspectos de infraestrutura de uma aplicação. O principal objetivo da incorporação de aspectos de infraestrutura a um arcabouço de componentes é possibilitar que o desenvolvedor de componentes concentre-se no domínio em que ele é especialista deixando aspectos como persistência de dados, para outros especialistas. EJB consegue atingir estes objetivos. Nas próximas seções são vistas as principais vantagens e desvantagens decorrentes do uso de Enterprise JavaBe-

ans.

Vantagens decorrentes do uso de Enterprise JavaBeans

EJB é um modelo de componentes que também trata de aspectos de infra-estrutura. As principais vantagens do uso de EJB decorrem principalmente da incorporação destes aspectos. O uso tanto de persistência gerenciada pelo contêiner (CMP) quanto de relacionamentos gerenciados pelo contêiner (CMR) em beans de entidade possibilitam que o desenvolvedor concentre-se na lógica de negócio em vez de concentrar-se na tecnologia que persiste os dados. Além do mais, o código da aplicação fica independente do sistema de gerenciamento de banco de dados (SGBD), pois todos os comandos para consultar, salvar, atualizar e remover entidades são gerados pelo contêiner.

Além disso, o uso de transações declarativas possibilita que o desenvolvedor de componentes construa o componente sem precisar demarcar transações, incentivando o reúso já que o escopo da transação é determinado sem que seja necessário recompilar o código fonte do componente. O uso de CMT também evita a introdução de erros que surgem decorrentes do esquecimento de sessões com o banco de dados abertas e não confirmadas, já que o contêiner se encarrega disto. Da mesma forma, o uso de segurança declarativa incentiva o reuso, pois o componente é customizado com uma política de segurança relativamente rígida, dependendo do escopo onde é usado, sem necessidade de mudanças no código fonte.

O modelo de componentes EJB também possibilita que componentes instalados em máquinas virtuais diferentes se comuniquem, de modo que os componentes da camada de negócio de uma aplicação podem ser instalados em máquinas diferentes. Este tipo de distribuição de componentes é útil, nos casos em que um componente reside no mesmo servidor em que está disponível um recurso de que ele necessita (por exemplo, quando um componente precisa acessar um recurso que só aceita conexões locais), no caso de componentes que demandam muito processamento (o componente pode ter um servidor dedicado a ele, não atrapalhando o desempenho dos outros componentes da aplicação) ou quando um componente precisa acessar outro componente do qual os desenvolvedores só tem acesso às interfaces (por exemplo, quando um componente precisa acessar um sistema de cartões de crédito).

Desvantagens decorrentes do uso de Enterprise JavaBeans

A principal desvantagem decorrente do uso de EJBs é o grande número de arquivos, incluindo código fonte escrito em Java e descritores XML, que precisam ser mantidos para

definir os beans de sessão e de entidade, os tipos de beans mais usados no desenvolvimento de aplicações com EJB. Para desenvolver um destes componentes é preciso pelo menos 3 arquivos de código fonte: a interface do componente, a interface home a classe do componente. Eventualmente, este número pode chegar a 6 arquivos de código fonte, contando apenas as classes previstas na especificação do modelo de componentes EJB.

A interface do componente define os serviços prestados pelo componente e pode ser local ou remota. Se a interface do componente for local, ela deverá estender a interface javax.ejb.EJBLocalObject, se for remota deverá estender a interface javax.ejb.EJBObject. A interface home é responsável por criar instâncias de componentes ou localizar instâncias existentes, no caso de beans de entidade. A interface home é definida como local ou remota, de acordo com a definição da interface do componente.

A classe do componente encapsula as regras de negócio e implementa também a interface javax.ejb.SessionBean no caso de beans de sessão ou a interface javax.ejb.EntityBean no caso de beans de entidade. A classe do componente pode, mas não é obrigada a implementar a interface do componente (a local ou a remota, sendo impossível implementar as duas). Se o desenvolvedor de componentes optar por desenvolver a classe do componente sem implementar a interface do componente, a sincronização entre interface e implementação deve ser realizada manualmente sem o suporte da linguagem Java, o que pode resultar em erros.

Por outro lado, se o desenvolvedor de componentes optar por desenvolver a classe do componente implementando a interface do componente, ele deverá implementar os métodos definidos nas interfaces EJBObject, se o componente for remoto, ou EJBLocalObject, se o componente for local. A implementação destes métodos nunca é executada, pois eles são sobrescritos pelo contêiner em tempo de implantação. A classe do componente deve ainda implementar métodos de *callback* definidos na especificação do EJB, como o ejbCreate e o ejbPostCreate, por exemplo. Além destes arquivos, no caso de beans de entidade com chave primária composta é preciso mais uma classe para representar esta chave.

Todos os enterprise beans também precisam ser acompanhados de um arquivo descritor em formato XML, onde são declaradas as regras de transação, segurança e outros. Dependendo do servidor de aplicações usado, podem ser necessários outros descritores, onde são declaradas regras como o mapeamento de beans de entidade em tabelas do banco de dados,

mapeamento de beans orientado a mensagens ao dispositivo de mensagem entre outros. Esta grande quantidade de arquivos traz problemas de manutenção. Esta desvantagem pode ser em parte amenizada pelo uso de ferramentas como o XDoclets, porém permanece sendo uma solução mais complexa do que usar uma interface Java e uma classe que a implementa.

Além da grande quantidade de arquivos que precisam ser mantidos, componentes Enterprise JavaBeans precisam ser executados dentro de um contêiner EJB. Isto implica em um número de opções quando necessita-se escolher o servidor de aplicações. Alguns servidores populares como o Tomcat não podem ser usados, pois não possuem contêiner EJB e servidores comerciais que possuem contêiner EJB em geral são mais caros. Além disso, os componentes EJB são dependentes do contêiner, com diversos métodos de *callback* chamados pelo contêiner EJB ao longo de seu ciclo de vida. Desta forma, componentes EJB são mais difíceis de testar unitariamente, pois precisam estar em execução dentro do contêiner para serem testados. No caso de componentes com interfaces locais, o próprio código de teste deve ser implantado junto com o componente para que este possa ser testado.

CCM

O padrão CORBA é o principal elemento da arquitetura definida pela OMG. Entre as especificações de maior relevância do padrão CORBA estão os mapeamentos de OMG IDL para diversas linguagens de programação, tais como C++, Java, C e Smalltalk, e as interfaces do seu *Object Request Broker* (ORB). A Figura A.3 mostra a estrutura geral da arquitetura CORBA.

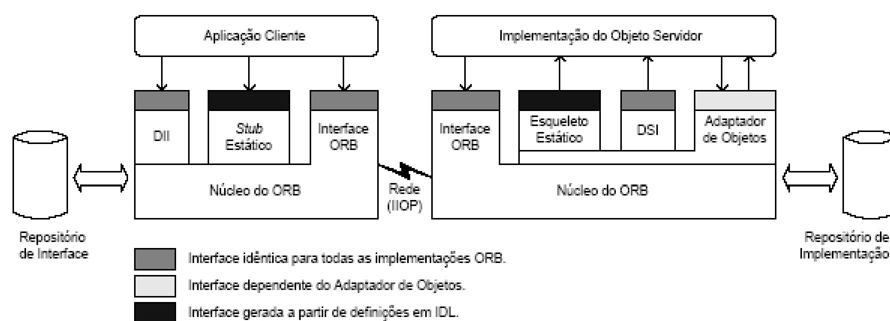


Figura A.3: Estrutura da arquitetura do MCC

O ORB é o elemento central dessa arquitetura. Ele é o responsável por estabelecer as conexões entre clientes e servidores, e por repassar as requisições de operações dos clientes para os objetos servidores. Vários aspectos relacionados à heterogeneidade entre plataformas e linguagens de programação são tratados internamente pelo ORB [OMG05].

O modelo de componentes de CORBA (CCM) estende o seu modelo de objetos com o intuito de tratar alguns problemas deixados em aberto pela arquitetura CORBA. Para tanto, são definidos novos recursos e serviços que permitem que desenvolvedores de aplicação possam implementar, gerenciar, configurar e implantar componentes de forma padronizada, facilitando a reutilização e a manutenção do sistema. No modelo CCM, componentes são os elementos básicos de construção de um sistema. Esses componentes são interconectados através de conexões orientadas a interface, onde a comunicação se dá através de chamadas de operações numa determinada interface; e de conexões orientadas a eventos, onde a comunicação se dá através da emissão e recebimento de eventos.

Os componentes são armazenados num pacote contendo sua implementação e a descrição

de suas características. Esse pacote é então utilizado para implantar o componente num servidor de componentes, onde esse pode ser então instanciando e utilizado por clientes.

No modelo CCM as conexões entre os componentes são feitas através de conectores denominados portas. Portas são pontos de comunicação do componente, que podem ser orientadas a interface ou a eventos. O modelo CCM define quatro tipos de portas: facetas que são conectadas a receptáculos para estabelecer conexões orientadas a interface; e fontes de eventos que são conectados a receptores de eventos para estabelecer conexões orientadas a eventos (canais de eventos). As portas definem os serviços oferecidos pelos componentes através de interfaces (faceta) ou de canais de emissão de eventos (fonte de eventos). Da mesma forma, as portas definem as dependências do componente, por exemplo definindo uma interface que é exigida pelo componente (receptáculo) ou um canal de recebimento de eventos (receptor de eventos).

Além de portas, os componentes CCM também podem oferecer interfaces e atributos, da mesma forma como objetos CORBA convencionais. As interfaces oferecidas e os atributos de um componente são destinados primordialmente à configuração do componente, através dos quais o componente é adaptado às necessidades da aplicação.

Os componentes CCM são descritos em IDL 3.0, que é uma extensão da linguagem de definição de interfaces de CORBA, que inclui novas estruturas para descrição de componentes, portas e outras estruturas relacionadas ao modelo de componentes CCM. A descrição de um componente também pode definir relações de herança, permitindo que o componente possa herdar interfaces oferecidas, atributos e portas de outro componente. Entretanto, cada componente só pode herdar de um único componente.

Componentes são declarados através da palavra *component*, de forma similar à definição de interfaces. A palavra *supports* define as interfaces oferecidas pelo componente. Herança de componentes é definida utilizando a mesma sintaxe da definição de interfaces, ou seja, através do operador : (dois pontos). Dentro da definição do componente, a declaração de atributos é feita através da palavra *attribute*, de forma similar à declaração de atributos em interfaces. As palavras *provides*, *uses*, *publishes*, *emits* e *consumes* são utilizados para declaração de portas.

Ao invés de definir regras para o mapeamento da definição de componentes para linguagens de programação, o modelo CCM define regras de mapeamento para IDL 2.3, gerando

uma IDL equivalente. Essa nova IDL gerada é constituída de um conjunto de interfaces que expõem todas as características e funcionalidades do componente (interfaces oferecidas, atributos e portas) denominada interface equivalente. Dessa maneira, é possível que clientes que não utilizam o modelo CCM possam acessar componentes de forma transparente através de interfaces CORBA comuns.

Executores

Na terminologia CCM, a implementação de um componente é denominada executor. Os executores devem implementar interfaces locais definidas na IDL equivalente do componente, denominadas interfaces de *callback*. é através dessas interfaces que os serviços do componente são invocados. Cada definição de componente resulta na geração de duas interfaces de *callback* distintas, uma para cada um dos tipos de executores definidos no modelo CCM: executor monolítico e executor segmentado.

Executores denominados monolíticos são tratados como um único módulo, ou seja, como um único pedaço íntegro e completo. Isso implica que componentes com executores monolíticos são carregados e ativados como um todo. O modelo CCM também permite definir executores segmentados, que são executores divididos em segmentos que podem ser carregados e ativados independentemente. Por exemplo, quando é feita uma requisição através de uma das portas do componente, apenas o segmento que implementa essa porta é ativado para responder a requisição, enquanto o resto do componente pode permanecer inativo.

Contexto

O modelo CCM define um conjunto de interfaces que fornecem serviços ao executor do componente, denominado interface de contexto. Através dessas interfaces, o executor pode ter acesso às conexões em suas portas, assim como utilizar serviços de objetos da arquitetura CORBA, tais como localização, transação, eventos, persistência e segurança. A interface de contexto é implementada pelo contêiner, que é o elemento do modelo CCM que fornece um ambiente de execução para as implementações de componentes com serviços e recursos que simplificam o desenvolvimento de componentes.

Eventos

Um novo recurso introduzido em IDL 3.0 é a definição de eventos através da palavra *eventtype*. Assim como a definição de componentes, esses eventos também são mapeados para IDL 2.3. Cada evento é mapeado para um *value type1* e uma interface. O *value type*

é utilizado para representar o evento propriamente dito. Por essa razão os eventos CCM são uma forma de value type mais restrito. Todos os value types que representam eventos CCM em IDL 2.3 devem ser uma especialização de um *value type* comum denominado *Components:EventBase*.

Protocolo de comunicação

CORBA especifica um padrão para o formato das mensagens enviadas entre objetos através de uma rede de comunicação. Esse padrão é denominado *General Inter-ORB Protocol* (GIOP). O GIOP é aplicado quando um cliente requisita uma operação de um objeto servidor através do ORB. O *Internet Inter-ORB Protocol* (IIOP) é um mapeamento específico do GIOP para o protocolo de transporte TCP/IP. ORBs podem implementar o GIOP para outros protocolos de transporte, mas todas as implementações de ORB devem oferecer pelo menos o IIOP.

Já que o IIOP é um requisito básico para que uma implementação de ORB seja considerada em conformidade com o padrão CORBA, é possível viabilizar a comunicação entre diferentes ORBs. Assim, o IIOP é um elemento essencial para a interoperabilidade entre diferentes implementações de CORBA. Essa interoperabilidade permite que um cliente desenvolvido usando um determinado ORB possa utilizar os serviços oferecidos por um objeto servidor desenvolvido para outro ORB [OMG05].

Interface entre cliente e servidor

Além do mecanismo tradicional de stubs, CORBA também oferece recursos para a construção dinâmica de requisições de operações, através de sua *Dynamic Invocation Interface* (DII). De forma semelhante, CORBA provê uma interface para a implementação dinâmica de objetos servidores: a *Dynamic Skeleton Interface* (DSI).

A requisição de uma operação pode obedecer a diferentes regras de execução. Tipicamente, uma requisição segue o modelo síncrono de chamada de procedimentos. Entretanto, devido a seu foco em sistemas distribuídos, CORBA também permite que operações sejam executadas de uma forma assíncrona. O modificador *oneway* de IDL define uma operação que não pode ter nenhum tipo de valor de retorno, podendo ser implementada de uma forma assíncrona pelo ORB. Essas operações tipicamente não têm nenhuma garantia de entrega

[OMG05].

Referências de objetos

Uma referência de objeto é o mecanismo utilizado para identificar e localizar um determinado objeto servidor. Para um cliente, referências de objetos são entidades opacas, ou seja, clientes usam essas referências para requisitar as operações dos objetos, mas não podem consultar ou modificar o conteúdo de uma referência. Uma referência de objeto só pode identificar um único objeto CORBA, mas um mesmo objeto pode ter várias referências para ele. Sob vários aspectos, essas referências são análogas aos ponteiros para instâncias de classes C++ [OMG05].

Toda referência de objeto contém uma indicação de qual interface seu objeto oferece. Isso permite que um ORB ofereça alguma segurança de tipos em tempo de execução, verificando se a interface oferecida por uma referência tem a operação sendo requisitada. Em linguagens estaticamente tipadas, como C++ e Java, segurança de tipos também é garantida em tempo de compilação. O mapeamento da linguagem não permite que seja requisitada uma operação, a menos que o objeto destino tenha garantidamente a operação em sua interface. Essa garantia em tempo de compilação só existe se o objeto servidor estiver sendo acessado através dos stubs gerados automaticamente a partir de uma definição em IDL. O uso de DII faz com que essa garantia seja perdida em tempo de compilação.

As regras de compatibilidade de tipos tradicionais entre uma interface e suas subinterfaces também valem para as referências de objetos, isto é, uma referência para um objeto com uma determinada interface derivada é considerada compatível com referências para objetos de suas super-interfaces.

Uma referência de objeto pode ser linearizada na forma de uma cadeia de caracteres (string), e essa cadeia de caracteres pode ser convertida de volta em uma referência que denota o mesmo objeto original. Essa cadeia de caracteres pode ser armazenada para um uso posterior. Por exemplo, um objeto servidor pode armazenar em um arquivo sua referência linearizada, e posteriormente um cliente pode ler esse arquivo para criar uma referência para esse objeto.

CORBA especifica um formato padrão para a representação de referências de objetos. Esse formato padrão é denominado *Interoperable Object Reference* (IOR). Uma IOR con-

tém todas as informações necessárias para estabelecer a conexão entre um cliente e um objeto servidor. Uma IOR identifica os protocolos disponíveis para conexão com um determinado servidor. No caso do protocolo IIOP, uma IOR armazena o nome da máquina servidora, o número da porta TCP/IP associada ao processo servidor, e um identificador de objeto que identifica unicamente o objeto destino naquele processo servidor. Isso significa que um ORB pode usar referências criadas por outras implementações CORBA, tanto através de referências passadas como parâmetros de operações, quanto através de referências importadas em sua forma linearizada [OMG05].

Repositórios de interfaces e implementações

Um ORB deve oferecer um repositório de interfaces, onde são armazenadas as descrições das interfaces dos objetos servidores disponíveis. O Repositório de Interfaces é o mecanismo básico de introspecção de CORBA, permitindo que um sistema consulte as interfaces dos objetos servidores disponíveis. Esse repositório é um objeto servidor CORBA, o que permite que ele seja utilizado por uma aplicação da mesma forma que outros objetos servidores. A sua interface define uma série de operações que permitem tanto a consulta de definições de interfaces quanto a alteração e criação de novas interfaces.

A arquitetura CORBA também define a existência de um repositório de implementações. Esse repositório deve auxiliar um cliente na obtenção de uma referência para um determinado objeto servidor. Além disso, um repositório de implementações pode oferecer serviços adicionais para auxiliar em tarefas como migração de objetos servidores, ativação automática de servidores e balanceamento de carga [OMG05].

Modelo de uso de CORBA

O modelo de uso de CORBA define basicamente a maneira de criação e utilização da implementação do componente. Cada modelo de uso define a forma que as requisições feitas a um componente são entregues à sua implementação. Por exemplo, no caso de um componente sem estado (Serviço), as requisições feitas a um componente podem ser entregues a um único executor ou a um repositório de executores daquele componente.

Por outro lado, no caso de um componente com identidade persistente (Entidade), as requisições feitas a um componente devem ser entregues a um executor específico, que ins-

tacia o estado persistente do componente. A partir dessas possibilidades são definidos três modelos de uso na especificação CCM, como descrito a seguir:

stateless - São criadas referências de componentes transientes e um grupo de referências são mapeadas a um único executor. Adicionalmente, o contêiner pode instanciar um único executor ou manter um repositório de executores para atender todas as requisições de componentes. Esse repositório pode ser aumentado de acordo com o número de requisições, sendo que tal política é definida pelo contêiner.

conversational - São criadas referências de componentes transientes e cada referência é mapeada a um único executor. O contêiner é responsável por criar uma instância do executor para cada componente criado e direcionar todas as requisições às referências do componente (referências de interfaces oferecidas e portas) ao executor correspondente.

durable - São criadas referências de componentes persistentes e cada referência é mapeada a um único executor que encarna o estado do componente associado a referência persistente. A recuperação do estado do componente associado a uma referência persistente pode ser feita pelo contêiner ou pelo próprio componente, denominadas respectivamente de persistência gerenciada pelo contêiner ou auto-gerenciada.