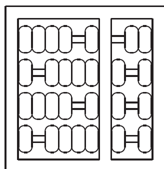

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Estudo sobre Estilos Arquiteturais para Sistemas de Software Baseados em Componentes

Patrick Henrique da Silva Brito
Paulo Asterio de Castro Guerra
Cecília Mary Fischer Rubira

Technical Report - IC-07-10 - 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.

Estudo sobre Estilos Arquiteturais para Sistemas de Software Baseados em Componentes

Patrick Henrique da Silva Brito   Paulo Asterio de Castro Guerra
Cecília Mary Fischer Rubira

Resumo. *Com o aumento do poder computacional, novos requisitos vêm sendo exigidos dos sistemas computacionais modernos. Além disso, tento em vista a flexibilidade e facilidade de evolução desses sistemas, o papel do software está cada vez mais importante nesse contexto de sistemas automatizados. Uma conseqüência da maior exigência de funcionalidades do software é o aumento da sua complexidade, o que pode comprometer a condução de atividades de manutenção e pode até mesmo reduzir a confiabilidade dos sistemas. Uma forma de controlar essa complexidade é através da arquitetura de software, que representa o sistema em um alto nível de abstração, em função dos seus elementos de alto nível e das regras de interação entre eles. Quando uma família de sistemas possui uma arquitetura de software semelhante, dizemos que elas seguem o mesmo estilo arquitetural. Este relatório apresenta um conjunto de estilos arquiteturais altamente utilizados na prática, exemplificando as principais características de cada um deles, assim como indicações de uso.*

Palavras-chave: *Arquitetura de software, estilos arquiteturais.*

1 Introdução

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 [20].

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 [2, 20].

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 [19, 13]. 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**.

Apesar da importância notória da arquitetura de software para o desenvolvimento de sistemas modernos, é necessário haver meios que orientem o arquiteto na definição dessa estrutura. Uma das maneiras mais utilizadas atualmente é a reutilização de soluções conhecidas, que deram certo em domínios específicos. Essa reutilização estrutural em alta granularidade é alcançada com a utilização dos **padrões arquiteturais**.

Este documento apresenta um estudo feito com os principais estilos arquiteturais, a fim de esclarecer o perfil de aplicação de cada um. A Seção 2 mostra alfuns fundamentos de arquitetura de software, tais como a relação estreita entre essa abstração e os requisitos não-funcionais do sistema e as diferentes visões que a arquitetura pode ser vista. A Seção 3 apresenta os cinco estilos arquiteturais tratados e um comentário a respeito de como esses estilos podem ser combinados. Finalmente, a Seção 4.1 apresenta um método para análise de arquiteturas, especificado pelo SEI.

2 Arquitetura de Software

A arquitetura de um sistema de software mostra como o sistema é realizado por um conjunto de **componentes arquiteturais** e as interações entre eles [17]. Seu foco principal é em como as responsabilidades do sistema são divididas entre os vários componentes arquiteturais e na forma como esses componentes interagem, abstraindo os detalhes de implementação internos aos componentes. Os componentes arquiteturais podem ser definidos em diferentes níves de abstração, podendo representar tanto rotinas de uma biblioteca procedural, como componentes de software como definidos no paradigma DBC ou grandes subsistemas. Em geral, o termo **componente** é utilizado, conforme o contexto, para designar tanto um componente arquitetural como um componente de software do paradigma DBC. Para que haja interação entre componentes é necessária alguma forma de comunicação entre eles. A responsabilidade pelo estabelecimento dessa comunicação é atribuída a elementos arquiteturais denominados **conectores**. Um conjunto de componentes interligados por conectores define uma **configuração** arquitetônica. A importância da abordagem de arquitetura do software é sentida à medida em que aumenta o porte e a complexidade dos sistemas, sendo um fator condicionante de diversos aspectos qualitativos do sistema, tais como desempenho, escalabilidade e segurança [5].

Embora a disciplina de arquitetura de software não se limite a sistemas desenvolvidos de acordo com o paradigma DBC, há uma forte sinergia entre essas duas disciplinas.

2.1 Os Princípios de uma Arquitetura de Software

Criar, manter e evoluir a arquitetura de um sistema é uma tarefa difícil, que normalmente é guiada pelas decisões de projeto do arquiteto e da equipe de desenvolvimento do software. Devido ao caráter subjetivo dessas decisões, a equipe de desenvolvimento e em especial o arquiteto de software precisam exercitar as suas habilidades de reconhecimento das melhores soluções. Para auxiliar nessa tarefa, o arquiteto de software pode se basear em uma lista de princípios que devem ser sempre almejados. A seguir, são apresentados os seis princípios básicos de uma arquitetura de software:

1. **Encapsulamento.** A arquitetura é organizada em partes relativamente independentes , no sentido de ser uma representação abstrata, onde os detalhes de implementação são omitidos.

2. **Baixo acoplamento.** Devido ao encapsulamento já discutido, a comunicação entre os componentes arquiteturais deve ocorrer através de protocolos de comunicação bem definidos.

3. **Alta granularidade.** Uma das principais vantagens de se especificar a arquitetura de software é a facilidade como se represeta estruturas complexas. O entendimento facilitado do sistema é conseqüência da abstração proporcionada pela arquitetura. Sendo assim, os componentes que compõe uma arquitetura normalmente representam um papel ou categoria, que pode possuir uma arquitetura interna independente.

4. **Alta coesão.** A coesão representa o nível de proximidade entre grupos de funcionalidades. A alta coesão dos componentes arquiteturais é conseqüência da definição de papéis claros e de fácil compreensão.

2.2 Requisitos Não-Funcionais do Sistema Realizados na Arquiteturas de Software

Nas discuções sobre arquiteturas de software, freqüentemente se faz referência às propriedades/requisitos não-funcionais do sistema que são satisfeitos pela arquitetura; em contraste, as propriedades funcionais da arquitetura são assumidas implicitamente. Uma propriedade funcional lida com algum aspecto particular da funcionalidade do sistema e está usualmente relacionada a um requisito funcional especificado. No passado, os desenvolvedores de software concentravam-se principalmente nas propriedades funcionais dos sistemas. Entretanto, nos sistemas de software modernos, as propriedades não-funcionais estão se tornando cada vez mais importantes. Um requisito não-funcional denota uma característica de um sistema que não é coberta pela sua descrição funcional. Uma propriedade não-funcional tipicamente está relacionada a aspectos como, por exemplo, confiabilidade, adaptabilidade, interoperabilidade, usabilidade, persistência, manutenabilidade, distribuição e segurança. As propriedades não-funcionais de uma arquitetura de software têm um grande impacto no seu desenvolvimento, manutenção e extensão. Quanto maior e mais complexo um sistema de software for e quanto maior for seu tempo de vida, maior será a importância das suas propriedades não-funcionais.

Como conseqüência, as decisões de projeto que devem ser tomadas no desenvolvimento de uma arquitetura de software geralmente são complexas e envolvem vários aspectos relacionados com as suas propriedades não-funcionais do sistema. Com o objetivo de se tomar boas decisões de projeto, é essencial que o contexto do problema seja bem definido, a fim de que seja possível identificar as melhores escolhas dentre as diversas opções de projeto possíveis. Um modo de clarificar esse contexto é aplicar a técnica de separação de interesse[1][14]. Para que os diversos aspectos envolvidos no projeto de uma arquitetura de software possam ser separados, é necessário inicialmente o estabelecimento dos limites conceituais entre eles. Cada partição obtida será responsável por resolver um subconjunto dos requisitos propostos. O arquiteto de software tem a responsabilidade de compor as diferentes partições de modo que todos os requisitos sejam conjuntamente satisfeitos.

2.3 Visões da Arquitetura de Software

Até o meio dos anos 90, devido à juventude da área de pesquisa de arquitetura de software, diversos trabalhos foram publicados nos quais representações arquiteturais falhavam em expressar as idéias de seus autores. Em 1995, Philippe Kruchten[11] publicou um artigo bastante influente em que argumentava que a causa para esse problema era a insistência dos autores em representar toda a arquitetura do sistema através de uma única abstração. Kruchten defendia a idéia de que a arquitetura de um sistema de software precisa ser representada através de quatro visões diferentes e ortogonais entre si: (i) lógica, (ii) de desenvolvimento, (iii) de processos e (iv) física. A conexão entre essas quatro visões é feita através de uma quinta visão que inclui elementos de todas elas, a visão de casos de uso. Essas cinco visões são descritas a seguir:

- **Visão lógica**: está relacionada aos requisitos funcionais do sistema. Na visão lógica, o sistema é decomposto em um conjunto de abstrações extraídas principalmente do domínio do problema, na forma de objetos ou classes que se relacionam. A visão lógica descreve tanto estrutura estática quanto estrutura dinâmica do sistema e é representada, na UML, através de diagramas de classes e de diagramas dinâmicos (seqüência, colaboração, atividades, etc.). Uma versão inicial da visão lógica da arquitetura do sistema é fornecida pelo diagrama de classes de análise e pelos diagramas de seqüência e colaboração produzidos durante a modelagem dinâmica.

- **Visão de desenvolvimento**: foca na organização do sistema em módulos de código. A visão de desenvolvimento é representada, na UML, através de diagramas de componentes contendo módulos, bibliotecas e as dependências entre esses elementos. É possível também representá-la, de forma mais simplificada, usando-se diagramas de pacotes.

- **Visão de processos**: a visão de processos lida com a divisão do sistema em processos e processadores. Ela trata de aspectos relacionados à execução concorrente e distribuída do sistema e, através dela, requisitos não-funcionais como performance, tolerância a falhas e disponibilidade são realizados. A visão de processos é representada,

[1] do inglês *separation of concerns*.

na UML, através de diagramas dinâmicos, assim como diagramas de componentes e implantação. Esta visão também é conhecida como visão de concorrência. A visão de processos costuma ser de especial importância para a construção de sistemas com requisitos ligados a temporização e sincronização, como sistemas de controle e de tempo real.

- **Visão física**: esta visão leva em consideração principalmente os requisitos não-funcionais do sistema, em especial os que estão relacionados com sua organização física, como tolerância a falhas, escalabilidade e performance. Na visão física, também conhecida como visão de implantação, o sistema é representado como um conjunto de elementos que se comunicam e que são capazes de realizar processamento (como computadores e outros dispositios). A UML dá suporte à visão física através de diagramas de implantação.
- **Visão de casos de uso**: esta visão, também conhecida como visão de cenários, é responsável por integrar as outras quatro e descreve a funcionalidade oferecida pelo sistema aos seus atores. Esta visão apresenta informação redundante com relação às outras visões e é usada desde o início do desenvolvimento, para expressar os requisitos, até o final, quando testes de aceitação são realizados. A visão de casos de uso é representada, na UML, através de diagramas e especificações de casos de uso.

Esse modelo para a representação de arquiteturas de software foi batizado de Modelo 4+1 e se tornou praticamente um padrão na indústria[10] e na academia[2]. A Figura 1 aparece em quase todos os textos nos quais o modelo 4+1 é mencionado e mostra como as cinco visões se relacionam. A Tabela 1 mostra um resumo das visões do modelo 4+1, com os respectivos requisitos e participantes de cada uma das visões.

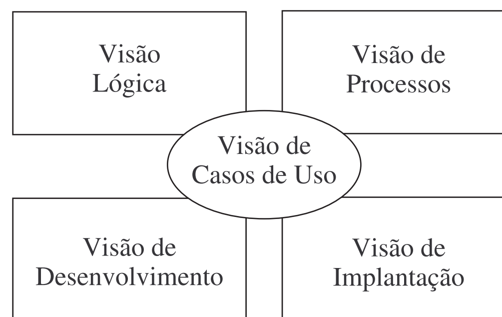

Figura 1: O modelo 4+1 para a representação de arquiteturas de software.

2.4 Linguagens de Descrição de Arquitetura

A arquitetura de uma sistema é normalmente expressa através de um conjunto de diagramas representando diferentes **visões arquiteturais** do sistema, como: a estrutura funcional, a estrutura do código, a estrutura de concorrência, a estrutura física, e a estrutura de desenvolvimento [11]. Embora tais diagramas sejam extremamente eficazes para a comunicação en-

Visão	**Requisito Principal**	**Participante Principal**
Visão Lógica	Funcionalidade	Usuário Final
Visão de Processo	Desempenho Escalabilidade Transferência de dados (taxa)	Integrador do Sistema
Visão de Desenvolvimento	Topologia do sistema Distribuição e instalação Comunicação	Engenheiro de Sistema
Visão Física	Gerenciamento do Software	Programador
Visão de Casos de Uso	Comportamento	Analista de sistemas Analista de testes

Tabela 1: Resumo das Visões do Modelo 4+1

tre arquitetos, desenvolvedores e usuários do sistema, não possuem o formalismo necessário para que possam ser empregados em outras etapas do desenvolvimento, como análise da arquitetura e geração de código, por exemplo. Para isso, são utilizadas linguagens formais de descrição de arquitetura (ADL) tais como UniCom [18], Acme [8] e xADL [6]. As abstrações principais de uma ADL são componentes, conectores e configurações. Os componentes são descritos através das suas interfaces. No tocante a essas interfaces, as mesmas limitações de expressividade das IDLs (Seção 2.4.1) são encontradas também nessas ADLs.

3 Estilos Arquiteturais e Padrões Arquiteturais

Um estilo arquitetural define um conjunto de tipos de componentes e conectores que podem ser utilizados para compor um sistema e um conjunto de restrições impostas às configurações que podem ser criadas dessa forma [16].

Os estilos arquiteturais mais conhecidos foram descritos inicialmente de maneira informal e não sistematizada, através de definições discursivas e diagramas ilustrativos. A comunidade de padrões se dedica a documentar soluções já comprovadas, incluindo estilos arquiteturais, no contexto de tipos de problemas específicos em que cada solução é aplicável [16]. Surgiram, assim, os chamados padrões arquiteturais [4], que são estilos arquiteturais organizados e documentados de forma sistemática. Um estilo arquitetural define uma família de arquiteturas de software, num nível de abstração mais elevado que o de uma configuração específica de componentes e conectores. A análise de um estilo arquitetural, nesse nível de abstração, permite que se deduzam propriedades que se irão se verificar em qualquer configuração que possa ser criada em conformidade com aquele estilo. São essas propriedades gerais dos vários estilos arquiteturais que, no desenvolvimento de sistema em particular, motivam a escolha de um ou mais estilos arquiteturais para guiar o projeto de sua arquitetura de software.

3.1 O Estilo Arquitetural de Camadas

No desenvolvimento de software é necessário particionar o sistema total em subsistemas coesos e fracamente acoplados[12]. Subsistemas normalmente são identificados pelos serviços que oferecem, definidos através das operações de suas interfaces. Consequentemente, uma das atividades mais importantes do projeto arquitetural é a especificação das interfaces dos subsistemas. Além disso, um subsistema pode depender de outros subsistemas, assumindo o papel de cliente, e oferecer serviços que são utilizados por outros subsistemas, assumindo o papel de servidor.

Pressman[15] apresenta uma lista de critérios de projeto com os quais subsistemas em qualquer projeto de desenvolvimento OO devem estar em conformidade:

- O subsistema deve ter uma interface bem definida através da qual ocorre toda a comunicação com o resto do sistema.
- Com exceção de um pequeno número de "classes de comunicação", as classes em um subsistema devem colaborar apenas com outras classes dentro do próprio subsistema.
- O número de subsistemas deve ser pequeno, onde o significado de "pequeno" depende do projeto e deve ser julgado pelo arquiteto de software, normalmente um membro experiente do time de desenvolvimento.
- Um subsistema pode (e deve) ser particionado internamente, a fim de reduzir sua complexidade.

Quando um sistema é particionado em subsistemas, uma outra atividade ocorre concorrentemente: a divisão em camadas. Cada camada de um sistema contém um ou mais subsistemas e é responsável por um conjunto de funcionalidades relacionadas. Normalmente, em sistemas estruturados em camadas, subsistemas localizados em uma determinada camada n usam os serviços oferecidos pelos subsistemas da camada $n+1$ (abaixo) e oferecem serviços para os subsistemas da camada $n-1$ (acima). As camadas $n-1$ e $n+1$ não "enxergam" uma à outra diretamente[2]. Sendo assim, como pode ser visto na Figura 2, uma camada esconde de suas camadas superiores, os detalhes das implementações das camadas inferiores, tornando possível que a implementação de uma camada seja trocada sem que o sistema inteiro seja afetado.

Uma estruturação em camadas muito utilizada para a construção de sistemas de informação, especialmente os baseados na web, consiste em utilizar três camadas: (i) uma responsável pela interação entre o sistema e o usuário, (ii) uma outra responsável por implementar as regras de negócio da aplicação e (iii) uma terceira que lida com o armazenamento dos dados. Arquiteturas de software organizadas de acordo com essa divisão são conhecidas como Arquiteturas *Three-Tier* (ou "arquiteturas em três camadas").

3.2 O Estilo Arquitetural C2

C2 [21] é um estilo arquitetural voltado para desenvolvimento baseado em componentes, que visa alcançar um elevado grau de reutilização de componentes e permitir a composição

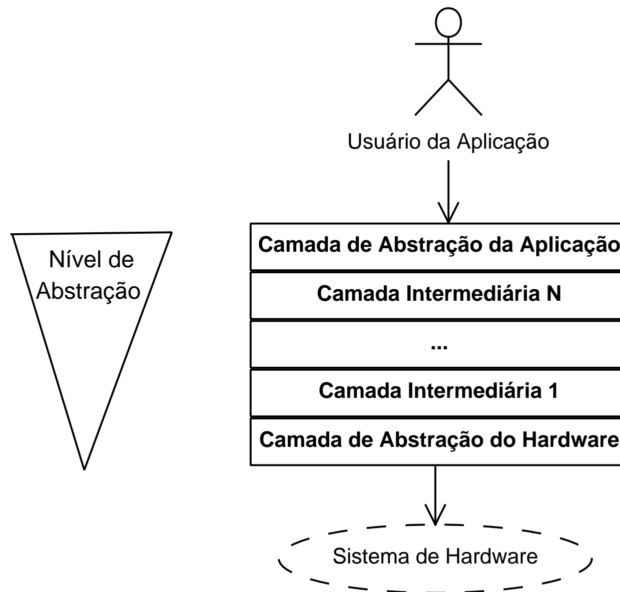

Figura 2: Exemplo do Estilo Arquitetural em Camadas

desses componentes de forma flexível, enfatizando o fraco acoplamento entre os componentes. O estilo C2 surgiu no desenvolvimento de interfaces de usuário gráficas. No estilo C2, os componentes interagem por meio de mensagens, em geral assíncronas, mediadas por conectores explícitos. Os conectores são responsáveis pelo roteamento, difusão e filtragem das mensagens. Os componentes podem ser encapsulados em invólucros que resolvem conflitos arquiteturais [7] que possam existir entre os vários componentes, adaptando as interfaces e serviços de infra-estrutura requeridos pelo componente às interfaces e serviços de infra-estrutura providos pela configuração onde está sendo inserido. Dessa forma, um componente pode interagir num sistema formado por componentes que lhe são desconhecidos e desenvolvidos com base em suposições arquiteturais conflitantes. Essa característica é especialmente adequada para integração de componentes de software desenvolvidos independentemente, como componentes de prateleira, que podem ser baseados em estilos arquiteturais, linguagens de programação e infra-estruturas de componentes heterogêneos.

No estilo C2, tanto componentes como conectores (Figura 3) possuem duas interfaces, que se convencionou chamar de interface de cima e interface de baixo. As configurações são formadas usando um estilo em camadas: uma interface de cima (baixo) só pode ser conectada a uma interface de baixo (cima) de outro elemento arquitetural. Um componente só pode ser conectado a, no máximo, dois conectores: um acima e outro abaixo. Um conector pode ser conectado a um número qualquer de componentes ou outros conectores.

São dois os tipos de mensagens em C2: requisições e notificações. Por convenção, as requisições fluem para cima, através das várias camadas do sistema, e as notificações fluem para baixo. Um componente emite requisições através da sua interface de cima. Essas

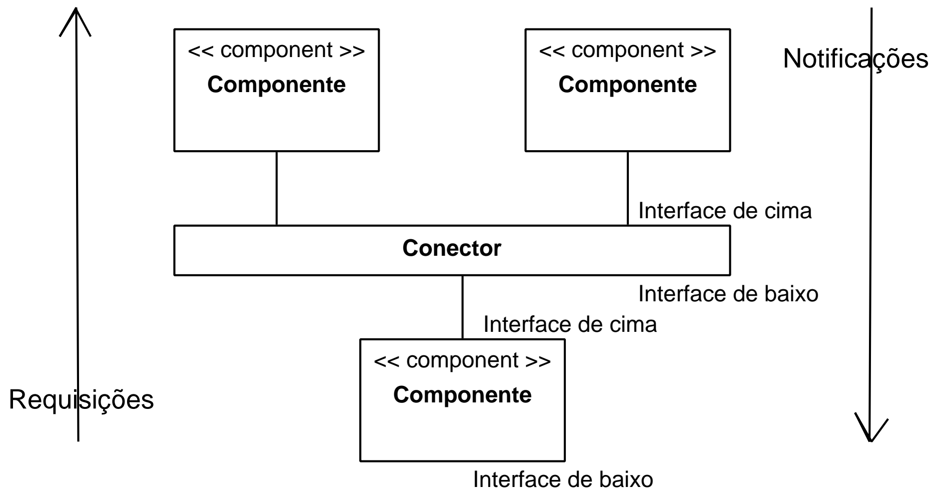

Figura 3: Elementos básicos do estilo arquitetural C2

requisições seguem, através do conector ligado à interface de cima do componente, para um ou mais elementos (componentes ou conectores) situados na camada imediatamente superior, que estejam ligados à interface de cima daquele conector. Dessa forma, e dependendo das regras de difusão e filtragem definidas pelos conectores, uma mesma requisição pode atingir um número qualquer de componentes, que as recebem através de suas interfaces de baixo. Ao receber uma requisição um componente pode executar uma de suas operações e, eventualmente, emitir uma notificação através de sua interface de baixo. Essas notificações seguem o caminho inverso das requisições, podendo também alcançar um número qualquer de componentes situados em camadas inferiores da arquitetura, incluindo o componente que emitiu aquela requisição. Ao receber uma notificação um componente pode reagir de forma similar à uma requisição, executando uma de suas operações e, eventualmente, emitindo uma nova notificação.

3.3 Repositório (*Repository*)

O estilo arquitetural repositório é um estilo que se caracteriza pela presença de dois tipos de componentes: (1) o repositório propriamente dito, uma estrutura que armazena e centraliza dos dados do sistema e (2) um conjunto de componentes independentes entre si que operam sobre os dados do repositório [19]. A Figura 4 ilustra uma arquitetura nesse estilo. Nessa figura, o Repositório representa e compartilha o estado geral do sistema, enquanto os componentes (C1, C2 e C3) realizam processamento, comunicando-se diretamente com o repositório.

Exemplos de sistemas que tipicamente utilizam esse estilo incluem ferramentas CAD (*Computer Aided Design* ou Projeto apoiado por computador) e ambientes integrados de desenvolvimento de software, onde diversas ferramentas de modelagem, visualização, apoio

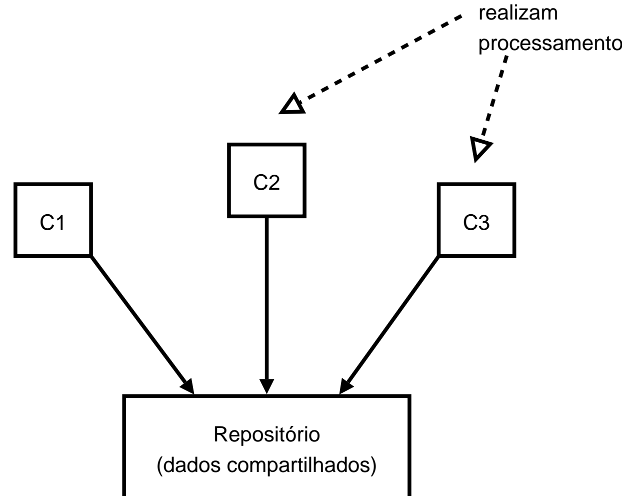

Figura 4: Exemplo de arquitetura seguindo o estilo repositório.

à edição, e outros operam de modo diferente sobre um mesmo conjunto de dados.

Com respeito ao fluxo de controle nesse estilo arquitetural, podemos ter principalmente duas variações: (1) quando uma entrada do sistema dispara os processos a serem executados e (2) quando o estado interno do repositório dispara os processos a serem executados. Nesse último caso, ilustra tipicamente o funcionamento de uma arquitetura *blackboard*.

3.3.1 Quadro Negro (*Blackboard*)

O estilo arquitetural *Blackboard* é um tipo de estilo arquitetural repositório. Ele possui os mesmos dois tipos de componentes: o repositório, chamado de *blackboard* e componentes que realizam processamento sobre o repositório [19].

Nesse estilo, o estado do *blackboard* dispara a escolha de qual componente será executado. Assim, cada componente contribui incrementalmente para a solução do problema que o sistema objetiva resolver. Os componentes comunicam-se exclusivamente através do repositório.

Esse estilo tem sido utilizado em aplicações que exigem interpretação de elementos complexos, tais como processamento de sinais, reconhecimento de padrões e de linguagem natural.

3.4 Pipes and Filters

No estilo *pipe-and-filter* cada componente possui um conjunto de entradas e um conjunto de saídas [19]. O componente lê fluxos de dados em suas entradas e produz fluxos de dados em suas saídas. São realizadas transformações locais sobre os fluxos de entrada de forma incremental, sendo possível o início da produção dos fluxos de saída antes que o fluxo de entrada tenha sido consumido completamente. Os componentes são denominados *filtros*.

Os conectores deste estilo servem como condutores para os dados, transmitindo as saídas de um filtro para as entradas de outros filtros. Os conectores são denominados *pipes*.

Uma importante característica deste estilo está na condição de que os filtros devem ser entidades independentes: um filtro não deve compartilhar seu estado com outros filtros. Outra característica importante é o desconhecimento do filtro sobre os demais filtros com que interage. Suas especificações devem ser restritas aos seus *pipes* de entrada e saída, não devendo identificar os componentes no final de cada *pipe*. A Figura 5 ilustra este estilo.

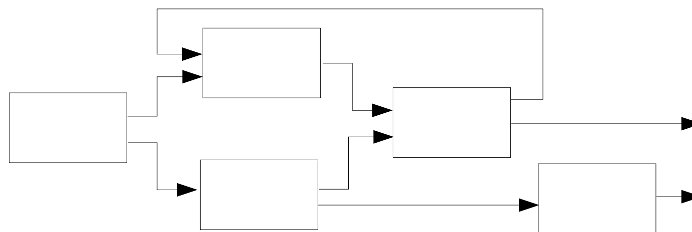

Figura 5: Estilo *Pipe-and-Filter*

O mais conhecido exemplo de uma arquitetura *pipe-and-filters* são os programas escritos no shell Unix. O Unix utiliza este estilo para realizar a conexão de seus componentes (representados pelos seus processos Unix). Outros exemplos de *pipe-and-filter* aparecem em sistemas de processamento de sinais, programação paralela, programação funcional e sistemas distribuídos.

Os sistemas construídos segundo o estilo *pipe-and-filter* possuem diversas vantagens. Uma vantagem é a facilidade de compreensão do sistema, dado que o sistema pode ser visto como uma simples composição de filtros individuais. Outra vantagem é a facilidade de reuso, dado que os filtros necessitam apenas especificar suas entradas e saídas. Existe ainda a facilidade de manutenção e evolução do sistema, através de trocas de filtros.

Por outro lado, sistemas que seguem esta abordagem podem impor regras de comunicação em seus *pipes*, de forma a padronizar a comunicação e facilitar a modificação de seus filtros. Estas regras em seus *pipes* podem ser uma desvantagem, pois podem exigir que os filtros trabalhem seus fluxos de entrada e saída para respeitarem as regras de comunicação, afetando o desempenho e a construção dos filtros. Outra desvantagem desses sistemas ocorre quando existem dependências entre fluxos de dados, sendo necessário que os filtros tratem as correspondências entre os fluxos.

3.5 Estilos Heterogêneos

As Seções 3.1 a 3.4 apresentaram alguns dos principais estilos arquiteturais “puros”. Apesar da importância de se entender a natureza individual de cada um, a implementação de sistemas reais quase sempre envolve uma combinação de vários estilos.

Estilos arquiteturais podem ser combinados de diversas maneiras. A maneira mais comum de se combinar estilos é através de hierarquias. Um componente de um sistema, que está já inserido em algum estilo arquitetural pode ter uma estruturação interna própria, que pode ser um estilo arquitetural totalmente diferente do sistema do qual ele faz parte.

A Figura 6 apresenta um sistema em três camadas. Porém, a camada de modelo utiliza o estilo *pipes and filters*.

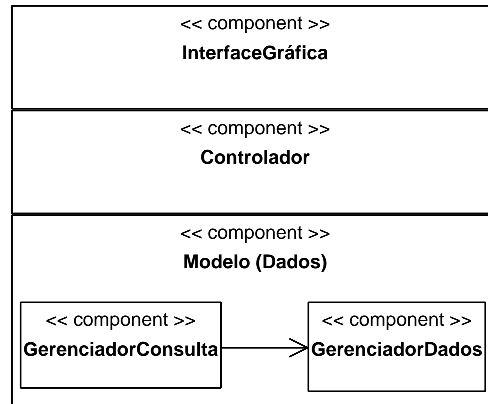

Figura 6: Exemplo de um Estilo Hierárquico Heterogêneo

O que é mais surpreendente é que os conectores, também, podem ser decompostos hierarquicamente. Por exemplo, um conector *pipe* pode ser implementado internamente como uma fila FIFO, que disponibiliza operações de inserção e remoção de itens.

Uma segunda maneira de combinar estilos arquiteturais é permitir que um componente simples utilize vários tipos de conectores. Em outras palavras, é possível que um determinado componente necessite se comunicar com outros dois componentes de maneiras diferentes.

Ainda existe uma terceira forma de se adotar um estilo arquitetural heterogêneo. Essa forma consiste na definição de um novo estilo, através de ADLs, discutidas brevemente na Seção 2.4.

4 Análise de Arquitetura

A análise da arquitetura de software de um sistema visa determinar, ainda num estágio inicial do desenvolvimento, problemas que possam vir a se manifestar após a implementação do sistema. A partir de estimativas de latência, velocidade de transmissão, capacidade de processamento e carga do sistema, por exemplo, devemos poder estimar o tempo de resposta esperado para uma determinada transação considerando-se diferentes arquiteturas de software.

Em [1] é descrito o método denominado ATAM (Architecture Tradeoff Analysis Method), baseado em modelos matemáticos para os diferentes atributos em que se está interessado, que são desenvolvidos para determinados cenários de uso considerados representativos para o sistema em questão. A Seção 4.1 apresenta mais detalhes desse método.

O uso efetivo de DBC para sistemas corporativos de grande porte depende não só de tecnologia para implementação de componentes como também para integração desses com-

ponentes de maneira sistemática e previsível. Isso requer técnicas para especificar o comportamento desse componentes de forma a permitir o raciocínio sobre os mesmos, a análise e comparação de componentes abstraindo-se dos seus detalhes de implementação [3].

Em [9] é proposta uma abordagem para composição de sistemas que diverge da visão de componentes e conectores. Nessa abordagem, denominada ACTI (Abstract, Concrete, Templates, Instances), a interação entre componentes (ou instances) é feita sempre através de gabaritos de software (templates) que definem novos componentes, cujas propriedades podem ser deduzidas diretamente das propriedades de suas instâncias. Essa propriedade de análise modular (modular reasoning) é considerada essencial para que se possa controlar a complexidade do trabalho de análise de arquitetura.

4.1 O Método de Avaliação de Arquitetura ATAM

O ATAM[2] 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 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[3]

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[4].** 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 entrega[5] 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 serão materializados.

Grupo 2: Investigação e Análise[6]

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.

[2]do inglês *Architecture Tradeoff Analysis Method*
[3]do inglês *presentation*
[4]do inglês *business deivers*
[5]do inglês *time to market*
[6]do inglês *Investigation and Analysis*

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]

7. ***Brainstorm* e priorização dos cenários.** Baseado nos cenários especificados na árvore de prioridades, 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.

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]

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.

Referências

[1] Mario R. Barbacci et al. Steps in architecture tradeoff analysis method: quality attribute models and analysis. Technical Report CMU/SEI-97-TR-029, Software Engineering Institute - Carnegie Mellon University, May 1998.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[7] do inglês *Testing*
[8] do inglês *tradeoff*
[9] do inglês *Reporting*

[3] A. W. Brown and K. C. Wallnau. The current state of cbse. *IEEE Software*, 15(5):37 – 46, September / October 1998.

[4] F. Buschmann, R. Meunier, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A system of patterns*. John Wiley & Sons, 1996.

[5] Paul Clements and Linda Nothrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, USA, 1996.

[6] Eric M. Dashofy, Andre Van der Hoek, and Richard N. Taylor. A highly extensible xml-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, 2001.

[7] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17 – 26, November 1995.

[8] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–68. Cambridge University Press, 2000.

[9] David S. Gibson, Bruce W. Weide, Scott M. Pike, and Stephens H. Edwards. Toward a normative theory for component-based system design and analysis. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 10, pages 211 – 230. Cambridge University Press, 2000.

[10] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] Phillipe Kruchten. The 4 + 1 view model of software architecture. *IEEE Software*, 12(6):42 – 50, November 1995.

[12] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall, 1998.

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

[14] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053 – 1058, 1972.

[15] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[16] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC97, First International Computer Software and Applications Conference*, April 1997.

[17] Mary Shaw, Robert Deline, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, 1996.

[18] Mary Shaw et al. Abstractions for software architecture and tools to support them. In *IEEE Transactions on Software Engineering*, volume 21, pages 314 – 335, 1995.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1st edition, 1996.

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

[21] R. N. Taylor et al. A component- and message-based architectural style for gui software. In *IEEE Transactions on Software Engineering*, volume 22, pages 390 – 406, June 1996.