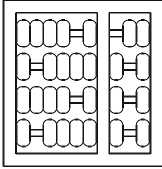

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Um Método para Modelagem da Arquitetura a Partir dos Requisitos do Sistema

Patrick Henrique da Silva Brito
Maria Antonia Martins Barbosa
Paulo Asterio de Castro Guerra
Cecília Mary Fischer Rubira

Technical Report - IC-07-11 - Relatório Técnico

March - 2007 - Março

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Um Método para Modelagem da Arquitetura a Partir dos Requisitos do Sistema

Patrick Henrique da Silva Brito   Maria Antonia Martins Barbosa
Paulo Asterio de Castro Guerra   Cecília Mary Fischer Rubira

Resumo. *No desenvolvimento de sistemas modernos, a fase de projeto arquitetural vem recebendo uma atenção cada vez maior. Por ser considerado o local ideal para a implementação dos requisitos não-funcionais de um sistema, mudanças tardias na arquitetura costumam ter um custo muito elevado. Além disso, uma arquitetura bem projetada, além de aumentar a qualidade do sistema, facilita a sua compreensão e manutenção. Porém, projetar a arquitetura de um software é uma tarefa complexa e dependente da experiência da equipe de desenvolvimento. Este trabalho propõe um método para sistematizar a especificação da visão lógica da arquitetura do sistema, refinada com o estilo arquitetural de componentes independentes. Dessa forma, esse método pretende reduzir a penalidade decorrente da pouca experiência da equipe de desenvolvimento. O método proposto foi validado através de um estudo de caso simples, apresentado no final do documento.*

Palavras-chave: *Arquitetura de software, projeto arquitetural, método de desenvolvimento, desenvolvimento centrado na arquitetura.*

Abstract. *The architectural design phase is being each time more valued in modern development processes. The software architecture is considered the ideal place to implement non-functional requirements of a system. Because this, delayed changes usually imply in a very high cost. Moreover, a well designed architecture, besides increasing the system quality, facilitates its understanding and maintenance. However, to project the software architecture of a system is a complex task which is dependent of the development team experience. This work considers a method to systemize the specification of the logical view of the software architecture, refined with the independent components architectural style. This method intends to reduce the decurrent penalty of a little experience development team. The proposed method was validated through a simple case study, presented ahead.*

Keywords: *Software architecture, architectural design, software development process, architecture-based software development.*

1 Introdução

Com o aumento crescente do tamanho e principalmente da complexidade dos sistemas de software, o projeto arquitetural assumiu um papel decisivo para o sucesso ou falha no atendimento dos requisitos, principalmente dos requisitos não-funcionais, que estão relacionados a aspectos qualitativos do sistema [28].

Diferentemente dos processos clássicos de desenvolvimento, nos métodos de desenvolvimento centrados na arquitetura, a fase de projeto arquitetural deve ser executada antes mesmo da fase de análise. A principal razão para isso é o fato de que na fase de análise acontece a identificação dos componentes que constituem o sistema; e durante essa identificação, é feito um posicionamento inicial de cada um dos componentes, de acordo com os componentes arquiteturais já identificados. Dessa forma, a antecipação da fase de projeto arquitetural possibilita uma maior contextualização dos componentes identificados em relação à estrutura geral do sistema.

As principais vantagens de se enfatizar o papel arquitetural do sistema são: (i) ter uma estruturação abstrata, que facilita o entendimento e faz da arquitetura um eficiente veículo de comunicação entre as partes interessadas[1] do sistema; (ii) alta granularidade de reutilização, que aliado ao DBC pode ser determinante para a redução do tempo de desenvolvimento, o que atende a um requisito importante do mercado desenvolvedor de software atual [24]; e (iii) maior facilidade para adaptar, manter, evoluir e portar o sistema para outras plataformas, como conseqüências do baixo acoplamento proporcionado pelo uso de conectores arquiteturais, que explicitam a comunicação entre os componentes da arquitetura. Essa comunicação explícita facilita a substituição de componentes, uma vez que torna possível a adaptação das mensagens que fluem entre eles [31, 10].

Um processo de desenvolvimento centrado na arquitetura, além de usufruir dessa abstração arquitetural em todas as fases do desenvolvimento, deve fornecer meios de especificar a arquitetura do sistema, obedecendo as restrições impostas pelos seus requisitos não-funcionais [31].

Durante a escolha dos estilos arquiteturais que serão utilizados, os interessados podem inclusive reutilizar arquiteturas de sistemas anteriores, baseado nas semelhanças entre os modelos conceituais, as suas restrições e os seus atributos de qualidade. Seguindo essa tendência, pesquisadores de várias partes do mundo estudam maneiras de sistematizar essa reutilização estrutural do sistema como forma de agilizar o desenvolvimento do software. Um exemplo muito em voga atualmente é a abordagem de desenvolvimento em linhas de produto de software[2] [1, 22, 11]. Devido à interferência da arquitetura em todo o desenvolvimento do sistema, este trabalho apresenta um método para identificar os elementos da arquitetura de maneira sistemática. Devido ao caráter recursivo do método proposto, cada um dos componentes arquiteturais pode ser refinado progressivamente, a fim de especificar os componentes de menor granularidade do sistema (componentes de desenvolvimento), caracterizando uma abordagem de desenvolvimento *top-down*.

A Seção 2 apresenta os fundamentos teóricos necessários para a contextualização do método proposto. A Seção 3 apresenta o método propriamente dito, classificado em duas

[1] do inglês *stakeholders*
[2] do inglês *software product lines*

fases principais: (i) especificação da visão lógica abstrata (Seção 3.1); e (ii) especificação da visão lógica detalhada (Seção 3.2). Para demonstrar a aplicabilidade do método proposto, a Seção 4 apresenta um estudo de caso onde é feita a especificação da arquitetura de um sistema de gerenciamento de hotéis. Finalmente, a Seção 5 apresenta as conclusões e alguns direcionamentos para trabalhos futuros.

2 Fundamentação Teórica

2.1 Desenvolvimento Baseado em Componentes

A idéia de utilizar o conceito de Desenvolvimento Baseado em Componentes (DBC) na produção de software data de 1976 [17]. Apesar desse tempo relativamente longo, o interesse pelo DBC só foi intensificado vinte anos depois, após a realização do Primeiro Workshop Internacional em Programação Orientada a Componentes, o WCOP'96 [30].

Hoje em dia, a popularização do DBC está sendo motivada principalmente pelas pressões sofridas na indústria de software por prazos mais curtos e produtos de maior qualidade. No DBC, uma aplicação é construída a partir da composição de componentes de software que já foram previamente especificados, construídos e testados, o que proporciona um ganho de produtividade e qualidade no software produzido.

Esse aumento da produtividade é decorrente da reutilização de componentes existentes na construção de novos sistemas. Já o aumento da qualidade é uma conseqüência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos. Porém, vale a pena ressaltar que apesar desses testes prévios serem benéficos, a reutilização de componentes não dispensa a execução dos testes no novo contexto onde o componente está sendo reutilizado.

Apesar da popularização atual do DBC, não existe um consenso geral sobre a definição de um componente de software, que é a unidade básica de desenvolvimento do DBC. Porém, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e implementação, além de oferecer interfaces bem definidas para o meio externo. O baixo acoplamento decorrente dessa política proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes COTS[3]; e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Uma definição complementar de componentes, adotada na maioria dos trabalhos publicados atualmente, foi proposta em 1998 [29]. Segundo Szyperski, um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (**interfaces providas**), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através de suas **interfaces requeridas**.

Além dessa distinção clara entre interfaces providas e requeridas, os componentes seguem três princípios fundamentais, que são comuns à tecnologia de objetos [7]:

[3]do inglês *Common-Off-The-Shelf*

1. **Unificação de dados e funções:** Um componente deve encapsular o seu estado (dados relevantes) e a implementação das suas operações oferecidas, que acessam esses dados. Essa ligação estreita entre os dados e as operações ajudam a aumentar a coesão do sistema;

2. **Encapsulamento:** Os clientes que utilizam um componente devem depender somente da sua especificação, nunca da sua implementação. Essa separação de interesse[4] reduz o acoplamento entre os módulos do sistema, além de melhorar a sua manutenção;

3. **Identidade:** Independentemente dos valores dos seus atributos de estado, cada componente possui um identificador único que o difere dos demais.

A fim de contextualizar o componente em relação aos diferentes níveis de abstração que ele pode ser analisado, um componente pode ser observado através de diferentes pontos de vista [7]. Em outras palavras, dependendo do papel que o interessado exerce no processo de desenvolvimento, a abstração como o componente é analisado pode variar. A Tabela 1 descreve os diferentes pontos de vista de um componente [7].

Tabela 1: Pontos de Vista de um Componente

Ponto de Vista	Descrição
Especificação	È constituído da especificação de uma unidade de software que descreve o comportamento de um componente. Esse comportamento é definido como um conjunto de interfaces providas e requeridas.
Plataforma	São definidas restrições que o componente deve seguir a fim de ser utilizado em alguma plataforma específica de desenvolvimento. As principais plataformas comerciais existentes são Enterprise Java Beans [20], Corba [16] e COM+ [26].
Implementação	É a realização de uma especificação. Sendo assim, a implementação de um componente está para a sua especificação, assim como uma classe está para a sua interface. Esse ponto de vista é compartilhado por todos os componentes já implementados em alguma linguagem de programação. Ele representa componentes prontos para serem instalados, como por exemplo, os componentes COTS.
Instalação	É uma instalação ou cópia de um componente implementado. Esse ponto de vista é utilizado na análise do ambiente de execução. Neste caso, pode ser feita uma analogia às classes localizadas no *classpath* do Java. O *classpath* é a lista de diretórios onde a máquina virtual procura as classes a serem instanciadas.
Instanciação	É uma instância de um componente instalado. Esta é a visão de execução do componente, com os seus dados encapsulados e um identificador único. É semelhante ao conceito de objetos no paradigma de orientação a objetos.

[4]do inglês *separation of concerns*

2.2 Arquitetura de Software

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de seus **componentes arquiteturais**, que representam unidades abstratas do sistema; a interação entre essas entidades, que são materializadas explicitamente através dos **conectores**; e os atributos e funcionalidades de cada um [28]. Por conhecerem o fluxo interativo entre os componentes do sistema, é possível nos conectores, estabelecer protocolos de comunicação e coordenar a execução dos serviços que envolvam mais de um componente do sistema.

Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidade de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [3, 28].

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [27, 19]. Um **estilo arquitetural** caracteriza uma família de sistemas que são relacionados pelo compartilhamento de suas propriedades estruturais e semânticas. Esses estilos definem inclusive restrições de comunicação entre os componentes do sistema. A maneira como os componentes de um sistema ficam dispostos é conhecida como **configuração**.

As propriedades arquiteturais são derivadas dos requisitos do sistema e influenciam, direcionam e restringem todas as fases do seu ciclo de vida. Sendo assim, a arquitetura de software é um artefato essencial no processo de desenvolvimento de softwares modernos, sendo útil em todas as suas fases [6]. A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece, uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável; e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável.

2.3 O Método de Avaliação de Arquiteturas ATAM

O ATAM[5] [13] é um método de análise de arquiteturas, que foca na identificação e priorização dos requisitos não-funcionais do negócio relacionados aos atributos de qualidade desejados. A partir da definição dos requisitos não-funcionais, o método ATAM pode ser

[5]do inglês *Architecture Tradeoff Analysis Method*

utilizado para analisar como os diversos estilos arquiteturais podem ser utilizados para alcançar cada um dos atributos de qualidade. O método ATAM apresenta nove atividades, classificadas em quatro grupos:

Grupo 1: Apresentação

1. **Apresentar o ATAM.** Nessa atividade, o método deve ser descrito para as partes interessadas, que tipicamente constituem um grupo formado pelo representante do cliente, o arquiteto de software, o testador, o gerente de projeto, e o responsável pela fase de manutenção.
2. **Apresentar os marcos do negócio**[6]. O gerente de projeto descreve quais são os objetivos pretendidos para o sistema. Essa descrição inicial deve servir de base para a escolha da arquitetura inicialmente proposta. A escolha dessa arquitetura deve se basear principalmente na expectativa de disponibilidade, tempo de mercado[7] e confiabilidade.
3. **Apresentar a arquitetura.** A partir da descrição do gerente de projeto, o arquiteto deve descrever a arquitetura inicialmente proposta, focando em como cada um dos objetivos descritos será materializado.

Grupo 2: Investigação e Análise

4. **Identificar as possibilidades para a arquitetura.** A lista de arquiteturas possíveis é identificada pelo arquiteto, mas ainda não são analisadas. Essa lista pode ser criada a partir das arquiteturas semelhantes à arquitetura inicialmente proposta.
5. **Gerar uma árvore com a classificação dos atributos de qualidade desejados.** Os atributos de qualidade especificados devem ser desmembrados em requisitos não-funcionais. Em seguida, são especificados cenários de exemplos que auxiliem a compreensão do que realmente se espera para cada um desses requisitos. Após a especificação dos cenários, esses requisitos devem ser priorizados.
6. **Analisar as possibilidades para a arquitetura.** A partir da lista de prioridades definida na atividade anterior, a lista de arquiteturas possíveis deve ser analisada, por exemplo, uma arquitetura que prioriza o desempenho pode ser menos desejada que outra que priorize a confiabilidade.

Grupo 3: Teste

7. ***Brainstorm* e priorização dos cenários.** Baseado nos cenários especificados na árvore de precedência, deve-se refinar essa lista de cenários a partir das idéias de cada um dos interessados. Após a identificação dos vários cenários possíveis, esses cenários devem ser priorizados através de uma votação entre os interessados.

[6]do inglês *business drivers*
[7]do inglês *time to market*

8. **Analisar as possibilidades para a arquitetura.** Este passo consiste em uma nova iteração da atividade 6. Mas nesse momento os principais cenários priorizados na atividade anterior devem gerar casos de teste para analisar a arquitetura. Com a execução desses casos de teste, podem ser descobertos riscos e relações de compromisso[8] entre as propostas de arquitetura listadas. Tudo isso deve ser documentado.

Grupo 4: DIVULGAÇÃO

9. **Apresentar resultados.** Com a execução das atividades do método ATAM, são coletadas algumas informações importantes, tais como arquiteturas candidatas, requisitos não-funcionais com prioridades, cenários, riscos e relações de compromisso. Com essas informações em mãos, deve ser gerado um relatório detalhando as possíveis estratégias para a estrutura da arquitetura do sistema.

2.4 Desenvolvimento Centrado na Arquitetura

A arquitetura de software é um artefato essencial no ciclo de vida do software e deve auxiliar todas as suas fases [6]. A principal motivação para isso é que a abstração oferecida pela arquitetura de software possibilita uma análise estrutural em alto nível e um melhor entendimento do sistema.

Um processo de desenvolvimento centrado na arquitetura, além de usufruir dessa abstração arquitetural em todas as fases do desenvolvimento, deve fornecer meios de especificar a arquitetura do sistema, obedecendo as restrições impostas pelos seus requisitos não-funcionais [31]. Além disso, enquanto os processos tradicionais consideram a composição do sistema como sendo uma atividade de implementação [7], nos processos centrados na arquitetura, a composição dos componentes deve ser considerada em diferentes fases do desenvolvimento. Em outras palavras, a composição deve ser pensada nos diversos níveis de abstração (modelo abstrato, especificação detalhada e implementação) [6].

As principais vantagens decorrentes da adoção de um processo de desenvolvimento centrado na arquitetura são: (i) facilidade de se especificar propriedades não-funcionais no sistema; (ii) reutilização em um nível maior de granularidade; (iii) redução do tempo de desenvolvimento; (iv) representação estrutural explícita; (v) compreensão facilitada do sistema; e (vi) facilidade de manutenção.

Motivadas principalmente pela reutilização de software em larga escala, é cada vez maior o número de empresas que utilizam alguma técnica de desenvolvimento centrado na arquitetura [15]. A Figura 1 mostra o cliclo de vida de um processo tradicional de desenvolvomento centrado na arquitetura, onde cada retângulo representa uma atividade. Inicialmente, logo após a especificação dos requisitos, antes mesmo da fase de análise, a visão lógica inicial da arquitetura é especificada (atividade 3). Essa visão proporciona uma representação estrutural do sistema em um alto nível de abstração. Em seguida, a arquitetura inicial passa por um processo de refinamento, com o intuito de detalhar os componentes arquiteturais, através da descoberta de sub-componentes, interfaces e conectores (atividade 4).

[8] do inglês *tradeoff*

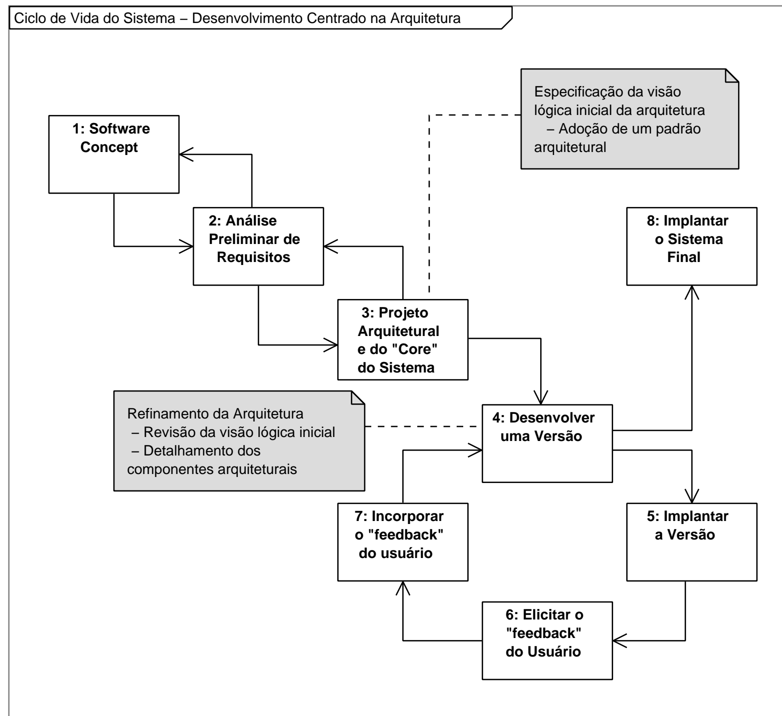

Figura 1: Ciclo de Vida do Sistema no Desenvolvimento Centrado na Arquitetura

3 Um Método para a Especificação da Arquitetura do Sistema

Esta seção apresenta um método para especificar a arquitetura de um sistema. Sendo assim, ele pode ser empregado para sistematizar a execução das atividades 3 e 4 apresentadas na Figura 1, que mostra o *workflow* de um processo tradicional de desenvolvimento centrado na arquitetura.

Como pode ser visto na Figura 2, o método é composto de três atividades. Inicialmente é feita uma especificação da visão lógica da arquitetura (Seção 3.1). Essa especificação se baseia unicamente nos requisitos especificados para o sistema e por isso pode ser executada antes da fase de análise. Em seguinda, a arquitetura deve ser detalhada de acordo com o estilo arquitetural baseado em componentes independentes[9] [3]. Em outras palavras, esse detalhamento consiste na identificação de sub-componentes, suas interfaces e os conectores arquiteturais que materializam a interação entre eles.

Após o detalhamento da arquitetura, pode ser necessário revisar a visão lógica inicial do sistema. Essa revisão visa uma melhor organização estrutural do sistema e consiste do seguinte procedimento: se existirem componentes arquiteturais com interfaces de aplicação e de infra-estrutura, ao mesmo tempo, esse componente deve ser revisto, no intuito de: (i) ou re-alocar a responsabilidade de implementação das interfaces; ou (ii)refatorar a arquitetura

[9] do inglês *independent components*

(dividir o componente arquitetural). Essa decisão deve ser tomada pelo arquiteto, que por sua vez pode julgar necessário convocar uma reunião de *brainstorm* entre as partes interessadas[10] do sistema.

Uma observação importante é o fato do método poder ser executado recursivamente para cada componente arquitetural especificado. Para isso, basta que o componente a ser detalhado seja tratado como um sistema particular.

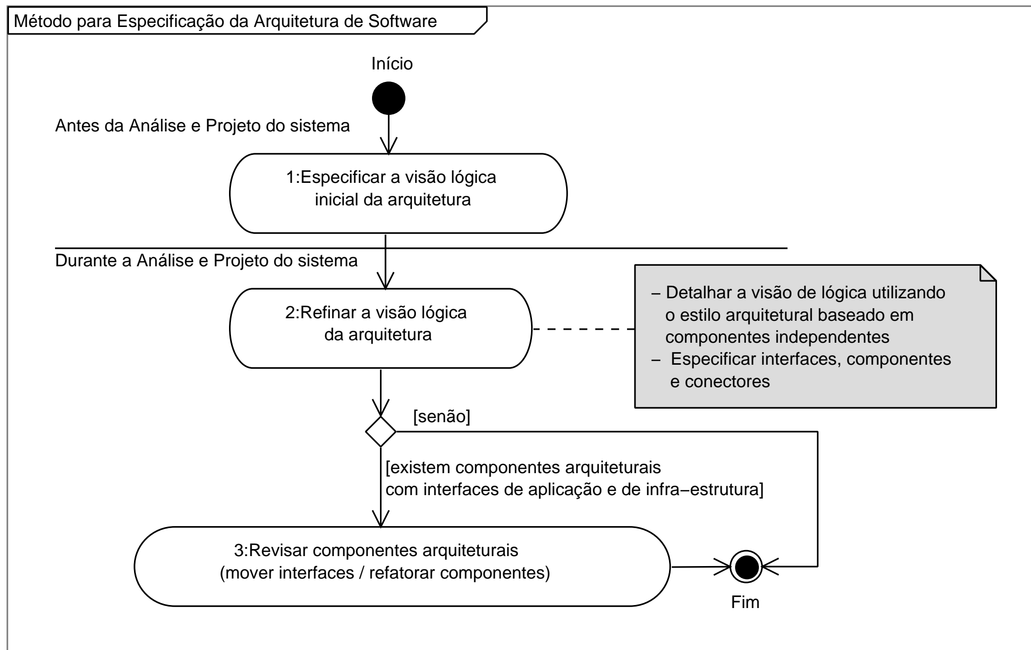

Figura 2: Método para Especificação da Arquitetura de Software

3.1 Especificar a Visão Lógica da Arquitetura (ativ. 1)

A relação direta entre a arquitetura de um sistema e seus requisitos não funcionais é uma característica bem conhecida entre os arquitetos de software [3, 28]. Apesar disso, não faz muito tempo que vem sendo investido esforço para sistematizar a maneira como essas estruturas podem ser materializadas, de forma a maximizar a satisfação desses requisitos [4]. Além disso, a possibilidade de existir requisitos antagônicos, como por exemplo confiabilidade extrema aliada ao desempenho crítico, evidencia a necessidade de alguma abordagem para alcançar o melhor ponto de equilíbrio de satisfação, onde não necessariamente todos os requisitos sejam satisfeitos.

O método proposto nesse documento se baseia em uma simplificação da abordagem adotada pelo método ATAM, que foi apresentado na Seção 2.3. Essa abordagem é baseada no entendimento de como os atributos de qualidade do sistema podem ser satisfeitos pela

[10] do inglês *stakeholders*

arquitetura. Basicamente, essa fase é constituída de cinco atividades principais, que seqüencialmente, orientam o arquiteto na escolha de uma arquitetura adequada. A Figura 3 mostra o *workflow* referente a esse método e em seguida, cada uma das atividades é explicada em maiores detalhes.

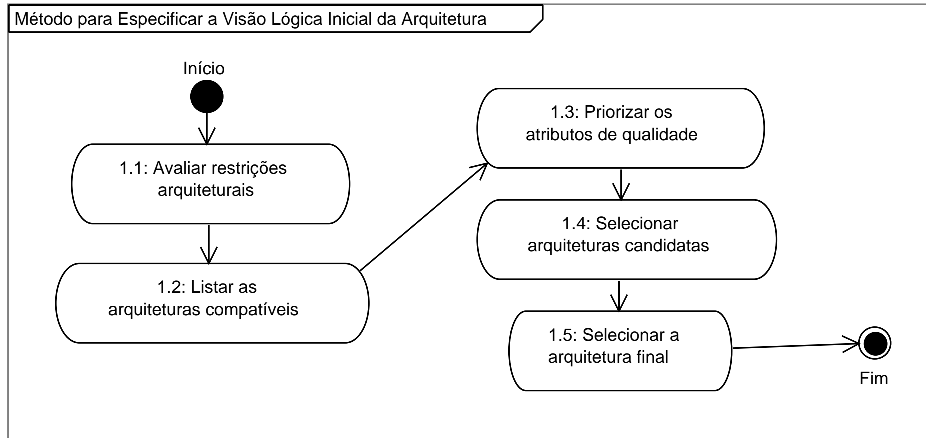

Figura 3: Método para Especificar a Visão Lógica Inicial da Arquitetura

1. **Avaliar restrições arquiteturais.** Nessa atividade, o arquiteto de software avalia as restrições definidas para o sistema, a fim de descartar as arquiteturas consideradas claramente inviáveis. O resultado final dessa atividade é uma lista de fatores que podem interferir diretamente na escolha da arquitetura, podendo inclusive inviabilizar alguma adoção específica. Alguns exemplos de fatores a serem analisados são: a adoção de um determinado *framework* ou de uma plataforma específica; ou até mesmo a preferência pela utilização de alguma arquitetura em particular.

2. **Listar as arquiteturas compatíveis.** Após a identificação dos principais fatores que influenciam a escolha da arquitetura, o arquiteto deve restringir a lista de estilos arquiteturais candidatos. Essa restrição tem o intuito de retirar os estilos considerados inviáveis de acordo com os novos fatores levantados. O resultado final dessa atividade é uma lista de estilos arquiteturais que são compatíveis com as restrições identificadas na atividade anterior. Esses estilos são considerados candidatos a satisfazer os atributos de qualidade do sistema, seguindo a idéia de se ter estilos arquiteturais baseados em atributos (ABAS[11] [14]). Para essa escolha, pode-se utilizar catálogos de estilos, também conhecidos como padrões arquiteturais [5]. A Seção 3.1.1 apresenta uma relação dos principais estilos arquiteturais utilizados atualmente.

3. **Priorizar os atributos de qualidade.** Depois de se ter a listagem dos estilos arquiteturais que podem ser utilizados, é chegada a hora de escolher entre eles, o estilo (ou combinação de estilos) mais adequado para o sistema.

[11] do inglês *Attribute-Based Architectural Styles*

Para isso, o método proposto sugere uma análise criteriosa dos atributos de qualidade desejados. O primeiro passo dessa análise não-funcional do sistema é estipular os critérios da avaliação, isto é, a relação de precedência entre os atributos de qualidade. Seguindo a proposta utilizada pelo método ATAM, o cliente juntamente com o arquiteto de software devem estabelecer uma árvore de precedência que represente as prioridades de escolha de uma maneira hierárquica *top-down*. Dessa forma, como mostrado na Figura 4, os requisito mais gerais são representados como nós de mais alto nível. No exemplo específico dessa figura, os requisitos de confiabilidade, disponibilidade, manutenibilidade e performance são considerados requisitos gerais. No modelo de árvore utilizado pelo MDCE+, a ordem em que os requisitos são dispostos (da esquerda para a direita) indica a sua precedência em relação aos demais. Por exemplo, na mesma árvore da Figura 4, a ordem de precedência dos atributos de qualidade é a seguinte: 1°- confiabilidade; 2°- disponibilidade; 3°- manutenibilidade; e 4°- desempenho.

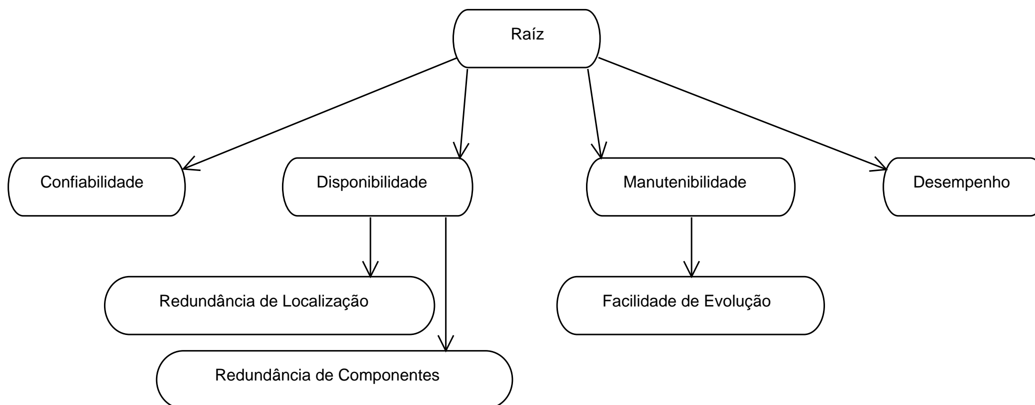

Figura 4: Árvore de Precedência dos Requisitos Não-Funcionais

Em seguida, a partir das descrições dos casos de uso, são selecionadas as afirmações que evidenciam exemplos da necessidade real de cada um desses requisitos. Essas frases são utilizadas para detalhar os interesses das partes interessadas e ajudam a repensar a ordem de precedência entre os atributos de qualidade. Por exemplo, atributos exigidos por alguma funcionalidade crítica tem, normalmente, uma prioridade maior. Para reduzir o número de cenários a serem analisados, o método proposto sugere a análise apenas dos cenários pertencentes às funcionalidades críticas do sistema, que devem ser identificados durante a especificação dos requisitos. A Figura 5 mostra a árvore de precedência da Figura 4 com alguns cenários representados.

4. **Selecionar arquiteturas candidatas.** Com a definição das prioridades entre os atributos de qualidade do sistema, o arquiteto deve fazer uma triagem entre os estilos arquiteturais existentes na saída da Atividade 2. Essa triagem tem o intuito de restringir a lista de arquiteturas candidatas que satisfaçam os requisitos não-funcionais do sistema. Para isso, a partir da avaliação da árvore de precedência com cenários, devem ser procurados estilos arquiteturais indicados para cada um dos requisitos deseja-

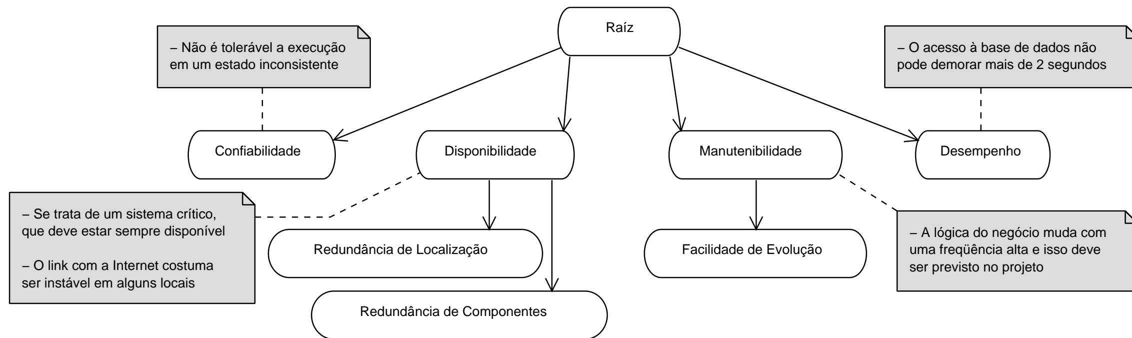

Figura 5: Árvore de Precedência de Requisitos Não-Funcionais com Cenários

dos. Para esse mapeamento, o método proposto recomenda que seja realizada alguma consulta a catálogos de padões arquiteturais, como o apresentado na Seção 3.1.1. A escolha do estilo arquitetural a partir dos requisitos do sistema caracteriza um método ABAS[12] [14], de acordo com a terminologia adotada pelo Instituto de Engenharia de Software (SEI) da Universidade de Carnegie Mellon (CMU). Em seguida, deve ser feita uma análise dos riscos envolvidos com a utilização de cada um deles. Essa análise deve considerar as relações de compromisso[13] entre os requisitos; lembrando que a relação de precedência deve ser levada em consideração nessa análise. O produto final dessa atividade é uma lista restrita dos estilos arquiteturais (ou combinações de estilos) mais adequados para o sistema.

5. **Selecionar a arquitetura final.** Esta é a última atividade referente à escolha da arquitetura do sistema. Por se tratar de uma das decisões mais importantes de todo o desenvolvimento, ela deve ser desempenhada em conjunto por toda a equipe de projeto: cliente, arquiteto de software, engenheiro de requisitos, testador, e mantenedor. Normalmente, essa atividade consiste de uma ou mais seções de *brainstorm* moderadas pelo arquiteto. Nessas reuniões, cada um dos participantes escolhe um dos estilos arquiteturais pré-selecionados e argumenta a respeito da sua decisão; em seguida, o grupo tenta chegar a um consenso, a partir da combinação de estilos sugerida pelo arquiteto. No final, após a escolha da arquitetura mais indicada, essa informação deve ser documentada no contrato.

Vale a pena salientar o fato de nem sempre ser possível satisfazer todos os requisitos desejados para o sistema. Nos casos onde os requisitos não são atendidos totalmente, o método proposto define que deve-se iterar novamente, a fim de refinar os requisitos, a arquitetura ou ambos e se necessário, o contrato com o cliente e a documentação inicial do sistema também devem ser atualizados, conforme definido no *workflow* mostrado na Figura 1.

[12] do inglês *Attribute-Based Architectural Styles*
[13] do inglês *tradeoff*

3.1.1 Principais padrões arquiteturais

O papel de um **padrão arquitetural** é fazer uma associação entre os estilos arquiteturais (e combinações de estilos) e as suas respectivas aplicabilidades. A seguir, é apresentada uma relação dos principais estilos arquiteturais encontrados atualmente.

1. **Baseada em fluxo de dados**
   - **Descrição**: Estilo onde o processamento é feito seqüencialmente, isto é, um componente do sistema só executa após o seu antecessor finalizar a execução por completo.
   - **Problema**: Sistemas onde o processamento deve ser realizado em partes seqüenciais.
   - **Vantagens**: (i) o fluxo de comunicação entre os componentes é relativamente simples e facilmente analisável; (ii) expressa uma arquitetura reconfigurável; (iii) expressa explicitamente a idéia de concorrência.
   - **Desvantagens**: (i) unidirecional; (ii) indicado para sistemas interativos; (iii) exige muitos estágios de conversão e serialização de dados; (iv) o compartilhamento de dados globais é muito caro.
   - **Variantes**:
     - *Pipes and filters*; Nessa variação, cada componente do sistema desempenha uma parte da transformação e em seguida, a sua saída é utilizada como entrada da próxima etapa do processamento (em outro componente)
     - Execução *batch*.
2. **Centrado em Dados**
   - **Descrição**: Em uma arquitetura centrada em dados, o comportamento do sistema é especificado em função dos dados que circulam nele. Em outras palavras, nesses sistemas a lógica do negócio é definida em função da variação do estado do sistema.
   - **Problema**: Sistemas onde há a necessidade de existir dados compartilhados.
   - **Vantagens**: (i) facilidade de cooperação entre os componentes (maior integridade); (ii) separação entre os dados e o processamento (maior modificabilidade).
   - **Desvantagens**: (i) não é possível encapsular os dados e o processamento juntos, os dois devem ser tratados obrigatoriamente (menor segurança); (ii) compreensão e manutenibilidade baixas, devido à dependência entre a lógica do negócio (processamento) e os dados.
   - **Variantes**:
     - Repositório – MVC [5]
     - *Blackboard*;
     - *Publish/Subscribe*.

3. **MVC**

   - **Descrição**: É um caso especial do repositório, que é uma variação do estilo arquitetural centrado em dados. No MVC existem três módulos principais no sistema: (i) *Model*: responsável pelas entidades do domínio da aplicação (estrutura de dados); (ii) *View*: responsável pela exibição dos dados; e (iii) *Controller*: responsável pela execução dos eventos (implementação das funcionalidades) e por avisar o *view* sobre modificações no *model*. Pode ser considerado uma particularidade especial de estilo em camadas. Porém, nesse caso, o *controller* deve desempenhar o papel de intermediário entre o *view* e o *model*.
   - **Problema**: Transparência de localização entre os dados, o processamento da lógica do negócio e a exibição dos resultados.
   - **Vantagens**: (i) separação entre os dados (*model*), a visualização (*view*) e o processamento (*controller*), o que acarreta numa melhor manutenibilidade.
   - **Desvantagens**: (i) o desempenho pode ser comprometido, devido ao nível de indireção existente; (ii) nem todos os sistemas são mapeados facilmente com MVC.

4. ***Call/return***

   - **Descrição**: A comunicação de uma arquitetura *call/return* acontece a partir da chamada e retorno de funções. Dessa forma, cada componente do sistema é considerado um módulo funcional, que solicita serviços de outros módulos e responde aos serviços solicitados a ele. Devido à sia similaridade com o funcionamento das rotinas de programação atuais, o *call/return* é o estilo arquitetural mais adotado atualmente.
   - **Problema**: Sistemas que não são centrados em dados, que necessite ser decomposto/modularizado e que tenha a figura do controlador de execução, isto é, um módulo responsável por orquestrar a execução e tratar os retornos dos módulos envolvidos.
   - **Vantagens**: (i) uni ou bidirecionais; (ii)iniciado pelo "chamador"; (iii) possibilita a decomposição/modularização do sistema.
   - **Desvantagens**: (i) dificulta o controle da propagação dos erros, uma vez que a comunicação entre os componentes pode ser relaxada; (ii) quando bidirecional, implica num maior acoplamento entre os módulos.
   - **Variantes**:
     - Sub-rotinas;
     - Orientada a objetos;
     - Cliente-Servidor;
     - *Peer to peer*;
     - Camadas.

5. **Cliente-Servidor**

   - **Descrição**: Os sub-sistemas servidores oferecem os serviços a múltiplas instâncias de sub-sistemas clientes. Os clientes não se comunicam entre si.
   - **Problema**: Sistemas com a necessidade de separação explícita entre as funcionalidades relativas à execução do sistema (servidor) e ao tratamento dos retornos do sistema (cliente). Normalmente é utilizado em aplicações Web, quer seja puro, quer seja em conjunto com o estilo arquitetural em camadas.
   - **Vantagens**: (i) arquiteturas cliente/servidor aumentam a escalabilidade do sistema, uma vez que é possível adicionar servidores gradativamente; (ii) promove transparência de plataforma e de localização entre o cliente e o servidor.
   - **Desvantagens**: (i) não possibilita uma comunicação *peer to peer*.

6. **Camadas**

   - **Descrição**: Uma arquitetura em camadas é organizada hierarquicamente. Cada camada oferece serviço para a camada imediatamente acima e utiliza os serviços oferecidos pela camada imediatamente abaixo. Essa restrição de comunicação pode ser relaxada por questões de desempenho.
   - **Problema**: Um sistema grande que é caracterizado por um conjunto de partes de alto e baixo níveis, onde as partes de nível mais elevado dependem das partes de nível menos elevado.
   - **Vantagens**: (i) baixo acoplamento entre os componentes arquiteturais (melhor manutenibilidade e flexibilidade), uma vez que a comunicação é restrita a um dos sentidos; (ii) melhora o entendimento, uma vez que reduz a complexidade do sistema (representação em níveis); (iii) maior grau de reutilização, tendo em vista a maior independência.
   - **Desvantagens**: (i) o desempenho pode ser comprometido, pois pode impor um nível de indireção desnecessário; (ii) nem todos os sistemas são mapeados facilmente em camadas.

7. **Máquina Virtual**

   - **Descrição**: Em uma arquitetura de máquina virtual, o sistema é estruturado de forma a dividir a sua execução em dois passos: (i) interpretação dos comandos; e (ii) execução propriamente dita. Por essa razão, esse estilo arquitetural define um componente responsável unicamente pelo processo de tradução. Esse estilo pode ser representado através do estilo em camadas, onde uma das camadas é responsável pelo processo de tradução .
   - **Problema**: Necessidade de desconhecer totalmente o ambiente onde o sistema será executado, como por exemplo máquinas virtuais e/ou sistemas multiplataforma.
   - **Vantagens**: (i) abstrai o *hardware* específico ou real que necessita para executar (maior portabilidade e testabilidade).

- **Desvantagens**: (i) *overhead* de tradução para a linguagem nativa (menor desempenho).

8. **Componentes independentes**

   - **Descrição**: Uma arquitetura de componentes independentes representa o sistema através de um conjunto de processadores independentes (componentes), que se comunicam através de mensagens. Vale ressaltar que um componente não tem controle sobre o outro.
   - **Problema**: Sistemas modulares, compostos de componentes autônomos, que se comunicam entre si através de troca de mensagens. Nesses sistemas, os componentes podem ser vistos como sub-sistemas, que em outros contextos podem ser utilizados como sistemas particulares.
   - **Vantagens**: (i) alta escalabilidade, como conseqüência da possibilidade de se adicionar componentes gradativamente ao sistema; (ii) o tempo de mercado é reduzido, uma vez que essas unidades de processamento (os componentes) podem ser reutilizados e/ou desenvolvidos incrementalmente.
   - **Desvantagens**: (i) o fluxo de interação do sistema pode ficar confuso (compromete o entendimento e a manutenibilidade). Para corrigir essa deficiência, costuma-se utilizar esse estilo associado a outro mais restrito, como por exemplo o estilo em camadas.
   - **Variantes**:
     - *Communicating process* – arquitetura orientada a serviço (SOA[14]);
     - Baseado em eventos.

9. **Heterogêneo**

   - **Descrição**: Uma combinação de vários estilos arquiteturais.
   - **Problema**: Situações onde a adoção de um estilo específico não atende todos os requisitos e/ou não resolve o problema satisfatoriamente.
   - **Vantagens**: (i) tenta conciliar as vantagens de vários estilos arquiteturais, amenizando as desvantagens através de um ponto de equilíbrio entre as características de vários estilos.
   - **Desvantagens**: (i) a qualidade de uma arquitetura heterogênea depende muito da experiência do arquiteto.

3.2 Refinar a Visão Lógica da Arquitetura (ativ. 2)

Esta atividade consiste no detalhamento dos componentes da visão lógica da arquitetura. Com a redução progressiva da granularidade dos componentes, esse processo de detalhamento proporciona um mapeamento entre as abstrações da arquitetura para o desenvolvimento baseado em componentes (DBC). Em outras palavras, o objetivo desse refinamento

[14] do inglês *service-oriented architecture*

sucessivo é descobrir os componentes DBC que compõem os componentes arquiteturais do sistema.

Como pode ser visto na Figura 6, esse processo recebe dois artefatos como entrada: (i) o modelo conceitual do negócio; e (ii) a especificação dos requisitos do sistema, através de casos de uso UML [25]. A partir desses artefatos de entrada, são executadas sete atividades: (i) identificar as interfaces providas; (ii) identificar os sub-componentes; (iii) identificar as dependências entre as interfaces; (iv) identificar as interfaces requeridas; (v) realizar a configuração dos componentes; e dependendo da decisão do arquiteto, (vi) criar componentes para a implementação da lógica do negócio; e (vii) criar componentes inerentes ao estilo. As duas primeiras atividades, isto é, a identificação das interfaces providas e dos componentes, são derivadas do processo *UML Components* [7], com algumas adaptações. As demais são adaptadas do método MDCE+ [8, 12]. A seguir, as Seções 3.2.1 a 3.2.5 explicam cada uma dessas sete atividades.

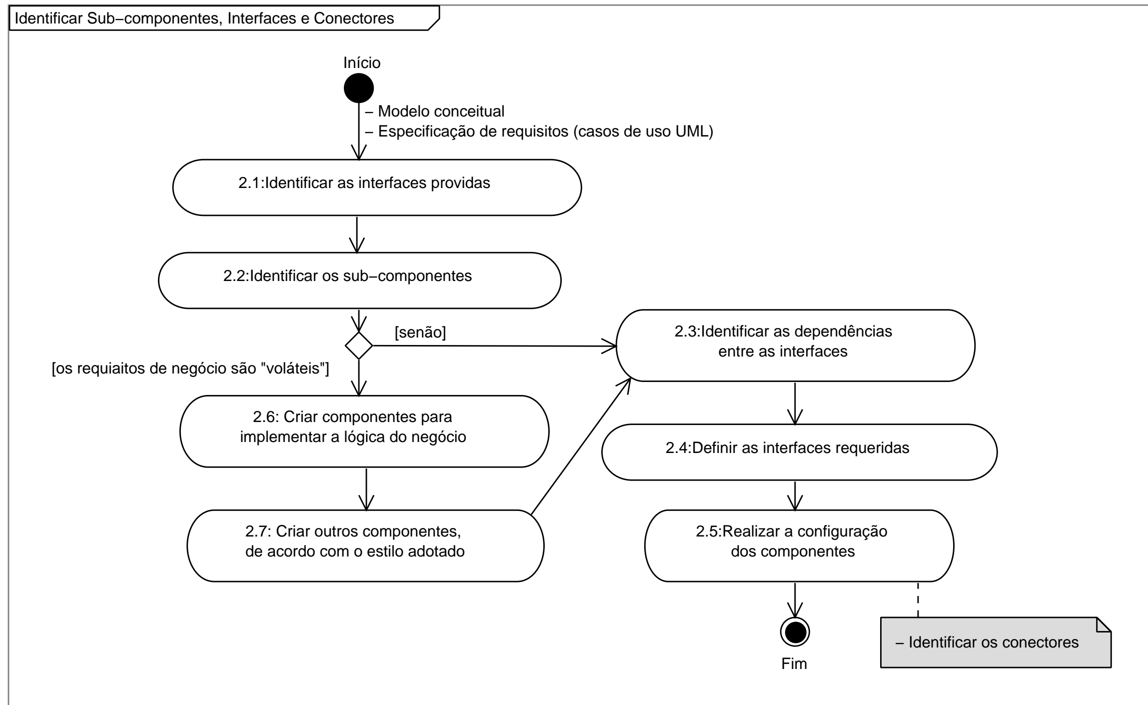

Figura 6: Identificação de Sub-componentes, Interfaces e Conectores

3.2.1 Identificar as interfaces providas (ativ. 2.1)

Assim como no processo *UML Components*, a identificação das interfaces providas se dá a partir do modelo conceitual e dos casos de uso do sistema. Como pode ser visto na Figura 7, essa atividade é executada em três passos: (i) processar os casos de uso; (ii) processar o modelo conceitual; e (iii) atribuir responsabilidade de implementação das interfaces. A seguir, cada um desses passos são detalhados.

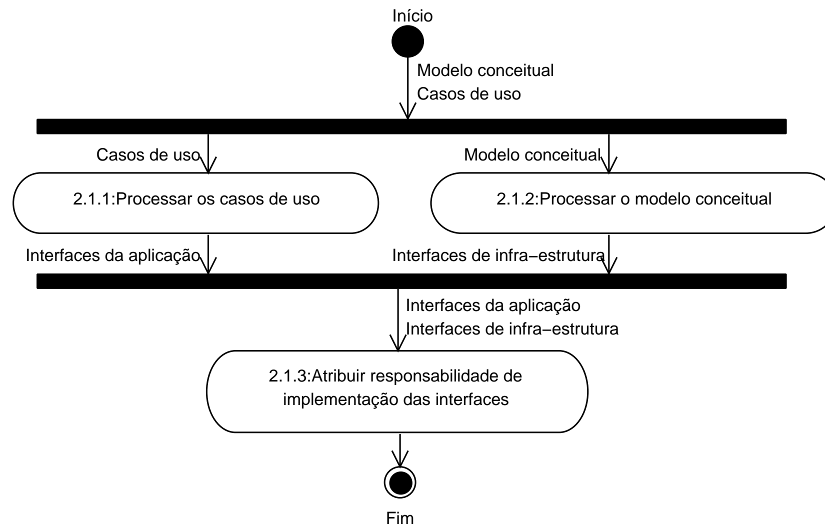

Figura 7: Identificação das Interfaces Providas

Ativ. 2.1.1 **Processar os casos de uso.** Como pode ser visto na Figura 8, para cada caso de uso especificado, é criada uma interface provida para o sistema. Em seguida, com o intuito de diferenciar as interfaces derivadas dos requisitos funcionais da aplicação, cada uma dessas interfaces descobertas é classificada como interface da aplicação.

Ativ. 2.1.2 **Processar o modelo conceitual.** Como pode ser visto na Figura 9, que mostra o *workflow* correspondente às atividades deste passo, os únicos artefatos de entrada necessários são: (i) o modelo conceitual; e (ii) a definição do escopo do sistema, representada através do diagrama de casos de uso. Ambos os artefatos são provenientes da fase de análise de requisitos do sistema.

A partir do modelo conceitual, o primeiro passo a ser executado é a restrição das entidades do domínio que realmente são relevantes para o escopo do sistema. Para isso, deve-se a partir do modelo inicial, adicionar e remover elementos, com o intuito de manter apenas as entidades que sejam relevantes para as funcionalidades de responsabilidade do sistema. Para complementar esse modelo estrito, pode ser necessário adicionar novos atributos que representem as restrições ou características específicas do sistema em questão.

Após a definição do **modelo de domínio do sistema**, o próximo passo é identificar as entidades principais desse modelo. Essas entidades são caracterizadas normalmente pelo seu papel de ligação entre outras entidades, possuindo assim um grande número de associações. Uma maneira de identificar as entidades principais é supor a inexistência de cada uma e tentar responder a seguinte pergunta: "caso ela não exista, as entidades associadas a ela continuam fazendo sentido (ainda que parcialmente)?" Se a resposta for sim, essa entidade não é uma candidata a ser considerada principal; caso contrário, se a resposta for não, essa entidade é uma séria candidata a ser consi-

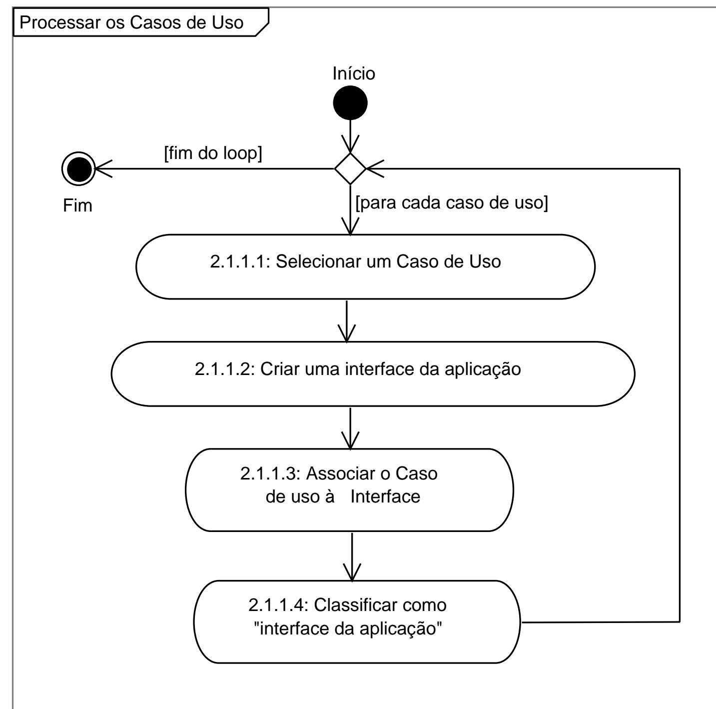

Figura 8: Processar os Casos de Uso

derada principal. Mas vale à pena lembrar que isso não é uma regra e que a decisão final é do analista.

A partir dessas entidades consideradas principais para o sistema, serão identificadas as interfaces de infra-estrutura. O procedimento é simples: para cada entidade principal, é criada uma interface, que é classificada como sendo de infra-estrutura, tendo em vista o seu papel básico para a implementação das funcionalidades do sistema.

A Figura 10 mostra um exemplo de um modelo de domínio de um sistema bancário.

Ativ. 2.1.3 **Atribuir responsabilidade de implementação das interfaces.** A atividade de atribuição de responsabilidade consiste na seleção do componente que implementará cada uma das interfaces identificadas. Como pode ser visto na Figura 11, para cada uma das interfaces identificadas, o desenvolvedor deve associar um componente como o responsável pela sua implementação.

3.2.2 Identificar os sub-componentes (ativ. 2.2)

Após a definição das interfaces dos componentes do sistema, pode-se iniciar o processo de decomposição dos componentes mais abstratos em sub-componentes. Como pode ser visto na Figura 12, o procedimento de decoposição dos componentes se baseia unicamente nas interfaces já identificadas.

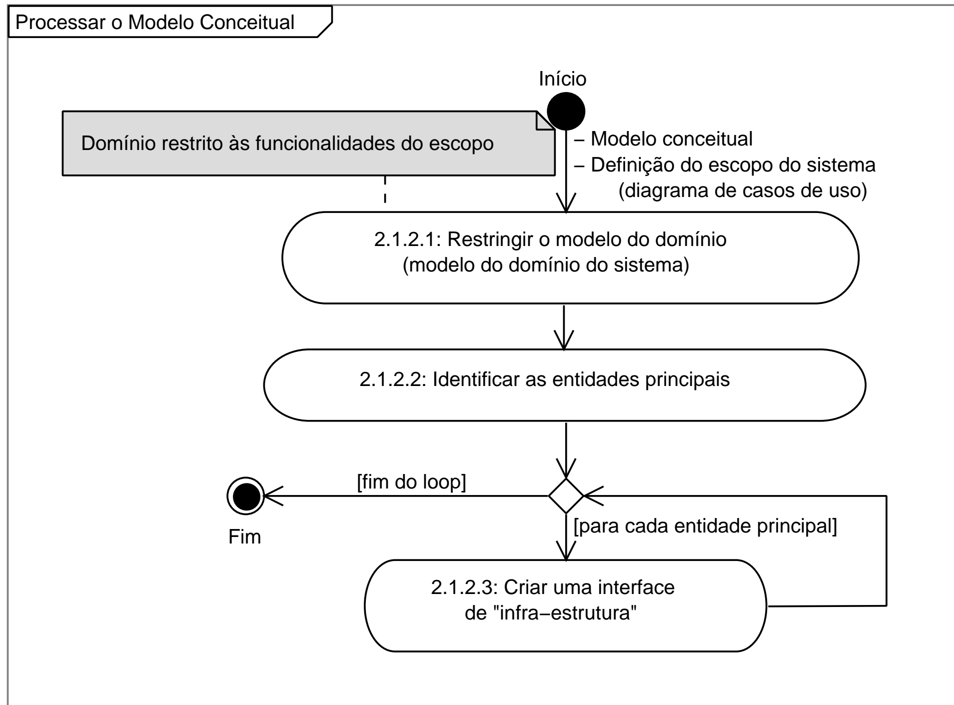

Figura 9: Processar o Modelo Conceitual

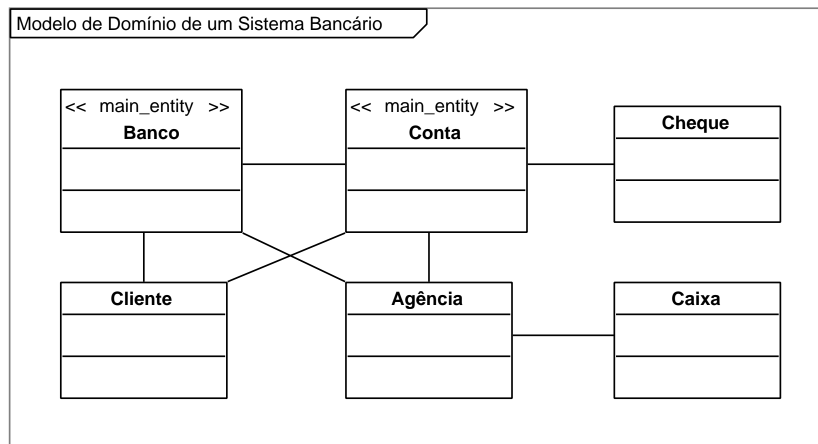

Figura 10: Modelo de Domínio de um Sistema Bancário

Para cada uma das interfaces, deve existir um sub-componente que a ofereça. Mas antes de se proceder a criação de um novo sub-componente, é necessário verificar a possibilidade de associar a interface a algum sub-componente já existente. No caso de não haver um sub-componente que possa oferecê-la, ele deve ser criado para que a associação seja feita. Caso

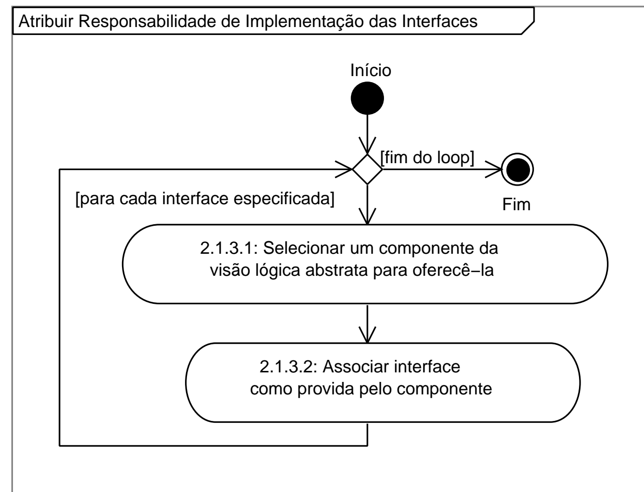

Figura 11: Atribuir Responsabilidade de Implementação das Interfaces

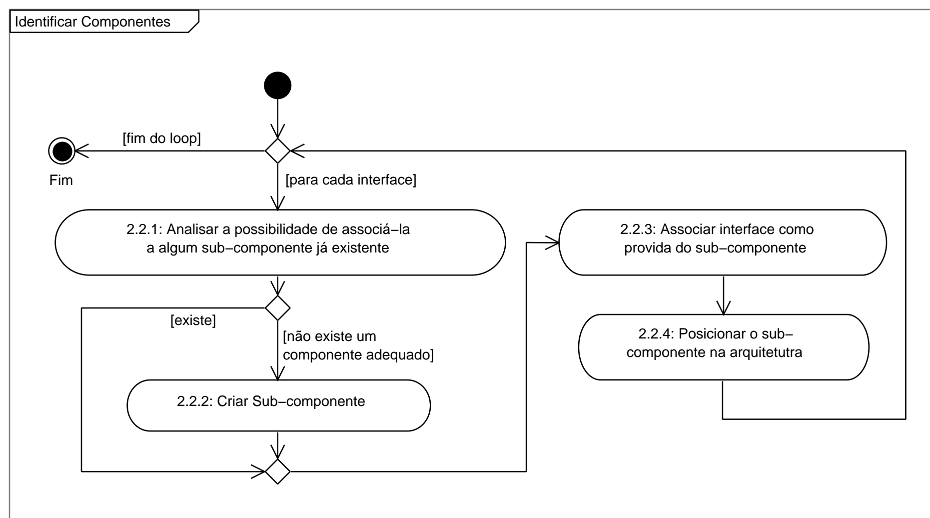

Figura 12: Identificar Componentes

a interface seja proveniente de um caso de uso, o sub-componente deve ser classificado como **componente da aplicação**, caso contrário, ele deve ser classificado como **componente de infra-estrutura**.

Em relação ao posicionamento na arquitetura, vale a pena salientar que os sub-componentes de infra-estrutura e da aplicação devem ser posicionados em componentes arquiteturais distintos.

Apesar do posicionamento dos componentes arquiteturais do sistema, ainda não se tem as informações necessárias para a representação das dependências entre eles. Por esse motivo, a arquitetura necessita ser refinada, o que será discutido na Seção 3.2.3.

3.2.3 Identificar as dependências entre as interfaces (ativ. 2.3)

O refinamento da arquitetura tem o objetivo de identificar os relacionamentos entre os componentes arquiteturais. Esse relacionamento engloba tanto a especificação das interfaces requeridas, quanto a conseqüente identificação dos conectores entre os componentes. Assim como o processo *UML Components*, este método propõe a especificação de simulações, tomando como foco central, a execução das funcionalidades especificadas através de casos de uso. A Figura 13 apresenta o *workflow* para a identificação das interfaces com as quais os componentes da aplicação necessitam interagir. Para cada uma das interfaces identificadas na Seção 3.2.1, deve-se analisar os possíveis cenários de execução dos seus serviços previstos. Com essa análise da execução seqüencial, é possível perceber de maneira mais intuitiva, quais interfaces com as quais esses componentes necessitam interagir. Em particular, se o sistema estiver em desenvolvimento, a informação das dependências pode ser extraída dos diagramas dinâmicos da UML, como por exemplo os diagramas de colaboração e de seqüência.

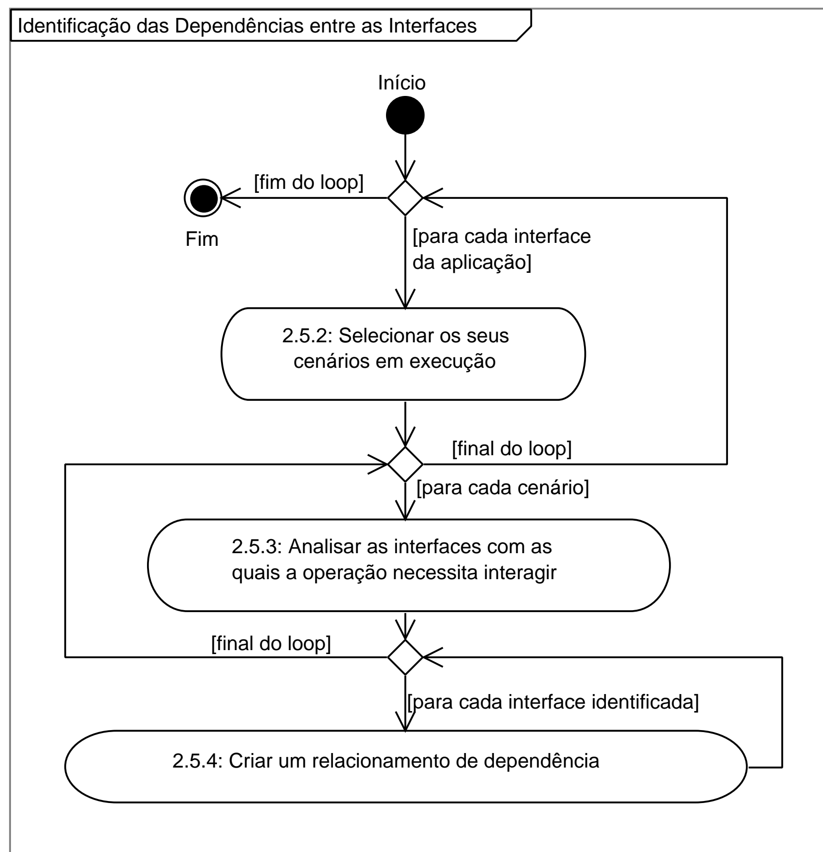

Figura 13: *Workflow* para Identificação das Dependências entre as Interfaces

3.2.4 Definir as interfaces requeridas (ativ 2.4)

Após a identificação das dependências entre as interfaces, é necessário definir as interfaces requeridas de cada componente arquitetural do sistema. Um número muito reduzido de interfaces requeridas de um componente pode provocar o agrupamento de operações pouco coesas. Além disso, como o método apresentado neste trabalho sugere a criação de um conector arquitetural para pada interface requerida do componente (Seção 3.2.5), o número de conectores seria reduzido e a complexidade de cada um seria elevada. Essa redução excessiva no número de conectores, aliada ao aumento da complexidade de cada um deles, representa uma desvantagem do ponto de vista dos aspectos relativos às atividades de manutenção do sistema.

De maneira análoga, o número exagerado de interfaces requeridas também pode dificultar o desenvolvimento do sistema. Dessa vez, a principal causa não é a dificuldade de manutenção, mas sim o aumento do custo da integração dos componentes do sistema, decorrente do número excessivo de conectores arquiteturais.

Tendo em vista essa situação de extremos, este método propõe uma solução intermediária, onde é criada uma interface requerida para cada camada da arquitetura que o componente interage, inclusive a sua própria camada, se for o caso. Por exemplo, supondo que o sistema possua uma arquitetura em camadas como a mostrada na Figura 14. Um componente arquitetural A, que pertence à camada de sistema, requer quatro operações de outros componentes. Dessas quatro, duas são oferecidas por componentes da própria camada de sistema e as outras duas por componentes da camada de negócios. Neste caso, o componente A teria duas interfaces requeridas, uma para cada camada (Figura 15).

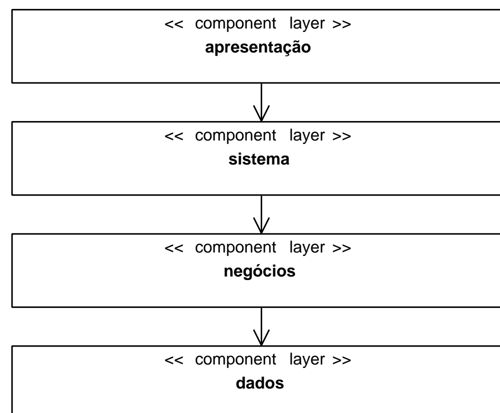

Figura 14: Uma Arquitetura em Quatro Camadas

3.2.5 Realizar a configuração dos componentes (ativ. 2.5)

Apesar das dependências entre os componentes e das interfaces providas e requeridas de cada um já estarem definidas, ainda nos falta materializar as conexões entre eles. Essas conexões são realizadas através de uma ligação entre as interfaces requeridas de um componente e as

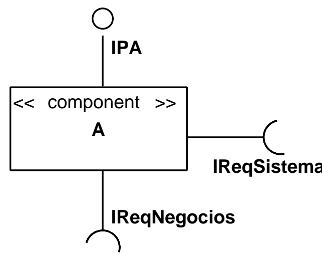

Figura 15: Exemplo do Componente A

respectivas interfaces providas de outros. Dessa forma, a materialização das configurações do sistema consiste basicamente de dois passos: (i) criação dos conectores; e (ii) ligação entre as interfaces providas e requeridas.

Por representar o elo de comunicação entre componentes arquiteturais, esses conectores são os únicos componentes do sistema que possuem o conhecimento dos seus fluxos interativos. Dessa forma, eles são considerados os locais ideais para a implementação dos atributos de qualidade do sistema, tais como confiabilidade e disponibilidade[15].

3.2.6 Criar componentes para implementar a lógica do negócio (ativ. 2.6)

Dependendo do nível de alteração que a lógica do negócio pode sofrer durante o ciclo de vida do software, vale a pena projetar a arquitetura do sitema de forma a minimizar o custo de alteração dessas regras. Essa minimização de esforço pode ser alcançada através da separação de interesse entre os componentes que implementam a lógica do negócio em si e os componentes menos dependentes dessas regras.

O método apresentado nesse documento propõe que essa separação de interesse seja realizada através da identificação de componentes com o papel explícito de implementar as operações relacionadas com a lógica do negócio. Em outras palavras, esses componentes devem implementar somente as operações que coordenam a execução dos fluxos dos casos de uso, determinando a ordem de execução de cada passo do cenário e as decisões evolvidas no processo de execução das funcionalidades. Essa organização é semelhante ao estilo arquitetural *call/return*, mais especificamente a variante sub-rotina, vistos na Seção 3.1.1.

3.2.7 Criar outros componentes de acordo com o estilo adotado (ativ. 2.7)

Dependendo do estilo arquitetural adotado pelo desenvolvedor, alguns componentes arquiteturais podem não ter sido analisados pelo método. Nesses casos, cada um dos componentes do estilo arquitetural (detalhados ou não detalhados) deve ser analisado, com o intuito de verificar a existência de sub-componentes não identificados e que possam ser necessários para desempenhar algum dos dois papéis seguintes: (i) representar uma quebra em módulos funcionais, que representem uma separação funcional resultante de um agrupamento de requisitos funcionais inerentes do próprio estilo; ou (ii) identificar os componentes resultantes da

[15] do inglês *availability*

influência de alguns requisitos não-funcionais especificados, como por exemplo a diversidade de interfaces gráficas, tais como sistemas com interfaces *stand alone* e *Web* simultaneamente.

O objetivo principal dessa atividade é possibilitar a identificação dos componentes essenciais do ponto de vista da arquitetura, cujo esquecimento possa representar um custo muito alto de manutenção corretiva. Mas nada impede que algum componente arquitetural não seja decomposto em sub-componentes. Em outras palavras, há casos em que um componente arquitetural de alto nível é mapeado diretamente em um componente de desenvolvimento (componente DBC). Um caso comum onde isso ocorre é o componente de peristência de dados.

4 Estudo de Caso: Sistema de Controle de Hotéis

Esta seção apresenta um estudo de caso que utiliza o método apresentado na Seção 3 para especificar a arquitetura de um sistema de controle de hotéis. Inicialmente, foi especificada a visão lógica mais abstrata do sistema, que representa os componentes arquiteturais em um nível de abstração bastante alto. Para essa etapa da especificação da arquitetura foi utilizado o método apresentado na Seção 3.1. Após a especficação da visão lógica inicial, os componentes arquiteturais identificados foram detalhados, de acordo com o estilo arquitetural de componentes independentes[16] [3]. Dessa forma, esse detalhamento consistiu na identificação dos componentes e as suas respectivas interfaces providas e requeridas, seguindo o método apresentado na Seção 3.2.

A Seção 4.1 apresenta a descrição do sistema especificado, mostrando seus principais requisitos. As Seções 4.2 e 4.3 apresentam as duas etapas do projeto arquitetural do sistema.

4.1 Requisitos do sistema

O sistema utilizado no estudo de caso tem a função de gerenciar reservas e ocupações de apartamentos de uma rede de hotéis. Em cada hotel, terá um ou mais terminais para controlar os serviços internos e a comunicação entre hotéis da mesma rede de forma a consultar sobre disponibilidade de vagas em outras unidades da mesma cidade ou região. Além disso, permite o cliente fazer reservas e cancelamento de reservas através da Web. Esse sistema também deve interagir com outros dois sistemas internos do hotel: controle de restaurante, controle de tarifação de telefone e controle de cadastro pessoal (clientes e funcionários).

As funções básicas de controle que devem ter são: cadastro de cliente, cadastro de apartamentos, gerenciamento de reservas e ocupações, gerenciamento de pagamento, emissão de nota fiscal, emissão de relatórios contábeis e reservas pela Web. As Seções 4.1.1 a 4.1.6 detalham os requisitos do sistema.

4.1.1 Modelo Conceitual

A Figura 16 apresenta o modelo conceitual desenvolvido para o sistema de controle de hotéis do estudo de caso. Nessa figura, já estão destacadas as entidades principais, representadas

[16] do inglês *independent components*

com o estereótipo << *main_entity* >>.

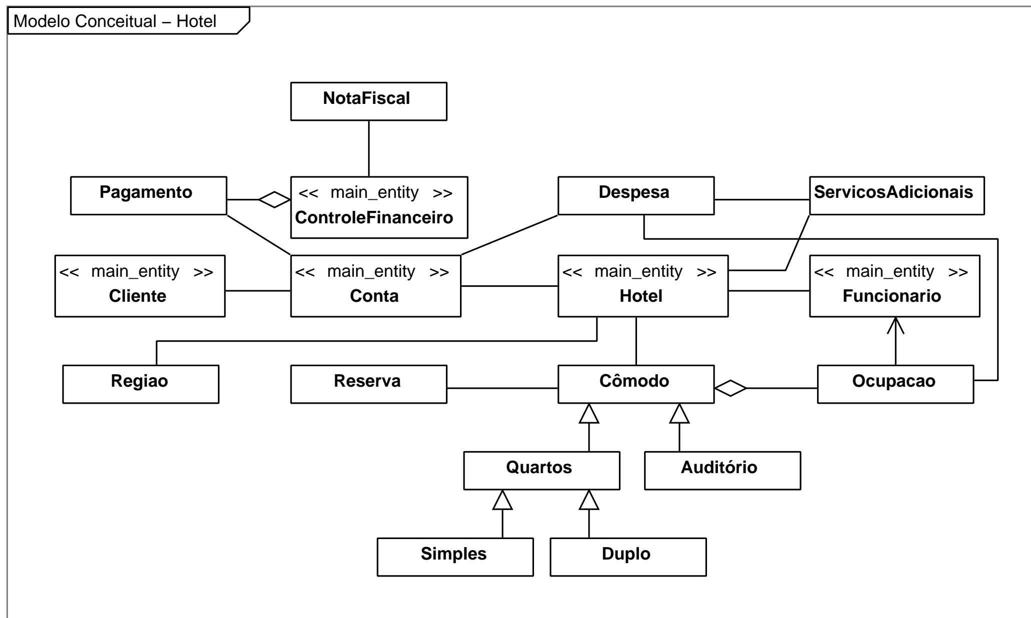

Figura 16: Modelo Conceitual do Sistema de Hotéis

4.1.2 Requisitos funcionais

- Consultas, reservas e cancelamento de reserva através da Web.
- Cadastro de apartamento: tipo de apartamento (suíte, standard, duplo, ar-condicionado), cidade ou local.
- Cadastro de salas e auditório.
- Serviços adicionais são também incluídos no sistema: telefone, TV paga, acesso à internet, frigobar, lavanderia, serviço de lanche e café da manhã.
- Conexão para consultas e reservas de vagas em outros hotéis do grupo.
- Controle de ocupação de apartamentos (reservado ou entrada do hóspede).
- Controle de ocupação de salas e auditório.
- Recebimento de pagamento (tipo de pagamento cheque, dinheiro, cartão, parcelado, moeda estrangeira).
- Registrar situações de pagamento (cheque compensado, transferência realizada, parcelado, em dinheiro, ou moeda estrangeira).

- Emissão de nota fiscal.
- Emissão da fatura parcial (somente para consulta).
- Emissão de relatórios contábeis.
- Emissão de relatórios de ocupação.
- Emissão de relatórios de hóspedes em débito.
- Relatórios parciais de consulta.
- Os relatórios e consultas deverão também ser visualizados pelo terminal.

4.1.3 Diagrama de Casos de Uso

A Figura 17 apresenta o diagrama de casos de uso relativo ao sistema de controle de hotéis. Para fins de entendimento e simplificação da especificação, só foram representados os atores primários, isto é, quem interage diretamente com cada uma das funcionalidades do sistema.

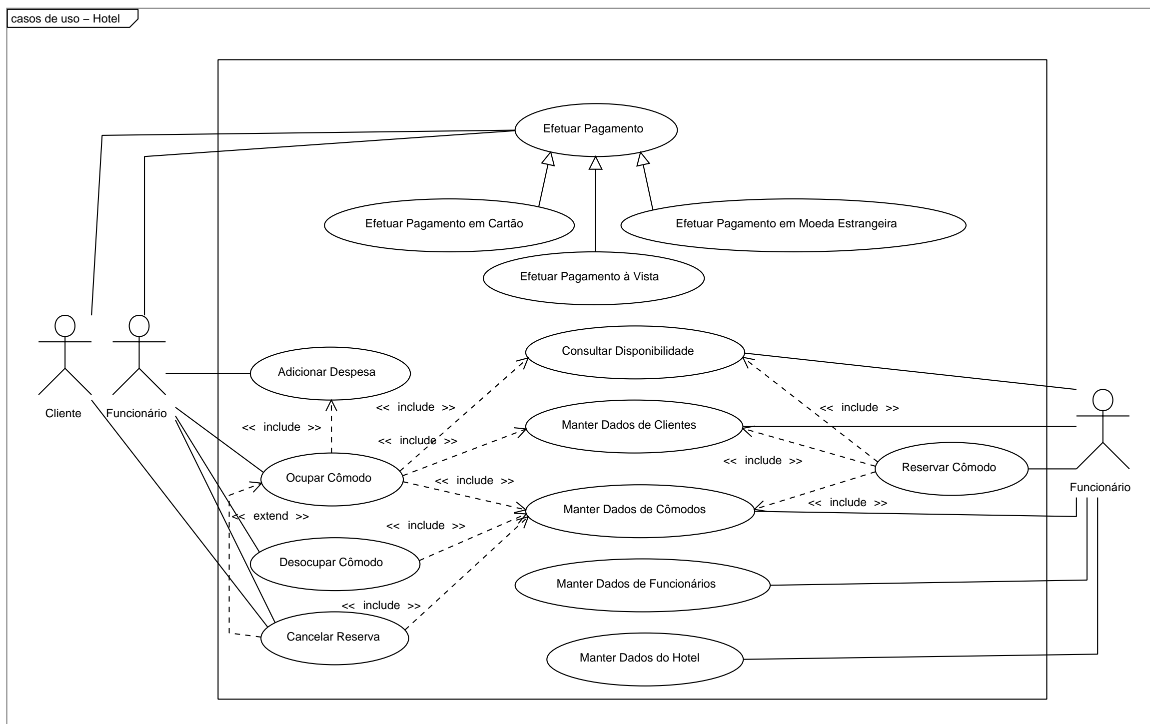

Figura 17: Diagrama de Casos de Uso do Sistema de Hotéis

4.1.4 Requisitos não-funcionais

- **Distribuição**: cada hotel da rede deve ser responsável por gerenciar seus cômodos, mas deve se comunicar com os demais sistemas da rede. Por questões de custo, a comunicação deve ser feita pela Web.
- **Desempenho**: o tempo de resposta desejável é menor que 10 segundos para consultas de vagas em outros hotéis da rede.
- **Portabilidade**: o módulo interno do sistema deve ser acessível a partir dos sistemas operacionais *Windows* e *Linux*. O módulo Web deve ser acessível a partir dos *browsers Internet Explorer* [18] e *Firefox* [21].
- **Manutenibilidade**: o sistema deve apresentar facilidades para: (i) adicionar novas funcionalidades: deve ser possível desenvolver o sistema incrementalmente, conforme as restrições de desenvolvimento apresentadas na Seção 4.1.5; (ii) evoluir: o custo de se trocar algum componente específico do sistema deve ser minimizado; e (iii) alterar a lógica do negócio: a lógica do negócio costuma ser alterada com freqüência. O custo dessa alteração deve ser minimizando.

4.1.5 Restrições do desenvolvimento

- Por questões de custo, a comunicação entre os hotéis da rede deve ser feita pela Internet.
- Devem ser utilizados computadores PCs do mercado.
- O sistema deve ser desenvolvido na linguagem JAVA.

4.1.6 Requisitos de desenvolvimento e manutenção

- O produto pode ser desenvolvido em etapas, mas deverá ter as funcionalidades básicas na primeira versão (gerenciar reservas e ocupação de apartamentos, cadastro de clientes, controle de pagamento, e reservas pelas Web.
- O prazo de desenvolvimento para as funcionalidades básicas é de seis meses.
- Após o desenvolvimento das funcionalidades básicas, o sistema deverá ser colocado em operação por três meses antes de se iniciar o desenvolvimento de outras funcionalidades (**desenvolvimento incremental**).
- Após os três primeiros meses de funcionamento, o produto deverá ser reavaliado para inserir melhorias, corrigir falhas do sistema e implementar as novas funcionalidades.
- O prazo estimado para implementação desta segunda fase é de seis meses.
- Após o desenvolvimento da segunda fase, o sistema deverá ser colocado em operação e terá três meses para corrigir eventuais falhas.

- Garantia: O desenvolvedor do produto deverá dar suporte gratuito durante um ano após a entrega do produto para casos de mau funcionamento do sistema.

- Deverá fornecer treinamento aos usuários.

- Deverá fornecer o manual de usuário do produto e de manutenção.

4.2 Visão Lógica Inicial da Arquitetura

A Figura 18 apresenta a visão lógica inicial da arquitetura, após a identificação dos componentes arquiteturais mais abstratos. Como pode ser visto nessa figura, foi adotado o estilo arquitetural em camadas. Dessa forma, cada componente arquitetural do sistema corresponde a uma camada. Além disso, as camadas superiores conhecem somente a sua camada imediatamente inferior, o que reduz o acoplamento entre os componentes e facilita a evolução do sistema.

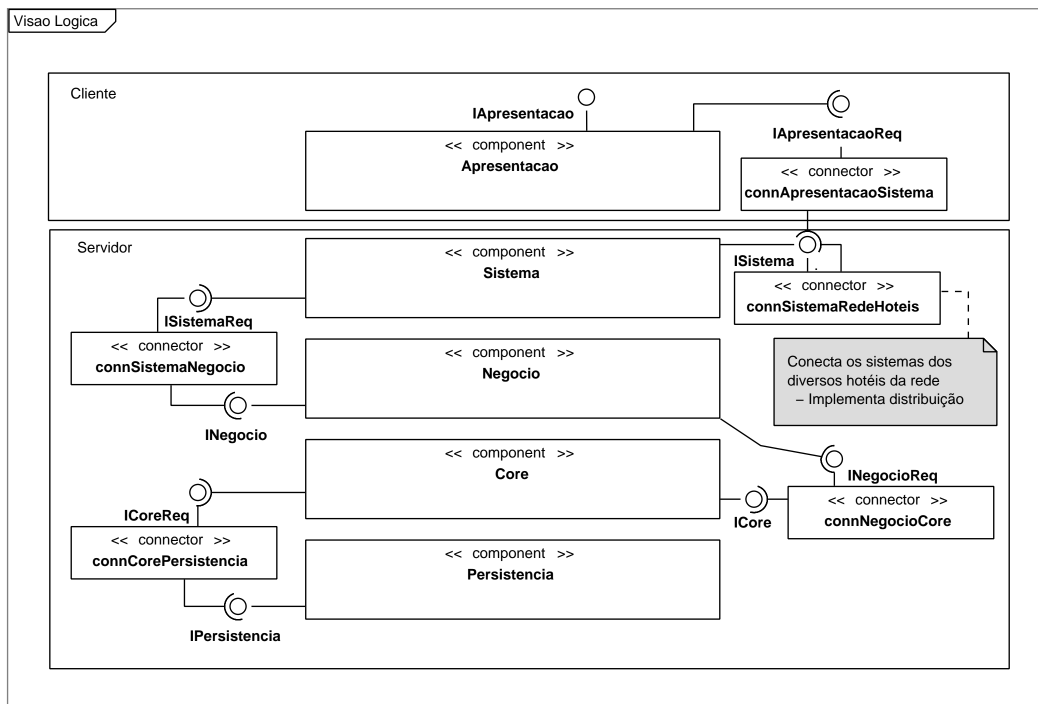

Figura 18: Visão Lógica Inicial da Arquitetura

A Figura 19 apresenta a árvore de precedência entre os requisitos não-funcionais identificados para o sistema. Além disso, são apresentados alguns cenários de uso que ilustram as necessidades desses atributos de qualidade, elicitados durante a especificação dos requisitos do sistema. A seguir, é dada uma explicação sobre como a arquitetura proposta atende cada um desses requisitos.

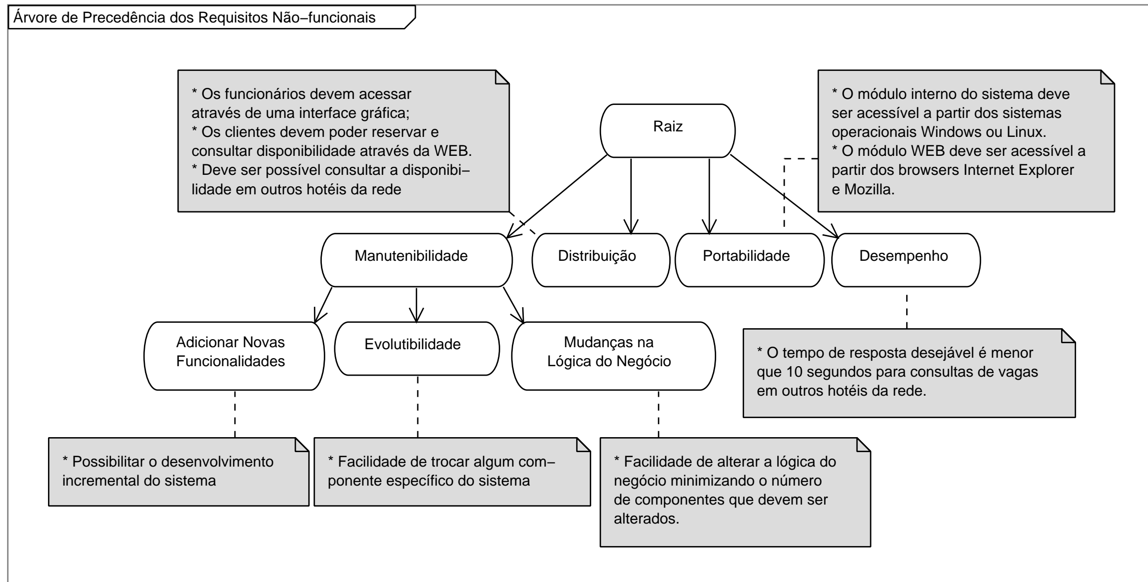

Figura 19: Árvore de Precedência entre os Requisitos Não-funcionais

1. **Manutenibilidade**: A facilidade para adicionar novas funcionalidades ao sistema deve ser alcançada através da possibilidade de "quebrar cada camada do sistema em subsistemas de granularidades menores, de acordo com a necessidade do desenvolvedor." A facilidade de evolução é alcançada através do baixo acoplamento decorrente da obrigatoriedade dos conectores arquiteturais; dessa forma, a adaptação necessária para a substituição de componentes fica centralizada e facilitada. Por fim, a camada de `Sistema` é responsável pela orquestração da lógica do negócio, representada através dos cenários dos casos de uso; dessa forma, uma mudança simples nas regras de negócio, como por exemplo a mudança de ordem de execução dos passos dos cenários, acarretaria unicamente na alteração dos componentes dessa camada.

2. **Distribuição**: A distribuição foi materializada de duas formas: através da adoção do estilo arquitetural `cliente-servidor`, que possibilita a execução remota da interface do cliente; e através da existência de um conector explícito para a comunicação entre hotéis da rede (`connSistemaRedeHoteis`). Isso fica mais claro após o detalhamento da arquitetura, apresentado na Seção 4.3.

3. **Portabilidade**: Esse requisito não-funcional será materializado através da tecnologia adotada. A utilização de Java para a implementação da aplicação (lado servidor) permitirá a utilização simutânea do sistema em ambientes *Windows* e *Linux*. Além disso, a portabilidade do acesso Web foi considerado uma particularidade inerente às tecnologias existentes atualmente.

4. **Desempenho**: O desempenho deverá ser alcançado através da utilização de tecnologias ágeis de comunicação remota. Inicialmente, o conector `connSistemaRedeHoteis`

deve ser implementado utilizando tecnologias de serviços Web[17] [2]. Porém, caso o desempenho não seja satisfatório, esse conector deve ser refatorado para proceder uma implementação em RMI[18] [23], específica da tecnologia Java.

4.3 Visão lógica detalhada da arquitetura

Seguindo o método apresentado na Seção 3.2, o primeiro passo a ser executado foi identificar as interfaces providas dos componentes. Para isso, foram processados tanto os casos de uso do sistema (Figura 17), quando o seu modelo conceital (Figura 16), descritos na Seção 4.1.

As Tabelas 2 e 3 apresentam respectivamente as interfaces providas pelos componentes da aplicação, derivadas dos casos de uso, e pelos componentes de infra-estrutura, derivadas do modelo conceitual.

Tabela 2: Interfaces Providas pelos Componentes da Aplicação (Casos de Uso)

Interfaces da Aplicação (Casos de Uso)	Caso de Uso que a Originou
`IOpPagamento`	Efetuar Pagamento Efetuar Pagamento em Cartão Efetuar Pagamento à Vista Efetuar Pagamento em Moeda Estrangeira
`IOpDespesas`	Adicionar Despesa
`IOpComodos`	Ocupar Cômodo Desocupar Cômodo Reservar Cômodo Cancelar Reserva Consultar Disponibilidade
`IOpCadastros`	Manter Dados de Clientes Manter Dados de Funcionários Manter Dados de Cômodos Manter Dados do Hotel

Tabela 3: Interfaces Providas pelos Componentes de Infra-estrutura (Modelo Conceitual)

Interfaces de Infra-estrutura (Modelo Conceitual)	Entidade Conceitual que a Originou
`IFinMgr`	ControleFinanceiro
`IContaMgr`	Conta
`IHotelMgr`	Hotel
`IClienteMgr`	Cliente
`IFuncionarioMgr`	Funcionario

[17] do inglês *Web services*
[18] do inglês *Remote Method Invocation*

- **Componentes identificados a partir das interfaces providas**
  - `OperacoesPagamento.` Implementa a interface `IOpPagamento` e depende das interfaces `FinanceiroMgr.IFinMgr`, `ContaMgr.IContaMgr`, `HotelMgr.IHotelMgr`, `ClienteMgr.ICadCliMgr` e `FuncionarioMgr.ICadFunMgr`.
  - `OperacoesDespesas.` Implementa a interface `IOpDespesas` e depende das interfaces `ContaMgr.IContaMgr` e `HotelMgr.IHotelMgr`.
  - `OperacoesComodos.` Implementa a interface `IOpComodos` e depende das interfaces `HotelMgr.IHotelMgr`, `ClienteMgr.ICadCliMgr` e `FuncionarioMgr.ICadFunMgr`.
  - `OperacoesCadastros.` Implementa a interface `IOpCadastros` e depende das interfaces `HotelMgr.IHotelMgr`, `ClienteMgr.ICadCliMgr` e `FuncionarioMgr.ICadFunMgr`.
  - `FinanceiroMgr.` Implementa a interface `IFinMgr` e depende da interface `Persistencia.IPersistencia`.
  - `ContaMgr.` Implementa a interface `IContaMgr` e depende da interface `Persistencia.IPersistencia`.
  - `HotelMgr.` Implementa a interface `IHotelMgr` e depende da interface `Persistencia.IPersistencia`.
  - `ClienteMgr.` Implementa a interface `ICadCliMgr` e depende da interface `Persistencia.IPersistencia`.
  - `FuncionarioMgr.` Implementa a interface `ICadFunMgr` e depende da interface `Persistencia.IPersistencia`.
- **Componentes derivados de sistemas já existentes**
  - `SistemaServicosAdc.` Os sistemas de controle de restaurante e de controle de tarifação telefônica foram encapsulados em um componente maior, o componente `SistemaServicosAdc`, que implementa as operações relativas a qualquer tipo de serviço adicional que o hotel possa oferecer. Esse componente implementa a interface `IServAdc` e depende das interfaces `OperacoesPagamento.IOpPagamento` e `OperacoesDespesas.IOpDespesas`.
  - `CadastroPessoalMgr.` O sistema de controle de cadastro pessoal está representado pelo componente `CadastroPessoalMgr`, que oferece a interface `ICadPMgr` e depende da interface `Persistencia.IPersistencia`. Dessa forma, esse componente é responsável pela manutenção dos cados dos clientes e dos funcionários, substituindo os componentes `ClienteMgr` e `FuncionarioMgr`, identificados anteriormente. Sendo assim, os componentes que dependiam de `ClienteMgr.ICadCliMgr` e `FuncionarioMgr.ICadFunMgr`, passaram a depender de `CadastroPessoalMgr.ICadPMgr`.

- **Identificação dos conectores**

  Seguindo as atividades do método, foi identificado um conector para cada camada que o componente depende. Ao todo, foram identificados 15 conectores.

  - `connAtendenteSist.` Liga o componente `TerminalAtendente` à camada de `Sistema`.
  - `connWebSist.` Liga o componente `TerminalWeb` à camada de `Sistema`.
  - `connServicosAdcNeg.` Liga o componente `SistemaServicosAdc` à camada de `Negócio`.
  - `connOcupSistema.` Liga o componente `SistemaOcupacao` à própria camada de `Sistema`. Esse conector implementa um protocolo de execução remota.
  - `connOcupNeg.` Liga o componente `SistemaOcupacao` à camada de `Negócio`.
  - `connConsResNeg.` Liga o componente `SistemaConsultaEReserva` à camada de `Negócio`.
  - `connCadastroNeg.` Liga o componente `SistemaCadastros` à camada de `Negócio`.
  - `connOpPagCore.` Liga o componente `OperacoesPagamento` à camada `Core`.
  - `connOpDespCore.` Liga o componente `OperacoesDespesas` à camada `Core`.
  - `connOpComodosCore.` Liga o componente `OperacoesComodos` à camada `Core`.
  - `connOpCadastrosCore.` Liga o componente `OperacoesCadastros` à camada `Core`.
  - `connFinancPers.` Liga o componente `FinanceiroMgr` à camada de `Persistencia`.
  - `connContaPers.` Liga o componente `ContaMgr` à camada de `Persistencia`.
  - `connHotelPers.` Liga o componente `HotelMgr` à camada de `Persistencia`.
  - `connCadastroPessoalPers.` Liga o componente `CadastroPessoalMgr` à camada de `Persistencia`.

- **Identificação dos componentes controladores**, que conterão as operações que implementam a lógica do negócio (que implementam os cenários cos casos de uso).

  Como o projeto arquitetural do sistema prioriza a modularidade e a facilidade de evolução do sistema, julgou-se necessário criar mais uma camada de componentes na arquitetura (camada de `Sistema`), como pode ser visto na visão lógica inicial, apresentada na Figura 18.

  - `SistemaOcupacao.` Implementa a interface `IOcupacao` e depende das interfaces `OperacoesDespesas.IOpDespesas` e `OperacoesComodos.IOpComodos`.
  - `SistemaConsultaEReserva.` Implementa as interfaces `IConsulta` e `IReserva` e depende da interface `OperacoesComodos.IOpComodos`.

  - `SistemaCadastros.` Implementa a interface `ICadastros` e depende da interface `OperacoesCadastros.IOpCadastros`.

- **Componentes criados de acordo com o estilo arquitetural adotado**
  - `TerminalAtendente.` Não possui nenhuma interface provida e depende das interfaces `SistemaServicosAdc.IServAdc`, `SistemaOcupacao.IOcupacao`, `SistemaConsultaEReserva.IConsulta`, `SistemaConsultaEReserva.IReserva` e `SistemaCadastros.ICadastros`.
  - `TerminalWeb.` Não possui nenhuma interface provida e depende das interfaces `SistemaConsultaEReserva.IConsulta`, `SistemaConsultaEReserva.IReserva` e `SistemaCadastros.ICadastros`.
  - `Persistencia.` Implementa a interface `IPersistencia` e não possui nenhuma interface requerida.

A Figura 20 apresenta a visão lógica final da arquitetura, após o detalhamento dos componentes arquiteturais mais abstratos.

5 Conclusões e Trabalhos Futuros

Por conhecer o fluxo interativo entre os componentes do sistema, a arquitetura do sistema é o local ideal para a implementação dos seus requisitos não-funcionais. Por esse motivo, a fase de projeto arquitetural deve receber uma atenção especial, uma vez que mudanças tardias na arquitetura costumam ter um custo elevado [28]. Apesar de levar em consideração principalmente os requisitos não-funcionais e os atributos de qualidade desejados para o sistema, a fase de projeto arquitetural também deve levar em consideração os requisitos funcionais especificados. Dessa forma, além de aumentar a qualidade do sistema, a arquitetura também tem o papel de facilitar a execução das atividades de manutenção, proporcionando uma melhor modularidade e um melhor entendimento da estrutura geral do sistema. Porém, projetar a arquitetura de um sistema é uma tarefa complexa e que, normalmente, depende da experiência dos desenvolvedores, especialmente do arquiteto.

Este relatório técnico apresentou um método para sistematizar a especificação da visão lógica da arquitetura do sistema, refinada com o estilo arquitetural de componentes independentes[19] [3]. Dessa forma, o objetivo desse método é auxiliar o desenvolvedor na tarefa de projetar a arquitetura do sistema, reduzindo a penalidade decorrente da pouca experiência da equipe de desenvolvimento.

Inicialmente, a Seção 3.1 apresenta uma maneira de identificar a visão lógica em um alto nível de abstração. Em seguida, a Seção 3.2 apresentou as atividades necessárias para o detalhamento da arquitetura especificada inicialmente.

Como pode ser percebido, o projeto arquitetural de um sistema de software é uma fase muito dependente da experiência do arquiteto. O desenvolvimento de métodos que auxiliam o desenvolvedor nesta fase é uma forma de reduzir essa dependência, oferecendo diretrizes

[19] do inglês *independent components*

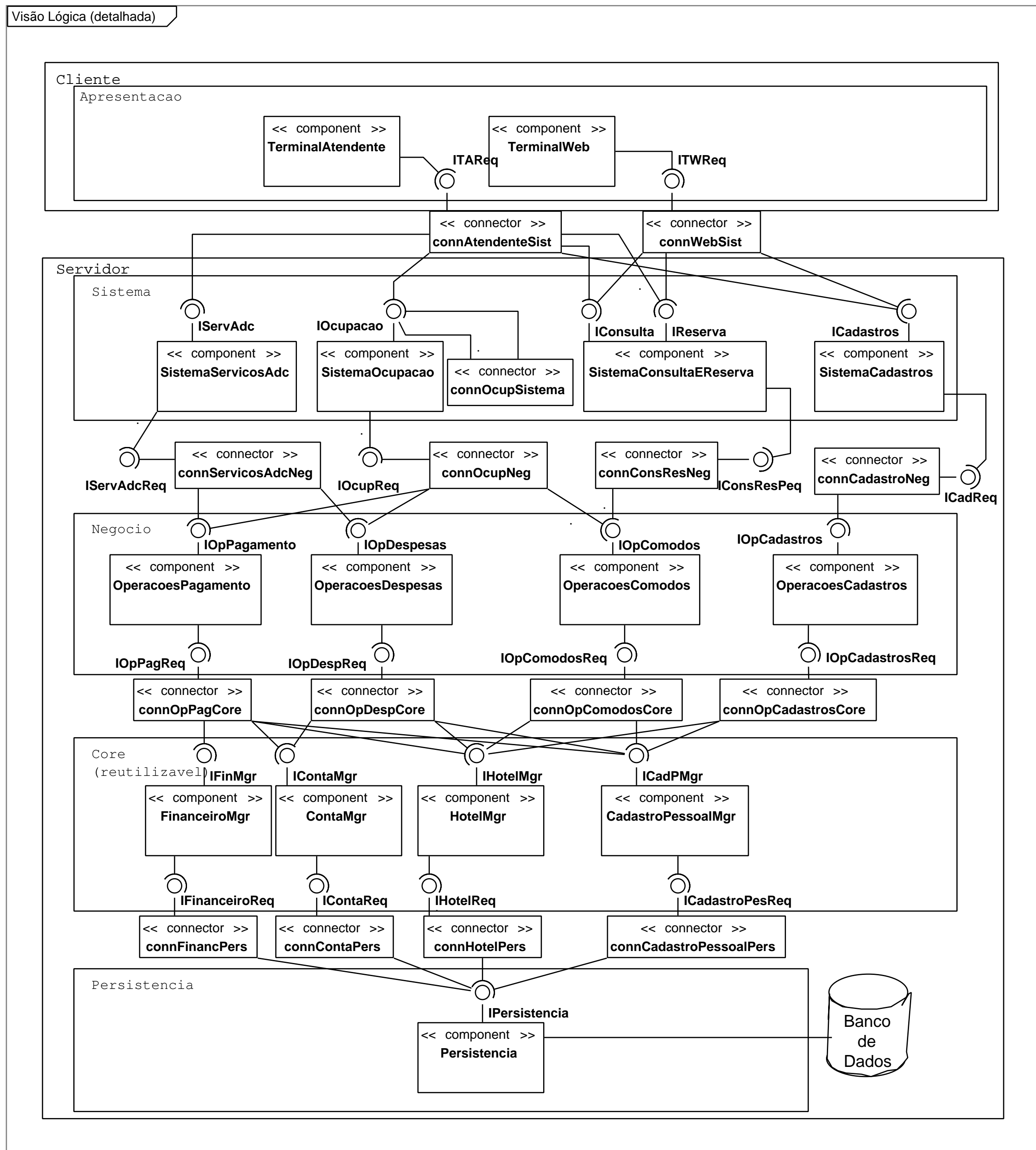

Figura 20: Visão Lógica Detalhada da Arquitetura

que facilitam a especificação de arquiteturas de qualidade, de acordo com os requisitos funcionais e não-funcionais estipulados pelo cliente.

A utilidade de um método de especificação de arquiteturas de software é evidenciada ainda mais no contexto dos processo de desenvolvimento centrados na arquitetura, como é o caso dos processos de desenvolvimento voltados para o reuso de componentes. Para concretizar a adaptação do método de especificação de arquiteturas nesses processos, seria necessário proceder as duas atividades seguintes: (i) identificar as atividades onde a arquitetura é especificada/refinada; e (ii) utilizar o método proposto no documento atual para efetuar o projeto arquitetural.

Como um exemplo de adaptação, a Figura 21 apresenta o *workflow* principal de execução do processo de desenvolvimento com reuso, proposto em um outro relatório técnico do Instituto de Computação da Unicamp [9]. Nesse processo, o método de especificação da arquitetura proposto neste documento, poderia ser utilizado como detalhamento da segunda atividade: projetar a arquitetura do sistema, destacada em cor cinza.

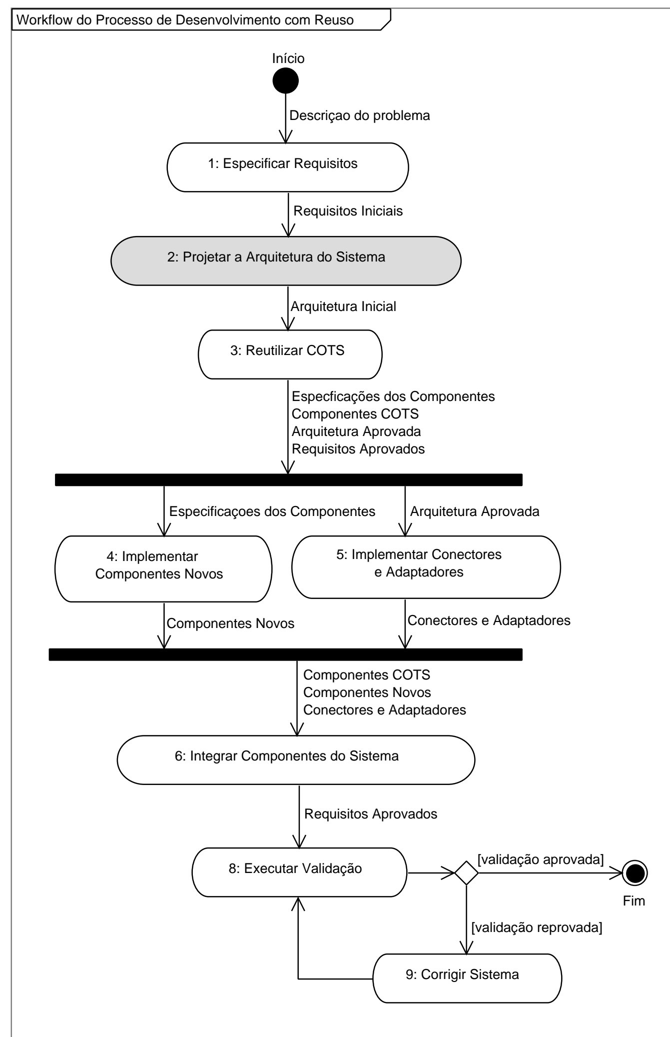

Figura 21: *Workflow* do Processo de Desenvolvimento com Reuso [9]

Em relação às extensões do trabalho, uma possibilidade imediata seria a elaboração de estudos de caso maiores e em empresas reais, com o intuito de verificar a viabilidade prática do método e, ao mesmo tempo, ajustá-la melhor às necessidades específicas do mercado desenvolvedor de software. Além disso, uma outra possibilidade de melhoria seria o detalhamento do catálogo de padrões e problemas típicos; o que contribuiria diretamente para a melhoria da qualidade da arquitetura abstrata proposta inicialmente (Seção 3.1).

Referências

[1] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: The kobrA approach. In P. Donohoe, editor, *Proceedings of theFirstSoftware Product Line Conference*, pages 289–309, 2000.

[2] Douglas K. Barry. *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide*. Elsevier Science, USA, 2003.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] Len Bass, Mark Klein, and F. Bachmann. Quality attribute design primitives. Technical report CMU/SEI-2000-TR-017, Pittsburgh, PA, USA, 2000.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[6] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. *26th Annual International Computer Software and Applications Conference*, August 2002.

[7] John Chessman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[8] Patrick Henrique da S. Brito, Camila Ribeiro Rocha, Fernando Castor Filho, Eliane Martins, and Cecília M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In Carlos Alberto Maziero, João Gabriel Silva, Aline Maria Santos Andrade, and Flávio Morais de Assis Silva, editors, *LADC*, volume 3747 of *Lecture Notes in Computer Science*, pages 61–79. Springer, 2005.

[9] Patrick Henrique da Silva Brito, Maria Antonia Martins Barbosa, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira. Um processo para o desenvolvimento baseado em componentes com reuso de componentes. Relatório Técnico (a publicar), Instituto de Computação, 2005.

[10] Paulo Asterio de C. Guerra, Fernando Castor Filho, Vinicius Asta Pagano, and Cecília M. F. Rubira. Structuring exception handling for dependable component-based software systems. In *EUROMICRO*, pages 575–582, 2004.

[11] L. Feijs. Architecture visualisation and analysis: Motivation and example. In *Intl. Workshop on Development and Evolution of Software Architectures for Product Families*, 1996.

[12] Gisele R. M. Ferreira. Um método para modelagem de exceções em desenvolvimento baseado em componentes. Master's thesis, IC, Unicamp, Outubro 2005.

[13] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical report CMU/SEI-2000-TR-004, Pittsburgh, PA, USA, August 2000.

[14] Mark Klein and Rick Kazman. Attribute-based architectural styles. Technical report CMU/SEI-99-TR-022, Pittsburgh, PA, USA, 1999.

[15] Kai Koskimies. Towards architecture-oriented programming environments. *First ASERC Workshop on Software Architecture*, August 2001.

[16] Raphael Marvie and Philippe Merle. Vers un modèle de composants pour CESURE - le CORBA Component Model. Technical Report 3, Projet RNRT 98 CESURE, Novembre 2000. http://www.gemplus.fr/cesure/.

[17] M. Douglas McIlroy. Mass-produced software components. In J. N. Buxton Peter Naur, Brian Randell, editor, *Software Engineering: Concepts and Techniques*, pages 88–94. Petrocelli/Charter, 1976.

[18] Microsoft. Internet explorer home. http://www.microsoft.com/windows/ie/. Last access in 2005.

[19] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Softw.*, 14(1):43–52, 1997.

[20] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[21] Mozilla.org. Mozilla firefox. http://www.mozilla.org/. Last access in 2005.

[22] Robert L. Nord, editor. *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*. Springer, 2004.

[23] Esmond Pitt, Kathleen McNiff, and Kathy McNiff. *java(TM).rmi: The Remote Method Invocation Guide*. Addison-Wesley, 2001.

[24] Brian Randell. Dependable pervasive systems. *23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, October 2004.

[25] Geri Schneider and Jason P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1st edition, 1998.

[26] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[27] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[28] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[29] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[30] WCOP'96. International workshop on component-oriented programming. http://sky.fit.qut.edu.au/ szypersk/WCOP96/. Last access in 2005.

[31] Zhang You-Sheng and He Yu-Yun. Architecture-based software process model. *Software Engineering Notes*, 28(2), March 2003.