Arquitetura de Software

By:
Guilherme Germoglio

Arquitetura de Software

By:
Guilherme Germoglio

Online:
< http://cnx.org/content/col10722/1.9/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

Mensagens do Livro[1]

O conteúdo deste livro está dividido em seis capítulos (além dos capítulos de estudos de caso) e cada capítulo serve para transmitir um conjunto específico de mensagens sobre a disciplina de Arquitetura de Software. Além de mensagens, há outros dois tipos de elementos que são essenciais para a composição de livro: definições, que descrevem os conceitos fundamentais, e boas práticas, que são recomendações a serem seguidas pelo leitor ao aplicar o conhecimento presente no livro. As recomendações têm um papel importante, principalmente, nos estudos de caso, quando o lidamos com os diversos *trade-offs* presentes na prática de Arquitetura de Sotware.

A seguir, apresentamos os capítulos e suas mensagens:

1.1 Cap. 2: Introdução a Design de Software

Neste capítulo apresentamos design de software e mostramos que ele é essencial no processo de desenvolvimento de software

[1] This content is available online at <http://cnx.org/content/m17526/1.7/>.
Available for free at Connexions <http://cnx.org/content/col10722/1.9>

independentemente do nível de detalhe em que ele é aplicado. No entanto, o design de alto nível é enfatizado, uma vez que projetar arquitetura é fazer design em alto nível. Mostramos também os elementos que compõem os problemas de design. As mensagens do capítulo são:

- O design é a estrutura ou o comportamento de um sistema que resolve ou contribui para a resolução das forças que atuam sobre esse sistema.
- Um design representa apenas um ponto no espaço de decisão.
- Um design pode ser singular, representando apenas uma folha na árvore de decisões,, ou coletivo, representando um conjunto de decisões.
- São cinco os elementos que compõem os problemas de design: objetivos, restrições, alternativas, representações e soluções.
- Design é necessário em todos os níveis de detalhe durante o processo de desenvolvimento do software.

1.2 Cap. 3: Estudo de Caso: SASF

Neste capítulo, ilustramos a necessidade de aplicar os conhecimentos de Arquitetura de Software por meio de um problema de design complexo. Nele, apresentamos tanto os requisitos de um sistema web de locação e transmissão de vídeos quanto seus stakeholders. Uma vez que este capítulo apenas descreve um caso, não há mensagens explícitas a serem transmitidas.

1.3 Cap. 4: Fundamentos de Arquitetura de Software

Este capítulo apresenta a definição de Arquitetura de Software usando um padrão ISO/IEEE. Como a definição apenas não é o bastante para entendermos o porquê de se aplicar os conhecimentos de arquitetura durante o ciclo de desenvolvimento, mostramos seus benefícios explicitamente através de exemplos e o estudo de caso. Além da definição ISO/IEEE, mostraremos outras definições que diversos autores fizeram ao longo da história, uma vez que elas expõem características importantes para o entendimento do assunto. As mensagens deste capítulo são:

- Arquitetura é design, mas nem todo design é arquitetural. É o arquiteto quem define a fronteira entre o design arquitetural e o não-arquitetural, definindo quais decisões serão necessárias para atender aos objetivos de desenvolvimento, de comportamento e de qualidade do sistema.
- A arquitetura também é um veículo de comunicação entre stakeholders.
- A arquitetura contém as decisões antecipadas de design, que têm o impacto mais caro (caso seja necessário mudá-las ou caso elas estejam erradas).
- A arquitetura é uma abstração transferível do sistema.
- A arquitetura facilita a construção do sistema.
- A arquitetura facilita o entendimento do sistema.
- A arquitetura facilita o reuso durante o ciclo de vida do sistema.
- A arquitetura facilita a evolução do sistema.
- A arquitetura facilita a análise do sistema.
- A arquitetura facilita a gerência durante o desenvolvimento do sistema.

- Documentar a arquitetura ajuda no controle intelectual do software.
- Documentar a arquitetura ajuda a manter a integridade conceitual do sistema.
- A arquitetura do software restringe o vocabulário de alternativas de design.
- Documentar a arquitetura permite a ligação entre os requisitos e as decisões de design do software.
- Documentar a arquitetura tem impacto negativo na imprecisão da especificação, que é fonte de complexidade do sistema.
- Documentar a arquitetura ajuda na divisão de tarefas entre os times de desenvolvimento.

1.4 Cap. 5: Stakeholders

Os stakeholders têm grande influência no design da arquitetura porque são eles que impõem os requisitos que o sistema deve atender. Por isso, para entendermos essa influência, devemos estudá-los. Os stakeholders demonstram essa influência porque possuem diversas responsabilidades durante o ciclo de vida do software. Neste capítulo apresentamos quem são os stakeholders do software mais comuns e suas características. As mensagens deste capítulo são:

- Stakeholders influenciam a arquitetura de diversas maneiras e não necessariamente estão de acordo entre si e é por isso que surgem os *trade-offs* durante o design do software.
- Os seguintes stakeholders devem ser considerados durante o projeto da arquitetura:
  - o próprio arquiteto ou outros futuros arquitetos;
  - os engenheiros de requisitos;
  - os designers;

    - os desenvolvedores;
    - os testadores;
    - os responsáveis pela integração do software com outros sistemas;
    - os mantenedores do software;
    - os designers de outros sistemas;
    - o gerente do desenvolvimento;
    - o time de controle de qualidade do software.
- O arquiteto deve ter pleno conhecimento de todo o ciclo de vida do software, para ser capaz de lidar com os *trade-offs* que surgirão entre os stakeholders.
- O arquiteto deve enteder a relação entre os stakeholders e os atributos de qualidade do software.

1.5 Cap. 6: Atributos de Qualidade

Uma vez que os atributos de qualidade do software são proporcionados, principalmente, por sua arquitetura e é por meio dos atributos de qualidade que o software atende aos requisitos não-funcionais, devemos estudar esses atributos. Este capítulo trata tanto dos requisitos não-funcionais quanto dos atributos de qualidade, enfatizando suas relações e ferramentas de design úteis para o alcance dos atributos. Usamos o modelo ISO/IEC 9126-1:2001 – mas não nos restringimos a ele – para definir a qualidade de software e os atributos que ele deve exibir para tanto. As mensagens deste capítulo são:

- A arquitetura se preocupa principalmente com os requisitos não-funcionais, não apenas técnicos, mas também relacionados a negócio.
- Não existe *a arquitetura correta*. Existems arquiteturas que são mais ou menos adequadas aos requisitos.
- A arquitetura permite uma forma de rastreamento entre a implementação do software e seus requisitos.

- A arquitetura de software permite diversos atributos de qualidade, entre eles:
  - desempenho;
  - escalabilidade;
  - confiabilidade;
  - disponibilidade;
  - segurança;
  - manutenibilidade;
  - portabilidade;
  - extensibilidade.

1.6 Cap. 7: Técnicas de Design Arquitetural

Ao introduzirmos design de software, citamos alguns princípios e técnicas que são fundamentais ao processo, pois facilitam a representação e a escolha da solução entre as alternativas de design. No entanto, não fomos explícitos sobre como estes princípios e técnicas são fundamentais ao processo de design arquitetural. Já no capítulo sobre atributos de qualidade, mencionamos a existência de táticas arquiteturais que ajudam na implementação de alguns requisitos de qualidade, mas não apresentamos essas táticas a não ser de forma breve e apenas por meio de exemplos.

Este capítulo, por sua vez, tem como objetivo tanto apresentar os princípios de design em nível arquitetural, quanto apresentar algumas táticas arquiteturais que implementam requisitos de qualidade. Neste capítulo, descrevemos os seguintes princípios de design arquitetural:

- uso da abstração ou níveis de complexidade
- separação de preocupações
- uso padrões e estilos arquiteturais

Além disso, apresentamos diversas táticas arquiteturais para alcançarmos os seguintes atributos de qualidade:

- desempenho e escalabilidade
- segurança
- tolerância a faltas
- compreensibilidade e modificabilidade
- operabilidade

1.7 Cap. 8: Documentação da Arquitetura

Depois de entender os conceitos, a importância e ter noções de design de arquitetura, o leitor precisar saber como capturar a informação arquitetura e documentá-la. Conceitos de visões arquiteturais são introduzidos para facilitar a documentação das diferentes dimensões que uma arquitetura apresenta. Este capítulo pretende ser agnóstico a linguagens ou modelos de documentação de arquitetura, mas apresenta um exemplo de como fazê-lo. As mensagens deste capítulo são:

- O documento de arquitetura auxilia no processo de design, é uma ferramenta de comunicação entre stakeholders e pode servir de modelo de análise do software.
- Toda informação presente numa arquitetura é uma decisão arquitetural.
- Decisões arquiteturais podem ser existenciais, descritivas ou executivas.
- Decisões arquiteturais se relacionam, podendo restringir, impedir, facilitar, compor, conflitar, ignorar, depender ou ser alternativa a outras decisões arquiteturais.
- Um único diagrama não é suficiente para conter a quantidade de informação que deve ser mostrada por um arquiteto. Por isso, a necessidade de múltiplas visões da arquitetura.

Chapter 2

Introdução a Design de Software[1]

Antes de começarmos o estudo e a prática na disciplina de Arquitetura de Software, é apropriado sabermos onde ela se encaixa ao longo do Corpo de Conhecimento em Engenharia de Software (*Software Engineering Body of Knowledge*). Design arquitetural, ou projeto da arquitetura, é a primeira das duas atividades que compõem a área de conhecimento de Design de Software (*Software Design Knowledge Area*). A atividade seguinte é design detalhado. Por ser uma atividade de Design, o design arquitetural se faz por uma mistura de conhecimento e criatividade. Como criatividade é algo que se obtém através da experiência, não é nosso objetivo ensiná-la. No entanto, buscamos ao longo desse livro transmitir o *conhecimento* necessário para a criação de arquiteturas de sistemas de software.

Certamente, uma base conceitual em Design de Software é necessária para uma melhor compreensão desse livro. Dessa maneira, este capítulo procura fundamentar o conhecimento do

[1]This content is available online at <http://cnx.org/content/m17494/1.26/>.

leitor nessa área, de forma que sua importância e seus benefícios proporcionados sejam reconhecidos. Em outras palavras, esse capítulo fará com que o leitor seja capaz de:

- Reconhecer os conceitos básicos de design de software
- Descrever problemas de design através de seus elementos fundamentais
- Identificar princípios de design de software e explicar seus benefícios
- Diferenciar design de baixo-nível (detalhado) de design de alto-nível (arquitetural) e saber quando aplicar cada um

2.1 Design de Software

A relevância de se projetar – ou fazer design de – software pode ser explicada pela complexidade crescente dos sistemas de software. Devido a essa complexidade, o risco de se construir um sistema que não alcance seus objetivos é eminente.

Para evitar tal risco, a prática comum de qualquer engenharia para se construir um artefato complexo, um sistema de software complexo em nosso caso, é construí-lo de acordo com um plano. Em outras palavras, *projetar* o sistema antes de construí-lo. O resultado dessa atividade, também conhecida como de atividade de design, é também chamado de design. O design facilita duas atividades que são essenciais no ciclo de vida de um sistema de software. Primeiro, ele possibilita a avaliação do sistema contra seus objetivos antes mesmo dele ser construído. Dessa maneira, ele aumenta a confiança de que o sistema construído, se de acordo com o design, alcançará seus objetivos. Obviamente, uma vez que nesse ponto há apenas o modelo do sistema – o design –, a avaliação não será completa, mas isso também não quer dizer que ela não ofereça resultados

importantes que levem ao sucesso do sistema. Já a outra atividade beneficiada pelo design é a própria construção do sistema, dado que ele também serve como guia para a implementação do software.

A seguir, mostramos um exemplo de quando o design permite a avaliação do software. O Exemplo 2.1 mostra parte da primeira versão do design de um sistema distribuído de armazenamento, o *HBase*[2] e, através de uma breve avaliação desse design, observamos uma grave limitação do software.

Exemplo 2.1

O HBase é um sistema de armazenamento distribuído. Isso quer dizer que os dados submetidos a ele não serão guardados em um único servidor, mas em vários. De forma simplificada, o design do HBase define dois tipos de entidades no sistema: o *data node*, que é o subsistema que armazena os dados, e o *master node*, que é o subsistema que sabe em quais *data nodes* os dados foram escritos e podem ser recuperados. Na primeira versão do HBase, só existia um *master node* que coordenava todos os *data nodes*. Assim, para recuperar ou escrever dados no HBase, um cliente realizava os seguintes passos: primeiro, o cliente se comunicava com o *master node* a fim de conseguir, de acordo com uma chave[3], o endereço do *data node* em que ele pode realizar a operação desejada (leitura ou escrita). Em seguida, o *master node*, que coordena onde os dados devem ficar, retorna o endereço do *data node* que deveria possuir dados para referida chave. A partir daí, o cliente, já com o endereço, se comunicava diretamente com o *data node* e realizava a operação desejada (escrita ou leitura).

[2] *Apache HBase*: urlhttp://hbase.org

[3] Os dados são inseridos no HBase na forma (chave,valor).

Se avaliarmos este design, podemos perceber duas características do HBase. A primeira, é que ele não adota o uso de um cliente magro (*thin client*). Com isso, a implementação e configuração do cliente se torna mais complexa, uma vez que o cliente precisa conhecer o protocolo de escrita e leitura do HBase, além de precisar acessar tanto o *master node* quanto os *data nodes*. Isto dificulta o desenvolvimento, a operabilidade e a eventual evolução do software, uma vez que mudanças no protocolo afetam clientes e servidores. Além disso, por possuir apenas um *master node*, a funcionalidade do HBase fica condicionada à sua disponibilidade. Afinal, se o *master node* estiver inacessível, nenhum cliente poderá ler ou escrever no sistema, o que o torna um ponto único de falhas.

2.1.1 O que é Design de Software

Para definir design de software, alguns autores o fazem em dois sentidos distintos: quando design de software é usado como *produto* e quando é usado como *processo*. Quando usado no primeiro sentido, o termo *design de software* indica o produto que emerge do ato (ou processo) de projetar um sistema de software e sendo assim algum documento ou outro tipo de representação do desejo do projetista (ou designer). Esse produto é o resultado das decisões do designer para formar uma abstração do sistema que é desejado no mundo real. Existem diversas formas de como representar essa abstração do sistema. Podemos citar, por exemplo, desenhos usando caixas e setas, textos descritivo, ou ainda uso de linguagens ou ferramentas criadas para este propósito, como linguagens de modelagem de software, redes de petri, pseudocódigo, etc. Já quando o termo é usado no segundo sentido, *fazer design* indica o pro-

cesso seguido para se obter um projeto. Esse é um processo que faz parte do processo de desenvolvimento e que é orientado aos objetivos do software. Ele deve ser realizado tendo em mente os diversos *stakeholders* do sistema e deve ser fundamentado no conhecimento do designer sobre o domínio do problema.

A partir da visão de design como artefato, podemos observar que ele deve descrever diversos aspectos do software para que, assim, possibilite sua construção. Entre estes aspectos, estão:

- a estrutura estática do sistema, incluindo a hierarquia de seus módulos;
- a descrição dos dados a serem usados;
- os algoritmos a serem usados;
- o empacotamento do sistema, em termos de como os módulos estão agrupados em unidades de compilação; e
- as interações entre módulos, incluindo as regras de como elas devem acontecer e porque elas acontecem.

Podemos perceber que, apesar dos exemplos anteriores descreverem apenas *parte* do design de dois sistemas, eles mostram boa parte dos aspectos que esperamos no design de um software.

Por fim, citamos uma definição de design que engloba todos estes aspectos:

Definição 2.1: design de software

> *"É tanto o processo de definição da arquitetura, módulos, interfaces e outras características de um sistema quanto o resultado desse processo."*[4]

[4] Freny Katki *et al*, editors. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers Inc., 1991.

2.1.2 Características de Design de Software

Projetar os diversos aspectos de um sistema de software é um processo trabalhoso. No entanto, pode proporcionar diversos benefícios.

Design de software permite avaliação prévia. Como desenvolver software custa tempo e dinheiro, não parece sensato alguém investir seus recursos no desenvolvimento de um sistema que não soluciona os problemas propostos pelos interessados. Dessa maneira, a avaliação prévia do sistema se torna imprescindível para garantir que ele alcance os objetivos desses interessados. Como o design descreve diversos aspectos que estarão presentes no sistema quando construído, ele permite esse tipo de avaliação. Além disso, fazer o design de um sistema é, geralmente, mais barato que construí-lo.

Exemplo 2.2
Considerando o sistema do Exemplo 2.1 e que um de seus objetivos fosse a alta disponibilidade, podemos avaliar que design apresentado não seria a melhor solução para o objetivo proposto. Isso ocorre porque seu design possui um ponto único de falhas, que é uma característica indesejável para sistemas que buscam alta disponibilidade. Note ainda que não foi necessário ter o *HBase* desenvolvido para percebermos esse problema (na época em que implementava tal design, ele possuía cerca de cem mil linhas de código e alguns anos de desenvolvimento e, portanto, não sendo um software de desenvolvimento trivial), bastou apenas estudarmos seu design.

Design de software estimula modelagem. Ao modelar um sistema, o designer se concentra no domínio do problema, ignorando temporariamente detalhes menos significativos para se alcançar a solução. Isso facilita na separação da complexidade essencial da complexidade acidental do problema. E, como já

dito por Fred Brooks em *The Mythical Man-Month*, essa separação é benéfica para a qualidade final do sistema projetado.

Design de software envolve planejamento. Uma vez que o design serve de guia para a construção do sistema, o designer deve então antecipar o que será necessário para tanto. Esse planejamento ajuda na estimativa dos diversos custos envolvidos no desenvolvimento do sistema. Entre esses custos, podemos citar:

- Quanto tempo durará todo o desenvolvimento,
- Quantos desenvolvedores serão necessários para o módulo A,
- Se comprado, quanto custará o módulo B, e se for implementado,
- Ou qual será o custo total do desenvolvimento do sistema.

Design de software facilita a comunicação, pois contém conhecimento sobre o sistema que pode ser gravado, transmitido e discutido entre os interessados. Um caso bem comum é o de apresentar um sistema a novos membros de um time de desenvolvimento. Informações valiosas, como por exemplo, quais os principais módulos e seus diversos comportamentos, lhes podem ser passadas através do design do sistema antes de mostrá-los o código-fonte. Dessa maneira, essas informações de alto nível de abstração ajudarão a situá-los no código posteriormente. No entanto, o design não serve apenas para os desenvolvedores. Um usuário do sistema pode procurar no design informações de um nível ainda maior de abstração, como quais funções o sistema é capaz de realizar, ou qual o desempenho delas.

Por outro lado, design de software também demanda algumas observações importantes.

O problema a ser resolvido pode não permanecer o mesmo durante todo o processo de design. Ao passo que o design é

implementado, o cliente, que é o stakeholder interessado em que o software construído solucione um problema em particular, (1) pode mudar de ideia quanto à natureza do problema; (2) pode ter descrito o problema incorretamente; ou ainda (3) pode decidir que o problema mudou ou mesmo que já fora resolvido enquanto o design é feito. Essas possibilidades não devem ser ignoradas durante o desenvolvimento, uma vez que elas podem ocasionar em perda de tempo e dinheiro durante a fase de design ou ainda ocasionar o fracasso no atendimento das necessidades do cliente.

Há diferenças entre o design e o sistema construído a partir dele. O design de um software é apenas um modelo, do qual o nível de detalhes pode não ser adequado para certos tipos de avaliação. Por sinal, avaliar um design insuficientemente detalhado pode levar a resultados errôneos e, consequentemente, há sistemas que não resolvem os problemas da forma esperada. Isso é comum acontecer, por exemplo, quando por erro do projetista, detalhes importantes para a avaliação não são incluídos no design. O exemplo a seguir ilustra um caso em que a avaliação inadequada resultou em um produto com problemas.

Exemplo 2.3

Um caso conhecido de produto com falhas por avaliação inadequada é o caso de um sistema de controle de armamento para cruzadores da marinha norte-americana que foi desenvolvido pela empresa Aegis. Depois de desenvolvido, o sistema de armamento foi instalado no cruzador U.S.S. Ticonderoga para o primeiro teste operacional. No entanto, os resultados do teste demonstraram que o sistema errava 63% dos alvos escolhidos devido a falhas no software. Posteriormente, foi descoberto que a avaliação e os testes do software de controle foram realizados numa escala menor do que as condições reais e que, além disso, os

> casos de teste incluíam uma quantidade de alvos menor que a esperada em campo de batalha.[5]

Por mais eficaz que um design seja, sua implementação pode não ser. O fato de haver um design bem elaborado para um determinado software não garante que na fase de implementação os desenvolvedores sigam as regras previamente especificadas e que o código produzido reflita fielmente o que foi especificado. Isto é certamente um grande problema na construção de sistemas de software, pois pode acarretar a construção de um produto que não era o esperado, e até mesmo levar ao insucesso em sua construção. Felizmente, na Engenharia de Software existem dois mecanismos que visam diminuir as divergências entre design e implementação. O primeiro mecanismo diz respeito à verificação de software, isto é, verificar se o software foi construído corretamente, se atendeu às especificações do design. Por outro lado, a validação de software está ligada à satisfação do cliente diante do produto, isto é, se o software construído é o desejado, se atende aos requisitos do cliente.

2.2 Elementos do processo de design de software

O processo de design pode ser descrito como o processo de escolha da representação de uma solução a partir de várias alternativas, dadas as restrições que um conjunto de objetivos envolve. Esse processo, ilustrado na Figura 2.1, pode ser dividido em duas fases: *diversificação* e *convergência*.

[5] Uma descrição mais completa deste caso pode ser encontrada no artigo *The Development of Software for Ballistic-Missile Defense*, de Lin.

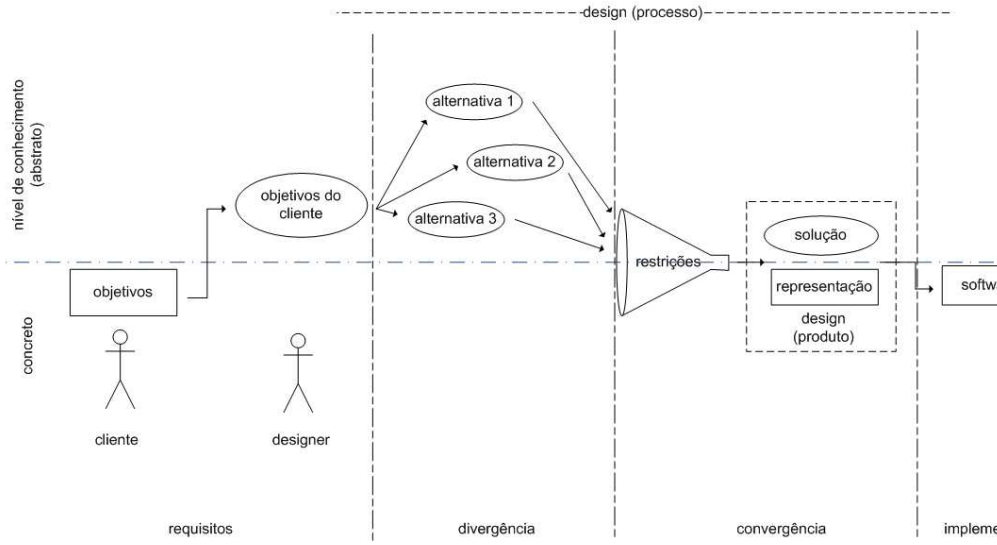

Figura 2.1: Ilustração do processo de design

É durante a fase de diversificação em que as *alternativas* são geradas. Por alternativas, não nos referimos necessariamente a documentos descrevendo uma possível solução, mas também a ideias de solução. Essas alternativas são soluções em potencial e são geradas/obtidas a partir do conhecimento e da experiência do designer. Já na fase de convergência, o designer escolhe a alternativa (ou combinação de alternativas) que satisfaz(em) aos *objetivos* esperados. A escolha comporá a *solução* que se sujeitará às *restrições* impostas pelo domínio do problema. Essa solução será descrita por meio de alguma *representação* e essa representação escolhida deve estar de acordo com seus propósitos: descrever a solução e permitir a construção do sis-

tema que melhor alcança os objetivos esperados.

Os elementos enfatizados no parágrafo anterior (objetivos, restrições, alternativas, representações e soluções), juntos, definem um arcabouço conceitual que nos ajuda a entender o processo de design de software.

2.2.1 Objetivos

O processo de design tem início com uma *necessidade*. Se algo é projetado, e consequentemente construído, é porque o produto proveniente do projeto suprirá essa necessidade. Em Engenharia de Software, a necessidade parte do cliente que especifica quais suas necessidades[6] e, portanto, quais os objetivos a serem atingidos pelo sistema de software a ser projetado. Assim, o objetivo do processo de design pode ser definido como:

Definição 2.2: objetivo de design
Aquilo que se pretende alcançar para resolver as necessidades do cliente.

Em design de software, objetivos também são chamados de requisitos. O design se preocupa com dois tipos de requisitos: requisitos funcionais e requisitos não-funcionais. Um requisito funcional especifica a funcionalidade que um sistema exibe.

Definição 2.3: requisito funcional
É a declaração de uma função ou comportamento providos pelo sistema sob condições específicas.

Em outras palavras, *o que* o sistema faz para alcançar às expectativas do cliente. Por exemplo, um requisito funcional de

[6]Vale lembrar que há transitividade nas necessidades do cliente. Um exemplo de quando acontece é quando clientes e usuários do sistema são entidades distintas. Então, entre as necessidades do cliente estarão: as necessidades do usuário devem ser atendidas. E, portanto, o software terá que atender terá que satisfazer também aos objetivos do usuário, além dos objetivos do cliente.

um programa de ordenação de números pode ser descrita como sua capacidade de ordenar inteiros; ou, se estamos falando de um sistema de informação de uma locadora de filmes em DVD, temos como requisitos funcionais, entre outros, a capacidade de buscar um filme usando palavras-chave, a capacidade de realizar o aluguel de um ou vários DVDs, ou a capacidade de realizar a devolução de um ou vários DVDs.

Por outro lado, um requisito não-funcional, especifica propriedades ou características que o sistema de software deve exibir diferentes dos requisitos funcionais. Os requisitos não-funcionais são atendidos pelos atributos de qualidade do software.

Definição 2.4: requisito não-funcional
É a descrição de propriedades, características ou restrições que o software apresenta exibidas por suas funcionalidades.

Em outras palavras, é basicamente *como* o sistema funcionará. De volta ao exemplo do programa de ordenar números, um requisito não-funcional que podemos mencionar é o tempo de execução da função de ordenação do sistema (por exemplo, é aceitável que o tempo de execução do algoritmo de ordenação tenha uma taxa de crescimento de $O(nlogn)$, onde n é a quantidade de elementos a serem ordenados). Já no sistema da locadora de filmes, um exemplo de atributo de qualidade é a exposição de algumas de suas funcionalidades via internet (e.g., busca e reserva de filmes através de um *site* disponibilizado pelo sistema).

Como os requisitos não-funcionais e os atributos de qualidade têm um papel importante na arquitetura do software, nós dedicaremos um capítulo a eles, onde serão descritos, categorizados e exemplificados em detalhes, além de serem relacionados aos stakeholders que os demandam.

2.2.2 Restrições

O produto de design deve ser viável. Dessa maneira, restrições são as regras, requisitos, relações, convenções, ou princípios que definem o contexto do processo de design, de forma que seu produto seja viável.

Definição 2.5: restrição de design
A regra, requisito, relação, convenção, ou princípio que define o texto do processo de design.

É importante saber que restrições são diretamente relacionadas a objetivos e que, em alguns casos, eles são até intercambiáveis. No entanto, uma vez que não são apenas os objetivos que guiam o processo de design, é necessário diferenciar objetivos de restrições. Em outras palavras, um sistema pode ter objetivos claros, mas seu design ou algumas alternativas dele podem ser inviáveis devido às restrições.

A seguir, apresentamos dois exemplos que nos ajudarão a entender o papel das restrições no design. No primeiro exemplo, apesar do sistema ter um objetivo claro, seu design não é viável devido a uma restrição.

Exemplo 2.4
Consideremos que um cliente deseje um sistema com um único objetivo: o sistema deve decidir se um programa, cuja descrição é informada como parâmetro de entrada, termina sua execução ou não.

Um designer inexperiente pode até tentar encontrar alguma alternativa de design para esse requisito – mas podemos ter certeza que a tentativa será em vão. Como é bem conhecido, há uma restrição teórica em Ciência da Computação, conhecida como *o problema da parada*, que impede o desenvolvimento de um programa capaz de alcançar o objetivo proposto. Como essa restrição impede a criação de qualquer alternativa

de design que satisfaça o cliente, podemos observar que um design pode ser se tornar inviável mesmo que seus objetivos sejam bem claros.

Já no segundo exemplo, o sistema também tem um objetivo claro. No entanto, uma restrição torna uma possibilidade de design inviável.

Exemplo 2.5

Um cliente especifica o seguinte requisito para seu sistema de software: ele deve ser capaz de ler dados de um leitor de cartões de um modelo específico. No entanto, ao estudar o requisito e, consequentemente, o leitor de cartões, o designer encontra a seguinte restrição. O fabricante do leitor em questão não o fornece *driver* necessário para um dos sistemas operacionais em que o sistema deve executar.

Podemos observar que, se não fosse por essa restrição, o design para o módulo de entrada de dados do sistema seria simples: apenas dependeria do *driver* do leitor para obter os dados dos cartões. No entanto, agora o designer terá que criar um design alternativo para contornar a restrição encontrada. Para isso, podemos citar algumas possibilidades desse design. Uma possibilidade seria emular um dos sistemas operacionais suportados quando o software estivesse executando num ambiente não suportado. Isso significa que seria necessária a criação de uma camada de abstração entre o *driver* do leitor e o sistema operacional onde o software está executando, onde essa camada representaria o ambiente operacional suportado. Essa camada de abstração, então, seria implementada pelo sistema nativo ou por um emulado, caso o nativo fosse o não-suportado pelo *driver*. Outra possibilidade de design seria o projeto e implementação do próprio *driver* para o ambiente não-

suportado.

2.2.3 Alternativas

Uma alternativa de design é uma possibilidade de solução. Uma vez que problemas de design geralmente possuem múltiplas soluções possíveis, é comum que sejam geradas mais de uma alternativa para a solução de um único problema. Note que o designer não necessariamente documentará todas as possibilidades de solução, mas, ao menos, considerará algumas delas para eleição de uma solução, mesmo que informalmente.

Definição 2.6: alternativa de design
Uma possibilidade de solução representada em nível de conhecimento.

O que precisamos observar é que o designer deve realizar duas tarefas essenciais após entender os objetivos e restrições envolvidos no problema de design: gerar alternativas de design e eleger a solução do problema dentre as alternativas geradas.

A geração de alternativas é o real desafio para os designers. Diferente dos problemas de decisão, onde alternativas são *conhecidas ou buscadas* através de métodos conhecidos, problemas de design pedem a *criação* de alternativas. O processo de criação deve ser controlado por princípios de design, pela experiência e imaginação do designer e deve ser guiado pelos objetivos do produto impostos pelos stakeholders. Alguns princípios essenciais de design serão apresentados ainda neste capítulo.

Já a eleição da solução é simplesmente a escolha de uma dentre as alternativas geradas, desde que essa sirva para a solução do problema. A escolha da solução deve ser realizada baseada em avaliações e experiência.

Exemplo 2.6

De volta ao nosso programa de ordenação, consideremos apenas uma de suas características: o algoritmo de ordenação a ser usado. Vamos observar quantas alternativas um designer poderia gerar só a partir dessa característica.

Uma rápida pesquisa na internet retorna nove algoritmos que respeitam o requisito imposto anteriormente de crescimento do tempo de execução ($O(nlogn)$): *binary tree sort*, *heapsort*, *in-place merge sort*, *introsort*, *library sort*, *merge sort*, *quicksort*, *smoothsort*, *strand sort*. Assim, esses nove algoritmos poderiam ser transformados em nove alternativas de design. Adicionalmente, um designer mais experiente em ordenação saberia que os dados de entrada podem definir o desempenho real do algoritmo, uma vez que uma das alternativas pode ter um ótimo desempenho para uma determinada entrada, enquanto outra alternativa, ainda que respeitando o mesmo desempenho assintótico $O(nlogn)$, pode ter um péssimo desempenho real para a mesma entrada. Neste caso, ele definiria que dois algoritmos serão usados no design, de forma que, de acordo com os dados de entrada, o algoritmo de melhor desempenho real para esses dados seja escolhido em tempo de execução. Assim, ainda mais alternativas de design são geradas.

Devemos observar que a geração de alternativas poderia continuar indefinidamente caso o designer considerasse outros aspectos do problema. Dessa maneira, *quando parar* a geração de alternativas é um problema também a ser resolvido pelo designer, uma vez que problemas de design geralmente têm um número infinito de soluções em potencial. Essa noção de quando parar o processo de geração de alternativas, certamente, é adquirida com a experiência.

2.2.4 Representações

A representação é a linguagem do design. Apesar do real produto do processo de design ser a representação de um sistema de software que possibilita sua construção, descrever o sistema não é o único propósito das representações. A representação também facilita o próprio processo de design, uma vez que ajuda na comunicação dos interessados e também serve como registro das decisões tomadas.

Definição 2.7: representação de design
A linguagem do processo de design que representa o produto do design para sua construção e também dá suporte ao processo de design como um todo.

A representação facilita a comunicação porque torna as alternativas em produtos manipuláveis, que podem ser comunicados, avaliados, e discutidos, não só por seus criadores, mas também por outros interessados.

É importante observar que existem diversas dimensões a serem representadas numa única alternativa de design. Essas dimensões abrangem comportamento, estrutura, relações entre entidades lógicas e entidades físicas, entre outros. Essas dimensões são normalmente descritas em diferentes tipos de representações, que, em outro momento, serão chamadas de *visões*.

Para exemplificar representações de design, apresentaremos duas dimensões derivadas do nosso programa-exemplo de ordenação usando duas representações diferentes. A primeira representação, ilustrada pela Figura 2.2, mostra a dimensão estrutural de uma alternativa de design usando UML[7]. Examinando essa representação, podemos observar alguns aspectos da solução: como a solução foi decomposta em classes funcionais, como as diversas classes da estrutura se relacionam entre si, ou até em que pontos poderíamos reusar pedaços de

[7] *Unified Modeling Language* (UML)

software prontos para a construção, desde que implementem as mesmas interfaces descritas na representação. No entanto, devemos também observar que essa representação não é auto-contida, uma vez que é necessário conhecimento em UML para entendê-la completamente.

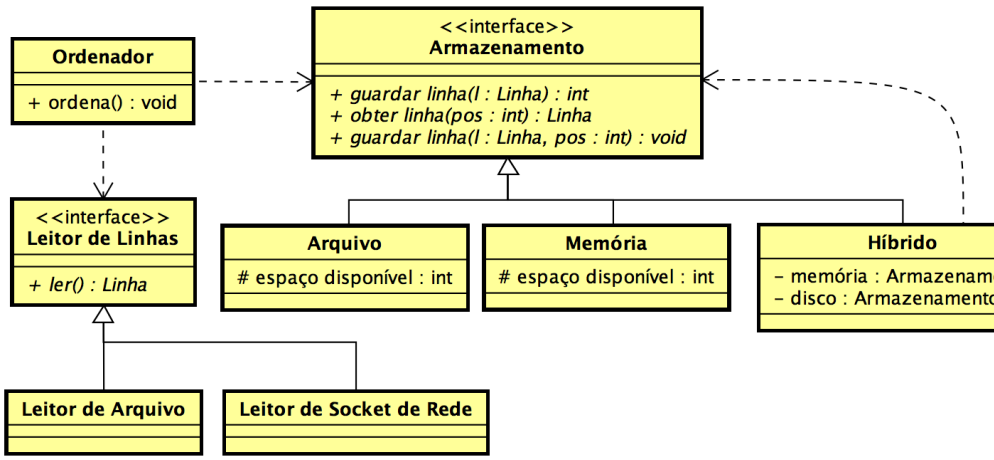

Figura 2.2: Representação estrutural do programa de ordenação

Já a segunda representação, Figura 2.3, mostra parte do comportamento do programa de ordenação com alto nível de detalhe. Apesar de não conseguirmos extrair dessa representação a mesma informação apresentada na figura anterior, essa nos permite analisar seu comportamento assintótico em relação ao crescimento do tamanho dos dados de entrada. Além disso, podemos também analisar o espaço consumido na execução do algoritmo.

```
function merge_sort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m

    var middle = length(m) / 2 - 1
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    if left.last_item > right.first_item
         result = merge(left, right)
    else
         result = append(left, right)
    return result


function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    end while
    if length(left) > 0
        append left to result
    else
        append right to result
    return result
```

Figura 2.3: Pseudocódigo do Merge sort

Ambas as representações mostram aspectos importantes do design de um software. No entanto, os stakeholders envolvidos no seu desenvolvimento podem ainda estar interessados em outros

aspectos além da estrutura ou análise assintótica do algoritmo. Por isso, outras representações podem ainda ser necessárias para mostrar outros aspectos do sistema, e é papel do processo de design – e do designer – provê-las.

Por fim, se considerarmos múltiplas versões ao longo do tempo de uma única representação, poderemos observar a evolução das decisões de design feitas ao longo desse período. Assim, se considerarmos as diversas versões obtidas até se alcançar o algoritmo descrito na Figura 2.3, perceberíamos a evolução desde o *merge sort* padrão até o *merge sort in-place* considerado pelo designer. Então, o histórico do design se torna peça fundamental para se entender quais decisões passadas levaram ao estado atual do design e, consequentemente, do sistema.

2.2.5 Soluções

A solução do design não é nada além do que a descrição que permite desenvolvedores construir um sistema de software a partir dos detalhes especificados por uma ou diversas representações. Suas principais características serão descritas nos parágrafos a seguir.

Definição 2.8: solução do design
A descrição do design que permite a construção do sistema de software que alcança os objetivos do design.

Soluções de design refletem a complexidade do problema, geralmente por mostrar diversos elementos e relações que compõem o problema. É possível observar essa característica quando, por exemplo, fazendo o design do sistema de informação de uma locadora que já mencionamos anteriormente. Qualquer que seja a solução, ela conterá elementos como filmes, DVDs, clientes ou gêneros de filmes, pois todos eles são inerentes ao problema em questão. No entanto, só os elementos não são o bastante

para compor a solução. A solução deve também conter relações do tipo: "um cliente pode alugar um ou mais DVDs", "um filme pode ter um ou mais gêneros", ou "um DVD pode conter um ou mais filmes". Em outras palavras, a solução deve conter relações similares às relações encontradas no domínio do problema. Vale lembrar que, quando diversos elementos têm diversas relações diferentes entre si, a complexidade emerge e é por isso que fazer design é difícil. Adicionalmente, para complicar ainda mais, é comum que os problemas tenham muitos elementos e relações que não são completamente conhecidos.

É difícil validar soluções de design. A complexidade inerente ao problema faz surgir diversos pontos de possível validação em relação aos objetivos de design. No entanto, o problema reside na precisão da descrição dos objetivos. Normalmente, para problemas complexos, objetivos são descritos num alto-nível de abstração que dificulta ou impossibilita bastante a avaliação das soluções.

E, por fim, *a maioria dos problemas de design aceita diversas soluções.* Isso é algo natural a problemas de design: uma vez que diversas alternativas podem ser geradas a partir de um único problema de design, diversas soluções podem ser obtidas.

2.3 Níveis de design de software

O produto do processo de design é sempre uma solução de design. Apesar de ser a descrição que permite a construção do sistema, nada foi dito sobre o nível de detalhe contido nessa solução. Acontece que, na verdade, o design pode ocorrer em diversos níveis de detalhe.

De acordo com o Guia para o Corpo de Conhecimento em Engenharia de Software, o processo de design de software consiste em duas atividades: *design de alto nível* e *design detalhado*.

O design de alto nível, também conhecido como design arquitetural, trata de descrever a organização fundamental do sistema, identificando seus diversos módulos (e sua relações entre si e com o ambiente) para que se alcancem os objetivos propostos pelo cliente.

Definição 2.9: design arquitetural
Descreve a arquitetura do software ou, em poucas palavras, como o software é decomposto e organizado em módulos e suas relações.

Ao contrário do design de alto nível, o design detalhado se preocupa com a descrição detalhada de cada módulo possibilitando a construção e se adequando ao design de alto nível.

Definição 2.10: design detalhado
Descreve o comportamento específico e em detalhes dos módulos que compõem o design arquitetural.

Apesar dessa divisão conceitual de design em duas atividades, essa divisão pode não acontecer durante o processo de desenvolvimento do software. Algumas vezes, o designer – ou quem assume seu papel – realiza ambas as atividades em paralelo, concebendo assim um produto de design que permitirá tanto o alcance dos requisitos de qualidade (que normalmente é tarefa da arquitetura), quanto a construção precisa do sistema por meio de seus detalhes. No entanto, adotaremos a separação conceitual das duas atividades de forma que possamos nos focar no design arquitetural, que é o principal assunto desse livro e que será discutido nos próximos capítulos.

2.4 Princípios e técnicas de design de software

Antes de iniciarmos nossos estudos em Arquitetura de Software, gostaríamos de lembrar alguns princípios e técnicas que

são essenciais ao design de software.

Há diversos princípios, técnicas, e abordagens nessa área que geralmente resultam em bons produtos de design de software. Uma vez que há muitos livros e artigos sobre esse assunto, gostaríamos apenas de fazer uma breve exposição do assunto nessa seção, fazendo com que o leitor relembre os princípios e técnicas – caso já os conheça – e indicando referências para um maior aprofundamento sobre o assunto. Os princípios, técnicas e abordagens essenciais para um designer que apresentaremos são as seguintes:

- Divisão e conquista
- Abstração
- Encapsulamento
- Modularização
- Separação de preocupações
- Acoplamento e coesão
- Separação de políticas da execução de algoritmos
- Separação de interfaces de suas implementações

2.4.1 Divisão e Conquista

Divisão e conquista é uma técnica para resolução de problemas que consiste em decompor um problema em subproblemas menores e *independentes* a fim de resolvê-los separadamente, para que, posteriormente, as soluções sejam combinadas e formem a solução do problema inicialmente proposto.

A estratégia é baseada na ideia de que atacar um problema complexo por diversas frentes é mais simples e factível de resolução do que tentar resolvê-lo completamente de uma só vez. A técnica de divisão e conquista possui três etapas bem definidas:

- Divisão: dividir o problema original em subproblemas menores;
- Conquista: resolver cada um dos subproblemas gerados na fase de divisão;
- Combinação: combinar as soluções de cada subproblema, compondo a solução para o problema inicial.

Em Ciência da Computação, essa estratégia é muito utilizada no projeto de algoritmos e, normalmente, é instanciada através do uso de recursão, uma vez que os problemas devem ser decompostos e as soluções dos subproblemas devem ser combinadas ao final da execução para compor a solução do problema inicial. Por exemplo, o algoritmo de ordenação *mergesort* se utiliza dessa técnica para ordenar uma sequência de inteiros de maneira eficiente. Esse algoritmo se baseia na idéia de quem dadas duas sequências ordenadas, é trivial ordená-las em uma única sequência. Portanto, a estratégia do *mergesort* é particionar uma sequência em várias subsequências até que seja trivial ordená-las, isto é, sequências de dois elementos. Por fim, o algoritmo combina as sequências em uma só sequência ordenada.

No entanto, como este livro foi escrito com foco em arquitetura de software, nada mais apropriado do que trazermos exemplos em nível arquitetural dos assuntos que abordamos. A estratégia de divisão e conquista também é aplicada constantemente em decisões de mais alto nível no projeto de software. Por exemplo, a decisão de organizar uma aplicação *web* em camadas nada mais é que dividir um problema maior em diferentes níveis de abstração, onde cada camada será responsável por implementar um serviço mais básico e específico (apresentação, lógica de negócio e armazenamento).

Vários são os benefícios providos pela estratégia de divisão e conquista. No nosso exemplo, a divisão da arquitetura em

camadas propicia a implementação de cada camada separadamente. Além disso, as camadas podem ser tratadas como componentes reusáveis de software, uma vez que implementam um serviço único e bem definido. Portanto, divisão e conquista também viabiliza o reuso de software.

2.4.2 Abstração

Abstração é um princípio essencial para se lidar com complexidade. Esse princípio recomenda que um elemento que compõe o design deva ser representado apenas por suas características essenciais, de forma que permita a distinção de outros elementos por parte do observador. Como resultado, temos a representação de um elemento do design mais simples, uma vez que detalhes desnecessários são descartados, facilitando então o entendimento, comunicação e avaliação.

O que poderemos observar é que a maioria das técnicas empregadas por designers ajudam na elevação do nível de abstração do design e, assim, baixam o nível de complexidade da solução.

2.4.3 Encapsulamento

Encapsulamento está relacionado à ocultação de detalhes de implementação de um elemento de um sistema aos que usarão esse elemento. Fazendo isso, o acoplamento entre os elementos é minimizado e sua contribuição para a complexidade do sistema é restringida às informações que eles expõem.

Encapsulamento pode ser obtido de diferentes maneiras: modularizando o sistema, separando suas preocupações, separando interfaces de implementações, ou separando políticas da execução de algoritmos.

2.4.4 Modularização

Modularização é a decomposição significativa do sistema em módulos. A modularização introduz partições bem-definidas e documentadas ao sistema ao decidir como estruturas lógicas do sistema serão divididas fisicamente. Podemos citar alguns benefícios da modularização:

- Facilita o entendimento, uma vez que cada módulo pode ser estudado separadamente;
- Facilita o desenvolvimento, uma vez que cada módulo pode ser projetado, implementado e testado separadamente;
- Diminui o tempo de desenvolvimento, uma vez que módulos podem ser implementados em paralelo, ou ainda reusados; e
- Promove a flexibilidade no produto, uma vez que um módulo pode ser substituído por outro, desde que implemente as mesmas interfaces.

2.4.5 Separação de preocupações

A separação de preocupações está fortemente ligada ao princípio da modularização. De certa maneira, a separação de preocupações define a regra para definir os módulos de um sistema: preocupações diferentes ou não-relacionadas devem se restringir a módulos diferentes. Assim, separando preocupações, obtemos benefícios semelhantes aos da modularização.

2.4.6 Acoplamento e coesão

Acoplamento e coesão são princípios usados para medir se módulos de um design foram bem divididos.

Acoplamento é a medida de interdependência entre módulos de software. Ou seja, quanto mais dependente um módulo A é da implementação do módulo B, maior é o acoplamento entre os módulos A e B. Alto acoplamento implica que (1) os módulos envolvidos serão mais difíceis de entender, uma vez que precisam ser entendidos em conjunto; (2) os módulos envolvidos serão mais difíceis de modificar, uma vez que as mudanças impactarão mais de um módulo; e (3) os módulos envolvidos serão mais difíceis de manter, uma vez que um problema num módulo se espalhará pelos módulos com quem está altamente acoplados.

Por outro lado, coesão é uma medida intramódulo. Ela é a medida da relação entre tarefas realizadas dentro de um mesmo módulo. As tarefas de um módulo podem estar relacionadas entre si por diferentes motivos. Esses motivos são usados para classificar os diferentes tipos de coesão:

***Coesão funcional*::** as tarefas estão agrupadas por suas funções serem similares.

***Coesão sequencial*::** as tarefas estão agrupadas por elas pertencerem à mesma sequência de operações. Elas compartilham dados a cada etapa da sequência, mas não realizam uma operação completa quando executadas juntas.

***Coesão comunicativa*::** as tarefas estão agrupadas porque usam os mesmos dados, mas não estão relacionadas de nenhuma outra maneira.

***Coesão temporal*::** as tarefas estão agrupadas por serem executadas no mesmo intervalo de tempo.

***Coesão procedural*::** as tarefas estão agrupadas porque elas devem ser executadas numa ordem específica.

***Coesão lógica*::** as tarefas estão agrupadas por compartilharem uma mesma *flag* de controle, que indicará qual tarefa será realizada durante a execução do sistema.

***Coesão coincidente*::** as tarefas estão agrupadas sem qualquer critério.

Para alcançarmos bons designs, podemos ordenar os tipos de coesão dos mais desejáveis para os menos desejáveis: funcional, sequencial, comunicativa, temporal, procedural, lógica, e coincidente.

2.4.7 Separação de Decisões de Execução de Algoritmos

Essa técnica realiza a separação de preocupações apresentando uma abordagem simples: ou um módulo deve se preocupar com as decisões sensíveis ao contexto do problema ou com a execução de algoritmos, mas não ambos. Em outras palavras, alguns módulos devem apenas executar algoritmos sem fazer qualquer decisão sensível ao domínio do problema. Essas decisões devem ser deixadas para os módulos específicos para realização dessas decisões e que também serão responsáveis por suprir parâmetros para os módulos de execução de algoritmos.

Essa separação facilita o reuso e manutenção, principalmente dos módulos de algoritmos, uma vez que eles são menos específicos que os módulos de decisões sensíveis a contexto.

2.4.8 Separação de Interfaces de suas Implementações

A separação entre interfaces e implementações também beneficia a modularização. Essa técnica recomenda a descrição da funcionalidade a ser implementada por algum módulo por meio de contratos, chamados interfaces. Assim, os módulos implementarão as interfaces de forma a comporem o sistema.

Usando essa técnica, o acoplamento entre módulos e seus clientes é diminuído, uma vez que os clientes estarão ligados apenas a interfaces – e não implementações –, e benefícios como facilidade no reuso, melhor entendimento do código, e menor custo de manutenção são alcançados.

2.5 Resumo

Esse capítulo expôs o conhecimento necessário sobre Design de Software para o estudo de Arquitetura de Software. Espera-se que, ao final desse capítulo, o leitor saiba:

- o que é design software, seja como produto ou como processo, e quais são suas características e benefícios;
- como os problemas de design de software podem ser decompostos; e
- o que são os princípios e técnicas de design de software e quais seus benefícios.

Pela existência de ótimos livros sobre Design de Software já escritos tendo em vista o mesmo público-alvo que nós (o leitor ainda inexperiente), nós preferimos não nos aprofundar nos assuntos expostos nesse capítulo, uma vez que nossa intenção foi de apenas introduzi-los. Para informações mais detalhadas, recomendamos os livros e artigos sobre Design de Software apresentados na seção de referências.

2.6 Referências

2.6.1 Teoria em Design de Software

Recomendamos o livro Software Design [23], de Budgen, aos interessados em mais informações sobre a teoria em design de software. Dois artigos que apresentam discussões úteis sobre

o assunto são *Software Design and Architecture – The Once and Future Focus of Software Engineering*[105], de Taylor e Van der Hoek, e *Conceptual Foundations of Design Problem Solving*[97], de Smith e Browne. Inclusive, este último é a nossa referência sobre o arcabouço conceitual de design usado neste capítulo.

2.6.2 Processo de Design

Em nível mais prático da execução do processo de design, citamos as seguintes referências: *The Mythical Man-Month: Essays on Software Engineering*[20], de Brooks, que discute as causas da complexidade que assola o processo de design de software; *Software Design: Methods and Techniques*[15], que descreve as etapas que podemos encontrar no processo de design; e o *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [5], que apresenta os níveis de design.

2.6.3 Técnicas e Ferramentas

Por fim, citamos referências que descrevem ferramentas e técnicas que podemos usar durante o processo de design.

Sobre a linguagem de modelagem UML, mais informações podem ser encontradas no site do *Object Management Group* (OMG) [80].

Já sobre técnicas de design, citamos o livro de Booch *et al*, *Object-Oriented Analysis and Design with Applications*[19], o de McConnell, *Code Complete*[76] e o de Buschmann et al, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*[26]. Este último é mais específico ao design arquitetural.

2.7 Exercícios

Exercício 2.1
Quais os benefícios de se projetar sistemas?

Exercício 2.2
Duas fases importantes do projeto de software são as fases de Divergência e Convergência. Descreva o que é feito em cada fase.

Exercício 2.3
Jack Reeves, em *What is Software Design?*[87], afirma que o código fonte é design. Qual a sua opinião a respeito da afirmativa?

Exercício 2.4
Qual padrão de projeto viabiliza a separação de política e implementação?

Exercício 2.5
Defina para coesão e acoplamento e sugira métricas para medi-las em software.

Exercício 2.6
Cite dificuldades que podem ser encontradas durante a aplicação de cada técnica de design apresentada no capítulo.

Exercício 2.7
Represente um design de software de duas maneiras diferentes. Para isso, antes será necessário descrever o problema que o software deve resolver.

Exercício 2.8
Elabore uma solução de design diferente para o problema descrito na resposta do anterior e descreva-a usando os mesmos tipos de representações usados anteriormente.

Chapter 3

Estudo de Caso: SASF[1]

Como muitos dos problemas relacionados à Arquitetura de Software só fazem sentido em sistemas complexos, é difícil ilustrar diversos aspectos dessa área apenas com exemplos simples. Assim, resolvemos adotar uma abordagem de ensino orientada a estudo de caso. Essa abordagem consiste em fazer com que o leitor acompanhe a construção da arquitetura de um sistema e, dessa maneira, possa observar na prática o uso do conhecimento encontrado no livro.

Ao longo deste livro, o leitor acompanhará um processo de design e documentação de uma arquitetura que possibilitará a implementação de um sistema complexo. Assim, será durante esse processo que serão apresentados os conceitos e generalizações essenciais para se formar um bom arquiteto de software. Também faremos uso de outros exemplos capazes de expor aspectos complementares ao estudo de caso.

[1] This content is available online at <http://cnx.org/content/m23389/1.5/>.

3.1 Apresentação do estudo de caso

O sistema através do qual acompanharemos o processo de design e documentação de sua arquitetura será o Sistema de Aluguel e *Streaming* de Filmes (SASF). O SASF é um sistema de informação com dois grandes objetivos: (1) gerenciar o processo de locação via *web* de vídeos e (2) proporcionar infraestrutura de software para realizar *streaming* de vídeos também via web. O SASF é um sistema fictício baseado no Netflix e na iTunes Store.

Esse sistema foi escolhido como estudo de caso por ter funcionalidades e atributos de qualidade não-triviais, além de pertencer a um domínio de problema mais comum do que os encontrados na literatura. A não-trivialidade de suas funções e de seus atributos de qualidade servirá como alvo para mostrar a necessidade e aplicação dos conhecimentos em Arquitetura de Software. Já em relação à abordagem de um domínio de problema relativamente simples, essa foi um dos fatores-chave para a escrita deste livro. Nos livros essenciais sobre Arquitetura de Software, é bastante comum acompanharmos estudos de caso de projetos militares ou da indústria aeroespacial. Esses projetos são riquíssimos em detalhes e se encaixam perfeitamente à necessidade de se estudar e aplicar os conhecimentos em Arquitetura de Software. Por outro lado, esses mesmos projetos, apesar de bem próximos à realidade dos autores dos livros em questão, são distantes da realidade do leitor ainda inexperiente em Engenharia de Software. Sendo assim, ao encontrar estudos de caso ou exemplos num domínio de problema pouco familiar, o leitor não se sente motivado e encontra dificuldades para concretizar os conceitos expostos ao longo dos livros.

3.2 Funcionalidades do SASF

Com a popularização da Internet, muitas empresas perceberam a oportunidade de, através dela, alcançar novos consumidores. O SASF se alinha com essa idéia. Seu papel é permitir que um cliente alugue um filme [2] usando apenas um navegador, sem estar presente fisicamente numa loja. Dessa maneira, esse sistema aumenta o número dos clientes em potencial da empresa, uma vez que eles deixam de ser apenas aqueles capazes de chegar à loja física para ser todos aqueles presentes na área de alcance da infraestrutura usada para entrega e coleta de vídeos.

3.2.1 Locação e *Streaming* de vídeo

O principal usuário do SASF é aquele interessado em alugar vídeos. Esse usuário, após se cadastrar no sistema, será capaz de (Figura 3.1):

- enfileirar filmes que serão enviados para sua casa obedecendo à política de sua assinatura,
- assistir a um filme via *streaming* em algum dispositivo ou aplicativo integrado e autorizado a comunicar com o SASF.

[2] Ao longo deste livro, apesar de usarmos a palavra "filme", estamos nos referindo a vídeos em geral, podendo ser também, seriados de TV, musicais, ou mesmo documentários, além de filmes. Assim, os termos "filme" e "vídeo" são intercambiáveis a não ser que sejamos explícitos quanto suas diferenças.

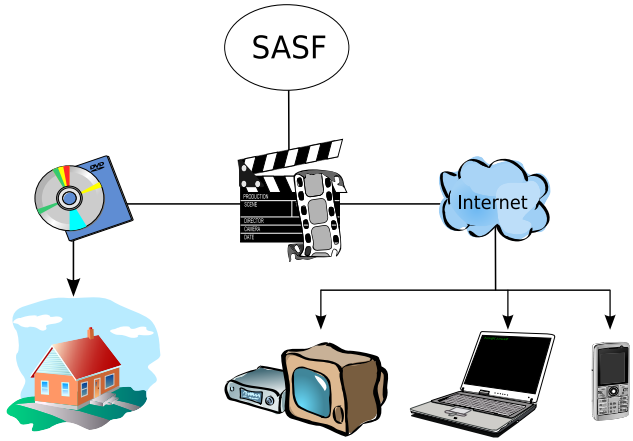

Figura 3.1: Principais funcionalidades do SASF: *streaming* de vídeo para diversos dispositivos e gerência do aluguel de filmes

As opções de assinatura disponíveis a esse tipo de usuário variam em relação ao número máximo de vídeos que ele pode ter em sua casa ao mesmo tempo. Dessa maneira, dado que o usuário enfileirou inicialmente 10 vídeos para aluguel, se sua assinatura permitir apenas um vídeo em sua casa por vez, o segundo vídeo da fila só será enviado para sua casa quando o primeiro for devolvido. De maneira análoga, se a assinatura for de três vídeos por vez, inicialmente, os três vídeos serão enviados e, à medida que eles forem devolvidos, a fila será esvaziada até que não sobre nenhum vídeo na fila ou o usuário esteja com três vídeos em sua casa. Vale notar que a fila pode crescer indefinidamente.

Os vídeos são entregues na casa do usuário pelos correios e ele pode ficar com as mídias o tempo que quiser. Não há qualquer tipo de penalidade se o usuário ficar com elas por muito tempo. O único incentivo para que ele as devolva é que ele não receberá mais vídeos de sua fila até que alguma mídia seja devolvida. Portanto, a devolução também é realizada pelos correios: o usuário envia as mídias para a empresa e, assim, possibilita que mais vídeos lhe sejam enviados.

Uma outra opção é assistir aos filmes através da Internet, via *streaming* de vídeo usando algum aparelho ou aplicativo que saiba conversar com o SASF. Assim, não é preciso esperar pela entrega da mídia pelos correios, nem esperar o tempo de *download* do vídeo para que se comece a assisti-lo, já que a idéia do *streaming* é consumir o arquivo de mídia transmitido à medida que ele é recebido. Isto, ainda por cima, economiza tempo do usuário, uma vez que ele pode assistir ao filme *agora*, e diminui gastos com logística por parte da empresa, uma vez que não precisa usar os serviços de uma transportadora para realizar a entrega do filme, além de economizar com compra e manutenção de mídias.

Ao prover um serviço de *streaming* para os usuários, surgem novas preocupações. O *streaming* de vídeo realizado pelo SASF pode ter vários destinos: uma aplicação cliente executando no navegador do usuário ou como aplicativo *stand-alone* disponibilizado para *download* no próprio site, decodificadores de TV por assinatura, celulares 3G ou outros sistemas capazes de receber dados e que sejam integrados ao SASF. Por existirem tipos diferentes de destinos, abrangendo de celulares a computadores pessoais, a qualidade do vídeo transmitido deve ser também variável de acordo com as características de cada destino. Assim, se o vídeo deve ser assistido num celular que tem resolução máxima de 240x320, não há a necessidade desse

ser transmitido em *1080p* [3].

Um vídeo numa resolução muito maior do que a capacidade de exibição do aparelho gera dois problemas. O primeiro é a necessidade de redimensionar o vídeo para que caiba na tela, processo que pode ser bastante custoso em termos de processamento. Já o segundo é o gasto desnecessário de banda passante, uma vez que, neste caso, um vídeo na resolução suficiente seria menor, significando menos *bytes* sendo transferidos. Por outro lado, se o destino for um decodificador de TV por assinatura, como a tela de aparelhos de televisão tem em geral uma resolução maior que telas de celulares, espera-se que o vídeo transmitido seja em maior resolução. Por isso, são disponibilizadas e sugeridas diversas opções de *streaming* para o usuário, cada uma se adequará melhor ao destino em questão.

3.2.2 Busca, feedback e sugestões ao usuário

O SASF é capaz de disponibilizar centenas de milhares de títulos de vídeos aos seus usuários. Em meio a um acervo deste tamanho, surge a necessidade de facilitar a vida do usuário para que esse encontre o que deseja. Assim, operações de busca por título (original e traduzido), atores, diretores, gênero, ano de lançamento ou palavras-chave são requisitos básicos de um sistema deste porte.

Além da funcionalidade básica de busca, o SASF sugere filmes ao usuário de acordo com seu histórico de aluguéis. Esse mecanismo de sugestão é alimentado não só pelos filmes assistidos, mas também pela nota dada pelo usuário após tê-lo assistido. Essa nota serve para refinar o motor de sugestão e fazer com que as sugestões automáticas se tornem mais precisas ao longo do tempo.

[3] Também chamado de *full HD*. Sua resolução é de 1920x1080 pixels.

Outra forma de sugestão é a manual, onde um usuário sugere um ou mais filmes a um amigo. Já que um membro pode receber sugestões de outros cadastrados, ele deixa de estar isolado dentro do sistema para estar agrupado com amigos de fora do SASF. Isto adiciona aspectos sociais ao sistema, além de melhorar a precisão das sugestões, uma vez que amigos terão, em potencial, mais informações sobre o usuário que o motor de sugestão. Isto também servirá para estimular mais aluguéis a fim da realização de sessões de cinema junto aos amigos.

Para prover ainda mais informações que ajudem na escolha de um vídeo, o SASF também disponibiliza trailers e críticas sobre os filmes. Os trailers são disponibilizados pelas distribuidoras assim como sua sinopse, fotos de divulgação e documentários "por trás das câmeras", todos disponíveis para *streaming* ou leitura independente da assinatura do usuário. Já as críticas podem ser feitas por qualquer um cadastrado no sistema.

3.2.3 Disponibilização de filmes e administração do sistema

Devemos nos lembrar que os usuários que alugam filmes não são os únicos do sistema. Há outros dois tipos de usuários essenciais para que o sistema tenha sucesso, são eles o Administrador e o Distribuidor de Filmes. Observe o diagrama apresentado na Figura Figura 3.2.

Figura 3.2: Diagrama de Casos de Uso simplificado do SASF

O primeiro é o usuário que representa uma empresa distribuidora de filmes. A visão do sistema para esse tipo de usuário é diferente da visão do usuário comum. A empresa ganha dinheiro disponibilizando e incentivando o aluguel de filmes. Dessa maneira, como há o interesse em saber como anda a popularidade de seus vídeos, o SASF provê para a empresa dados sobre aluguéis ao longo de intervalos de tempo customizáveis. Esses dados contêm informações sobre o perfil de cada usuário que alugou o filme (por exemplo, idade declarada ou sexo), mas não contêm sua identidade, por motivos de respeito à privacidade. Esses dados servirão para a distribuidora poder direcionar a divulgação de seus filmes ou verificar se a campanha de publicidade foi efetiva. Para cada filme disponibilizado pela distribuidora, é possível também adicionar sinopses, trailers, fotos de divulgação e documentários "por trás das câmeras" para tornar o filme mais atrativo. Toda essa informação extra se torna disponível a todos os usuários do SASF.

Já o segundo tipo de usuário essencial do SASF é o administrador do sistema. Ele está interessado em manter o SASF funcionando. Sua interação com o sistema consiste em obter informações de monitoração (por exemplo, quantos servidores estão no ar, quantas requisições por segundo cada um está recebendo no momento, o histórico de falhas de comunicação

entre servidores, etc.) e, de acordo com estas informações, atuar sobre ele. As ações do administrador sobre o SASF englobam: iniciar novos servidores para atender uma demanda crescente ou isolá-los para manutenção, habilitar ou desabilitar funcionalidades por excesso de carga ou fins de teste e habilitar ou desabilitar contas de usuários mal-comportados.

3.3 Capacidades do SASF

O desafio de desenvolver o SASF não está na implementação de suas funcionalidades, uma vez que o desafio de desenvolver um sistema de locadora é pequeno e que também já existem vários aplicativos que realizam *streaming* de vídeos. O desafio está no atendimento aos seus atributos de qualidade. Dessa maneira, para passarmos uma noção do tamanho do problema, citaremos alguns números presentes no SASF.

3.3.1 Números de usuários e aspectos de segurança

O SASF é desenvolvido para atender a 10 milhões de usuários cadastrados, onde cerca de 20% desses usam o sistema a cada dia. Como há diversos tipos de usuários e cada um possui informações confidenciais (por exemplo, o número de cartão de crédito usado para pagar a assinatura), cada usuário deve ser autenticado para acessá-las. A autenticação servirá também para identificar o tipo de usuário e, assim, autorizá-lo a realizar apenas o conjunto de funções permitidas à sua categoria.

3.3.2 Tamanho do inventário e número de operações por dia

A principal função do sistema é executada pelo usuário que aluga filmes, e consiste na colocação de filmes em sua respectiva

fila de aluguéis. O acervo do SASF é estimado em cem mil vídeos cadastrados, que estão disponíveis em 55 milhões de DVDs. Deste acervo, são realizados dois milhões de aluguéis por dia. Isto significa dois milhões de execuções por dia do processo de aluguel: (1) relacionar o vídeo a ser enviado ao específico usuário, (2) descobrir de qual ponto distribuidor será enviada a mídia a partir do endereço do usuário, (3) notificar a responsabilidade de entrega da mídia ao ponto distribuidor responsável, e (4) realizar o envio pelos correios da mídia em questão.

3.3.3 Transmissões simultâneas

Já o *streaming* de vídeos é realizado numa taxa menor que o aluguel, o que não representa uma baixa taxa de execuções. São cerca de 150 mil vídeos transmitidos por dia, ou seja, um *stream* de vídeo sendo iniciado a cada 0.57 segundos caso a as transmissões fossem distribuídas uniformemente ao longo do dia. Se considerarmos que a transmissão de um vídeo dura em média uma hora, isso gera uma carga de pouco mais de seis mil usuários simultâneos fazendo *streaming*. O acervo de vídeos para *stream* é menor que o de mídias convencionais, apenas 20 mil títulos, mas cada título está disponível em alguns níveis de resolução e diversas taxas de amostragem. Os títulos, inicialmente, estão disponíveis em dois níveis de resolução: *Standard Definition* (SD) e *High Definition* (HD), ou 720x480 e 1280x720 pixels respectivamente para vídeos em *widescreen* (16:9). É importante notar que *widescreen* não é o único aspecto de vídeo presente no sistema, uma vez que outros podem estar presentes, como por exemplo, o aspecto *Cinemascope* (2.35:1). Dessa maneira, o fator determinante para qualidade do vídeo é sua taxa de amostragem. Inicialmente, o SASF provê três taxas de amostragem para vídeos em SD e duas para vídeos em HD. Assim, o usuário receberá o vídeo com aquela que mel-

hor se adequar à sua conexão de internet. A Figura 3.3 mostra uma tabela com o tamanho esperado para vídeos de duas horas de duração em cada taxa de amostragem disponível. A partir dessa tabela, podemos também ter uma noção do espaço gasto no armazenamento de mídias.

Níveis de definição	SD			HD	
Amostragem (em kbps)	375	500	1000	2600	3800
Tamanho*(em GB)**	0.314	0.419	0.838	2.179	3.185

* Considerando um vídeo de 2 horas de duração
** 1 GB = 1.073.741.824 bytes

Figura 3.3: Tamanhos e amostragens disponíveis

3.3.4 Adição de informações sobre os vídeos

Os usuários do SASF também podem avaliar e escrever críticas sobre os vídeos que já assistiram. Essa avaliação é feita através de uma nota dada ao filme. O SASF possui cerca de dois bilhões de notas já registradas, que podem ou não vir acompanhadas de uma crítica escrita sobre o filme. Essas críticas, inicialmente, não possuem limites de tamanho. Vale também observar que apenas cerca de 5% da notas são acompanhadas de críticas escritas, mas que totalizam cerca de 100 milhões de textos sobre filmes do acervo do SASF.

Note que avaliação e críticas não são as únicas informações relacionadas a cada vídeo do acervo. Cada vídeo possui ainda fotos de divulgação, trailers, sinopse, lista de usuários que já alugaram e lista de usuários com o filme na fila de locação.

Essas informações devem sempre estar disponíveis ao usuário para ajudá-lo na decisão de alugar um filme.

3.3.5 Tempos de resposta

Por fim, observamos que o SASF disponibiliza um grande volume de informação, seja para o usuário comum, através do *streaming*, da busca ou do aluguel de filmes, seja para o administrador, através da monitoração e ação sobre o estado do sistema. Esse volume de informação aumenta naturalmente o tempo de resposta dessas operações. Por outro lado, tempos de resposta acima do especificado ou acima da expectativa do usuário contribuem para o fracasso de um sistema. Assim, as diversas operações providas pelo SASF devem ser realizadas na velocidade da satisfação de cada usuário em questão.

No SASF, diferentes classes de usuários têm diferentes operações à sua disposição. Além disso, essas operações são bastante diferentes entre classes de usuários. Por exemplo, um administrador pode obter a velocidade média da realização da operação de aluguel executada por dado conjunto de servidores ao longo da última semana, enquanto um usuário quer apenas ver as cinco últimas críticas a determinado filme. Por isso, todas as operações disponíveis não terão o mesmo tempo de resposta, mas tempos diferentes de acordo com o volume de dados que opera, sua criticidade, e o *stakeholder* envolvido. Isto será observado ao longo do livro, além de estar descrito de forma mais estruturada no Apêndice XXX, que apresenta fragmentos do documento de requisitos do SASF. [4]

[4] Ainda falta escrever o apêndice, mostrando requisitos funcionais não-funcionais numerados, para tracking, e com valores quantificáveis.

3.4 Resumo

Como podemos observar através de suas capacidades, o SASF se mostra um estudo de caso significativo devido à sua relativa complexidade de seus requisitos. Esses requisitos dificultam ou mesmo impossibilitam seu desenvolvimento se não houver um mínimo de planejamento para atendê-los ou ainda caso não seja adotada uma abordagem, digamos, *arquitetural* para atendê-los.

Nos próximos capítulos estudaremos os aspectos fundamentais para que possamos desenvolver um sistema como o SASF e, passo a passo, mostraremos como esses aspectos se aplicam ao estudo de caso em questão.

Chapter 4

Fundamentos de Arquitetura de Software[1]

No capítulo introdutório, mencionamos que o Design de Software pode ser dividido em duas atividades: design de alto-nível ou arquitetural e design detalhado e que ambas as atividades têm um papel importante no ciclo de desenvolvimento do software. Como no objeto de estudo deste livro é Arquitetura de Software, voltamo-nos agora para a primeira atividade em questão.

Este capítulo tem como objetivo expor o leitor aos fundamentos de Arquitetura de Software ou, em outras palavras, fazer com que seja capaz de:

- Reconhecer, entender, e comparar as diferentes definições existentes do termo *arquitetura de software*
- Relacionar as diferentes definições de arquitetura de software com o padrão ISO/IEEE 1471
- Identificar as características e benefícios proporcionados por uma boa arquitetura

[1] This content is available online at <http://cnx.org/content/m17524/1.21/>.

- Avaliar os benefícios de explicitamente projetar a arquitetura durante o desenvolvimento do software

4.1 Motivação para desenvolver melhores sistemas

Desenvolver software não é uma tarefa fácil. É por esse motivo que muitos projetos de software fracassam durante seu desenvolvimento ou ao obter seus resultados. Entre esses maus resultados, encontramos os que custaram muito acima do orçamento, os incompletos e os que não solucionam os problemas *como* deveriam resolver.

Não é fácil alcançar um bom produto de software devido à complexidade envolvida em seu processo de desenvolvimento. Além de lidar com a complexidade inerente ao problema, devemos também nos preocupar em *como* o software resolve esse problema. Assim, o software deve, além de resolver o problema, resolvê-lo da forma esperada. Ou em outras palavras:

Espera-se que, além de função, o produto de software possua os atributos de qualidade esperados.

Exemplo 4.1
Considere um programa que realize as quatro operações: soma, subtração, multiplicação e divisão. Se o tempo de resposta de suas operações for sempre maior do que o tempo que seu usuário está disposto a esperar, esse programa não terá utilidade mesmo que sempre retorne o resultado correto.

Podemos observar no Exemplo 4.1 que o programa funciona corretamente, mas, por não exibir o desempenho esperado, acaba sendo abandonado. Por outro lado, consertar esse programa para que seja útil é relativamente fácil. Por exemplo,

se o programa não multiplica rápido o bastante, basta apenas reimplementar a função de multiplicação para que tenha um melhor desempenho.

Exemplo 4.2
Considere agora o SASF, já apresentado no Capítulo XX. Considere também que ele se mostra incapaz de responder em menos de dois segundos às operações de aluguel de filmes. Uma vez que os usuários não estão dispostos a esperar esse tempo pela principal operação do sistema, isso resultará numa má experiência de uso, que será motivo para que seus usuários deixem de usá-lo e também deixem de pagar pelo serviço.

Acontece que diminuir o tempo de resposta de uma funcionalidade no SASF, dado o tamanho do sistema, pode não ser tão simples quanto diminuir o tempo de execução de uma função matemática. O alto tempo de resposta de um serviço no SASF pode ser função de uma ou mais decisões tomadas ao longo do desenvolvimento que resultaram na sua estrutura e organização interna. Essa estrutura e organização é o que chamamos de arquitetura. Como o atendimento aos atributos de qualidade do software se deve em grande parte à sua arquitetura, surge a necessidade de estudá-la. E, por fim, é através do estudo das características e técnicas de projeto de arquitetura que poderemos projetar e desenvolver melhores produtos de software.

4.2 O que é Arquitetura de Software

Desde sua primeira menção em um relatório técnico da década de 1970 intitulado *Software Engineering Tecnhiques* [28], diversos autores se propuseram a definir o termo arquitetura de software. Por esse motivo, ao invés de criarmos nossa própria definição do termo, faremos uso de quatro definições exis-

tentes a fim de ressaltar suas diferentes características. As três primeiras que usaremos são as definições de facto do termo. Elas foram formuladas por autores que se destacam na área desde sua introdução e são usadas atualmente pela grande maioria dos professores, alunos e praticantes da área. Por outro lado, também mostraremos a definição de jure de arquitetura de software. Ela é parte do padrão ISO/IEEE 1471-2000 [53] e teve sua criação motivada justamente para fazer com que estudantes, professores e praticantes de arquitetura de software concordem sobre o termo.

4.3 Definição de Arquitetura de Software por Perry e Wolf

Perry e Wolf introduziram sua definição para arquitetura de software em seu artigo seminal *Foundations for the Study of Software Architecture* [84]. A definição que eles propõem consiste na Fórmula (4.1) e na explicação de seus termos:

Fórmula

$$Arquitetura = \{Elementos, Organizao, Decises\} \quad (4.1)$$

De acordo com essa definição, a arquitetura de software é um conjunto de elementos arquiteturais que possuem alguma organização. Os elementos e sua organização são definidos por decisões tomadas para satisfazer objetivos e restrições. São destacados três tipos de elementos arquiteturais:

***Elementos de processamento*: :** são elementos que usam ou transformam informação;

***Elementos de dados*: :** são elementos que contêm a informação a ser usada e transformada; e

***Elementos de conexão*: :** são elementos que ligam elementos de qualquer tipo entre si.

Já a organização dita as relações entre os elementos arquiteturais. Essas relações possuem propriedades e restringem como os elementos devem interagir de forma a satisfazer os objetivos do sistema. Adicionalmente, essas relações devem ser ponderadas de modo a indicar sua importância no processo de seleção de alternativas.

Exemplo 4.3
Um elemento de dados muito presente no SASF e em sistemas de informação em geral é o banco de dados. Ele é o responsável por guardar e recuperar dados no sistema.

No SASF, inicialmente, estão presentes três tipos de dados:

1. Informação textual: informações cadastrais dos usuários e informações textuais sobre os filmes;
2. Imagens: imagens que compõem a identidade visual do sistema, foto do usuário presente em seu perfil e imagens de divulgação dos filmes;
3. Vídeos: filmes completos, *trailers* e documentários "por trás das câmeras" disponíveis para *streaming*.

Por isso, consideramos um elemento de dados para cada tipo. Assim, temos o banco de dados responsável por informações textuais, o banco de dados responsável por imagens e o banco de dados responsável por vídeos. Essa separação de responsabilidades permite que a implementação de cada elemento de dados disponha de serviços diferenciados ou mesmo tire proveito da natureza de seus dados para atender a algum atributo de qualidade (desempenho, escalabilidade, etc.). Dessa maneira, o elemento responsável por texto pode ser otimizado para busca por palavras-chave, enquanto o

responsável por vídeos pode ser otimizado para recuperar grandes massas de dados a cada requisição. Por outro lado, também faz sentido dividir logicamente os elementos de dados em: elemento de dados de usuários e de dados de filmes. Vale notar que essa divisão é ortogonal à divisão em elementos de texto, imagens e vídeos e, portanto, o elemento de dados de usuários pode ser composto por um elemento de dados textuais e outro elemento de dados de imagens, da mesma maneira que o elemento de dados de filmes pode conter o elemento de dados textuais, de imagens e de vídeos.

Como exemplo de elemento de processamento, citamos a lógica de negócio do SASF. Ela contém as regras de negócio que compõem o SASF. Note que podemos ainda dividir esse elemento de processamento em elementos mais especializados: o elemento de processamento responsável por criar, editar, recuperar e remover usuários, o responsável por criar, editar, recuperar e remover informações de filmes, o responsável pelo aluguel de filmes e o responsável por controlar a sessão de *streaming*, entre outros. Essa divisão, assim como a divisão dos elementos de dados, pode ser feita em prol do atendimento aos atributos de qualidade [2].

No entanto, um elemento não é capaz de criar, editar, recuperar ou remover usuários sem se comunicar com os dados dos usuários. Da mesma maneira, o elemento responsável por manipular as informações dos filmes deve se comunicar com os elementos que guardam os dados dos filmes. Ou ainda, para controlar a sessão de *streaming*, o responsável deve obter o filme do elemento de dados que contém os filmes completos. Essa comu-

[2] Trataremos melhor desse assunto no capítulo sobre atributos de qualidade

nicação é feita pelos diversos elementos de conexão do SASF. Entre eles, podemos citar: o driver JDBC [3], que permite a comunicação com o banco de dados responsável pelos usuários; o protocolo FTP, para transferência de vídeos; o protocolo HTTP, para transferências a partir do banco de imagens; ou o REST [4], que é uma especialização do HTTP e é usado para comunicação entre elementos de processamento. A Figura 4.1 ilustra alguns elementos que formam a arquitetura do SASF.

[3] Java Database Connectivity. http://java.sun.com/javase/technologies/database/
[4] REpresentational State Transfer [41]

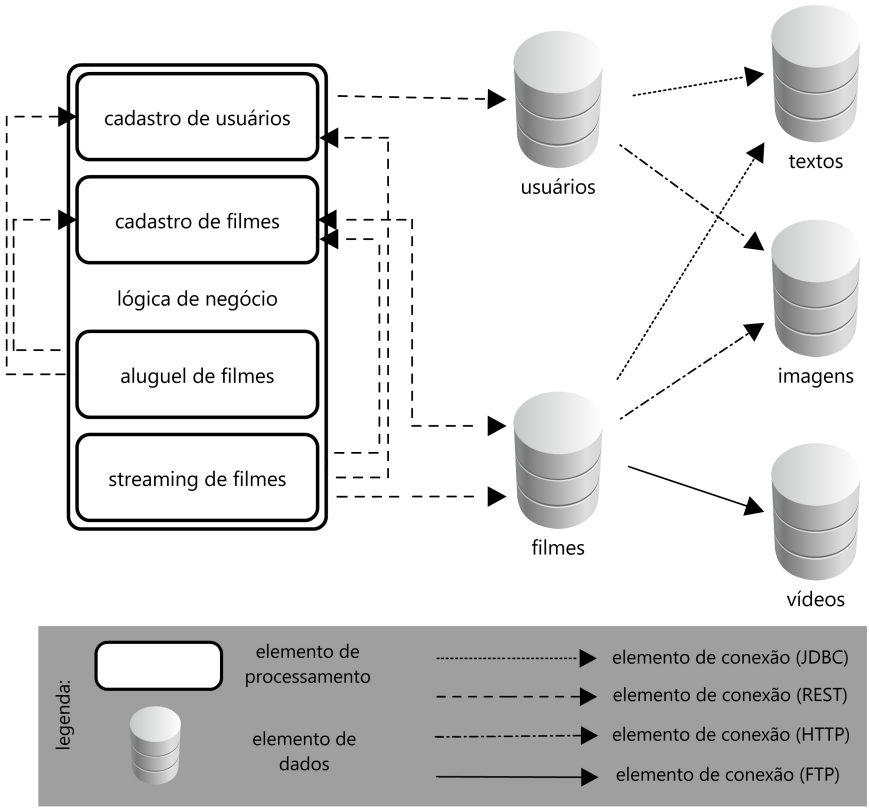

Figura 4.1: Alguns elementos de processamento, de dados e de conexão do SASF

4.4 Arquitetura de Software por Garlan e Shaw

Além de terem uma visão mais concreta sobre arquitetura que Perry e Wolf, Garlan e Shaw são mais explícitos quando mencionam o propósito de se aplicar conhecimentos de arquitetura num sistema de software. Para eles, arquitetura de software se torna necessária quando o tamanho e a complexidade dos sistemas de software crescem. Assim, o problema de se construir sistemas vai além da escolha dos algoritmos e estruturas de dados certos. Esse problema envolverá também decisões sobre as estruturas que formarão o sistema, a estrutura global de controle será usada, protocolos de comunicação, sincronização e acesso a dados, atribuição de funcionalidade a elementos do sistema, ou ainda sobre distribuição física dos elementos do sistema. Além disso, o problema envolverá decisões que impactarão no comportamento do sistema em termos de escala e desempenho, entre outros atributos de qualidade [45].

A visão sobre arquitetura de software de Garlan e Shaw se torna importante por conter três aspectos. O primeiro é por eles serem explícitos em quando devemos aplicar conhecimentos de arquitetura de software – quando lidamos com grandes sistemas. O segundo é por serem claros na separação de tarefas entre design detalhado e design arquitetural – o primeiro se preocupa com algoritmos e estruturas de dados, enquanto o segundo se preocupa com os elementos e organização do sistema como um todo, sendo em relação à estrutura do sistema, controle, comunicação, ou implantação. E, por fim, é por eles citarem que o processo de design da arquitetura precisa se preocupar com atributos de qualidade do sistema – alcançar escalabilidade ou desempenho, por exemplo.

Exemplo 4.4
A arquitetura de um sistema operacional, para atingir

atributos de desempenho e portabilidade, deve se preocupar com diversos aspectos que comporão o sistema. É claro que alguns algoritmos serão também responsáveis pelo desempenho do S.O. em questão, como o responsável pela ordenação por prioridade dos processos em execução ou o de alocação de memória para um novo processo; mas a organização do sistema em camadas de abstração (abstração de hardware, sistema de arquivos e drivers, gerência de processos, API do sistema, bibliotecas e aplicações), a comunicação entre elas (uma camada só pode se comunicar com a camada seguinte, ou aplicações e bibliotecas só podem se comunicar com a API do sistema, etc.) e a sincronização (um aplicativo sugere o arquivamento de dados, mas o sistema de arquivo decidirá quando isso será feito) também impactarão no seu desempenho. Note que essa organização também tem impacto na portabilidade: quanto menos acoplado o resto das camadas for da camada de abstração de hardware, mais fácil será de realizar mudanças para que o sistema operacional esteja disponível para uma nova plataforma de hardware – idealmente, só havendo que se reimplementar essa camada.

4.5 Arquitetura de Software por Bass *et al*

Como veremos a seguir, a definição de Bass et al é bastante similar à encontrada no padrão ISO/IEEE 1471-2000. No entanto, sua especificidade sobre quais propriedades dos elementos arquiteturais devem ser consideradas a faz ser mencionada:

> *A arquitetura de um programa ou de sistemas computacionais é a estrutura ou estruturas do sistema, a qual é composta de elementos de software, as propriedades externamente visíveis desses elementos, e os relacionamentos entre eles. [13]*

Como já observado por Gorton [47], essa definição é explícita quanto ao papel da abstração na arquitetura (quando fala de propriedades externamente visíveis), e também quanto ao papel das múltiplas visões arquiteturais (estruturas do sistema). Devemos também mencionar o uso do termo "elementos de software" como as peças fundamentais da arquitetura. Na edição anterior dessa definição [10], seus autores usavam "componentes de software" ao invés de "elementos de software". Essa mudança foi feita para deixar a definição mais geral, principalmente pelo termo "componente de software" ter um sentido específico na área de Engenharia de Software baseada em Componentes.

Exemplo 4.5
Podemos observar a arquitetura do SASF através de uma visão de partes funcionais (Figura 4.2):

1. módulo responsável pelo cadastro de usuários,
2. módulo responsável pelo cadastro de filmes,
3. módulo responsável pelo aluguel de filmes,
4. módulo responsável pela transmissão de filmes,
5. módulo responsável pela sugestão de filmes, etc.

Esses módulos proveem serviços e informações a outras partes do sistema: por exemplo, uma operação de aluguel ou de transmissão de filmes deve atualizar o histórico presente na conta do usuário. Isso ocorre porque o módulo de sugestão usará periodicamente esse histórico a fim de gerar listas de filmes de acordo com as preferências do usuário.

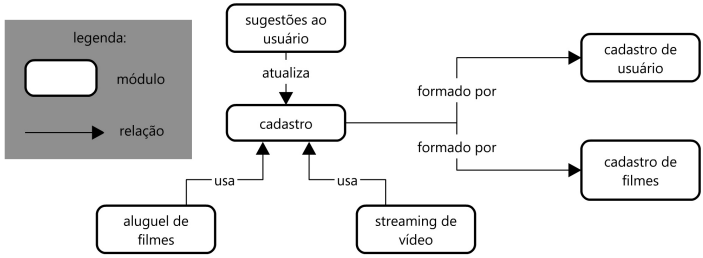

Figura 4.2: Módulos funcionais do SASF

Mas essa não é a única maneira de observarmos o sistema. Podemos também ver o sistema como um conjunto de processos executando e se comunicando em máquinas diferentes, como observado na Figura 4.3. O navegador do usuário, que pode ser considerado parte do sistema e que está executando em uma máquina, se comunica usando o protocolo HTTPS com um servidor de aplicações, que está executando em outra máquina e que contém parte da lógica do negócio (e os módulos de cadastro, autenticação, e atualização do usuário, entre outros). O servidor de aplicações, por sua vez, se comunica de forma diferente com cada um dos sistemas de armazenamento presentes. Ele usa JDBC para obter dados de usuários, FTP para obter vídeos e HTTP para obter imagens. Já o motor de sugestão é visto como outro processo executando numa máquina diferente do servidor de aplicação. Esse processo, de tempos em tempos, lê, processa e atualiza informações do banco de usuários a fim de gerar a lista de filmes sugeridos. Ele também usa JDBC para se comunicar com o banco de usuários.

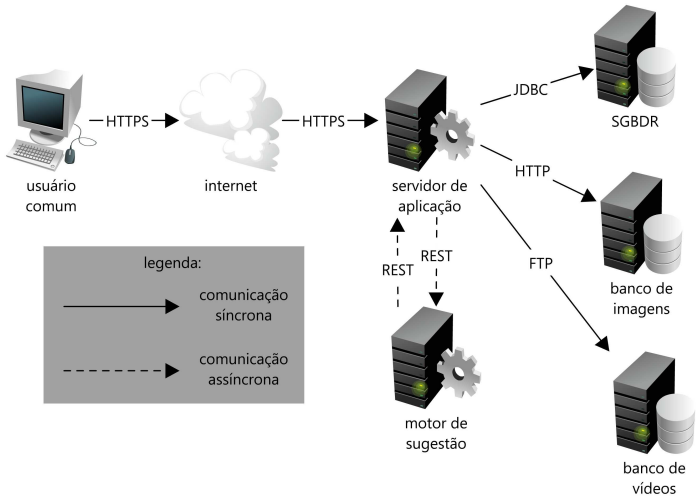

Figura 4.3: Processos presentes no SASF

Na visão em que dividimos o sistema em partes funcionais, podemos perceber aspectos do software como a composição entre elementos ou pontos de reuso. Já na visão em que dividimos o sistema em processos, podemos observar outros aspectos, como propriedades de comunicação e de interação entre as partes do sistema. Por exemplo, na primeira visão, os cadastros de filmes e de usuários podem compor um módulo maior responsável por todos os cadastros. Já na segunda visão, percebemos que a comunicação entre o navegador e o servidor de aplicações é síncrona, enquanto a comunicação entre o motor de sugestão e o banco de dados é assíncrona em relação às ações dos usuários.

4.6 Arquitetura de Software pelo Padrão ISO/IEEE 1471-2000

O propósito da criação do padrão ISO/IEEE 1471-2000 [53] foi o de ajudar no consenso entre autores, estudantes e profissionais sobre o que é e para que serve arquitetura de software. Assim, esse padrão não só define arquitetura de software, mas também introduz um arcabouço conceitual para descrição arquitetural. Sua definição de arquitetura de software, a qual nós adotaremos ao longo do livro, é a seguinte:

> **Definição 4.1: arquitetura de software**
> Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução.

Podemos perceber que a definição acima é consistente com as anteriores por também mencionar que arquitetura compreende estrutura (ou elementos ou componentes), relações, e decisões (ou princípios). No entanto, ela vai além adicionando mais uma preocupação à arquitetura: conduzir a evolução do software.

Evolução de software é o fenômeno de mudança que ocorre no software ao longo dos anos e das múltiplas versões, desde seu início até o completo abandono do sistema. Essa mudança não está só relacionada com a adição e remoção de funcionalidades, mas também está relacionada com a manutenção do código ao longo do ciclo de vida do software. Essa manutenção pode melhorar ou deteriorar tanto atributos externos de qualidade do software, os quais são percebidos pelos usuários (e.g., desempenho, tolerância a falhas, disponibilidade), quanto atributos internos de qualidade do software, os quais são percebidos pelos envolvidos no desenvolvimento (e.g., testabilidade, legibilidade, reusabilidade).

Uma vez que um dos principais objetivos de se projetar uma arquitetura é o de atingir a qualidade desejada pelos interessados no sistema, se torna claro o papel da arquitetura em conduzir a evolução do software, uma vez que ela conterá decisões que contribuirão para a preservação da qualidade do sistema durante seu ciclo de vida.

Antes de entrarmos em detalhes sobre os diversos aspectos de arquitetura de software, devemos entrar em consenso sobre o termo "componente de software". Em Engenharia de Software, "componentes" têm vários significados divergentes. Um significado, de acordo com o *Standard Computer Dictionary* [1], é que um componente é uma das partes que compõem o sistema. Dessa maneira, "componente" pode ser substituído por "módulo", "unidade", ou mesmo "elemento" de software. É esse o significado de "componente" usado no padrão ISO/IEEE 1471-2000 e que será usado ao longo deste livro.

Por outro lado, um componente também pode ter o significado como o descrito por Kai Qian, em *Component-Oriented Programming* [8]: "um pedaço de código autocontido e autoimplantável com uma funcionalidade bem definida e que pode ser agregado com outros componentes através de sua interface." Esse outro significado é estritamente ligado à Engenharia de Software baseada em Componentes e não será usado a não ser que sejamos explícitos sobre ele.

O padrão ISO/IEEE 1471-2000 também define outros termos fundamentais para o entendimento de arquitetura de software, em especial visões (views). Esse termo será brevemente descrito na Seção "Visões da Arquitetura" (Section 4.8: Visões da Arquitetura) e então detalhado no Capítulo XX.

4.7 Decompondo a definição de Arquitetura de Software

A arquitetura de software é mais bem entendida através de suas partes. Considerando as definições expostas acima, podemos ressaltar seus dois principais aspectos, que serão os meios para alcançar os *atributos de qualidade*: *elementos* e *decisões arquiteturais*. Detalharemos cada aspecto a seguir.

4.7.1 Elementos arquiteturais

A arquitetura de um sistema deve definir os elementos que formarão o software. Tais elementos definem como o software é particionado em pedaços menores e, assim, definem como o software é entendido. Elementos arquiteturais são divididos em dois tipos: elementos estáticos e elementos dinâmicos.

Os elementos estáticos de um sistema de software definem as partes do sistema e qual sua organização. Esse tipo de elemento reflete o sistema durante o design e é constituído de elementos de software (e.g., módulos, classes, pacotes, procedimentos, ou ainda serviços autocontidos), elementos de dados (e.g., entidades e tabelas de bancos de dados, arquivos de dados, ou classes de dados), e elementos de hardware (e.g., computadores em que o sistema vai executar, ou outros tipos de hardware que o sistema usará: roteadores, cabos, ou impressoras).

Elementos estáticos não consistem apenas das partes estáticas do sistema, mas também como eles se relacionam entre si. Associações, composições, e outros tipos de relações entre elementos de software, de dados, e de hardware formam o aspecto estático que compõe a arquitetura do sistema. O exemplo a seguir ilustra elementos estáticos de um sistema de software.

Exemplo 4.6
Voltando ao SASF, observar sua arquitetura sob uma

ótica estática expõe seus elementos estáticos. Em tempo de design, alguns elementos estáticos são cada pacote, módulo ou conjunto de classes responsáveis por cada função do sistema. Alguns desses elementos são os responsáveis por: criação, edição, remoção e recuperação de usuários e filmes, aluguel de filmes, autenticação e autorização dos usuários, entre outros.

Por outro lado, elementos dinâmicos definem o comportamento do sistema. Esse tipo de elemento reflete o sistema durante a execução e nele estão incluídos processos, módulos, protocolos, ou classes que realizam comportamento. Elementos dinâmicos também descrevem como o sistema reage a estímulos internos e externos, como mostrado no exemplo a seguir.

Exemplo 4.7
Ainda na arquitetura do SASF, podemos também observar o sistema sob uma ótica dinâmica. Essa exibe seus elementos dinâmicos, a exemplo dos diversos processos executando nas diversas máquinas que compõem o sistema. Esses processos pertencem aos servidores de aplicação, aos serviços de armazenamento, ou mesmo aos navegadores dos usuários.

4.7.1.1 Elementos Arquiteturais e Atributos do Sistema

Note que quando examinamos os elementos arquiteturais de um sistema, tanto os estáticos quanto os dinâmicos, devemos também prestar atenção nas relações que os ligam. Essas relações são importantes, pois especificam a comunicação e o controle da informação e do comportamento que formam o sistema. Assim, as relações definem diversos aspectos do sistema, por exemplo, quais dados do objeto da classe A são visíveis pelos objetos da classe B; ou quantas leituras concorrentes são

feitas no elemento C; ou ainda como o elemento D é autorizado a escrever dados no elemento E. Dessa maneira, essas relações têm efeito sobre atributos de qualidade do sistema, sejam os percebidos pelos usuários, ou os percebidos pelos desenvolvedores. Os exemplos seguintes mostram casos de como relações entre elementos arquiteturais afetam atributos de qualidade.

Exemplo 4.8

Se dividirmos a arquitetura do SASF em três camadas (apresentação, lógica de negócio, e persistência), a camada de persistência pode ser um recurso compartilhado por diversas instâncias da lógica de negócio. Se temos diversas instâncias da lógica de negócio, mesmo que algumas saiam do ar, as restantes proverão disponibilidade ao sistema, desde que a camada de persistência (e.g., o banco de dados) não falhe. Além disso, o compartilhamento do banco de dados pode significar também o acesso concorrente ao mesmo. Assim, quando uma instância da lógica de negócio lhe faz uma requisição, essa requisição lhe será respondida mesmo que outras instâncias estejam fazendo o mesmo (obviamente, isso só ocorre se alguma instância da lógica de negócio não esteja realizando alguma requisição que precise de acesso exclusivo aos dados).

Exemplo 4.9

A separação do sistema em três camadas (Figura 4.4) pode também facilitar a manutenção. Se, além de adotar essa divisão, a camada de apresentação apenas se comunicar com a lógica de negócio, mas não com a de persistência, mudanças na camada de persistência afetarão apenas a camada de negócio. Portanto, caso seja necessário mudar o fornecedor da camada de persistência, a assinatura dos métodos disponíveis, ou mesmo o protocolo de comunicação, apenas a lógica de negócio será afetada por essas mudanças, uma vez que não ex-

iste acoplamento entre a apresentação e a persistência.

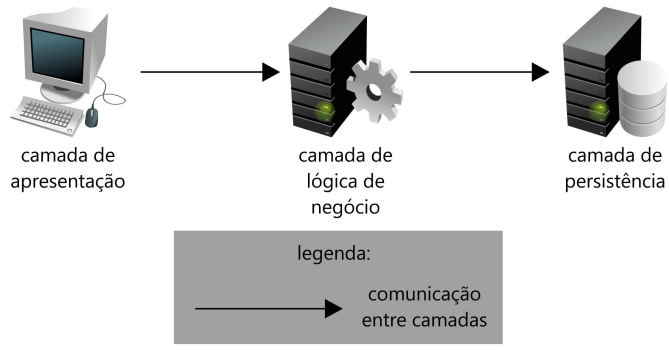

Figura 4.4: Ilustração da divisão de uma arquitetura em três camadas.

4.7.2 Decisões arquiteturais

Uma arquitetura não deve ter suas estruturas definidas aleatoriamente, uma vez que são elas que permitem o sucesso relativo aos objetivos do sistema. Dessa maneira, é trabalho do arquiteto definir essas estruturas em meio às alternativas de design arquitetural existentes. O arquiteto deve decidir entre as alternativas, particionando o sistema em elementos e relações que possibilitarão o atendimento aos atributos de qualidade. Essas decisões são chamadas *decisões arquiteturais*.

Definição 4.2: decisão arquitetural
Uma escolha entre as alternativas de design arquitetural. Essa escolha se propõe a alcançar um ou mais atributos de qualidade do sistema, por meio da(s) estrutura(s) arquiteturais que ela envolve ou define.

4.7.2.1 Características

As decisões arquiteturais têm, basicamente, três características que devem ser consideradas: descrição, objetivos e fundamentação.

A primeira característica é bem clara. É simplesmente a descrição do que foi decidido para o sistema, seja a descrição um elemento, módulo, classe, ou serviço que existirá da arquitetura, a descrição da comunicação de um elemento da arquitetura com outro, a descrição da agregação de diversos elementos diferentes da arquitetura para formar um serviço, ou a descrição de um princípio ou mais princípios que conduzirão a evolução do sistema.

Exemplo 4.10
[Decisão Arquitetural 001] A arquitetura do SASF é dividida em três camadas lógicas: apresentação, lógica de negócio e persistência de dados. A camada de apresentação se comunica apenas com a lógica de negócio e apenas a lógica de negócio de comunica com a camada de persistência de dados.

Toda decisão é feita com um ou vários objetivos. Assim, a segunda característica trata de explicitar qual o objetivo de dada decisão, normalmente, permitindo ou restringido um conjunto de atributos de qualidade do sistema. Vale notar que, para atender aos atributos de qualidade do sistema (que podem ser muitos), uma arquitetura poderá possuir dezenas ou mesmo centenas de decisões arquiteturais.

Exemplo 4.11

(continuação da [Decisão Arquitetural 001]) Objetivo: Essa divisão diminui o acoplamento entre os elementos internos da arquitetura, facilitando o desenvolvimento e a manutenção. [5]

Por fim, uma decisão arquitetural só pode ter sido alcançada em meio a alternativas com algum embasamento ou fundamentação. Então, cabe ao arquiteto explicitar por que tal decisão foi tomada, seja por ser um padrão conhecido na indústria, seja por conhecimento prévio de como satisfazer os objetivos em questão, ou pela atual decisão ter mostrado os melhores resultados em meio a uma avaliação prévia das alternativas.

Exemplo 4.12

(continuação da [Decisão Arquitetural 001]) Motivação: Projetar os elementos internos do sistema de modo que cada um pertença a apenas uma camada lógica ajuda a aumentar a coesão e diminuir o acoplamento. A coesão aumenta, pois cada elemento será desenvolvido com o objetivo de ser parte da apresentação, da lógica ou da persistência do sistema. Dessa maneira, cada elemento terá sua responsabilidade bem definida, mesmo que em alto nível. Como a comunicação entre as camadas é pré-definida, a de seus elementos também é: elementos da camada de apresentação não se comunicarão com elementos da camada de persistência, por exemplo. Assim, o acoplamento entre elementos internos será análogo ao acoplamento entre camadas. Com o baixo acoplamento, o desenvolvimento e a manutenção dos elementos também é facilitado, seja por possibilitar o desenvolvimento independente, seja por mudanças em um elemento terem

[5] TODO: Adicionar os *drivers* dessa decisão: o Requisito(s) Não-Funcional(is) XX presente(s) no Apêndice. Exemplo de requisito: [RNF 01] Ter mais de uma interface gráfica.

menor impacto nos outros.

4.7.2.2 Rastreabilidade

Vale notar que decisões definem que elementos comporão o sistema. No exemplo anterior, podemos observar que a decisão define elementos como plug-ins, pontos de extensão, etc. Assim, por relacionarem atributos de qualidade (ou requisitos) a elementos arquiteturais, as decisões contidas numa arquitetura facilitam o chamado *rastreamento de requisitos*.

> **Definição 4.3: rastreamento de requisitos**
> É o processo/capacidade de ligar requisitos do sistema a estruturas arquiteturais.

A possibilidade de se rastrear requisitos na arquitetura é uma característica importante porque facilita o entendimento e a manutenção do sistema representado pela arquitetura. O entendimento do sistema é facilitado porque uma arquitetura permite que um interessado qualquer navegue pelos elementos que compõem o sistema em dois sentidos: tanto do nível mais abstrato do sistema para seus níveis mais concretos, ou seja, dos requisitos para os elementos arquiteturais, como módulos, bibliotecas, serviços, ou classes; quanto dos níveis concretos da arquitetura para os níveis mais abstratos, ou seja, dos elementos arquiteturais para os requisitos do sistema.

O entendimento do sistema é facilitado porque uma arquitetura permite que um interessado qualquer navegue pelos elementos que compõem o sistema em dois sentidos: tanto do nível mais abstrato do sistema para seus elementos mais concretos, ou seja, dos requisitos para as estruturas arquiteturais, como módulos, bibliotecas, serviços, ou classes, quanto dos elementos concretos da arquitetura para os níveis mais abstratos, ou seja, das estruturas arquiteturais para os requisitos do sistema.

Exemplo 4.13

Se observarmos a arquitetura do SASF e procurarmos pelas decisões responsáveis por facilitar a manutenção do sistema, encontraremos entre elas a decisão do Exemplo 4.12. Essa decisão sugere uma divisão do sistema em camadas lógicas, mas também influencia na divisão em pacotes, serviços ou mesmo processos. Assim, a satisfação do requisito de manutenibilidade está diretamente ligada à correta divisão das partes do sistema em apresentação, lógica de negócio e persistência.

Da mesma maneira, se partirmos das partes que formam as camadas de apresentação, lógica de negócio e persistência, observaremos que elas estão ligadas à divisão do sistema (e à decisão arquitetural) que se propõe a atender a requisitos de manutenibilidade.

Além de permitir a navegação, um aspecto que merece ser ressaltado é que se os requisitos do sistema forem eventualmente ordenados por importância para o sucesso do sistema, os elementos arquiteturais também possuirão diferentes níveis de importância. Essa ordenação, então, significará diferentes níveis de investimento, seja em tempo ou dinheiro, na construção dos elementos arquiteturais para o sucesso do sistema.

Adicionalmente, a manutenção do sistema é facilitada de uma forma análoga ao seu entendimento. Se algum requisito é atendido insatisfatoriamente, por meio da arquitetura é possível descobrir quais elementos do sistema estão envolvidos na insatisfação desses requisitos. Da mesma maneira, a arquitetura possibilita descobrir quais requisitos serão afetados por um dado elemento arquitetural caso esse sofra uma mudança ou manutenção.

Exemplo 4.14

Se uma modificação na camada de apresentação só

pode ser feita se a camada de persistência também for modificada, isso pode significar que a decisão arquitetural do Exemplo 4.12 não está sendo seguida corretamente. Portanto, o requisito de manutenibilidade também não está sendo atendido corretamente e essa divergência da arquitetura deve ser corrigida o quanto antes.

4.7.2.3 Evolução

Devido às suas características, se torna fácil perceber que o registro das decisões arquiteturais na forma de um documento – o documento arquitetural – agrega valor ao ciclo de vida do software, uma vez que facilita o processo de rastreamento de requisitos. Adicionalmente, se algum tipo de registro histórico das decisões arquiteturais existir, o processo de rastreamento pode também ser realizado para as diversas versões do sistema, facilitando assim o entendimento da evolução do mesmo.

Além de descreverem estruturas arquiteturais, as decisões também descrevem princípios que conduzirão a evolução do sistema. Isso significa que uma decisão não necessariamente descreverá módulos, classes, ou serviços, mas também poderá descrever regras que deverão ser seguidas ao longo do desenvolvimento do sistema. A seguir, citamos e exemplificamos alguns tipos de regras a serem descritas pelas decisões arquiteturais.

- Regras para adição de funcionalidade ao sistema.

  Exemplo 4.15
  Uma nova funcionalidade do SASF não poderá adicionar uma carga maior que *mil* requisições por segundo ao banco de dados de usuários, con-

siderando a média atual de dez mil usuários simultâneos no sistema.

Exemplo 4.16
Uma nova funcionalidade de um editor de imagens só será adicionada implementando o ponto de extensão *ProcessImagePlugin*. Esse ponto de extensão permite obter a imagem que está aberta no workspace do usuário e seus atributos, além de permitir a exibição de uma caixa de diálogo que permitirá ao usuário entrar com parâmetros que servirão para a execução do *plug-in*. O retorno dessa nova funcionalidade sempre será uma imagem (processada ou não). A nova funcionalidade, para ser adicionada, deve conter um arquivo de configuração em texto sem formatação que conterá o atributo *extension-class* que indicará o caminho para a classe da nova funcionalidade que implementa *ProcessImagePlugin*.

Exemplo 4.17
Uma nova funcionalidade do sistema de edição de texto não poderá modificar a *GUI* de forma que adicione mais do que um botão na área de trabalho em sua configuração padrão.

- Regras para remoção ou desativação de funcionalidades, seja durante o desenvolvimento, implantação, ou execução do sistema.

  Exemplo 4.18
  No SASF, a remoção de um serviço do módulo responsável pelo *streaming* para outros dispositivos será feita em duas etapas. Na primeira etapa, o serviço será marcado como *deprecated*, retornando assim, além da resposta padrão, uma

flag avisando que na próxima versão ele será descontinuado. Será ainda disponibilizada uma solução que contorne a ausência desse serviço (serviços alternativos, por exemplo). Na segunda etapa, que deverá acontecer no mínimo 1 mês depois da primeira etapa, o serviço será desativado, retornando uma mensagem padrão de erro avisando que o serviço deixou de existir.

Exemplo 4.19
Caso o consumo de recursos computacionais do SASF ultrapasse 80% do total, alguns de seus serviços podem ser completamente ou parcialmente desativados. Um serviço que pode ser desativado temporariamente sem que os usuários percebam é o motor de sugestão de filmes. Como cada usuário está acostumado a ter sua lista de sugestões atualizada apenas "de tempos em tempos", mas não tem certeza qual é o real intervalo entre cada atualização, se dada atualização demorar alguns minutos ou horas a mais para acontecer, dificilmente o atraso será notado. Em casos extremos, devido ao seu grande consumo de recursos, o serviço de *streaming* de vídeo também pode ser desativado. No entanto, essa decisão deve também levar em conta o alto grau de insatisfação de usuários que causará e que, fatalmente, poderá ser convertida em perda de faturamento. Uma alternativa é desativar a transmissão de vídeo para apenas algumas opções de resolução. Assim, o grau de insatisfação será menor, uma vez que apenas uma parte dos usuários não será atendida pelo serviço de *streaming*.

- Regras para modificação ou manutenção de funcionali-

dades.

Exemplo 4.20
Não haverá modificação do *Web Service* que realiza busca e aluguel de filmes no SASF que é disponibilizado para uso por serviços externos. Se for realmente necessária a modificação, dois *Web Services* ficarão disponíveis: o antigo, completamente suportado, e o novo, que passará a ser adotado por novos sistemas a partir da data de seu lançamento. O antigo só será desativado depois da adoção do novo serviço ser feita por *todos* os serviços externos.

- Regras de atendimento a atributos de qualidade.

  Exemplo 4.21
  No Exemplo 4.19, a disponibilidade de parte das funcionalidades, i.e., construção da lista de sugestões de filmes ou transmissão de vídeos, é mais importante do que a indisponibilidade de todas as funções: caso o uso dos recursos computacionais alcance 100%, usuários começarão a não ser atendidos de forma descontrolada. Assim, prefere-se que uma menor parte dos usuários não seja atendida, apenas os que desejam assistir a filmes em alta definição, do que a maior parte, que são os que desejam alugar filmes ou assisti-los em definição padrão.

  Exemplo 4.22
  A disponibilização de uma nova funcionalidade no SASF será feita em etapas para 10%, 25%, 50%, 100% desses usuários. Dessa maneira, será possível avaliar o comportamento da nova função no sistema sob carga real. Além disso, a desati-

vação da funcionalidade poderá ser feita através de uma *flag* de controle, permitindo o retorno às funcionalidades anteriores do sistema em caso de sobrecarga dos recursos por parte da nova funcionalidade.

Exemplo 4.23
Antes da implantação de uma nova versão de um serviço de infraestrutura, digamos, um novo banco de dados, a carga gerada pelos usuários da versão antiga será espelhada para a nova versão. Assim, será possível avaliar seu comportamento com uma carga real e, portanto, saber o que esperar quando o novo banco de dados substituir a versão em produção.

No Capítulo XX, Documentação da Arquitetura, voltaremos às decisões arquiteturais, onde aprenderemos a categorizá-las e documentá-las.

4.7.3 Atributos de qualidade

Uma das principais preocupações da arquitetura é o atendimento aos atributos de qualidade do sistema. Atributos de qualidade, como já introduzidos no capítulo anterior, são a maneira *como* o sistema executará suas funcionalidades. Esses atributos são impostos pelos diversos interessados no sistema e podem ser classificados em três tipos: atributos do produto, atributos organizacionais, e atributos externos.

Atributos de qualidade do produto são aqueles que ditam como o sistema vai se comportar. Exemplos clássicos desse tipo de atributo de qualidade são escalabilidade, desempenho, disponibilidade, nível de entendimento ou mesmo portabilidade. Podemos observar requisitos de escalabilidade no Exemplo 4.24 e requisitos de portabilidade no Exemplo 4.25.

Exemplo 4.24
Sistemas de redes sociais costumam ter uma grande massa de usuários. Como, a partir do lançamento de um sistema desse tipo, sua massa de usuários cresce bastante, é desejável que o crescimento do consumo de recursos em relação ao crescimento do número de usuários não seja muito acentuado – de forma que a escala seja viável para a gerência do sistema. Para atender esse requisito, a arquitetura deve ser muito bem pensada em termos de consumo de recursos por usuário, tirando proveito de diversas técnicas como *caching*, processamento assíncrono, replicação, entre outras.

Exemplo 4.25
Um requisito desejável em um jogo de videogame é que ele esteja disponível para diversas plataformas de entretenimento. Como diferentes plataformas têm diferentes especificações ou ainda usam diferentes tipos de *hardware*, atingir a portabilidade pode não ser trivial. Entre as técnicas de portabilidade, a mais usada acaba sendo a abstração dos aspectos específicos à plataforma – principalmente o *hardware*, mais especificamente primitivas de desenho em tela ou armazenamento em disco – da lógica do jogo. Assim, toda ou boa parte da camada lógica é reusada, enquanto as camadas de níveis mais baixos de abstração são portadas para as diferentes plataformas.

Já atributos de qualidade organizacionais, por outro lado, são consequência de políticas ou procedimentos organizacionais. Em outras palavras, o sistema deve respeitar padrões ou regras impostas por uma ou mais organizações envolvidas para atender a esses requisitos.

Exemplo 4.26

Se um sistema que servirá de infraestrutura será produzido para uma organização ou empresa que já possui diversos sistemas que implementam o padrão *Web Service Distributed Management* (Gerência Distribuída de *Web Services*), a adoção desse padrão na arquitetura do novo sistema é um requisito a ser atendido, por ser imposto pela organização em questão. A adoção desse padrão implica na disponibilização via *Web Service* de serviços de ativação, consulta e desativação do sistema ou parte dele, que terá impacto na arquitetura do sistema como um todo.

Por fim, restam os chamados atributos de qualidade externos, que não são impostos pelo processo de desenvolvimento nem pelo projeto do sistema. Neles se encaixam leis impostas sobre software ou requisitos de interoperabilidade entre sistemas.

Exemplo 4.27

Para o SASF atrair usuários de outros sistemas (p. ex., redes sociais), percebeu-se que ele deve ser capaz de agregar o perfil do usuário existente nos outros sistemas. Esse tipo de agregação (que permitiria não só a visualização dos perfis compartilhados entre os diversos serviços, mas também sua edição), impactará profundamente na arquitetura do sistema, uma vez que será necessário organizar dados locais e dados compartilhados por terceiros, além de manter todos os dados sincronizados ao longo do tempo e das eventuais modificações.

4.7.3.1 Medindo atributos de qualidade

É importante notar que para se definir o sucesso do software em relação aos atributos de qualidade, precisamos medir o quanto

o sistema satisfaz esses atributos. Em primeiro momento, essa medição de sucesso parece simples: "basta considerar o valor esperado do atributo de qualidade, digamos, 'o sistema deve estar disponível 99,999% do tempo'; medir se ele atinge os valores esperados, 'num período de 1 ano, o sistema esteve parado por 1 hora'; e, por fim, atestar seu sucesso ou fracasso: '1 hora equivale a 0,0114% e, portanto, o sistema não atendeu ao requisito de disponibilidade."' No entanto, não é fácil estabelecer métricas quantitativas para atributos de qualidade como testabilidade, usabilidade, ou manutenibilidade são bem mais difíceis de estabelecer métricas quantitativas e, portanto, não é fácil atestar o sucesso em relação a esses atributos.

4.7.3.2 Relacionando atributos de qualidade

Além de serem difíceis de medir, atributos de qualidade se relacionam entre si de forma que um pode permitir, ajudar ou mesmo dificultar o atendimento de outros. Essas relações entre atributos acontecem mesmo que eles sejam de tipos diferentes.

No Exemplo 4.28, notamos que o atributo de qualidade desempenho está afetando os níveis de testabilidade e entendimento do sistema.

Exemplo 4.28

Uma forma de aumentar o desempenho do sistema é diminuir os níveis de indireção usados na comunicação entre dois elementos quaisquer no SASF. Um caso simples seria fazer com que algumas chamadas presentes na camada de apresentação usassem diretamente a camada de persistência, sem usar a lógica de negócio. Essa medida tornaria as chamadas da apresentação mais rápidas, uma vez que menos chamadas remotas seriam executadas. No entanto, quando diminuímos as camadas de abstração entre dois elementos inicialmente distintos, aumentamos o acoplamento entre eles

e, portanto, dificultamos seu entendimento ou mesmo sua testabilidade.

Já no exemplo a seguir, o atributo de segurança afeta dois atributos distintos: o desempenho e a usabilidade do sistema.

Exemplo 4.29

Uma forma de aumentar a segurança de um sistema operacional é requerer autorização do usuário para a realização de certas operações. No entanto, o processo de verificação do usuário (além de todos os elementos e abstrações do sistema relacionados à segurança: unidade certificadora, unidade verificadora, listas de controle de acesso, entre outros.) deteriorará o desempenho da aplicação, dado que consumirá recursos que poderiam ser destinados à operação em si - não a um aspecto dito não-funcional dela. Além disso, o sistema vai ficar menos usável, uma vez que pedirá uma verificação, seja senha, impressão digital, ou certificado, para cada operação sensível a ser executada.

O principal motivo que faz com que atributos de qualidade conflitem é por eles serem impostos por mais de um interessado no software. Assim, como preocupações de diferentes interessados podem conflitar, os atributos de qualidade também conflitarão. Assim, cabe à arquitetura resolver, ponderar, ou ao menos mediar esses conflitos, considerando assim os diversos *trade-offs* envolvidos para se alcançar os objetivos do software. O exemplo seguinte mostra atributos de desempenho e portabilidade conflitando.

Exemplo 4.30

Um cliente de um jogo para celular requisitou que o jogo tivesse um bom desempenho nos diversos aparelhos disponíveis no mercado. No entanto, o gerente de projeto sugere que o tempo gasto para portar o software de um aparelho para outro seja mínimo, uma

vez que o prazo do projeto em questão é curto. Podemos então observar dois requisitos conflitantes: desempenho e portabilidade.

Esse conflito ocorre porque as técnicas para alcançar ambos os requisitos são divergentes. Para alcançar portabilidade, normalmente é necessário o uso de diversas camadas de abstração, principalmente de *hardware*. No entanto, a adição dessas camadas de abstração significa uma perda em desempenho, uma vez que aumentará o número de chamadas necessárias para se realizar qualquer operação. E isso se torna ainda mais significativo no caso dos aparelhos celulares, que podem ser limitados em termos de recursos computacionais como processador ou memória.

Assim, a arquitetura do sistema terá que ponderar entre as técnicas disponíveis de modo que atenda em parte cada requisito e, assim, ambos os interessados fiquem satisfeitos.

Dois outros atributos de qualidade que normalmente conflitam são os atributos usabilidade e segurança, como veremos no exemplo a seguir. Nesse caso, ambos os atributos foram requisitados pelo mesmo interessado, o usuário, e, mesmo assim, se tornaram conflitantes.

Exemplo 4.31

Quando usando um sistema operacional, um mesmo usuário procura atributos de segurança e usabilidade para suas operações. Para segurança, ele deseja que suas operações no sistema ou seus resultados não sejam afetados por ações de outros usuários. Esse atributo, que na arquitetura implicará em soluções de autenticação, verificação, listas de permissões, etc., imporá que as tarefas realizadas por qualquer usuário eventualmente terão sua autenticidade e permissão verifi-

cadas. Essa interrupção para realizar as devidas autorizações deteriora o atendimento do atributo de usabilidade, uma vez que o usuário terá suas atividades interrompidas por algo que não gera resultado para ele.

Veremos mais sobre atributos de qualidade de software, suas relações, como alcançá-los, e seus interessados no Capítulo Y.

4.8 Visões da Arquitetura

Como consequência da existência dos diversos interessados nos objetivos alcançados pelo software, a arquitetura também possuirá diversos interessados. No entanto, uma vez que os interessados no sistema têm diferentes preocupações e níveis de conhecimento, a arquitetura não deve ser exposta da mesma maneira para interessados diferentes. Para resolver esse problema, surge o conceito de *visões arquiteturais*.

Exemplo 4.32

Considerando a arquitetura do SASF, vejamos as preocupações de dois interessados diferentes: o implementador e o responsável pela disponibilidade do sistema em produção. O implementador está preocupado com módulos, classes e algoritmos que ele e seu time terão que construir, como e com quais subsistemas esses módulos irão se comunicar ou ainda quais restrições de comunicação foram impostas em seu design. Já o responsável pela disponibilidade está preocupado em como o SASF está distribuído entre as máquinas, que funcionalidades serão afetadas caso um conjunto específico de máquinas deixe de funcionar, ou como será possível realizar a troca de um servidor sem afetar o tempo de início de uma transmissão de vídeo.

Podemos observar que há preocupações bem diferentes entre os dois interessados e assim perceber que dimensões bem difer-

entes da arquitetura são necessárias para satisfazê-los. Para o primeiro, a arquitetura deve mostrar que módulos lógicos (pacotes, classes, bibliotecas) compõem o sistema, além das relações de comunicação e restrição entre eles. Já para o segundo, a arquitetura deve mostrar como o sistema está dividido fisicamente, quais partes do sistema estão executando em quais computadores, quais os links físicos entre esses computadores, etc.

Uma visão arquitetural é uma representação da informação (ou parte dela) contida na arquitetura de forma que se adéque às necessidades de um ou mais interessados. Ela facilita o entendimento da arquitetura por parte do interessado, uma vez que vai filtrar e formatar a informação de acordo com as necessidades e preocupações do interessado em questão.

Definição 4.4: visão arquitetural
É a representação do sistema ou de parte dele da perspectiva de um conjunto de interesses relacionados.

Não podemos esquecer que o próprio arquiteto também pode tirar proveito desse conceito durante o processo de design da arquitetura. Quando um arquiteto faz design, ele usa o conceito de visões arquiteturais para assim endereçar as diferentes preocupações do sistema por vez. Dessa maneira, ele divide o problema de design em problemas menores e, consequentemente, menos complexos: ele endereça cada atributo de qualidade – cada aspecto do sistema – que serão alcançados por essa arquitetura. Atacando uma visão por vez, o arquiteto pode, por exemplo: primeiro definir as partições lógicas, ou seja, os módulos funcionais que comporão o sistema – e assim considerar uma visão lógica do sistema; definir as partições dinâmicas do sistema, ou seja, quais processos, *threads* e protocolos estarão presentes no sistema – considerar uma visão de dinâmica; definir as partições do ponto de vista de implementação, ou seja, que classes, pacotes e bibliotecas comporão o

sistema – considerar uma visão de desenvolvimento; e, por fim, definir onde as partes dinâmicas executarão, ou seja, onde e em quais máquinas os diversos "executáveis" do software estarão implantados, além de como eles vão se comunicar – considerar uma visão de implantação do sistema.

4.9 O Documento de Arquitetura

Considerando o que mencionamos até agora sobre arquitetura de software, percebemos que ela provê diversos benefícios: proporciona atendimento de atributos de qualidade, ajuda na comunicação entre os interessados no sistema e guia a evolução do sistema. No entanto, até agora, só falamos da arquitetura como algo abstrato. Ou seja, apenas falamos dela como uma propriedade imposta ou emergente de um sistema, mas não falamos em como documentá-la, nem fomos específicos quanto aos benefícios proporcionados por sua documentação.

4.9.1 Benefícios

Um documento de arquitetura não é nada mais que um documento que descreve a arquitetura do sistema e, portanto, descreve elementos, relações, e decisões arquiteturais do sistema em questão. Assim, os benefícios de se documentar a arquitetura se tornam análogos aos benefícios proporcionados pela própria arquitetura. No entanto, pelo documento de arquitetura ser um artefato concreto, ele poderá ser reproduzido, reusado, comunicado e analisado contra o código gerado a partir da arquitetura em questão.

Em resumo, a documentação da arquitetura proporcionará os seguintes benefícios:

- Ajudará na introdução de novos membros ao time de desenvolvimento do sistema, uma vez que é um documento

que abstrai o sistema a diferentes visões que representam diferentes preocupações;

Exemplo 4.33

Um novo desenvolvedor acabou de ser contratado e passou a integrar o time de desenvolvimento de um sistema que já soma 250 mil linhas de código. Para esse desenvolvedor se familiarizar com o sistema, não é uma boa idéia para ele mergulhar no código de cabeça, mas entender por partes como as coisas funcionam. Esses diversos níveis de abstração até chegar ao código propriamente dito devem estar disponíveis na arquitetura do sistema, que se mostrará um bom ponto de partida para o entendimento do sistema.

- Servirá de ponte para a comunicação entre os diversos interessados do sistema. Uma vez que a arquitetura é projetada para satisfazer diversos interessados, sua documentação também o será. O documento de arquitetura servirá de arcabouço conceitual para comunicação entre diferentes interessados no sistema, uma vez que define seus elementos e relações que o compõem.

  Exemplo 4.34

  Usando a arquitetura para mapear custos às funcionalidades que o sistema proverá, o gerente pode justificar ao financiador do projeto a necessidade de se adquirir uma licença para um banco de dados específico. Ou ainda citar quais as consequências caso essa licença não seja adquirida: a funcionalidade provida pelo banco deverá ser então implementada pelo time de desenvolvimento, que precisará de dois meses para tanto. Essa possibilidade de "navegar" pelo sistema e pelas diversas visões, seja a de gerente, seja a de financiador,

ou de desenvolvedor, é facilitada pelo documento de arquitetura.

- Servirá como modelo do sistema para a análise. Uma vez que é uma representação manipulável do sistema, a documentação poderá ser analisada, desde que contenha informação suficiente para tanto.

  Exemplo 4.35
  A arquitetura do SASF, dividido em três camadas (apresentação, lógica de negócio e persistência), descreve que cada camada estará executando em máquinas diferentes. É certo que a descrição de cada camada possui informações de quantas máquinas serão necessárias para determinada carga de usuários, como máquinas da mesma camada se comunicarão e também como elas se comunicarão com máquinas de diferentes camadas. Assim, com essas informações, é possível algum tipo de análise e estimativa do custo do sistema em produção (e.g., número de CPUs por hora, banda passante entre as máquinas, ou banda passante disponível para os usuários), inclusive com base no crescimento do número de usuários, mesmo que o sistema ainda não tenha sido construído.

- Dificultará uma especificação imprecisa. Quando o arquiteto projeta a arquitetura, mas não a materializa em um documento, pode haver pontos de discordância que eventualmente não serão avaliados por, simplesmente, não estarem explícitos.

  Exemplo 4.36
  Num sistema de controle de vôo, onde vidas estão em risco, o documento da arquitetura é

também um contrato. Ele é avaliado por cada interessado em questão, que deve consentir com a forma de como serão realizadas as funções do sistema e como serão medidos seus atributos de qualidade de forma a garantir o sucesso do sistema antes mesmo que esse seja construído.

4.9.2 Dificuldades

No entanto, documentar a arquitetura é tão ou mais difícil que criá-la. Os principais motivos são três: o documento reflete a complexidade da arquitetura, que geralmente é alta; o documento reflete o tamanho da arquitetura, que o torna custoso para construir e ser lido; e o documento, por seu tamanho e complexidade, é difícil de manter consistente com o sistema que ele descreve.

A complexidade do documento surge principalmente da necessidade de mostrar de diferentes maneiras os diferentes aspectos da arquitetura, ou seja, da necessidade de mostrar as diferentes visões da arquitetura. Cada visão possui uma forma de melhor ser representada e também deve estar consistente com as outras visões.

Exemplo 4.37

Na documentação da arquitetura do SASF podemos observar, entre outras, duas visões diferentes: uma visão que mostra aspectos dinâmicos e outra que mostra o sistema estaticamente.

A visão estática mostra os principais módulos funcionais do software e, na Figura 4.5, foi representada por um diagrama de classes em Unified Modeling Language (UML) contendo os módulos funcionais e sua descrição. Entre esses módulos funcionais, podemos

encontrar o responsável pelo cadastro de usuários, o responsável pelo cadastro de filmes, o responsável por sugerir novos filmes a usuários, e o responsável pelo *streaming* de filmes.

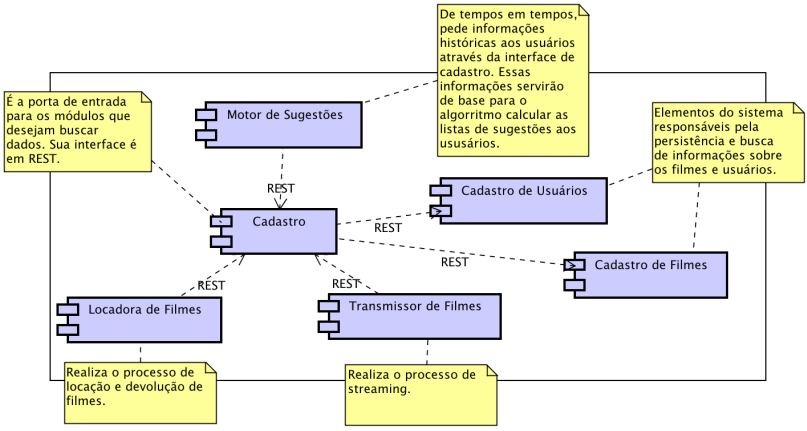

Figura 4.5: Uma visão estática da arquitetura do SASF

Já a visão dinâmica da arquitetura se preocupa em mostrar os módulos que possuem comportamento dinâmico no sistema. Aqui, eles foram representados por um diagrama de sequência, também em UML, que mostra seu comportamento e suas interações com outros módulos (Figura 4.6). Obviamente, os módulos usados nessa visão devem ter correspondentes na visão estática.

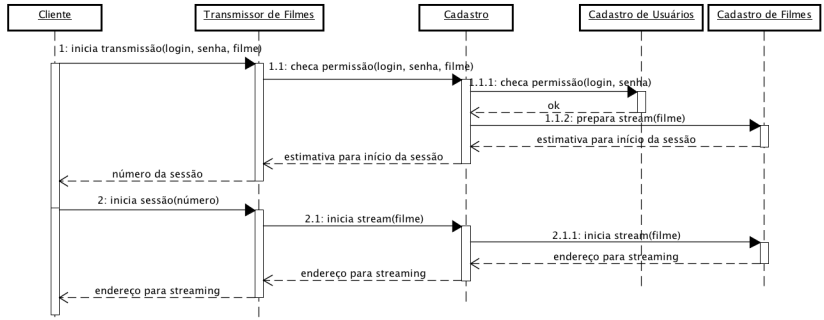

Figura 4.6: Uma visão dinâmica da arquitetura do SASF, mostrando o comportamento de alguns módulos durante o processo de transmissão de um filme.

Documentos grandes levam tempo para serem construídos. Além disso, documentos grandes, na prática, não são usados a não ser que proporcionem para o desenvolvimento um benefício maior que o custo de lê-lo. Essa realidade pode ser traduzida em duas fases. Na primeira, é feito um grande esforço para se construir o documento de arquitetura. Ainda nessa fase, o documento é completo e consistente com o sistema, além de ter o potencial para prover os benefícios de uma arquitetura bem documentada. No entanto, a segunda fase consiste no processo de desatualização do conteúdo do documento, que ocorre por falha no processo ou pelo alto custo de se manter o documento consistente, e que tem por consequência a inutilização do documento de arquitetura e o possível aumento da entropia no sistema.

O problema da inconsistência da arquitetura com o código acontece porque, em muitos processos de desenvolvimento, arquitetura evolui ao longo do tempo, seja uma evolução planejada ou não. Uma evolução não-planejada pode acontecer da

forma descrita no exemplo a seguir.

Exemplo 4.38

Lembrando da arquitetura do SASF, que foi dividida em três camadas: apresentação, lógica de negócio e persistência, uma das decisões impostas dita que a camada de apresentação só pode se comunicar com a lógica de negócio. No entanto, um desenvolvedor, medindo que a exibição da interface está demorando porque o carregamento das imagens necessárias está lento, resolve modificar a interface para que proceda da seguinte maneira. O pedido das imagens é feito diretamente à camada de persistência, contornando assim o *overhead* da camada lógica para tanto. Uma vez que ele nota que o desempenho da exibição da interface com o usuário agora está satisfatório, ele adiciona essa mudança ao código.

Acontece que, com isso, ele adicionou uma mudança também na arquitetura do sistema. A partir daí, há comunicação entre o módulo de interface e de persistência, fazendo assim que a documentação da arquitetura esteja inconsistente em relação ao código do sistema.

4.10 Por que documentar a arquitetura de software?

Como já foi mencionado no padrão ISO/IEEE 1471-2000, a arquitetura de um sistema existe independentemente dela ter sido documentada ou planejada. No entanto, em pequenos sistemas, pensar, planejar, documentar e manter a arquitetura pode não ser necessário: um conjunto de classes e pacotes ou de módulos com suas relações e evolução minimamente pensados (ou uma Big Ball of Mud) pode atender aos requisitos

funcionais e os atributos de qualidade do sistema. Normalmente, isso acontece quando os requisitos não são difíceis de serem atendidos. Assim, todos os interessados ficam satisfeitos – que podem não ser muitos ou conflitantes – e o sistema atinge o sucesso esperado.

Exemplo 4.39

Pensemos num pequeno sistema que servirá para a organização de uma modesta locadora de filmes. Ele será capaz de cadastrar, recuperar, atualizar e remover filmes, cadastrar, recuperar, atualizar e remover DVDs de filmes, cadastrar, recuperar, atualizar e remover clientes, realizar locações, devoluções e reservas.

Se a execução desse sistema estiver restrita apenas a uma única loja física, seus requisitos serão simples o suficiente para nem precisarmos de uma documentação abrangente (ou mesmo precisar de qualquer documentação!): ele será desktop, terá apenas um usuário atuando sobre o sistema, sua carga, por ter apenas um usuário, será baixíssima, além dos dados armazenados no sistema, que por maior que seja a loja, não chegará a limites intratáveis por um sistema simples. Podemos observar que um sistema com esses requisitos pode ser desenvolvido e mantido até por um programador menos experiente.

Em casos assim, realmente, os custos de planejar, documentar e manter a arquitetura seriam maiores que os benefícios proporcionados por ela.

No entanto, quando os sistemas crescem, pensar em arquitetura – nos atributos de qualidade e nas múltiplas visões e interessados envolvidos –, e documentá-la se tornam necessários. Observaremos essa necessidade nos dois exemplos seguintes: apesar de serem exemplos de sistemas funcionalmente semelhantes ao do exemplo anterior, eles têm requisitos não-

funcionais que impõem a necessidade de uma arquitetura bem pensada e documentada.

Exemplo 4.40

O sistema de locadora agora tem que servir para mais duas filiais. Assim, o sistema deve estar rodando nas três lojas e deve existir um cadastro único de novos filmes, novos DVDs e novos clientes, e tanto a locação quanto a devolução podem ser feitas em qualquer loja da rede de locadoras. O sistema se torna multiusuário, por agora mais de um balconista usá-lo ao mesmo tempo, e distribuído, por ter que manter seu estado consistente entre as diversas lojas físicas existentes. Surgem agora preocupações de desempenho, tolerância a falhas e backup e consistência de dados. Outras dúvidas também surgem: Será um banco de dados central para as três lojas? Será um banco distribuído? Se for central, o que fazer caso não seja possível se comunicar com ele? Se for distribuído, como manter a consistência entre os dados? Um balconista de uma loja pode acessar o sistema de outra loja? O que um balconista de uma loja tem permissão para fazer na instância do sistema executando em outra loja? A reserva de um filme está restrita a uma loja física, ou será válida para todas? E assim por diante.

Assim, podemos perceber que uma simples visão de decomposição de classes deixa de ser o único artefato necessário para entender o sistema. Precisamos agora de um artefato que represente os estados do sistema durante a execução, seja em condições normais de operação (e.g., como funciona o procedimento de reserva de filmes entre as lojas da rede de locadoras) , ou seja quando surgem problemas (e.g., o link de comunicação entre as lojas caiu), apenas para exemplificar algumas

poucas preocupações.

Podemos notar que todas essas perguntas afetarão como o sistema estará organizado internamente, mas não afetarão suas funcionalidades, que continuarão sendo as do exemplo anterior. Inferimos também que a arquitetura desse sistema e sua documentação serão mais complexas que a do Exemplo 4.39.

No entanto, no caso do SASF, percebemos que a arquitetura pode se complicar ainda mais, mesmo considerando quase as mesmas funcionalidades. Uma arquitetura ainda mais complexa necessita de uma documentação ainda mais completa para ajudar no desenvolvimento e manutenção desse sistema de software.

Exemplo 4.41

A organização interna do SASF mudará ainda mais em relação aos Exemplo 4.39 e Exemplo 4.40. As decisões que antes permitiam que o sistema rodasse para as três lojas numa mesma cidade não serão mais válidas quando falamos de diversos pontos de distribuição espalhados pelo país.

Dessa maneira, observamos que as decisões de desempenho, disponibilidade dos dados, e políticas de acesso mudam e, como aumentam também em quantidade, se torna mais evidente a necessidade do registro dessas decisões em algum tipo de documento para consulta, resolução de discussões e verificação de conformidade.

Adicionalmente, num sistema como o SASF, o número de interessados aumenta: desde o usuário que deve entender quais tipos de locação e reserva estão disponíveis, passando pelos responsáveis pelo suporte ao usuário, os responsáveis pela disponibilidade dos diversos subsistemas (aluguel, *streaming*, dados, *backup*, etc.), gerente de marketing, time de desenvolvimento,

gerente de projeto, gerente da empresa. Aumentando assim a responsabilidade de se obter um sistema capaz de satisfazer a todos eles.

Cada um terá um conjunto diferente de preocupações sobre o sistema. Seja o responsável por manter o sistema no ar, que precisa saber quantos recursos estão sendo consumidos a cada momento; seja o time de implementação, que precisa descobrir como adicionar uma nova funcionalidade sem quebrar as anteriores; seja o gerente do projeto, que deve decidir por contratar mais desenvolvedores para implementação ou comprar soluções prontas.

Cada um desses estará preocupado também com qualidades diferentes do sistema: o responsável pela disponibilidade do sistema quer saber como o sistema escala se a base de usuários duplicar; já o time de implementação está preocupado em deixar o sistema mais testável para que a implementação da nova funcionalidade seja mais fácil; e, por outro lado, o gerente quer saber o desenvolvimento do sistema é possível com um time de desenvolvedores menor que o atual.

Essas preocupações serão endereçadas pelo documento de arquitetura do SASF, que contém diversas visões direcionadas às diversas preocupações dos interessados. Uma visão de implementação interessará ao responsável pela disponibilidade, assim como uma visão de decomposição interessará ao time de desenvolvimento, assim como uma visão de implementação interessará ao gerente do projeto, fazendo então que o documento de arquitetura possua diversas visões e se torne um documento complexo.

O mais importante a se observar nesse exemplo (e no estudo

do SASF) é que o design e a documentação da arquitetura não são atividades fáceis nem baratas. O arquiteto escolhido para resolver esse problema deve (1) conhecer os interessados, (2) conhecer os atributos de qualidade impostos ao sistema por esses interessados, (3) conhecer as relações e trade-offs entre interessados e atributos de qualidade, (4) conhecer técnicas, padrões e ferramentas que permitam o atendimento aos atributos, e (5) documentar a solução do problema, de forma que os interessados entendam e tirem proveito do documento gerado.

4.11 Resumo

O objetivo deste livro é fazer com que o leitor seja capaz de endereçar todos os aspectos da arquitetura citados anteriormente, podendo realizar algumas das diversas funções realizadas por um arquiteto de software. Dessa maneira, o objetivo deste capítulo foi dar uma visão geral do conhecimento necessário para tanto, fundamentando-o com alguns exemplos e definições. Assim, esperamos que o leitor, a partir de agora:

- entenda e exemplifique os principais conceitos relacionados à arquitetura de software; e
- entenda e exemplifique as principais características e benefícios proporcionados pela arquitetura de software no processo de desenvolvimento.

Já no próximo capítulo, conheceremos os principais interessados que devem ser contemplados pela arquitetura, além de suas características e relações. No capítulo seguinte, entenderemos melhor os atributos de qualidade impostos por esses interessados, além de apresentarmos algumas técnicas para atender esses atributos. Em seguida, teremos um capítulo focado em padrões arquiteturais, uma vez que o uso de padrões no design da arquitetura é uma técnica essencial ao arquiteto. Por fim,

no último capítulo, aprenderemos a documentar a solução que atenderá aos interessados e atributos do sistema.

4.12 Referências

4.12.1 Histórico da área

Apesar da ênfase em Arquitetura de Software como disciplina ter acontecido apenas durante a década de 1990 com autores a exemplo de Perry e Wolf [84] e Garlan e Shaw [46], podemos encontrar trabalhos das décadas de 1960 e 1970 que já citam algumas técnicas e benefícios da área. Entre eles, encontramos Dijkstra [36], Parnas [82] e outros. Mais informações sobre o histórico da disciplina podem ser vistas em *The Past, Present, and Future for Software Architecture*, de Kruchten, Obbink e Stafford [60].

4.12.2 Evolução de software

A evolução de Software é bem estudada no livro editado por Mens e Demeyer, *Software Evolution* [2] e nos trabalhos de Parnas [83], van Gurp e Bosch [107] e Eick *et al* [40]. Mais informações sobre a *Big Ball of Mud* podem ser encontradas em Foote e Yoder [42].

4.12.3 Elementos de uma arquitetura

A divisão dos elementos arquiteturais em estáticos e dinâmicos é feita originalmente por Rozanski e Woods em *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* [88]. Já a discussão sobre classificação dos atributos de qualidade pode ser encontrada no livro *Software Engineering*, de Sommerville [99]. Por fim, podemos citar algumas referências importantes sobre visões arquite-

turais: *The 4+1 View Model of Architecture* de Kruchten [61], *Documenting Software Architectures: Views and Beyond Clements* de Clements *et al* [30] e o padrão ISO/IEEE 1471-2000 [53].

Chapter 5

Stakeholders[1]

O ciclo de vida do software é composto por diversas responsabilidades atribuídas a pessoas, grupos e entidades a quem chamamos de *stakeholders* ou interessados. Entre essas responsabilidades, podemos citar o financiamento, o projeto, o desenvolvimento, o teste, o uso e a manutenção do software. A arquitetura, por sua vez, tem como objetivos tanto facilitar o cumprimento das responsabilidades dos stakeholders, quanto atender às suas necessidades. Entre as necessidades, citamos a urgência por desempenho, diversos aspectos de segurança e usabilidade. Por sua vez, o cumprimento desses objetivos tem impacto direto nos atributos de qualidade exibidos pelo software. Logo, os stakeholders têm forte influência sobre a arquitetura do software e também sobre os atributos de qualidade que este exibirá ao longo de seu ciclo de vida e é por isso que dedicamos um capítulo a eles.

Este capítulo tem como objetivo fazer com que o leitor seja capaz de:

- Entender o conceito de stakeholders da arquitetura de

[1] This content is available online at <http://cnx.org/content/m26195/1.3/>.

um software
- Identificar alguns stakeholders e sua influência em uma arquitetura
- Relacionar stakeholders com os atributos de qualidade impostos a um software
- Entender que stakeholders também se relacionam entre si, podendo, inclusive, ser conflitantes

5.1 Quem são os interessados em um sistema de software?

É comum termos como principais interessados no ciclo de vida de um software os seus usuários e desenvolvedores. Acontece que eles não são os únicos envolvidos ou, ao menos, são grupos homogêneos em termos de interesses e necessidades. Entretanto, para termos um ponto de partida, vamos considerar um cenário em que existem apenas esses dois grupos e algumas simplificações. Nesse cenário, eles ambos os grupos são homogêneos, ou seja, todos os usuários e desenvolvedores apresentam os mesmos interesses e necessidades, e os usuários se encarregam de impor as necessidades, enquanto os desenvolvedores cuidam para que essas necessidades sejam alcançadas através do produto de software. Para montarmos esse cenário, vamos partir de um sistema parecido com o nosso estudo de caso e, pouco a pouco, retirar interesses e necessidades dos envolvidos para observar suas influências no software e em sua arquitetura. Esse processo é ilustrado através do Exemplo 5.1.

Exemplo 5.1
Vamos considerar uma simplificação do SASF que chamaremos SSF (Sistema de Streaming de Filmes). Ele é mais simples porque realiza apenas uma das duas principais funcionalidades do SASF: transmitir

de filmes. Por sua semelhança, consideramos que ele possui um conjunto de interessados parecido com o do SASF. Entretanto, para compormos um cenário sem conflitos, vamos começar descartando as distribuidoras de filmes desse conjunto. Com isso, passamos a responsabilidade de disponibilizar filmes aos usuários que inicialmente usam o software apenas para assisti-los. Contudo, as distribuidoras não são consideradas interessadas apenas por disponibilizarem filmes. Elas têm também a preocupação com que o software respeite os direitos autorais desses filmes. Para tanto, o SASF e o SSF são obrigados a só permitir a transmissão de filmes a pessoas autorizadas e impedir a redistribuição de vídeos por parte dos usuários. Essas obrigações têm efeito na arquitetura de ambos os produtos, que tem que prover não só meios de autenticar e autorizar usuários, para distinguir usuários que assistem dos usuários que distribuem filmes, como também prover meios de impedir ou dificultar a redistribuição do conteúdo transmitido.

A autenticação e autorização são feitas por um módulo responsável pelo cadastro e autenticação de usuários e criação de sessões de uso. Esse módulo provê opções para se cadastrar como distribuidor ou consumidor de filmes. Para o cadastro, o usuário deve prover informações para contato qualquer que seja seu papel. Porém, enquanto a conta para um consumidor é criada assim que o número de seu cartão de crédito seja verificado junto a operadora, o mesmo não acontece para a conta do distribuidor. Para o cadastro de um consumidor ser efetivado, é necessária uma verificação não-automática de sua autenticidade. Essa verificação é iniciada a partir de uma notificação por e-mail, que indica o distribuidor recém-cadastrado e que é envi-

ado às pessoas do departamento responsável pela verificação de usuários.

A proteção contra redistribuição do conteúdo transmitido, por sua vez, é feita por meio da Gestão de Direitos Digitais (GDD) [2]. Por isso, a arquitetura não só define o servidor de stream, mas também o aplicativo cliente e reprodutor de filmes que é o único capaz de decodificar o vídeo.

Por outro lado, ao descartarmos as distribuidoras de filmes de seu grupo de interessados, o SSF fica livre das restrições impostas por elas e passar a não necessitar de uma arquitetura que permita autenticação e autorização para distribuição de filmes, nem proteção do conteúdo distribuído. Por isso, sua arquitetura pode ser simplificada. Uma forma de simplificar é não mais usar a GDD. Dessa maneira, fica decidido que a transmissão será feita usando qualquer formato de vídeo amplamente adotado por reprodutores de mídia. Essa decisão exclui até o que antes era a necessidade: implementar um reprodutor de filmes próprio, mas também melhora a usabilidade, uma vez que agora o usuário está livre para assistir a filmes com o reprodutor que desejar.

A desconsideração de apenas um grupo de interessados causou mudanças profundas tanto nos atributos de segurança, quanto nos de usabilidade do sistema e, como consequência, causou mudanças também na arquitetura. Se continuarmos a simplificação de nosso cenário e desconsiderarmos o cliente do software, poderemos então descartar a necessidade de um baixo custo de desenvolvimento e operação. Assim, para alcançarmos

[2] *Digital Rights Management* (DRM)

desempenho esperado pelos consumidores de filmes, a arquitetura do SSF poderia adotar uma técnica simples, porém cara: para servir mais rápido, basta apenas dispor de mais recursos computacionais, por exemplo, processadores, HDs, memória e conexões maiores, mais rápidos e em maior número. Com essa decisão de aumentar os recursos não se importando com o preço, o SSF poderá não só servir os usuários mais rápido, como também servir a mais usuários. [3]Essa abordagem de apenas melhorar o hardware para servir a uma maior demanda é o que no próximo capítulo chamamos de escalabilidade vertical. A escalabilidade vertical costuma ser bem cara e ter um limite menor de crescimento em relação à sua alternativa, que é a escalabilidade horizontal. Nesse segundo tipo de escalabilidade, a organização do software e como ele se comunica realiza um papel essencial para atender à grande demanda de usuários, mesmo quando executando em hardware de menor capacidade. Em outras palavras, há um melhor aproveitamento dos recursos disponíveis, algo que só pode ser alcançado por meio de uma arquitetura bem pensada.

É importante lembrar que dentro de um mesmo grupo de interessados podem existir interesses conflitantes entre si. Afinal, um grupo pode se organizar em subgrupos de interesses comuns, mas um subgrupo pode demonstrar interesses conflitantes com outro subgrupo. Portanto, subgrupos diferentes de usuários ou de desenvolvedores resultam em requisitos diferentes, que significam atributos de qualidade diferentes e que são frutos de arquiteturas diferentes. Podemos observar isso

[3]Desempenho é um atributo comumente esperado pelos usuários, que nunca querem esperar pelo serviço. Já escalabilidade não é um atributo requerido explicitamente por eles, mas se torna necessária quando o número de usuários aumenta e não se aceita que o desempenho degrade.

no estudo de caso (também citado no Exemplo 5.1), quando o grupo de usuários se organiza em dois subgrupos: os que se cadastram no sistema para alugar filmes e as distribuidoras de filmes. O resultado dessa divisão e o conflito podem também ser observados no exemplo (e na Figura 5.1). Por um lado, as distribuidoras impõem seus requisitos de proteção aos direitos autorais. Por outro, os usuários têm a forma de interação com o sistema modificada, uma vez que devem usar um reprodutor de filmes específico para que os requisitos das distribuidoras sejam alcançados. Em resumo, mesmo fazendo parte de um mesmo grupo de envolvidos, a influência de cada subgrupo não pode ser desconsiderada, uma vez que ela pode ser grande o bastante para modificar, inclusive, a forma de outros subgrupos interagirem com o sistema.

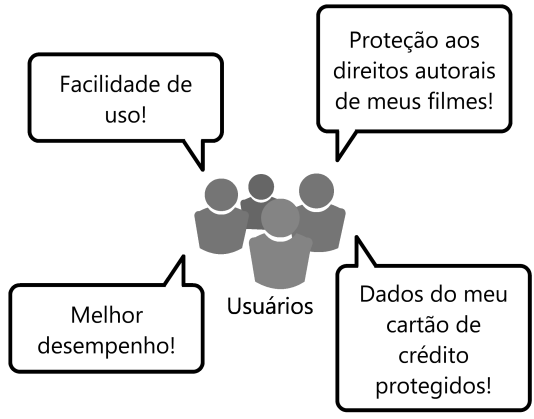

Figura 5.1: Stakeholders de um mesmo grupo, mas divergindo nos requisitos.

5.1.1 Importância dos interessados

Podemos observar por meio do Exemplo 5.1 que a presença ou ausência de um interessado tem grande influência na arquitetura. Além disso, é comum que sua ausência dê espaço para simplificações nas decisões arquiteturais. [4] Entretanto, no mundo real, os envolvidos não se limitam a usuários e desenvolvedores. Há diversos outros tipos de envolvidos que influenciam o desenvolvimento do software de diversas maneiras diferentes. Esses envolvidos que influenciam o ciclo de vida do software também são chamados *stakeholders*. Devido ao conceito de stakeholder ser bastante amplo e transcender a Engenharia de Software, preocupamo-nos apenas com aqueles que impactam a arquitetura e, por isso, usamos a definição dada por Rozanski e Woods:

> **Definição 5.1: stakeholder**
> "Um stakeholder em uma arquitetura de software é uma pessoa, grupo ou entidade com um interesse ou preocupações sobre a realização da arquitetura." [5]

Alguns stakeholders têm diferentes responsabilidades durante o ciclo de vida do software. Entre as responsabilidades, podemos citar financiamento, projeto, desenvolvimento, teste, uso, manutenção e até passagem de conhecimento sobre ele. Outros stakeholders, por sua vez, esperam que o software funcione de alguma forma específica: eles têm necessidades em relação ao software. Por exemplo, é comum para um usuário esperar que o resultado alcançado pelo software seja confiável ou que seja alcançado em um tempo hábil. Quando estamos no espaço do problema, costumamos chamar essas responsabilidades e necessidades de requisitos do software. Por outro lado, quando esta-

[4] Note que uma arquitetura mais simples não necessariamente significa um produto com desenvolvimento mais barato ou execução mais rápida.

[5] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley Professional 2005.

mos no espaço da solução, costumamos chamá-las de *atributos de qualidade*. Logo, os stakeholders têm forte influência sobre a arquitetura de um software porque ela é uma ferramenta essencial para proporcionar seus atributos de qualidade e atender aos requisitos, como, por exemplo: custo, reusabilidade, testabilidade, manutenibilidade, legibilidade, desempenho, escalabilidade, segurança, confiabilidade, entre outros.

5.2 Tipos de stakeholders e sua relação com os atributos de qualidade

Entre os diversos tipos de stakeholders que influenciam a arquitetura, podemos citar os usuários, os desenvolvedores, os gerentes, os testadores, os clientes (que podem ou não ser usuários), os designers de outros sistemas e os mantenedores, além dos analistas e o próprio arquiteto do sistema. Considerando que esse é um conjunto heterogêneo de papéis, é natural que cada papel possua diferentes necessidades e responsabilidades que têm efeito sobre a arquitetura e que, eventualmente, resultem em conflitos.

Resolver conflitos de interesses entre stakeholders está entre as obrigações de um arquiteto de software. Ele deve ser consciente de que muitas vezes não será possível agradar perfeitamente a todos os interessados, uma vez que esses conflitos podem impedir o projeto de uma solução ótima. Portanto, sua obrigação será a de produzir uma arquitetura boa o suficiente e que faça todos os stakeholders ficarem satisfeitos. Por isso, é importante que cada envolvido seja informado de como a solução de seu interesse foi restringida pelos interesses de outros envolvidos.

A seguir, podemos observar duas situações de divergências entre stakeholders que resultam em conflitos entre os atributos de qualidade.

Exemplo 5.2

As distribuidoras esperam que os direitos autorais de seus filmes sejam protegidos, já os usuários querem apenas assistir a seus filmes sem dificuldades ou interrupções. A forma encontrada para proteger os direitos autorais foi por meio da Gestão de Direitos Digitais. Essa decisão implica em restringir o reprodutor de mídia que pode ser usado e obrigar o usuário a se autenticar no sistema para assistir a algum filme. Tanto a restrição do reprodutor de mídia, quanto a autenticação do usuário dificultam a tarefa de assistir a um filme, uma vez que o usuário pode não se lembrar de seu login ou senha ou ele pode não estar acostumado com o reprodutor de filmes permitido. Por isso, essa decisão de segurança tem impacto negativo na usabilidade. Portanto, podemos observar aqui um conflito entre segurança e usabilidade.

Exemplo 5.3

Ainda no SASF e também pela decisão de proteger os direitos autorais usando GDD, o arquivo contendo o filme é transmitido encriptado para o cliente. Essa encriptação é uma forma de dificultar a reprodução do vídeo em programas não autorizados. No entanto, o reprodutor de vídeo autorizado deve pagar um preço por isso: para decodificar um arquivo com GDD, é necessário mais processamento e, portanto, maior consumo de recursos. Isso ocasiona perda de desempenho, o que pode ser crítico em dispositivos com menos recursos, como celulares. Por isso, a decisão de segurança também tem impacto negativo no desempenho, caracterizando um conflito entre esses dois atributos.

Note que para afirmarmos que uma arquitetura alcançou algum sucesso, os stakeholders devem se mostrar satisfeitos com o sistema desenvolvido a partir dela. Para tanto, espera-se que o

arquiteto seja capaz de projetar uma arquitetura que alcance dois principais objetivos: atendimento de requisitos e resolução de conflitos.

5.2.1 Atendimento aos requisitos como medida de sucesso

O primeiro objetivo, atender aos requisitos dos stakeholders, acaba sendo óbvio, pois para satisfazer os interessados, o sistema deve fazer o que eles esperam dele. Mas apesar de óbvio, enfatizar esse objetivo serve para o arquiteto iniciante perceber que seu objetivo principal é projetar uma arquitetura com atributos de qualidade capazes de atender aos requisitos do sistema *impostos e esperados pelos stakeholders* e não só por ele próprio. No exemplo a seguir, mostramos um caso quando isso não acontece.

Exemplo 5.4

Em alguns celulares e outros aparelhos que executam software embarcado, espera-se que esse software tenha um bom desempenho, principalmente considerando a escassez de recursos do ambiente de execução. Afinal, o usuário não quer pressionar uma tecla e esperar vários segundos pela resposta. Por outro lado, não se espera que o software seja extensível, uma vez que alguns desses aparelhos nem ao menos permitem atualizações de software. Considerando que, nesse caso, desempenho e economia de recursos são requisitos mais críticos que extensibilidade, de nada adianta o arquiteto do software para aparelhos que não permitem atualizações projete uma arquitetura que torne o software extensível, com diversos níveis de abstração, quando esses níveis impactam negativamente no desempenho.

Pode parecer ingenuidade tomar decisões em favor da extensibilidade quando se espera desempenho, como ilustrado no

Exemplo 5.4. No entanto, esse erro é muito comum e não é só cometido por arquitetos inexperientes. Muitos arquitetos não consideram o real impacto de suas decisões e se deixam levar por modas de padrões, *frameworks* [6] ou abordagens que prometem resolver todos seus problemas. Às vezes, é apenas considerado que assim será mais fácil "vender" a arquitetura ao gerente do projeto.

Por fim, poderíamos ainda afirmar a partir do primeiro objetivo: não importa o quanto "de acordo com as boas práticas" a arquitetura de um software está, se ela não atende aos requisitos que esperam que ela atenda. Ela, simplesmente, estaria acertando o alvo errado. [7] Portanto, a medida de atendimento aos requisitos do sistema é a melhor medida de sucesso da arquitetura, desde que se conheçam os requisitos.

5.2.2 Conflitos entre requisitos e atributos de qualidade

Situações de conflito surgem quando requisitos de stakeholders divergem ou afetam atributos de qualidade comuns. Podemos observar que esse tipo de situação está presente, inclusive, em alguns exemplos anteriores (Exemplo 5.2 e Exemplo 5.3). Nesses exemplos são ilustrados conflitos entre atributos de segurança e usabilidade e entre segurança e desempenho. A seguir, citamos outros atributos de qualidade e relacionamos a alguns stakeholders que têm requisitos que comumente divergem durante o ciclo de vida do software.

[6] É comum que a adoção de um *framework* seja seguida de decisões arquiteturais impostas por ele e essas decisões podem comprometer ou conflitar com os objetivos traçados pelo arquiteto e esperados pelos stakeholders.

[7] Mas é claro que as boas práticas serão *ferramentas* para resolver os problemas propostos pelos stakeholders.

Exemplo 5.5: Desempenho versus custo
Usuários buscam por maior desempenho, enquanto clientes e gerentes costumam preferir menor custo de desenvolvimento. Esses atributos divergem porque é comum que maior desempenho resulte em uma solução que necessite de mais recursos computacionais ou ainda desenvolvedores mais qualificados na sua construção.

Exemplo 5.6: Desempenho versus escalabilidade
O cliente, que espera ganhar dinheiro a partir da popularização do software, impõe o requisito que ele seja capaz de servir a demanda crescente de usuários. Já os usuários continuam buscando por desempenho do software, não se importando se há dez, mil ou um milhão de usuários usando-o ao mesmo tempo. Uma forma simples de servir a demanda crescente de usuários, ou escalar, seria não se preocupar com o tempo de resposta do serviço para cada usuário e aumentá-lo drasticamente. No entanto, o aumento do tempo de resposta é um indício de perda de desempenho, caracterizando o conflito.

Exemplo 5.7: Usabilidade versus segurança
Em um último exemplo, citamos o conflito entre usabilidade e segurança. Usuários esperam realizar suas tarefas rapidamente, sem dúvidas e sem erros causados pela dificuldade de usar, ou seja, esperam usabilidade do software. Por outro lado, auditores, clientes e os próprios usuários esperam que suas informações estejam a salvo, tanto para casos de ataques, quanto para manipulação indevida. Medidas de segurança devem ser projetadas e o software deve prover meios de autenticação, autorização, confidencialidade e auditabilidade. Ao tomar essas medidas, a usabilidade é afetada negativamente, uma vez que mais passos serão

necessários para se realizar as mesmas ações. Por exemplo, para começar a usar o software, agora será necessário inserir uma senha para que o usuário seja autenticado. Portanto, a adoção de políticas de segurança costuma afetar negativamente a usabilidade do sistema.

5.2.3 Responsabilidades dos stakeholders

Como já foi mencionado anteriormente, stakeholders têm responsabilidades durante o ciclo de vida do software. A seguir, agrupamos as responsabilidades em quatro grandes tipos e citamos seu principais interessados:

- Uso ou aquisição do sistema, que são responsabilidades de usuários e clientes;
- Desenvolvimento, descrição e documentação da arquitetura do sistema, que são responsabilidades do arquiteto do sistema;
- Desenvolvimento e manutenção do sistema, que são responsabilidades que envolvem o maior número de stakeholders: arquitetos, projetistas, programadores, mantenedores, testadores, engenheiros de domínio, gerentes de projetos e desenvolvedores, entre outros;
- Avaliação do sistema e do seu desenvolvimento, que são responsabilidades de CIOs [8], auditores e avaliadores independentes.

Por fim, descrevemos alguns dos stakeholders citados e qual sua influência da arquitetura e em sua documentação. Para tanto, mencionamos quais são seus interesses comuns e o que eles esperam da documentação da arquitetura.

[8] Chief Information Officer ou CIO é o nome dado ao diretor do departamento de Tecnologia da Informação de uma empresa.

5.2.3.1 Usuários

A principal preocupação dos usuários é com as funcionalidades providas pelo sistema, pouco importando como o software foi dividido em módulos ou como esses módulos se comunicam entre si. Podemos afirmar que um usuário só pensa em um atributo de qualidade, por exemplo, em desempenho ou em segurança, quando algum desses lhe faltar.

Essa despreocupação com a organização interna do software poderia nos fazer afirmar ingenuamente que a arquitetura não interessa ao usuário. No entanto, ela interessa, ainda que indiretamente, uma vez que o sistema deve possuir uma arquitetura que proporcione os atributos de qualidade esperados pelos usuários para que funcione de forma satisfatória.

Já em relação à documentação, os usuários estão interessados em saber as capacidades e o comportamento do sistema. Vale notar que essa informação pode estar em outros documentos, como em um manual do usuário, mas esse e outros documentos devem ser escritos tendo por base o documento de arquitetura, que deve conter essas informações.

5.2.3.2 Clientes

Da mesma forma que os usuários, os clientes não costumam se preocupar em detalhes técnicos da arquitetura. Eles estão interessados nas características da arquitetura ligadas ao seu negócio: se o sistema faz o que deveria fazer, seus custos, sejam de desenvolvimento ou de execução, e o planejamento de seu desenvolvimento. Isso se faz necessário para justificar o dinheiro investido no software.

Clientes também se mostram interessados na justificativa de resolução dos eventuais conflitos, principalmente se essa resolução tem impacto no negócio.

5.2.3.3 Arquiteto

Uma vez que é o principal responsável por projetar a arquitetura, o arquiteto tem a obrigação de conhecer os stakeholders envolvidos no sistema. Isso permitirá que ele saiba o que os stakeholders esperam do sistema e, por fim, seja capaz de projetar o sistema de acordo com os requisitos esperados. O arquiteto também é responsável por negociar os conflitos de interesses entre os stakeholders, o que resultará numa arquitetura com atributos de qualidade que agradem a vários, mesmo que parcialmente.

A necessidade de conhecer e dialogar com os diversos stakeholders faz com que o arquiteto precise de habilidades tanto sociais quanto técnicas. Em relação ao conhecimento técnico, ser experiente no domínio do problema o ajudará a identificar previamente as dificuldades e soluções a serem encontradas ao longo do desenvolvimento. Já as habilidades sociais o ajudam tanto na descoberta de requisitos, quanto na negociação de divergências.

5.2.3.4 Desenvolvedor

O desenvolvedor vê a arquitetura como base para construir o sistema. Há dois extremos de como a arquitetura pode ser apresentada para ele. Ela pode ser apresentada como uma especificação, onde não há qualquer liberdade de design durante o desenvolvimento. Ou ela pode ser apresentada como um guia, que apresenta algumas restrições essenciais para que o software alcance o sucesso, mas também possui diversas liberdades para as decisões de implementação e design de baixo-nível que ficam a cargo do time de desenvolvimento. Ao longo de todo o espectro, o desenvolvedor espera pela ideia geral do sistema, onde as funcionalidades serão implementadas, quem serão os responsáveis por elas e quais as decisões de design de alto-nível relacionadas a elas.

Um desenvolvedor comumente espera que a arquitetura também seja viável e de acordo com suas habilidades, além de que possua as decisões de design escritas de forma clara e objetiva. Ele também espera que o documento de arquitetura possibilite a associação dos requisitos do sistema às partes que o compõem. Essa associação é o que chamamos de rastreabilidade, que torna mais fácil tanto a manutenção quanto o entendimento do sistema.

5.2.3.5 Testador

O testador procura no documento de arquitetura as restrições as quais o software deve obedecer. Além disso, ele espera que o software seja *testável* e, para tanto, sua arquitetura deve proporcionar tal atributo de qualidade.

O nível de testabilidade de um software está diretamente ligado a capacidade dele (ou de suas partes) ser posto em execução em ambiente de desenvolvimento e de seu comportamento, interno ou externo, ser verificável a partir do esperado.

5.2.3.6 Gerente de projeto

O gerente de projeto, assim como o cliente, está interessado em custos e planejamento. No entanto, ele também se preocupa com detalhes técnicos da arquitetura e como ela ajudará no desenvolvimento do software. A arquitetura o ajudará a resolver problemas do tipo: como dividir o time de desenvolvimento a fim de paralelizar a construção dos módulos, quais partes do software podem ter o código reusado, terceirizado ou comprado, ou ainda como as funcionalidades serão divididas entre os múltiplos *releases* do software.

5.3 Resumo

De maneira alguma esgotamos o assunto sobre stakeholders. Por outro lado, não devemos nos aprofundar ainda mais para não perdermos nosso foco que é o design da arquitetura. Entretanto, acreditamos alcançar os objetivos deste capítulo, mesmo com uma abordagem superficial sobre o assunto. Assim, esperamos que o leitor, a partir de agora:

- entenda e exemplifique o conceito de stakeholders da arquitetura de um software;
- entenda a influência desses stakeholders;
- relacione os stakeholders aos atributos de qualidade esperados pelo software; e
- entenda que os stakeholders se relacionam entre si, podendo, inclusive, gerar demandas conflitantes.

Nos capítulos a seguir, voltamos nosso foco aos atributos de qualidade e às técnicas de como se projetar uma arquitetura que atenda a esses atributos. Ainda assim, não podemos esquecer que nossos objetivos como arquitetos são descritos explicita ou implicitamente pelos stakeholders e por sua influência sobre arquitetura do software.

5.4 Referências

5.4.1 Definição e papel dos stakeholders

O padrão ISO/IEEE 1471-2000 [54], além de definir stakeholders, modela seu papel em relação aos vários elementos relacionados à arquitetura de software. É nesse padrão que Rozanski e Woods se baseiam para para chegar à definição de stakeholders no livro *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* [89], onde dedicam um capítulo inteiro sobre o assunto. Outras duas

importantes referências sobre o papel dos stakeholders ao longo da vida da arquitetura são dois livros publicados pelo Software Engineering Institute, da Carnegie Mellon University: *Software Architecture in Practice* de Bass *et al* [14] e *Documenting Software Architecture: Views and Beyond* de Clements *et al* [31]. Ambos mostram a importância de considerar os stakeholders, sendo o segundo mais completo pois também trata das escolhas que o arquiteto deve fazer ao criar as visões da arquitetura de acordo com os interessados.

5.4.2 Arquiteto como stakeholder

Taylor *et al*, no livro *Software Architecture: Foundations, Theory, and Practice* [102], dedicam todo um capítulo sobre a responsabilidade do arquiteto como stakeholder, incluindo os diversos papéis que ele pode assumir durante o ciclo de vida do software.

Outros artigos importantes sobre o papel do arquiteto de software que podemos citar são o *The Software Architect – and The Software Architecture Team* de Kruchten [59], o *Who Needs an Architect?* de Fowler [43] e o *The Software Architect: Essence, Intuition, and Guiding Principles* de McBride [71].

5.4.3 Responsabilidades dos stakeholders

Booch discute sobre a "despreocupação" dos usuários em relação à arquitetura em *The Irrelevance of Architecture* [18]. Já em *Goodness of Fit* [17], ele escreve sobre o que seria uma arquitetura de sucesso.

Por fim, Hohmann, no livro *Beyond Software Architecture* [51] trata das diversas preocupações ligadas aos atributos de qualidade do software. Preocupações que não são apenas do ar-

quiteto, mas também são dos diversos envolvidos no desenvolvimento do software.

5.5 Exercícios

Exercício 5.1

Qual a importância da identificação dos stakeholders na arquitetura de um sistema de software?

Exercício 5.2

A identificação de muitos stakeholders em uma arquitetura aumenta a chance de sucesso. No entanto, os interesses dos stakeholders muitas vezes não são claros e podem ser conflitantes e/ou contraditórios. Cite mais exemplos desses conflitos/contradições.

Exercício 5.3

É impossível capturar as características funcionais e as propriedades de qualidade de um sistema complexo com um simples modelo. De que forma poder-se-ia representar arquiteturalmente sistemas complexos de forma que seja gerenciável e compreensível por uma faixa de stakeholders técnicos e de negócio?

Chapter 6

Atributos de Qualidade[1]

Um software tem como objetivo atender aos seus requisitos funcionais e não-funcionais. Os requisitos funcionais descrevem as funções que o software deve ser capaz de realizar, ou seja, *o que* o sistema faz. Já os requisitos não-funcionais descrevem as qualidades e restrições de como o sistema realiza suas funções, ou seja, *como* o sistema funciona. Um software, portanto, deve exibir atributos de qualidade que atendam aos seus requisitos.

Por sua vez, a arquitetura de software contém a descrição de como esse alcança aos atributos de qualidade. Essa descrição de como o software atende aos requisitos não-funcionais é feita pelas diversas decisões presentes na arquitetura. Para conceber essas decisões arquiteturais – e, portanto, para projetar a arquitetura – é de fundamental importância que o arquiteto conheça tanto os objetivos a serem alcançados pelo software, quanto as ferramentas para alcançá-los. Em outra palavras, é essencial que ele conheça tanto os atributos de qualidade, quanto técnicas e padrões de design arquitetural que, ao serem implementados, possibilitam ao software que exiba os atributos

[1] This content is available online at <http://cnx.org/content/m17527/1.5/>.

de qualidade desejados.

Considerando a importância dos atributos de qualidade de software, dedicamos dois capítulos a eles. Neste capítulo, mostramos uma visão geral do assunto, abordando diversos atributos que devem ser alcançados. Este capítulo tem como objetivos:

- Identificar o que são atributos de qualidade e qual é sua influência na arquitetura de software;
- Relacionar atributos de qualidade a decisões arquiteturais que os proporcionam;
- Entender que os atributos de qualidade se relacionam e como eles se relacionam.

No capítulo seguinte, apresentamos técnicas de design arquitetural e uma série de estudos de como alguns atributos foram alcançados na prática em diferentes sistemas de software. Esses estudos mostram que técnicas e padrões de design arquitetural foram aplicados para alcançar tais atributos e quais seus benefícios e limitações apresentados.

6.1 Requisitos Funcionais e Não-Funcionais

O único objetivo de um software é o de atender a seus requisitos. Esses requisitos são definidos ao longo de seu ciclo desenvolvimento e costumam ser classificados em *requisitos funcionais* e *requisitos não-funcionais*.

Os requisitos funcionais descrevem as funções que o sistema é capaz de realizar, ou seja, descrevem *o que* o sistema faz.

Definição 6.1: requisito funcional
É a declaração de uma função ou comportamento providos pelo sistema sob condições específicas.

Os requisitos do software são impostos pelos seus diversos *stakeholders*. No entanto, os requisitos funcionais costumam ser ditados pelos clientes do software, afinal são eles que esperam ter seus problemas resolvidos pelas funcionalidades do software.

Exemplo 6.1
Se estamos falando do SASF, entre suas funções, podemos citar:

(RF-01): O usuário deve ser capaz de inserir um filme da sua lista de aluguéis;
(RF-03): O usuário deve ser capaz de assistir a um filme via *streaming*;
(RF-06): O usuário deve ser capaz de adicionar um comentário sobre um filme.

Se o problema de desenvolver software fosse apenas o de atender aos requisitos funcionais, desenvolver software já poderia ser considerado uma tarefa difícil. Isso porque, para serem atendidos, muitos dos requisitos funcionais necessitam de conhecimento que ultrapassa os limites da Engenharia de Software, da Ciência da Computação ou mesmo da Matemática. Afinal, para se implementar sistemas para *Computer-Aided Design* (CAD) ou sistemas que analisam os dados extraídos do *Large Hadron Collider* (LHC) [2] é preciso grande conhecimento específico ao domínio do problema, ou seja, grande conhecimento de outras engenharias (*por ex.*, Engenharia Mecânica e Civil) ou de outras ciências (*por ex.*, Física e Química), respectivamente.

Além da necessidade de conhecimento específico ao domínio do problema, há outra dificuldade no desenvolvimento de software

[2] http://public.web.cern.ch/public/en/LHC/LHC-en.html (<http://public.web.cern.ch/public/en/LHC/LHC-en.html>)

para atender apenas aos requisitos funcionais: o cliente pode não ter certeza sobre o que ele quer do software. Esta condição é bem conhecida pela Engenharia de Requisitos, que nos provê algumas técnicas para resolvê-la ou contorná-la. Mas isso não quer dizer que não possa se tornar um problema durante o ciclo desenvolvimento. Afinal, se o principal interessado não sabe bem quais funções se espera que o sistema realize, não podemos afirmar que será fácil desenvolver esse sistema.

Por outro lado, há também os requisitos não-funcionais. Esses estão relacionados à qualidade da realização dos requisitos funcionais, ou seja, *como* essas funções são realizadas.

Definição 6.2: requisito não-funcional
É a descrição de propriedades, características ou restrições que o software apresenta exibidas por suas funcionalidades.

Esses requisitos também são impostos pelos diversos *stakeholders* do software e estão normalmente relacionados a interfaces com o usuário, capacidades, consumo de recursos e escalas de tempo.

Exemplo 6.2
Podemos citar alguns exemplos de requisitos não-funcionais do SASF:

(RNF-01): O sistema deve permitir o uso por diversas interfaces diferentes: navegador de internet, celular, TV (usando um decodificador de TV por assinatura compatível) e aplicação-cliente compatível com as famílias de sistemas operacionais Windows, Mac OS e Linux;

(RNF-04): O sistema deve suportar até 3 milhões de inserções na fila de aluguéis por dia (34,7 operações por segundo);

(RNF-09): Uma transmissão de vídeo via *streaming* não pode ser iniciada em mais do que 30 segundos.

As restrições feitas pelos requisitos não-funcionais são várias e podem incluir restrições ao processo de desenvolvimento, restrições para atingir ou manter compatibilidade, e restrições legais, econômicas ou de interoperabilidade. As restrições ao processo de desenvolvimento podem ser feitas pela imposição de padrões de desenvolvimento ou mesmo de linguagens a serem utilizadas pelo sistema. Por exemplo, um requisito não-funcional de um sistema pode ser que ele deva ser implementado usando a linguagem Java™, uma vez que a equipe responsável pela operação e manutenção após seu desenvolvimento seja é experiente nessa linguagem. Por fim, podemos ainda citar requisitos não-funcionais conhecidos que foram impostos em prol de compatibilidade e interoperabilidade e – por que não dizer – de questões econômicas, que é um caso relacionado ao sistema operacional Windows NT. O Windows NT possui requisitos não-funcionais que ditam que ele deve ser capaz de executar aplicativos originalmente escritos para DOS, OS/2, versões anteriores do Windows e aplicativos de acordo com o padrão POSIX [3]. Assim, satisfazendo aos requisitos de poder executar aplicativos originalmente escritos para sistemas operacionais anteriores, o Windows NT teria um custo de adoção mais baixo, uma vez as empresas não precisariam renovar seu ecossistema de aplicativos para poder usá-lo. Já o requisito de aderência ao padrão POSIX se mostra necessário para eventuais contratos com cláusulas do tipo: "o sistema operacional a ser utilizado deve estar de acordo com o padrão POSIX".

[3]Note que não tivemos acesso ao documento de requisitos do Windows NT. No entanto, a estrutura de seu kernel deixa bem clara esta necessidade de retrocompatibilidade e aderência ao padrão POSIX. Para mais informações sobre esse assunto, recomendamos a leitura do capítulo *A Tale of Two Standards* do livro *Open Sources 2.0* [35].

Os requisitos não-funcionais podem ainda ser divididos em três tipos: de produto, de processo e externos. Os requisitos não-funcionais de produto podem, à primeira vista, nos parecer os únicos que deveríamos estudar. Isso se dá por eles estarem diretamente relacionados à qualidade do software e serem definidos como os requisitos que especificam as características que o software deve possuir. No entanto, devemos lembrar que a arquitetura de software não influencia apenas a qualidade final do software, mas também influencia (e é influenciada pela) a forma com que ele é desenvolvido e até mesmo a organização em que ela está inserida.

Definição 6.3: requisito não-funcional de produto
Requisito que especifica as características que um sistema ou subsistema deve possuir.

Os requisitos não-funcionais de produto, como já dito anteriormente, são relacionados à qualidade do software e são alcançados pelo que chamamos de atributos de qualidade. Portanto, quando existem requisitos em que o software deve ter algum grau de confiabilidade, certo nível de eficiência, ou ser portável para diversos sistemas operacionais, estamos descrevendo quais atributos de qualidade que o software deve exibir. Todos requisitos presentes no Exemplo 6.2 podem ser classificados como sendo de produto. Ainda retornaremos a esse assunto neste capítulo, mas antes devemos mostrar os outros tipos de requisitos não funcionais.

Os requisitos não-funcionais de processo são definidos como as restrições ao processo de desenvolvimento.

Definição 6.4: requisito não-funcional de processo
Requisito que restringe o processo de desenvolvimento do software.

Esse tipo de requisito é encontrado em muitas situações, principalmente em grandes empresas ou organizações. Por exemplo, é comum que o desenvolvimento de sistemas de software para o Exército Americano tenham como requisito ter o processo de desenvolvimento de acordo com a *Joint Technical Architecture* [4]

Por fim, há os requisitos não-funcionais externos. Esses, muitas vezes, podem se classificar tanto como de produto quanto de processo e são extraídos do ambiente em que o sistema é desenvolvido. Esse ambiente pode ser tanto a organização, com políticas que devem ser seguidas ou seu atual ecossistema de software com o qual ele deve interoperar, quanto a legislação vigente do país em que o sistema está operando.

Definição 6.5: requisito não-funcional externo
Requisito derivado do ambiente em que o sistema é desenvolvido, que pode ser tanto do produto quanto do processo.

Por fim, como exemplo de requisitos externos, podemos citar:

Exemplo 6.3
O sistema de recomendação de livros deve ler as informações do sistema de aluguel de livros de uma biblioteca, onde cada registro de livro está de acordo com o padrão *Dublin Core*. Um requisito não-funcional externo desse sistema de recomendação é:

(RNF-01): O sistema deve guardar os dados dos livros recomendados em um modelo mapeável para o modelo de dados definido pelo padrão Dublin Core [85].

[4] A *Department of Defense Joint Technical Architecture* (DoD JTA) [6] é um documento que descreve um conjunto de normas a que um sistema deve aderir para facilitar a interoperabilidade com outros sistemas do Exército Americano. A título de curiosidade, o DoD JTA contém algumas centenas de normas.

Note que o uso do *Dublin Core* só é realmente necessário porque a comunicação entre os dois sistemas é esperada e que um sistema já adota esse padrão.

6.1.1 Diferenças entre requisitos funcionais e não-funcionais

Apesar da classificação dos requisitos de software em requisitos funcionais e não-funcionais ser bem aceita, devemos observar que na prática essa divisão pode não ser tão clara. Isso ocorre devido ao nível de detalhes contido em sua descrição ou mesmo devido ao tipo de sistema desenvolvido.

Podemos ilustrar o caso em que o nível de detalhes faz a diferença com o seguinte exemplo:

Exemplo 6.4
Se considerarmos um requisito de segurança de confidencialidade (e normalmente considerado não-funcional):

(RNF-01): O sistema deve possibilitar o envio de mensagens de modo que não possam ser lidas a não ser pelos destinatários.

Uma vez que não especifica nenhuma funcionalidade, esse pode ser considerado um requisito não-funcional. Por outro lado, poderíamos deixar essa evidente característica de requisito não-funcional um pouco mais turva se adicionarmos um pouco mais de detalhes a ele:

(RF-01): O sistema deve permitir aos usuários que criptografem suas mensagens usando as chaves públicas dos destinatários.

Agora, esse requisito seria melhor classificado como funcional, uma vez que especifica uma função do sistema, apesar do atributo de qualidade exibido pelo software ao final do desenvolvimento será o mesmo: segurança, mais especificamente confidencialidade das mensagens enviadas.

Já quando mencionamos que o tipo do sistema pode influenciar em como classificamos um requisito, basta apenas lembrarmos dos sistemas de tempo-real. Neles, a corretude do comportamento do sistema não depende só do resultado lógico da função, mas também *quando* esse resultado é obtido. Portanto, uma resposta cedo ou tarde demais pode estar tão incorreta quanto uma resposta logicamente errada.

Exemplo 6.5
Em um sistema de informação, consideramos o requisito não-funcional:

(RNF-01): A busca por nome deve retornar os resultados em no máximo 100 milissegundos.

Já em um sistema de controle de voo *fly-by-wire*, devemos considerar o requisito a seguir como funcional, uma vez que respostas que não respeitam o intervalo de tempo especificado são tão inúteis quanto a falta de resposta dos sensores (podem causar a queda do avião):

(RF-01): Novas amostras de dados dos sensores da aeronave devem ser obtidas a cada 20 milissegundos.

Apesar disso, vale notar que ambos os requisitos presentes no Exemplo 6.5 ditam que tanto o sistema de informação quanto o sistema *fly-by-wire* devem exibir o atributo de qualidade desempenho, mesmo que em graus diferentes.

6.1.2 Conflitos entre requisitos

Como requisitos de software têm impacto em um ou mais atributos de qualidade, pode acontecer de impactarem em atributos relacionados a outros requisitos. Quando isso ocorre, o impacto pode resultar em reforço do atributo ou em conflito.

Podemos perceber que não surgem grandes problemas quando dois ou mais requisitos reforçam o mesmo atributo de qualidade. Afinal, caso isso ocorra, o design da solução que atenda a um dos requisitos afetará apenas positivamente o design da solução que atenda aos outros requisitos.

Apesar do caso de requisitos que se reforçam não ser muito comum, podemos ilustrá-lo com requisitos que afetam à segurança do software, mais precisamente autenticidade e confidencialidade:

Exemplo 6.6
Se temos um sistema de mensagens instantâneas com os seguintes requisitos:

(RNF-01): O sistema deve prover meios de autenticar os seus usuários.
(RNF-02): Uma mensagem enviada a um usuário não pode ser lida a não ser pelo destinatário.

Podemos observar que os requisitos *RNF-01* e *RNF-02* se relacionam, uma vez que afetam a alguns aspectos de segurança do sistema. Eles se reforçam visto que é possível encontrarmos uma solução para *RNF-01* que facilite *RNF-02* e vice-versa. A solução no caso é a utilização criptografia de chave pública: tanto ela pode ser usada para autenticação de usuários quanto pode ser usada para encriptação de mensagens.

Por outro lado, requisitos conflitantes são mais comuns e adicionam dificuldade durante o design das soluções. Isso ocorre

porque a solução para um requisito conflitante afeta negativamente outro requisito. Assim, o design do software terá que considerar diversos *trade-offs* a fim satisfazer melhor aos requisitos mais importantes, já que atender a todos de forma ótima não é possível.

Se adicionamos alguns requisitos de usabilidade ao Exemplo 6.6, esses novos requisitos certamente afetarão negativamente à solução apresentada. Isso ocorre porque é comum que soluções de segurança afetem aos requisitos de usabilidade, visto que essas soluções adicionam conceitos não familiares aos usuários (por exemplo, chaves criptográficas) ou adicionam mais passos para que os usuários realizem suas tarefas (por exemplo, inserir *login* e senha).

6.1.3 Expressando requisitos não-funcionais

Grande parte do trabalho de um arquiteto consiste em projetar sistemas que devem satisfazer requisitos não-funcionais. No entanto, a Engenharia de Requisitos é limitada quanto a métodos de análise e derivação de requisitos não-funcionais. Essa limitação, muitas vezes, obriga ao arquiteto a trabalhar com requisitos que carecem de métricas e valores-alvo. Isso dificulta o processo de design, uma vez que desconhecer requisitos é o mesmo que desconhecer os objetivos do design. Por este motivo, recomenda-se aos arquitetos que sempre busquem por requisitos que possuam valores e métricas bem definidos e, desta maneira, conheçam e possam medir os objetivos e o sucesso de seu design.

Todavia, nem sempre é possível trabalhar com requisitos bem definidos, uma vez que encontramos alguns problemas ao expressá-los. Os principais motivos da dificuldade de expressar requisitos não-funcionais são os seguintes:

- Alguns requisitos simplesmente não são conhecidos em

etapas iniciais do ciclo de desenvolvimento. Por exemplo, a tolerância a faltas ou o tempo de recuperação pode ser muito dependente da solução de design.
- Alguns requisitos, como alguns relacionados à usabilidade, são muito subjetivos, dificultando bastante a medição e o estabelecimento de valores-alvo.
- E, por fim, há os conflitos entre requisitos. Como já foi apresentado, requisitos podem influenciar atributos de qualidade comuns ou relacionados, até fazendo com que requisitos sejam contraditórios.

Mesmo sendo difícil lidar com os requisitos não-funcionais, é obrigação do arquiteto projetar o software de modo que, ao fim do desenvolvimento, este exiba os atributos de qualidade esperados pelos *stakeholders*.

6.2 Atributos de qualidade

Apesar de afirmarmos que o software possui requisitos não-funcionais [5] *a serem atendidos*, é comum dizermos que o software *exibe* atributos de qualidade que atendem aos requisitos em questão. Portanto, atributos de qualidade estão mais relacionados aos objetivos já alcançados, enquanto requisitos são os objetivos propostos.

Podemos chamar de atributos de qualidade do software suas propriedades externamente visíveis. Essas propriedades podem se manifestar como:

- *capacidades ou restrições de suas funções*. Por exemplo, tempo de resposta de uma determinada função ou capacidade de execução de certa quantidade de chamadas simultâneas;

[5] Alguns autores preferem o termo *requisitos de qualidade*.

- *características não diretamente relacionadas às suas funções.* Por exemplo, usabilidade ou adoção de padrões para interoperabilidade; ou ainda
- *características relacionadas ao ciclo de desenvolvimento.* Por exemplo, testabilidade ou mesmo a capacidade de facilitar o desenvolvimento por múltiplos times geograficamente distribuídos.

Definição 6.6: atributo de qualidade
É uma propriedade de qualidade do software ou de seu ciclo de desenvolvimento, podendo se manifestar como características, capacidades ou restrições de uma função específica ou de um conjunto de funções do software.

Podemos perceber a importância dos atributos de qualidade, em especial, quando comparamos dois produtos de software que têm as mesmas funcionalidades, como fazemos no exemplo a seguir:

Exemplo 6.7
Vamos considerar um projeto para construção de sistemas de buscas de sites web chamado Hounder [6]. Para deixarmos nosso exemplo ainda mais significativo em termos de diferenças entre atributos de qualidade, vamos considerar um sistema construído usando o Hounder, mas em que todos os seus módulos executam em apenas um servidor. Vamos chamar esse serviço de busca de HSearch [7].

Uma vez que o Google Web Search [8] também é um

[6] http://hounder.org/ (<http://hounder.org/>)
[7] Caso o leitor deseje criar um clone do HSearch, basta seguir o tutorial de cinco minutos presente em http://code.google.com/p/hounder/wiki/5minuteTutorial (<http://code.google.com/p/hounder/wiki/5minuteTutorial>)
[8] http://www.google.com (<http://www.google.com>)

serviço de busca de *web sites*, podemos afirmar que ambos os serviços têm o principal requisito funcional em comum:

(RF-01): O sistema deve retornar endereços de *web sites* que se relacionem às palavras-chave inseridas pelo usuário.

Já que ambos os serviços funcionam, percebemos que ambos atendem ao requisito (RF-01), o que poderia significar algum grau de equivalência entre os serviços. No entanto, se compararmos *como* ambos os sistemas atendem a esse requisito, perceberemos que eles são bem diferentes, justamente pela diferença entre os atributos de qualidade que exibem.

Para funcionar, um serviço de busca de *web sites* deve executar basicamente três atividades: (a) *crawling*, que é a coleta de páginas que servirão de resultados, (b) indexação, que é a organização da informação obtida na atividade de *crawling* de forma que facilite a busca (principalmente em termos de desempenho), e (c) busca, cujo resultado é a realização do requisito RF-01. Note que as três atividades são *I/O bound*, ou seja, as atividades têm uso intensivo de entrada e saída. Portanto, elas têm seu desempenho limitado pela capacidade de entrada e saída dos recursos computacionais em que executam.

Se compararmos as capacidades de ambos os sistemas, o HSearch está limitado à capacidade do único computador em que está executando. Isso significa que ele executa as três atividades usando o mesmo recurso. Por outro lado, é bem conhecido que a arquitetura do Google Web Search permite que o sistema utilize diversos *data centers* ao redor do mundo, usando muitos mil-

hares de processadores simultâneos e, assim, podendo dividir a execução das três atividades entre esses recursos. Por essa diferença de utilização de recursos, algumas métricas de vários atributos de qualidade, como tempo de resposta, capacidade de atender a buscas simultâneas, tamanho do índice de busca ou tolerância a falhas de hardware serão bem diferentes entre os dois sistemas.

Quando comparamos as bilhões de consultas diárias que o Google Web Search é capaz de realizar com as apenas milhares ou poucos milhões do HSearch, dizemos que o desempenho do primeiro é melhor. Mas o desempenho não é diferente apenas em termos de operações por unidade de tempo, mas também quando comparamos os tempos de resposta para cada operação ou número de usuários simultâneos no sistema. Se considerarmos que o Google Web Search realiza um bilhão de buscas por dia e cada busca dura em torno de 300 milissegundos, pela Lei de Little [67], temos cerca de 3500 buscas simultâneas a qualquer momento ao longo da vida do sistema. Já o HSearch só consegue realizar 3,5 buscas simultâneas ao realizar 1 milhão de buscas por dia a 300 milissegundos cada.

Mas há outros atributos que podem ser mencionados. O HSearch é dependente do funcionamento de um único servidor. Portanto, se esse servidor falhar, todo o sistema ficará fora do ar. Já o Google Web Search é capaz de tolerar falhas de hardware, uma vez que não depende de apenas um servidor para funcionar. Assim, podemos dizer que o grau de confiabilidade ou tolerância a falhas do Google Web Search é maior que o do HSearch. As respostas do HSearch são formadas apenas pelo título e pequenos trechos dos web sites que

contêm as palavras-chave. Já o Google Web Search ajuda ao usuário também mostrando imagens contidas no site ou mesmo trechos de vídeo, contribuindo assim para sua usabilidade. Por fim, citamos também que o Google Web Search apresenta o atributo de integrabilidade, dado que ele contém diversos serviços além da busca numa mesma interface: entre eles calculadora, previsão do tempo, conversão de medidas, definição de palavras, busca de sinônimos, entre outros.

É a arquitetura que permite que o software exiba os atributos de qualidade especificados. Já que a especificação dos atributos é feita pelos requisitos (normalmente não-funcionais), requisitos e atributos de qualidade partilham diversas características. Tanto que alguns autores usam ambas as expressões com o mesmo sentido.

As principais características dos atributos de qualidade são as seguintes:

- Atributos de qualidade impõem limites às funcionalidades;
- Atributos de qualidade se relacionam entre si; e
- Atributos de qualidade podem tanto ser de interesse dos usuários quanto dos desenvolvedores.

6.2.1 Limites às funcionalidades

Os limites às funcionalidades acontecem da mesma forma que os requisitos podem restringir ou mesmo impedir funcionalidades, pois atributos de qualidade não se manifestam isolados no ciclo de vida do software, mas influenciam e são influenciados pelo meio. Por exemplo, para que o SASF tenha um *time to market* pequeno, ele deve ser lançado inicialmente sem possuir um cliente de *streaming* para dispositivos móveis, deixando

para implementar essa funcionalidade em outras versões. Isso é uma limitação na funcionalidade de transmissão de filmes em benefício do atributo de qualidade *custo e planejamento*. É também bastante comum encontrarmos sistemas que têm funcionalidades podadas simplesmente porque, se estas existissem, o software não exibiria os atributos de segurança esperados.

6.2.2 Relações entre atributos de qualidade

Como já foi observado, os atributos não existem isoladamente e, por afetarem partes em comum da arquitetura, afetam também outros atributos de qualidade. Eis que surgem os *trade-offs* entre os atributos de qualidade. Por exemplo, um sistema mais portável terá seu desempenho afetado negativamente, pois necessita de mais camadas de software que abstraiam o ambiente que pode ser mudado. Já no caso do SASF, para se obter um nível de segurança capaz de realizar autorização e autenticação, a usabilidade do software é prejudicada, uma vez que o usuário deve ser obrigado de lembrar sua senha ou mesmo ter o fluxo de ações interrompido para que insira suas credenciais.

É papel do arquiteto conhecer e resolver os *trade-offs* entre os atributos de qualidade durante as fases de design e implementação. Por isso, ao apresentar algumas técnicas para alcance da qualidade, apresentaremos também quais atributos são influenciados positiva e negativamente.

6.2.3 A quem interessa os atributos de qualidade

Uma grande gama de atributos podem ser citados. Tanto que, a seguir, quando apresentamos uma lista deles, restringiremonos a apenas um modelo de qualidade. Esses atributos podem interessar a vários envolvidos no ciclo de vida do soft-

ware, como usuários e desenvolvedores. Dos exemplos citados anteriormente, podemos dizer que desempenho e usabilidade são atributos importantes a usuários, enquanto custo e planejamento são mais importantes aos desenvolvedores.

6.3 Modelo de Qualidade

Para avaliar a qualidade de um software, o ideal seria usar todos os atributos de qualidade que conhecemos. No entanto, é inviável adotar esta abordagem em um processo de desenvolvimento que possua tempo e dinheiro finitos devido à grande quantidade de dimensões [9] do software que poderíamos avaliar. Para facilitar o processo de avaliação durante o desenvolvimento, foram desenvolvidos o que chamamos de *modelos de qualidade*. Modelos de qualidade têm como objetivo facilitar a avaliação do software, organizando e definindo quais atributos de qualidade são importantes para atestar a qualidade geral do software. Alguns exemplos significativos de modelos de qualidade são os de Boehm [16], o de McCall [72] e o contido no padrão ISO/IEC 9126-1:2001 [4]. Vamos descrever melhor este último, para assim termos uma melhor noção de quais atributos de qualidade procuramos que a arquitetura permita ao software.

Definição 6.7: modelo de qualidade
Modelo que define e organiza os atributos do software importantes para a avaliação de sua qualidade.

[9] Em Inglês, alguns autores se referem aos atributos de qualidade usando o sufixo *-ilities*, que é comum ao nome de vários atributos. Podemos perceber isso na lista de qualidades presente no endereço: http://en.wikipedia.org/wiki/Ilities (<http://en.wikipedia.org/wiki/Ilities>). Em Português, poderíamos nos referir a *-idades*, mas preferimos usar *dimensões*, *propriedades* ou mesmo *qualidades*.

6.3.1 Padrão ISO/IEC 9126-1:2001

Ele é um padrão internacional para avaliação de software. O que nos interessa dele é o conteúdo de sua primeira parte, que é o que é chamado de qualidades internas e externas do software. Essas qualidades são apresentadas na forma de uma lista exaustiva de características ou atributos de qualidade. Os atributos que um software deve possuir para que possamos dizer que ele é de *qualidade* são os seguintes:

- Funcionalidade
- Confiabilidade
- Usabilidade
- Eficiência
- Manutenibilidade
- Portabilidade

É importante enfatizar que essa lista tem como objetivo ser exaustiva. Portanto, de acordo com a norma, todas as qualidades que venham a ser requisitadas ao software estão presentes nessa lista. No padrão, cada característica é ainda quebrada em subcaracterísticas, que são mais específicas, a fim de facilitar o entendimento e a avaliação. A seguir, definimos cada atributo de qualidade e mostramos algumas subcaracterísticas mais importantes ao atributo.

6.3.1.1 Funcionalidade

Funcionalidade é a capacidade do software de realizar as funções que foram especificadas. Esse primeiro atributo pode parecer óbvio, mas seu propósito é claro quando passamos a avaliar um sistema de software: se esse sistema faz menos que o mínimo que é esperado dele, ele não serve, mesmo que o (pouco) que ele faça, ele faça de forma usável e confiável ou eficientemente.

Para caracterizarmos melhor a funcionalidade do software, devemos ainda considerar as características de:

- *adequação*, ou capacidade de prover as funções necessárias para os objetivos dos usuários. Podemos observar que a métrica deste atributo de qualidade é a satisfação ou não dos requisitos funcionais do sistema.

  Exemplo 6.8
  Para se adequar às necessidades de seus usuários, basta que o SASF atenda a seus requisitos funcionais. Se ele realizar a locação e a transmissão de filmes, ele está adequado às necessidades de seus usuários comuns. Por outro lado, para se adequar às necessidades dos usuários que distribuem os filmes, uma das funções que ele deve prover é a função de *upload* de filmes.

- *precisão*, ou capacidade de prover os resultados com o grau de precisão adequado. Para que seja possível medir a precisão, é necessário que ela esteja especificada – possivelmente no documento de requisitos.

  Exemplo 6.9
  Podemos observar diferentes necessidades de precisão quando comparamos como os números são tratados em um sistema de software bancário e numa calculadora. No primeiro, os números são tratados apenas como racionais e truncados na quantidade de casas decimais relativa à moeda do país. No Brasil, por exemplo, o software bancário só reconhece até centavos de Real. Portanto, se é necessário dividir R$ 1,00 em três parcelas, cada parcela não será representada pela dízima R$ 0,33333..., mas sim por R$ 0,34. Essa mesma precisão não poderia ser adotada em um software

de calculadora. Nesse, sendo uma calculadora comum, é esperado que os números seja representados da forma mais próxima aos números reais [10].

- *interoperabilidade*, ou capacidade de interagir com outros sistemas. Para medir o grau de interoperabilidade, o ideal é que esteja especificado quais sistemas devem interagir. Já para facilitar a satisfação desse atributo, a solução mais utilizada é a adoção de padrões *de facto*. Alguns tipos de padrões são os de representação de dados, como o *Dublin Core* ou formatos de arquivos de vídeo, ou padrões de especificação de funcionalidades, como os padrões WS-*. [11]

  Exemplo 6.10

  É uma qualidade do SASF ser capaz de interagir com diversos sistemas capazes de reproduzir o vídeo transmitido. Para isso, foi escolhido o padrão para transmissão de vídeo amplamente adotado entre sistemas.

- *segurança*, ou capacidade de funcionar segundo os princípios de autenticação, autorização, integridade e não-repudiação. Autenticação é a capacidade de o sistema verificar a identidade de usuários ou de outros sistemas com que se comunica. Autorização é a capacidade de garantir ou negar direitos de uso a recursos a usuários autenticados. Integridade é a capacidade de garantir que

[10] Possivelmente, a calculadora implementará o padrão para aritmética de ponto-flutuante IEEE 754-2008 [3]

[11] A comunidade interessada em *web services* especificou uma série de padrões que facilitam a interoperabilidade entre os serviços. Podemos encontrar uma grande lista deles no seguinte endereço: http://bit.ly/kIEXs (<http://bit.ly/kIEXs>).

os dados não foram alterados indevidamente, principalmente durante a comunicação. E não-repudiação é a capacidade de prover meios para a realização de auditoria no sistema. No entanto, é importante observar que nem todos os sistemas precisam estar de acordo com todos os princípios.

Exemplo 6.11

Uma vez que recebe o número do cartão do usuário para receber o pagamento, o SASF deve garantir que apenas o sistema de cobrança da operadora de cartão de crédito seja capaz de verificar as informações necessárias para a autorização. Outro aspecto de segurança do SASF é que ele precisa diferenciar os usuários que ainda não estão registrados (e, consequentemente, que não pagaram a assinatura), dos já registrados. Para isso, ele deve realizar a autenticação do usuário.

- *estar de acordo com padrões*, ou a capacidade de aderir a normas, convenções ou leis relacionadas à funcionalidade.

Exemplo 6.12

Para ser executado no Brasil, o SASF é obrigado por lei a emitir o cupom fiscal do pagamento da assinatura do usuário.

6.3.1.2 Confiabilidade

Quando afirmamos que um sistema é confiável, estamos afirmando que esse sistema é capaz de manter algum nível de desempenho quando funcionando sob circustâncias determinadas. A confiabilidade é normalmente definida sob períodos de tempo. Ou seja, dizer apenas que o SASF deve ser confiável

não é suficiente. Temos, por exemplo, que dizer que o SASF é capaz de transmitir vídeos para 6 mil usuários simultâneos sob condições normais durante 99% do ano e para mil usuários simultâneos durante o 1% do ano reservado para o período de manutenção dos servidores. Vale observar que, para uma loja *online*, faz mais sentido que a medida de confiabilidade seja a de servir aos seus usuários com o tempo de espera das operações de compra e busca de 50 milissegundos durante períodos normais do ano, mas, durante as semanas próximas ao Natal, ter o tempo de espera das mesmas operações em torno dos 150 milissegundos, uma vez que o número de usuários simultâneos nessa época do ano aumenta consideravelmente.

A confiabilidade pode ainda ser dividida nas seguintes características:

- *maturidade*, ou capacidade de se prevenir de falhas resultantes de faltas de software. Isso é comum em sistemas distribuídos, onde um componente não confia completamente no resultado provido por outro. Isso pode ser verificado em sistemas com sensores de software, onde um módulo pode ser responsável por julgar os valores gerados pelos sensores. Caso os valores sejam julgados inválidos, o módulo pode simplesmente desligar o sensor defeituoso. A medição do grau de maturidade de um sistema é bem difícil, mas podemos ter uma noção ao analisarmos decisões que foram feitas com este objetivo.

  Exemplo 6.13
  No caso do SASF, o módulo de transmissão de vídeo pode verificar quantas conexões estão abertas para um mesmo destinatário. Uma grande quantidade de conexões para um mesmo destinatário pode significar um ataque ou mesmo um *bug* no reprodutor de vídeo no lado do cliente que, eventualmente, pode consumir todos os re-

cursos disponíveis para *streaming*. Assim, ao detectar esse problema, o SASF pode recusar abrir novas conexões para esse cliente, prevenindo-se de um problema maior, como uma completa parada por *DoS* [12]

- *tolerância a faltas*, ou capacidade de manter alguma qualidade de serviço em caso de faltas de software ou comportamento imprevisto de usuários, software ou hardware. Em outras palavras, a medida de funcionamento do software, mesmo que de forma restrita, em caso de a parada de servidores, partições de rede, falhas de discos rígidos, inserção ou leitura de dados corrompidos, etc. Considerando a grande quantidade de eventos que o software deve tolerar, também são muitas as formas de medir o grau de satisfação a este atributo de qualidade. As formas mais comuns são: medir se o serviço continua funcionando em caso de falha de n servidores, medir qual a variação no tempo de resposta para as operações mais comuns ou quantos usuários simultâneos o sistema é capaz de servir em caso de falhas de servidores ou ainda verificar como o sistema se comporta se dados inválidos são inseridos no sistema.

  Exemplo 6.14
  A forma mais comum de melhorar o grau de tolerância a faltas em um serviço web é fazer com que não dependa de um único recurso. Seja esse recurso hardware, como um único processador, roteador ou disco rígido, seja esse recurso software, como depender de um único banco de dados, um único serviço de cadastro ou um único

[12] O *Denial of Service* ou DoS ocorre quando o sistema não pode atender a novas requisições porque todos os seus recursos estão sendo consumidos, possivelmente devido a um ataque de um ou vários agentes maliciosos.

serviço de inventário. Assim, o SASF possui seus módulos replicados em diferentes servidores. Desta maneira, ele evita a dependência de um único recurso, ou o chamado *ponto único de falhas* e pode continuar a funcionar mesmo que um desses módulos pare por completo. Note que para a replicação funcionar, devem ser adicionados à arquitetura módulos responsáveis pela verificação de estado dos servidores e, assim que forem detectados problemas em algum servidor, o tráfego possa ser redirecionado para réplicas sadias. Para isso ser possível, há ainda outras complicações, como a manutenção da consistência de estado entre o servidor original e sua réplica. Falaremos mais sobre a eliminação do ponto único de falhas quanto estivermos tratando das diversas técnicas para a obtenção de atributos de qualidade.

- *recuperabilidade*, também chamada de resiliência, é a capacidade de o sistema voltar ao nível de desempenho anterior a falhas ou comportamento imprevisto de usuários, software ou hardware e recuperar os dados afetados, caso existam. É comum medirmos o grau de recuperabilidade ao medirmos quanto tempo o sistema leva para voltar aos níveis normais de desempenho. Quanto menor esse tempo, melhor a qualidade do sistema neste sentido.

  Exemplo 6.15
  No SASF, podemos medir o tempo de substituição de um servidor de *streaming* pelo tempo da detecção da falha, somado ao tempo de inicialização do servidor e somado ao tempo de redirecionamento das requisições de transmissão. Uma forma de ter o tempo total de recuperação minimizado seria manter o servidor auxiliar ligado,

apenas esperando a detecção da falha do servidor principal. No entanto, essa decisão significaria mais custos, uma vez que seriam dois servidores ligados ao mesmo tempo, gastando mais energia, diminuindo a vida útil do hardware e possivelmente consumindo licenças de software.

6.3.1.3 Usabilidade

Usabilidade é a medida da facilidade de o usuário executar alguma funcionalidade do sistema. Essa facilidade está ligada diretamente à compreensibilidade, à facilidade de aprendizado, à operabilidade, a quanto o usuário se sente atraído pelo sistema e à adesão de padrões de usabilidade, que são as subcaracterísticas desse atributo de qualidade. Apesar de muitos desses critérios serem subjetivos, há maneiras de medi-los para termos noção da usabilidade do software. A seguir, mostramos as subcaracterísticas da usabilidade:

- *compreensibilidade*, ou a capacidade de o usuário entender o sistema. Esta característica está ligada à quantidade de conceitos que o usuário precisa saber previamente para lidar com o sistema ou à qualidade ou quantidade da documentação do sistema. A compreensibilidade serve para o usuário dicernir se o software serve para ele ou não.
- *facilidade de aprendizado* está ligada diretamente à compreensibilidade. No entanto, neste caso, a qualidade é a de o usuário aprender a usar o software, caso ele saiba que o software serve para ele. As métricas dessa qualidade também estão relacionadas à quantidade de conceitos ou operações que o usuário precisa aprender para fazer com que o software funcione.

- *operabilidade* é a capacidade de o usuário operar ou controlar o sistema. Esta qualidade é muito importante em grandes sistemas de software, onde há um tipo de usuário que é o administrador do sistema. O administrador deseja ser capaz de realizar operações sobre o sistema que, comumente, não estão entre as funções que interessam aos usuários mais comuns: ligar, desligar ou verificar estado de servidores, realizar *backup* dos dados, etc. Em sistemas de redes sociais, por exemplo, entre os serviços providos ao operador, ainda estão a possibilidade de expulsar usuários do sistema ou moderá-los, não permitindo que esses usuários realizem algumas funções, como enviar mensagens ou mesmo barrando conexões de acordo com o endereço de origem.

6.3.1.4 Eficiência

A eficiência ou desempenho é talvez a qualidade mais buscada durante o desenvolvimento de software, uma vez que ela é a mais percebida pelos usuários. Ela é a qualidade relacionada ao uso de recursos do sistema quando esse provê funcionalidade e é também a com que os desenvolvedores mais se preocupam. Quando queremos medir eficiência, medimos basicamente duas características:

- *comportamento no tempo* ou *desempenho*, ou a capacidade do sistema de alcançar a resposta dentro do período de tempo especificado. Aqui, referimo-nos a tempos de resposta, latência, tempo de processamento, vazão (*throughput*), etc. Vale observar que, ao medir essa característica, devemos também entender as condições em que o sistema está operando. Afinal, no Exemplo 6.7, mesmo que o HSearch tenha um tempo de resposta menor que o

Google Web Search, o primeiro é capaz de servir a apenas um milésimo da quantidade de usuários servida pelo segundo.

- *uso de recursos*, que é a capacidade de o software exigir mais ou menos recursos de acordo com suas condições de uso. Normalmente, essa característica também é chamada de *escalabilidade* e pode também ser vista de outra maneira: como a adição ou remoção de recursos no sistema vai melhorar ou piorar as condições de uso. Existem dois tipos mais comuns de escalabilidade, que também servem para facilitar o entendimento dessa característica: *escalabilidade vertical* e *escalabilidade horizontal* . Eles podem ser melhor explicados por meio de um exemplo:

  Exemplo 6.16

  Vamos considerar um sistema servidor de arquivos. Esse servidor de arquivos usa apenas um disco rígido e é capaz de servir a cinco usuários simultâneos, cada um usando 10 MB/seg de banda passante (fazendo *upload* ou *download*). Vamos desconsiderar os efeitos da rede que liga os clientes ao servidor ou qualquer outro gargalo. Podemos dizer que as condições de uso do software são: 5 usuários simultâneos a 10 MB/seg cada.

  No Exemplo 6.16, uma forma de melhorar as condições de uso, ou mais especificamente, aumentar a quantidade de usuários simultâneos, seria *seria substituir um dos recursos do sistema por outro com maior capacidade*. Ou seja, escalar verticalmente.

  Exemplo 6.17: (continuação do exemplo anterior)

  Vamos substituir o disco rígido do servidor por um que seja capaz de transferir arquivos no do-

bro da velocidade do anterior. Desta maneira, se o disco rígido fosse o único fator limitante, conseguiríamos não mais servir 5 usuários a 10 MB/seg, mas sim 10 usuários simultâneos a 10 MB/seg, como ilustrado na Figura 6.1. Além disso, poderíamos seguir melhorando verticalmente o sistema até encontrarmos um limite, que pode ser tanto o limite na velocidade possível para um disco rígido quanto o limite financeiro de comprarmos um disco mais rápido.

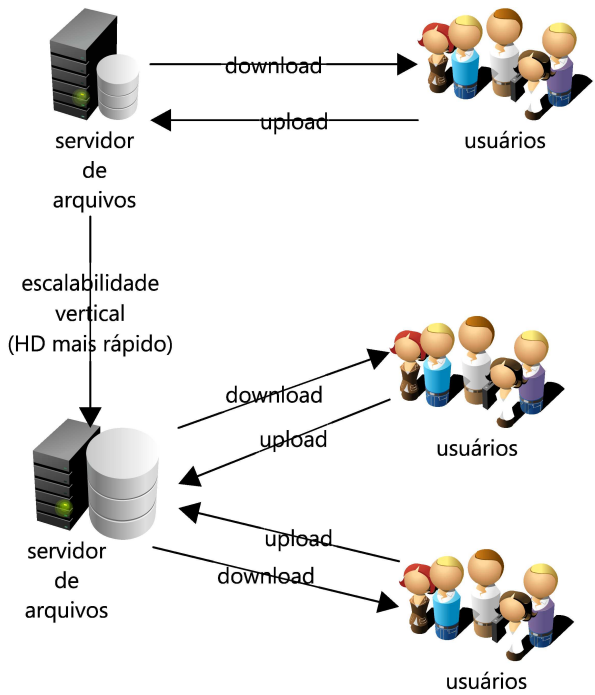

Figura 6.1: Escalando verticalmente um sistema.

Outra forma de escalar o sistema seria horizontalmente. Desta maneira, não substituímos um recurso por um melhor, mas *adicionamos um novo recurso ao sistema de modo que ele faça uso tanto do recurso velho quanto do novo.*

Exemplo 6.18: (continuação do exemplo anterior)

Ao invés de necessariamente comprar um disco rígido mais rápido, compramos um novo disco (que pode até ser igual ao anterior) e fazemos com que o software divida a carga de escrita e leitura entre os dois discos rígidos. Esta abordagem está ilustrada na Figura 6.2.

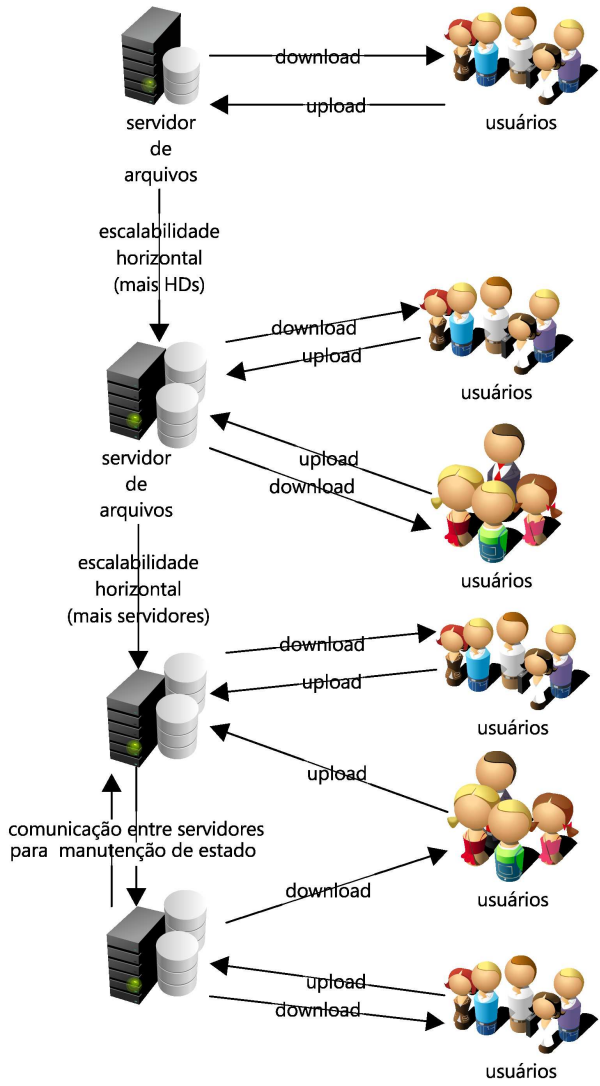

Figura 6.2: Escalando horizontalmente um sistema.

Note que a solução do Exemplo 6.18 ((continuação do ex-

emplo anterior)) não vem de graça: além da camada de software ficar mais complicada, há o impacto na eficiência – possivelmente, o tempo de resposta será afetado, uma vez que uma operação do usuário terá que agora decidir qual disco rígido usar. No entanto, a vantagem desta solução reside no fato de que o teto de desempenho com a adição de novos discos será mais alto que o teto alcançável com discos mais rápidos. Além disso, há um limite de discos rígidos que podem ser utilizados por um mesmo sistema operacional. Para expandir ainda mais o limite de discos rigídos sendo usados simultaneamente, o próximo passo seria adicionar mais uma máquina servidora, o que deixaria o software ainda mais complexo, pois este agora teria que decidir entre discos presentes em máquinas diferentes e assim por diante. Esse é apenas um exemplo de técnica de se alcançar escalabilidade horizontal. No próximo capítulo, quando falarmos de técnicas de design, apresentaremos outras abordagens e padrões de design para a escalabilidade.

6.3.1.5 Manutenibilidade

A manutenibilidade é uma qualidade, às vezes, negligenciada pelos usuários, mas muito importante aos desenvolvedores. Ela é a capacidade de o software ser modificado em seu processo de evolução. Podemos citar as seguintes características do atributo de manutenibilidade: a analisabilidade, a modificabilidade e a testabilidade.

- *analisabilidade*: é o grau de facilidade com que podemos procurar por deficiências no software ou por partes que devem ser modificadas para algum fim. Os níveis de modularidade, de separação de preocupações e de acomplamento do software se relacionam a essa característica.

- *modificabilidade*: é a capacidade de realizar mudanças de implementação no sistema. Essa característica também está relacionada às métricas clássicas de software, como níveis de coesão e acoplamento e complexidade ciclomática. Quanto mais modificável o software, menor o impacto da mudança em áreas – teoricamente – não relacionadas às mudanças.

  Exemplo 6.19
  No SASF, por termos o módulo de transmissão de vídeos separado do gestor de usuários, qualquer mudança ou adição nos formatos suportados para transmissão não deve afetar ao módulo de usuários. Outra separação comum em sistemas web que também foi adotada no SASF é a aplicação do padrão *Model-View-Controller* (MVC) [13], que separa as interfaces de usuário de lógica de negócio. Isso permite modificações na lógica de negócio que não afetam as interfaces de usuário e vice-versa.

- *testabilidade*: é a capacidade de o software ter suas mudanças validadas. Para um software ser testável, antes de tudo, devemos conhecer seus objetivos. Mas, além disso, precisamos que o sistema seja capaz de executar de forma controlada a fim de podermos medir os resultados obtidos a partir de entradas conhecidas. Sistemas pouco testáveis são aqueles os quais sua execução é muito cara, pode custar vidas ou, simplesmente, não podemos medir seu comportamento deterministicamente. Vale observar que muitos sistemas distribuídos, se mal projetados, podem se encaixar nesse último tipo.

[13] Falamos mais do padrão arquitetural MVC quando apresentamos as ferramentas de design de software no capítulo sobre técnicas de design.

6.3.1.6 Portabilidade

O último atributo de qualidade presente no padrão ISO/IEC 9126-1:2001 é o de portabilidade. Esse atributo é a medida de adaptações necessárias para que o sistema tenha seus requisitos ou ambientes de execução modificados, podendo ser o ambiente de software, de hardware ou organizacional. Esse atributo é importante, por exemplo, para jogos, uma vez que é desejável que eles sejam capazes de executar no maior número de plataformas, mas também é desejável que o custo para tornar isso possível seja baixo. Algo similar acontece com aplicativos para celulares. A necessidade de um aplicativo para celulares ser portável existe porque é comum que seus desenvolvedores queiram que ele esteja disponível em dezenas de modelos diferentes. Isso significa que um mesmo aplicativo deve estar disponível para dezenas de ambientes de hardware diferentes. Portanto, não faz sentido que o mesmo aplicativo seja reimplementado diversas vezes, mas sim que seja projetado de forma a minimizar o esforço para alterar o ambiente de hardware.

A portabilidade pode ainda ser dividida nas seguintes características:

- *adaptabilidade*: é a capacidade de o software ser portado para outro ambiente sem precisar de modificações além das previstas.

  Exemplo 6.20

  O Vuze [14] é um aplicativo escrito na linguagem de programação Java e que, por isso, é capaz de executar em qualquer sistema operacional em que a máquina virtual Java (JVM) esteja disponível. No entanto, apesar da portabilidade provida pela linguagem de programação em que foi escrito, ele necessita de uma pequena modificação específica

[14] http://www.vuze.com (<http://www.vuze.com>)

para cada novo sistema operacional suportado pela JVM. Essa modificação consiste na criação de um instalador específico para o S.O., uma vez que diferentes sistemas possuem diferentes formas de instalação de software. No entanto, essa modificação é prevista na arquitetura do Vuze e não afeta significativamente sua adaptabilidade a novos sistemas operacionais.

- *instalabilidade*: é a capacidade de o software ser instalado em algum ambiente específico. A instalabilidade é medida junto com o ambiente-alvo. Portanto, por exemplo, antes do *Apple Bootcamp*, o sistema operacional *Windows XP* não era instalável em ambientes *Apple*. Já o sistema *GNU/Linux*, por sua vez, era instalável tanto em PCs quanto em Macs.
- *co-existência*: é a capacidade de o software compartilhar recursos em um mesmo ambiente com outros sistemas.

6.3.2 Conflitos entre atributos de qualidade

Assim como os interesses de cada stakeholder não são isolados e podem afetar os de outro por meio dos requisitos não-funcionais, os atributos de qualidade não surgem isolados no software. Uma decisão arquitetural feita com o objetivo de alcançar um atributo de qualidade pode ter efeito em outros atributos. Por uma decisão arquitetural nunca ser isolada no design da arquitetura, o arquiteto deve sempre entender quais atributos a decisão afeta, seja positivamente ou negativamente, e fazer as devidas concessões caso ela afete atributos de qualidade conflitantes. No capítulo sobre técnicas de design, observaremos melhor as relações entre os atributos de qualidade ao apresentarmos algumas técnicas de design arquitetural para

alcançá-los. Isso acontece porque é comum que essas técnicas não afetem cada atributo de software isoladamente.

6.4 Atributos de Negócio

Apesar de a lista de atributos de qualidade apresentada anteriormente ter sido criada a fim de ser exaustiva, há alguns atributos adicionais que merecem ser citados. São chamados os *atributos de qualidade de negócio*, que, apesar de não serem ligados diretamente ao software, têm grande influência sobre sua arquitetura. Eles são importantes porque influenciam principalmente as decisões de resolução de conflitos dos atributos apresentados anteriormente. Os atributos de negócio são:

- mercado-alvo
- *time-to-market*
- custo e benefício
- vida útil do sistema
- agenda de lançamento

6.4.1 Mercado-alvo

O arquiteto só é capaz de priorizar os atributos de qualidade em seu design se conhecer o público e o mercado para o qual o software está sendo construído. Por exemplo, portabilidade e funcionalidade são buscados para o público geral de um pacote de aplicativos de escritório e, portanto, priorizados neste caso. Por outro lado, ao se construir um sistema de infraestrutura para uma empresa específica, o arquiteto pode priorizar a eficiência em detrimento da portabilidade e até mesmo da usabilidade, uma vez que os usuários comuns desse sistema são operadores qualificados.

6.4.2 *Time-to-market*

Time-to-market é o tempo entre a concepção do software e sua entrega no mercado. Esse atributo se torna importante, principalmente, quando a janela de oportunidade é pequena devido a produtos concorrentes. O *time-to-market* influencia e, quando curto, prioriza decisões de compra e reuso de módulos em detrimento do desenvolvimento *in house* ou de investimento em decisões que dizem respeito a atributos considerados secundários ao negócio.

6.4.3 Custo e benefício

Como os recursos financeiros para se desenvolver o software são limitados, cada decisão arquitetural deve ter seu custo e o benefício proporcionado analisados e, com base nessa análise, priorizados ou até mesmo descartados. Essa análise deve levar em conta o ambiente de desenvolvimento em questão: capacidades do time de desenvolvimento, ferramentas disponíveis para o reuso e os objetivos do software.

6.4.4 Vida útil

O design de sistemas de grande vida útil deve priorizar diferentes atributos de qualidade se os compararmos com o design de sistemas de vida mais curta, como protótipos. No primeiro tipo de sistemas, atributos de manutenibilidade e portabilidade são mais valorizados; no segundo, são priorizados atributos de eficiência e funcionalidade.

6.4.5 Agenda de lançamento

O design do software é muito dependente de como ele vai ser disponibilizado a público. Por exemplo, se o software será disponibilizado em fases distintas que englobarão diferentes

conjuntos de funcionalidades, ele deve ser dividido de modo que funcione sem as partes que ainda não foram disponibilizadas, mas que também facilite tanto a modificabilidade, uma vez que é desejável que novas funcionalidades sejam adicionadas com menor esforço, quanto a interoperabilidade entre diferentes versões, que eventualmente ocorrerá. Já se o software será disponibilizado sem possibilidade de posterior atualização, como acontece em muitos sistemas embarcados, preocupações de modificabilidade e interoperabilidade entre versões podem ser descartadas.

6.5 Design Arquitetural para Qualidade de Software

A principal responsabilidade do arquiteto é a de conceber o design que possibilite ao software ser construído de modo que satisfaça os requisitos de qualidade impostos pelos stakeholders. Para que o processo de design arquitetural tenha sucesso, é essencial que o arquiteto conheça os objetivos do software, ou seja, conheça os requisitos funcionais e de qualidade para os quais ele está projetando. Além disso, ele deve conhecer tanto as técnicas e práticas de design arquitetural que podem ajudá-lo na concepção da arquitetura. Ele deve também conhecer como documentar a arquitetura projetada, uma vez que é preciso comunicá-la aos outros membros do time de desenvolvimento.

Neste capítulo, nós aprendemos sobre os objetivos que devem ser alcançados pelo design da arquitetura e esperamos que o leitor agora seja capaz de:

- Identificar o que são atributos de qualidade e qual é sua influência na arquitetura de software;

- Relacionar atributos de qualidade com algumas decisões arquiteturais que os proporcionam; e
- Entender quais os atributos de qualidade se relacionam e como eles se relacionam.

A seguir, apresentaremos técnicas e práticas de design que o arquiteto deve conhecer para projetar sistemas com determinados atributos de qualidade. Por fim, no capítulo seguinte, apresentaremos como documentar o design arquitetural.

6.6 Referências

6.6.1 Requisitos funcionais e não-funcionais

Os livros *Software Engineering* [100], de Sommerville, *Requirements Engineering: Processes and Techniques* [58], de Sommerville e Kotonya, *Software Engineering: A Practitioner's Approach* [86], de Pressman, dedicam alguns capítulos a este assunto. No entanto, o foco desses livros é no papel dos requisitos de software no processo de desenvolvimento. Já o artigo *Defining Non-Functional Requirements* [70], de Malan e Bredemeyer, é mais voltado à influência dos requisitos na arquitetura.

6.6.2 Diferenças entre requisitos funcionais e não-funcionais

A discussão sobre a inexistência de diferenças práticas entre requisitos funcionais e não-funcionais pode ser encontrada tanto no livro *Requirements Engineering: Processes and Techniques* [58], de Sommerville e Kotonya, quanto no artigo *Distinctions Between Requirements Specification and Design of Real-Time Systems* [56], de Kalinsky e Ready, e no livro *Real-Time Systems: Design Principles for Distributed Embedded*

Applications [57], de Kopetz. Essa discussão se mostra bastante presente em sistemas de tempo-real porque os requisitos de desempenho definem a funcionalidade desses sistemas – ao contrário do que encontramos, por exemplo, em sistemas de informação, onde os requisitos de desempenho são considerados requisitos não-funcionais.

6.6.3 Atributos de Qualidade

Bass *et al*, no livro *Software Architecture in Practice* [11], mostra o papel dos atributos de qualidade na arquitetura de software. Além dele, Gorton faz uma pequena introdução a este assunto ao tratar do estudo de caso presente em *Essential Software Architecture* [48]. Os livros *Software Systems Architecture* [90], de Rozanski e Woods, e *Code Complete* [73], de Steve McConnell, também dedicam seções aos atributos de qualidade de software, sendo o primeiro em nível de design arquitetural e o segundo em nível de design detalhado.

6.6.4 Atributos de Negócio

Por fim, podemos encontrar informações sobre atributos de qualidade de negócio nos livros *Software Architecture in Practice* [11], de Bass *et al*, e *Beyond Software Architecture* [52], de Hohmann.

Chapter 7

Técnicas de Design Arquitetural[1]

Ao introduzirmos design de software, citamos alguns princípios e técnicas que são fundamentais ao processo, pois facilitam a representação e a escolha da solução entre as alternativas de design. No entanto, não fomos explícitos sobre como estes princípios e técnicas são fundamentais ao processo de design arquitetural. Já no capítulo sobre atributos de qualidade, mencionamos a existência de táticas arquiteturais que ajudam na implementação de alguns requisitos de qualidade, mas não apresentamos essas táticas a não ser de forma breve e apenas por meio de exemplos.

Este capítulo, por sua vez, tem como objetivo tanto apresentar os princípios de design em nível arquitetural, quanto apresentar algumas táticas arquiteturais que implementam requisitos de qualidade. Neste capítulo, descrevemos os seguintes princípios de design arquitetural:

- uso da abstração ou níveis de complexidade;

[1] This content is available online at <http://cnx.org/content/m17523/1.5/>.

- separação de preocupações; e
- uso de padrões e estilos arquiteturais.

Em relação às táticas arquiteturais, apresentamos as que implementam os seguintes atributos de qualidade:

- desempenho e escalabilidade;
- segurança;
- tolerância a faltas;
- compreensibilidade e modificabilidade; e
- operabilidade.

7.1 Princípios e Técnicas de Design Arquitetural

Há alguns princípios e técnicas que, quando aplicados, geralmente resultam em boas soluções de design. Entre eles, podemos citar: divisão e conquista, abstração, encapsulamento, modularização, separação de preocupações, acoplamento e coesão, separação de interfaces de suas implementações, entre outros. Inclusive, muitos destes já foram apresentados no capítulo sobre Design, mas sem o devido foco em design arquitetural. Por isso, nesta seção, descrevemos novamente alguns deles, desta vez mostrando seu papel na arquitetura. Os princípios e técnicas que apresentamos a seguir são três: uso da abstração ou níveis de complexidade, separação de preocupações e uso de padrões e estilos arquiteturais.

7.1.1 Abstração

Abstração é a seleção de um conjunto de conceitos que representam um todo mais complexo. Por ser um modelo do software, a arquitetura já elimina, ou em outras palavras, *abstrai* naturalmente alguns detalhes do software. Por exemplo,

é comum que não tenhamos decisões em nível algorítmico na arquitetura. Mesmo assim, podemos tirar proveito do uso de níveis de detalhe (ou de abstração) ao projetá-la.

Podemos nos beneficiar do uso da abstração ao realizarmos o processo de design de forma iterativa, onde cada passo é realizado em um nível de detalhe. De forma simplificada, podemos dizer que a sequência de passos pode ocorrer seguindo duas estratégias de acordo com os níveis de abstração do software.

A primeira estratégia é a *top-down* (do nível mais alto de abstração para o mais baixo). Se o design ocorre no sentido *top-down*, o arquiteto usa elementos e relações arquiteturais descritos em alto nível de abstração para iniciar o projeto da arquitetura. No primeiro nível de abstração, o mais alto, é comum que os elementos arquiteturais usados no projeto mostrem apenas *o que* realizam e não *como* realizam suas responsabilidades. A partir daí, a cada passo do processo, o arquiteto segue refinando o design, adicionando mais detalhes aos elementos arquiteturais e às suas relações, até que possuam informações sobre *como* realizar suas responsabilidades. Neste ponto, é comum termos elementos arquiteturais que realizam funções e serviços mais básicos ou de infraestrutura e que, eventualmente, farão parte da composição das funcionalidades em níveis mais altos.

Um problema recorrente ao se aplicar a estratégia *top-down* é o de *quando parar*. Afinal, podemos notar que o arquiteto poderia seguir indefinidamente adicionando detalhes à arquitetura até que o design deixe de ser um modelo para ser o próprio sistema. Para definir o ponto de parada do processo de adição de detalhes, o arquiteto deve avaliar se o nível atual de abstração contém ou não informações suficientes para guiar o time de desenvolvimento na implementação dos requisitos de qualidade do software. Devemos ainda observar que os dois extremos da

condição de parada podem trazer desvantagens: se as informações presentes na arquitetura são insuficientes, a liberdade proporcionada ao design de baixo nível pode resultar numa solução que não implementa os requisitos de qualidade esperados. Por outro lado, se são excessivas, a arquitetura pode: (1) custar mais tempo do que o disponível para ser projetada; (2) desmotivar o time de desenvolvimento, por "engessar" o design de baixo nível pela grande quantidade de restrições; e (3) ser inviável, por ter sido projetada sem o conhecimento que muitas vezes só pode ser obtido durante o processo de implementação[2].

A outra estratégia, mais usada por quem possui experiência no domínio do problema, é a *bottom-up*. Esta estratégia consiste em definir elementos arquiteturais básicos e com maior nível de detalhe (serviços ou funções de infraestrutura, por exemplo), e compor serviços presentes em maiores níveis de abstração a partir desses elementos. A experiência no domínio do problema é necessária justamente na definição dos elementos mais detalhados, ou seja, experiência é necessária para definir o nível de abstração mais baixo que servirá de ponto de partida do processo de design. Nesta estratégia, detalhes excessivos ou insuficientes no nível mais baixo de abstração trazem as mesmas desvantagens já apresentadas quando falamos sobre o ponto de parada da estratégia *top-down*.

7.1.2 Separação de preocupações

A separação de preocupações é a divisão do design em partes idealmente independentes. Entre estas partes, podemos citar aspectos funcionais e não-funcionais do sistema. Os aspectos funcionais, como é de se esperar, são o que o sistema é capaz

[2]Devemos nos lembrar que alguns requisitos de qualidade não são completamente conhecidos em etapas iniciais do ciclo de desenvolvimento. Por exemplo, a tolerância a faltas ou o tempo de recuperação podem ser muito dependentes da solução de design de baixo nível.

de fazer. Já os não-funcionais são os aspectos de qualidade do sistema, como desempenho, segurança, monitoração, etc. A separação dos diferentes aspectos permite que cada uma das partes seja um problema de design a ser resolvido de forma independente, permitindo maior controle intelectual por parte do arquiteto, uma vez que agora ele só precisa se focar em um aspecto da arquitetura de cada vez.

Vale observar que a separação completa das diferentes preocupações (ou dos diferentes aspectos) da arquitetura do software é o caso ótimo da aplicação deste princípio, mas não é o caso comum. Isto ocorre porque, como já vimos anteriormente, diferentes funcionalidades e qualidades do software se relacionam entre si. Portanto, apesar de ser vantajoso pensar na solução de design de cada aspecto separadamente, o arquiteto deve também projetar a integração desses aspectos. Esta integração é fundamental por dois motivos. O primeiro, mais óbvio, é que o software é composto por seus aspectos trabalhando em conjunto – e não separadamente. Já o segundo motivo é que a própria integração influencia nas diferentes soluções de design dos aspectos do software. Por exemplo, aspectos de armazenamento devem estar de acordo com aspectos de segurança do software, ou aspectos de desempenho devem trabalhar em conjunto com aspectos de comunicação ou mesmo localização dos elementos da arquitetura.

7.1.3 Padrões e estilos arquiteturais

Outro princípio muito usado durante o processo de design arquitetural é o uso de padrões. Os padrões podem ser considerados como experiência estruturada de design, pronta para ser reusada para solucionar problemas recorrentes. Um padrão de design arquitetural define elementos, relações e regras a serem seguidas que já tiveram sua utilidade avaliada em soluções de problemas passados.

A principal diferença entre um padrão arquitetural[3] e um padrão de design é que o primeiro lida com problemas em nível arquitetural, se tornando assim mais abrangente no software. Por outro lado, a aplicação de um padrão de design tem efeito mais restrito na solução. Mais uma vez, devemos lembrar que essa divisão não é absoluta e que podemos encontrar padrões inicialmente descritos como arquiteturais tendo efeito apenas local no design e vice-versa.

De acordo com McConnell no livro *Code Complete*[4], podemos citar os seguintes benefícios do uso de padrões em um projeto:

- *Padrões reduzem a complexidade da solução ao prover abstrações reusáveis.* Um padrão arquitetural já define elementos, serviços e relações arquiteturais, diminuindo assim a quantidade de novos conceitos que devem ser introduzidos à solução.
- *Padrões promovem o reuso.* Como padrões arquiteturais são soluções de design para problemas recorrentes, é possível que a implementação (parcial ou total) do padrão já esteja disponível para reuso, facilitando o desenvolvimento.
- *Padrões facilitam a geração de alternativas.* Mais de um padrão arquitetural pode resolver o mesmo problema, só que de forma diferente. Portanto, conhecendo diversos padrões, um arquiteto pode avaliar e escolher qual ou quais padrões irão compor a solução do problema, considerando os benefícios e analisando as desvantagens proporcionadas por eles.
- *Padrões facilitam a comunicação.* Padrões arquiteturais facilitam a comunicação da arquitetura porque descrevem conceitos e elementos que estarão presentes no

[3] Também chamado de estilo arquitetural.

[4] McConnell, S. *Code Complete.* Microsoft Press, Segunda edição, Junho 2004.

design. Portanto, se uma solução de design contém padrões que são conhecidos por todos os participantes da comunicação, os elementos e conceitos definidos pelos padrões não precisam ser explicitados, uma vez que os participantes já devem conhecê-los também.

A seguir, citamos alguns padrões arquiteturais que foram popularizados no livro *Pattern-Oriented Software Architecture*[5], de Buschmann *et al*:

***Layers* (ou Camadas)::** este padrão define a organização do software em serviços agrupados em *camadas de abstração*. As camadas são relacionadas de modo que cada uma só deve se comunicar com a camada adjacente acima ou abaixo dela. Se apresentamos graficamente as camadas empilhadas, as camadas dos níveis superiores apresentam um nível de abstração maior, mais próximas aos serviços disponíveis aos usuários. Enquanto isso, nas camadas inferiores, temos serviços mais básicos, normalmente de infraestrutura, e que servem para compor os serviços de camadas mais acima. Como exemplo de arquitetura que usa este padrão, podemos citar a arquitetura da pilha de protocolos TCP/IP. Ela é organizada em cinco camadas, sendo elas: Aplicação, Transporte, Rede, Enlace e Física.

***Pipes & Filters*::** este padrão organiza o software para processar fluxos de dados em várias etapas. Dois elementos básicos são definidos: os chamados *filters*, que são os elementos responsáveis por uma etapa do fluxo de processamento; e os chamados *pipes*, que são os canais de comunicação entre dois *filters* adjacentes. Note que a arquitetura pode conter diferentes *pipes* e *filters*, de

[5]Buschmann, F. *et al*, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, Agosto 1996.

modo que possam reusados e recombinados para diferentes propósitos. O exemplo canônico de uso do padrão *Pipes & Filters* é a arquitetura de um compilador, que pode ser dividida nos seguintes *filters*: analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário e otimizador, que são conectados por diferentes *pipes*. Entre eles, encontramos o *pipe* que conecta o analisador léxico ao sintático e que transmite um fluxo de *tokens*; o *pipe* que transporta a árvore de derivação sintática do analisador sintático para o analisador semântico; o *pipe* que transporta a árvore de sintaxe do analisador semântico para o gerador de código intermediário; e, por fim, o *pipe* que conecta o gerador de código intermediário ao otimizador.

***Model-View-Controller*::** este padrão, por sua vez, divide a arquitetura em três elementos distintos: a lógica de negócio (ou *model*), que representa as funcionalidades e os dados do sistema; visões (ou *views*), que representam a forma de exibir o estado da lógica de negócio ao usuário; e os controladores (ou *controllers*), que são responsáveis pela entrada de dados dos usuários. O padrão também define que deve existir um mecanismo de propagação de mudanças, de forma que a interface com o usuário (composta das visões e dos respectivos controladores) se mantenha consistente com a lógica de negócio. Este padrão é comum em sistemas interativos e foi também popularizado em sistemas *web* por meio de *frameworks*, a exemplo de *JSF*[6], *Struts*[7] e *Spring MVC*[8].

[6] *JavaServer Faces Technology (JSF)*: http://java.sun.com/javaee/javaserverfaces/ (<http://java.sun.com/javaee/javaserverfaces/>)

[7] *Apache Struts*: http://struts.apache.org/ (<http://struts.apache.org/>)

[8] *Spring Framework*: http://www.springsource.org/ (<http://www.springsource.org/>)

***Microkernel*::** este padrão é a base de arquiteturas extensíveis orientadas a *plugins*. Ele define um elemento arquitetural que será o núcleo do sistema e elementos chamados pontos de extensão. Este núcleo provê serviços de infraestrutura para compor as funcionalidades mais básicas do sistema e um serviço de registro e configuração de componentes em tempo de execução. O serviço de registro e configuração tem como responsabilidade a adição de novas funcionalidades a partir dos pontos de extensão pré-definidos. Estes pontos de extensão servem para guiar e restringir os tipos de funcionalidades a serem adicionadas. Como exemplo de aplicação do padrão *Microkernel*, podemos citar o sistema operacional *MINIX*[9], o ambiente de desenvolvimento *Eclipse*[10] e diversos sistemas de manipulação de imagens que são extensíveis por meio de *plugins*, como o *GIMP*[11] e o *ImageJ*[12].

7.2 Táticas de Design

Por meio da aplicação de padrões, somos capazes de reusar a experiência de outros projetistas por meio de soluções estruturadas de design. No entanto, há outra forma de reuso de experiência de design e que não é propriamente definida como padrões. Esta forma é chamada de tática de design e, apesar de cada tática ter objetivos bem definidos, seu conteúdo é menos estruturado, normalmente contendo apenas ideias ou dicas de projeto que ajudam na implementação de atributos de qualidade. A principal diferença entre táticas e padrões de

[9] *MINIX*: http://www.minix3.org/ (<http://www.minix3.org/>)
[10] *Eclipse*: http://www.eclipse.org (<http://www.eclipse.org/>)
[11] *The GNU Image Manipulation Program* (GIMP): http://www.gimp.org (<http://www.gimp.org>)
[12] *ImageJ - Image Processing and Analysis in Java*: http://rsbweb.nih.gov/ij/ (<http://rsbweb.nih.gov/ij/>)

design é que, ao contrário dos padrões, as táticas não necessariamente descrevem elementos arquiteturais que devem existir na solução. Desta maneira, é responsabilidade do arquiteto defini-los de forma a seguir as dicas contidas nas táticas.

Ao aplicar as táticas ao design, assim como durante a aplicação de padrões, o arquiteto deve também considerar os *trade-offs* existentes: por um lado, uma tática pode aumentar o grau de atendimento a um atributo de qualidade, mas, por outro lado, pode afetar negativamente outros atributos. Por isso, para facilitar a avaliação dos *trade-offs* durante o design, apresentaremos algumas táticas de acordo com as qualidades que elas implementam, mas também seremos explícitos sobre o que é afetado negativamente.

A seguir, apresentamos táticas de design de acordo com os seguintes atributos de qualidade:

- desempenho e escalabilidade;
- segurança;
- tolerância a faltas;
- compreensibilidade e modificabilidade; e
- operabilidade.

7.2.1 Desempenho e escalabilidade

Para melhorar desempenho de uma aplicação ou facilitar a adição de recursos computacionais para atender a uma maior demanda, podemos citar as seguintes táticas arquiteturais.

7.2.1.1 Não mantenha estado

Se os elementos da arquitetura são projetados de forma a não manter estado (*stateless*), ou seja, que eles sejam capazes de realizar suas funções apenas com os parâmetros presentes nas

requisições, fica mais fácil replicá-los para dividir a carga de requisições entre as réplicas. Basta apenas que seja definido uma balanceador de carga para distribuir as chamadas entre estes elementos. Note que se a demanda aumenta, pode-se também aumentar o número de elementos *stateless* para suprimir a demanda sem muito esforço. Basta então informar ao balanceador sobre os novos elementos para que ele os considere na distribuição de novas requisições.

É importante observar que nem todos os elementos arquiteturais podem ser *stateless*. Por exemplo, elementos de dados essencialmente mantêm estado (e, portanto, são *stateful*). Assim, é possível que, em algum ponto da arquitetura, os diversos elementos *stateless* precisem de dados ausentes nos parâmetros das requisições e portanto terão que fazer novas requisições aos elementos *stateful*. Se os elementos que mantêm estado não forem capazes de responder a esta carga de novas requisições, eles se tornarão o gargalo da arquitetura, prejudicando o desempenho de todo o sistema.

7.2.1.2 Partição de dados

Para melhorar o desempenho e a escalabilidade de elementos de dados, podemos dividir o conjunto de dados entre elementos de execução. Cada um destes elementos que possui *parte* dos dados é chamado de partição (ou *shard*). Há duas técnicas de partição de dados que merecem ser citadas: a partição horizontal e a partição vertical.

Primeiro, vamos apresentar a partição horizontal por meio de um exemplo. Se pensamos em dados relacionais, que estão organizados em linhas e colunas, a partição horizontal é a divisão em grupos de linhas entre os elementos arquiteturais de dados em execução. Por exemplo, se temos um banco de dados com dois milhões de usuários e temos dois servidores, A e B, execu-

tando esse banco de dados, os usuários com índices de zero a um milhão devem estar localizados no servidor A e o restante dos usuários devem estar localizados no servidor B. A partir desta divisão, para um cliente do banco de dados encontrar as informações de um dado usuário, agora ele deve ser capaz de localizar em qual servidor os dados estão de acordo com o índice que procura. Note que isso é uma forma de dividir a carga de requisições entre elementos de execução, mesmo usando elementos *stateful*.

Já a partição vertical consiste na seleção de algumas colunas do modelo de dados para serem servidas por elementos de execução diferentes. Assim, se temos novamente os servidores A e B, informações sobre todos os usuários estão em ambos os servidores. No entanto, informações mais requisitadas (por exemplo, nome do usuário e grupo de permissões o qual ele pertence no sistema) podem ser encontradas no servidor A, que dispõe de hardware melhor, enquanto informações menos requisitadas podem ser encontradas no servidor B. Da mesma forma que no caso anterior, o cliente deve ser capaz de localizar em qual servidor os dados estão. Só que agora, a localização é feita de acordo com o tipo de dados requisitados e não o seu índice.

7.2.1.3 *Caching*

Em um sistema, existem algumas informações que são mais requisitadas que outras. Por exemplo, a página de alguém muito popular numa rede social ou as notícias de primeira página de um portal de notícias. Portanto, podemos nos aproveitar desta característica ao projetar sistemas.

Se algumas informações são muito mais requisitadas que outras, o desempenho aparente de um sistema pode ser melhorado se conseguirmos servir essas informações com melhor desem-

penho. Uma forma de conseguir isso é usando um *cache*. Um *cache* é um elemento arquitetural capaz de servir informações com maior desempenho do que o elemento de dados que guarda essas informações originalmente. Portanto, ao requisitar alguns dados, o cliente pode primeiro requisitar ao *cache*. Caso o *cache* possua os dados requisitados, eles serão retornados mais rapidamente do que se o cliente tivesse requisitado apenas ao elemento de dados original. No entanto, precisamos observar que para desempenhar melhor do que os servidores de dados, o *cache* normalmente armazena um conjunto limitado de dados. Esta limitação o obriga a implementar as chamadas políticas de *caching*, que são diferentes formas de comportamento para maximizar a quantidade de "acertos" nas requisições de disponibilidade de informação e manter a consistência entre o *cache* e o elemento de dados original.

7.2.1.4 Táticas de processamento

Entre as táticas de processamento para melhorar o desempenho da aplicação (em oposição às táticas de dados vistas anteriormente: partição de dados e *caching*), podemos citar: partição, paralelização e distribuição de processamento.

A partição de processamento é a divisão do processamento entre elementos arquiteturais distintos para tirar proveito das características de cada elemento de execução do software. Um exemplo simples é distribuir um grande processamento de dados entre os elementos da arquiteturais *mais próximos* a esses dados, com a finalidade de evitar ao máximo a transferência de arquivos. Assim, a característica do elemento de execução procurada para realizar a distribuição é se o elemento possui ou não os dados necessários para o processamento. Por exemplo, se observarmos a arquitetura de um sistema de processamento de grandes conjuntos de dados chamado *MapReduce*[13] (ou de

[13] A arquitetura do *MapReduce* é brevemente apresentada por Dean e

sua implementação *open source*, o *Hadoop*[14]), percebemos que ele divide o processamento em tarefas menores e tenta associar cada tarefa ao processador que esteja mais próximo dos dados necessários. Com esta política de atribuição de tarefas, o *MapReduce* consegue processar grandes massas de dados em tempo relativamente pequeno.

Já a paralelização de processamento consiste em permitir que linhas de execução independentes, por exemplo, chamadas de usuários diferentes em um sistema web, ocorram simultaneamente. Essa paralelização pode ser realizada de diferentes maneiras: em diferentes *threads* dentro de um mesmo processo, em diferentes processos dentro de um mesmo sistema operacional e em diferentes elementos de execução de um sistema (tipicamente, em diferentes servidores). Esta paralelização melhora o desempenho porque aumenta a vazão de respostas e pode utilizar recursos, inicialmente, ociosos.

Por fim, há a distribuição de processamento ao longo do tempo. Esta tática consiste em permitir que algumas tarefas de processamento requisitadas pelo usuário não sejam executadas sincronamente e, portanto, não fazendo com que ele espere pelo processamento de algo que não utilizará no momento. Assim, aumentamos o desempenho aparente do software. Um exemplo de distribuição de processamento ao longo do tempo é o de tratamento de imagens em sistemas de redes sociais. Quando um usuário faz o *upload* de uma imagem, essa imagem precisa ser otimizada para ocupar menos espaço de armazenamento no sistema. No entanto, este tratamento não é feito de forma síncrona, ou seja, quando o usuário envia a imagem, mas sim é agendado para ser executado em algum momento no futuro.

Ghemawat no artigo *MapReduce: Simplified Data Processing on Large Clusters*[34].

[14] *Apache Hadoop*: http://hadoop.apache.org/ (<http://hadoop.apache.org/>)

7.2.1.5 Menos camadas de abstração

Apesar de projetar um sistema em diversas camadas de abstração melhorar o reuso (pela possibilidade das camadas serem reusadas), o entendimento (porque diferentes camadas representam diferentes níveis de abstração, facilitando o controle intelectual da complexidade) e até mesmo a testabilidade do sistema (dado que as camadas podem ser desenvolvidas e testadas separadamente), a presença de muitas camadas em um sistema pode prejudicar seu desempenho. Isto ocorre porque quanto mais camadas de abstração existem no design, principalmente se desnecessárias, mais recursos serão consumidos. Entre os recursos consumidos, podemos citar a memória, uma vez que mais camadas de implementação significam mais camadas a serem carregadas durante a execução, e mais ciclos de processamento, para realizar a comunicação entre diferentes camadas.

7.2.1.6 Desvantagens das táticas de desempenho e escalabilidade

Podemos observar que as táticas que acabamos de apresentar aumentam a complexidade da arquitetura, uma vez que apresentam novos elementos tanto em nível de design, quanto em nível de execução. Em nível de design, os novos elementos podem prejudicar a modificabilidade e a compreensibilidade do software, dado que adicionam novas relações e conceitos e até sugerem a diminuição dos níveis de abstração. Já em nível de execução, novos elementos podem dificultar: a segurança, porque agora os dados estarão ainda mais distribuídos no sistema e mais entidades poderão acessá-los; a tolerância a falhas, porque podem surgir mais pontos únicos de falhas; e a operabilidade, considerando que os novos elementos de execução impõem mais tarefas de configuração.

7.2.2 Segurança

Para implementar a segurança em um sistema de software, o arquiteto deve conhecer, além de técnicas de autorização, autenticação, criptografia e auditabilidade, os seguintes princípios.

7.2.2.1 Princípio do menor privilégio

O princípio do menor privilégio consiste em garantir ao usuário, cliente do software ou módulo do sistema apenas os privilégios necessários para que sejam capazes de concluir suas tarefas. Assim, caso este usuário, cliente ou módulo sejam comprometidos (passem a se comportar de forma nociva ao sistema), a quantidade de dano que poderão causar ao sistema será limitada.

7.2.2.2 Princípio da falha com segurança

O princípio de falha com segurança (*fail-safe*) é o de garantir que em caso de qualquer problema, seja de comunicação, autenticação ou falta em um serviço, o comportamento padrão seja um comportamento *seguro*. Por exemplo, se um usuário com privilégios de acesso tenta ler um arquivo privado e o sistema de autorização está indisponível, o comportamento padrão do sistema de leitura deve ser o de negar o acesso ao arquivo. Dessa maneira, mesmo que usuários autorizados sejam privados do acesso aos seus arquivos, os não-autorizados não conseguirão acesso indevido. O mesmo princípio deve ser aplicado, por exemplo, em sistemas de controle de tráfego. Se os indicadores de estado dos semáforos estão com problemas, os semáforos devem falhar no estado "pare", uma vez que fazer com que todos os veículos parem nas vias de um cruzamento é mais seguro do que fazer com que mais de uma via seja indicada para seguir.

7.2.2.3 Princípio da defesa em profundidade

O princípio da defesa em profundidade sugere que a arquitetura deve aplicar diferentes técnicas de segurança em diferentes níveis do software. Por exemplo, um cliente autenticado do software deve não só ser autorizado a chamar uma função, mas a função chamada deve também ser autorizada a acessar as informações necessárias para o dado cliente. Esta técnica tanto permite que medidas de segurança mais específicas ao contexto possam ser utilizadas, quanto permite manter a segurança do software mesmo durante a falha de alguma medida de segurança adotada.

7.2.2.4 Desvantagens das táticas de segurança

Podemos observar que, assim como as táticas de desempenho e escalabilidade, as táticas de segurança aumentam a complexidade da arquitetura. Isto ocorre porque também adicionam novos elementos arquiteturais à solução. Estes novos elementos, por serem novos conceitos, prejudicam a compreensibilidade do sistema em tempo de design e a operabilidade durante a execução. Além disso, as táticas de segurança também requerem a execução de passos adicionais de processamento (por exemplo, criptografar uma mensagem ou checar se senha inserida é válida), o que prejudica o desempenho da aplicação.

7.2.3 Tolerância a Faltas

A área de sistemas distribuídos contribui com muitas técnicas que podem ser aplicadas à arquitetura para que os sistemas sejam projetados para serem mais tolerantes a faltas. Entre estas técnicas, podemos citar as seguintes.

7.2.3.1 Evitar ponto único de falhas

Se muitas funcionalidades dependem de apenas um serviço que executa em apenas um recurso computacional, todo o sistema estará comprometido se esse único serviço falhar. Este único serviço ou recurso computacional no qual o sistema depende é o que chamamos de ponto único de falhas. Portanto, para que o software não seja completamente dependente de um único elemento, o arquiteto deve se preocupar em evitar os pontos únicos de falhas a partir do design. Para isso, ele pode distribuir responsabilidades entre diferentes elementos da arquitetura ou mesmo replicar processamento, de forma que o ponto único seja eliminado.

7.2.3.2 Partição de dados

Já mostramos que a partição de dados é benéfica para o desempenho e a escalabilidade do sistema. No entanto, ao particionarmos os dados por diversos elementos de armazenamento, distribuímos também as responsabilidades do servidor de dados. Portanto, se um dos elementos de armazenamento falha, ainda podemos ter o sistema disponível para parte dos usuários (aqueles os quais as informações ainda estão disponíveis por meio dos elementos de armazenamento que não falharam).

7.2.3.3 Partição e distribuição de processamento

Obtemos benefícios semelhantes aos de particionar os dados quando particionamos e distribuímos processamento por diferentes elementos da arquitetura. Diferentes responsabilidades atribuídas a diferentes elementos da arquitetura permitem que o software continue funcionando, mesmo que parcialmente, em caso de faltas.

Além disso, quando usamos processamento síncrono, amarramos a confiabilidade no processamento aos dois ou mais el-

ementos que estão relacionados sincronamente. Por exemplo, se o elemento A realiza uma função que precisa chamar uma função síncrona no elemento B, a função de A só será executada com sucesso caso B também esteja disponível. No entanto, se a chamada a B for assíncrona, a função chamada em A pode ser executada com sucesso mesmo que B esteja indisponível temporariamente. Dessa maneira, assim que B estiver novamente disponível, sua função poderá ser executada.

7.2.3.4 Redundância

Não só podemos distribuir diferentes responsabilidades de processamento a diferentes elementos da arquitetura, como também podemos atribuir a *mesma* responsabilidade a diferentes elementos. Assim, durante a execução, em caso de qualquer problema com um dos responsáveis, outro pode assumir seu lugar e retornar corretamente a resposta. Isso é o que chamamos de atribuir redundância a alguns elementos da arquitetura, sejam elementos de dados ou de processamento. Vale observar que não basta apenas replicar a responsabilidade do elemento em questão, mas decidir (1) se o elemento redundante ficará sempre ativo ou apenas entrará em execução quando a falha do original for identificada, (2) como as falhas serão identificadas durante a execução e (3) como os clientes do elemento que falhou redirecionarão suas chamadas para o elemento redundante.

7.2.3.5 Desvantagens das táticas de tolerância a faltas

Como as táticas de tolerância a faltas se aproveitam de algumas táticas de desempenho e escalabilidade, elas proporcionam as mesmas desvantagens em relação à compreensibilidade, modificabilidade e operabilidade, uma vez que aumentam a complexidade da solução de design.

7.2.4 Compreensibilidade e Modificabilidade

Algumas técnicas que aumentam a compreensibilidade e a modificabilidade da arquitetura já foram mencionadas anteriormente:

- uso de camadas de abstração;
- separação de preocupações;
- aplicação de padrões;
- alta coesão e baixo acoplamento.

No entanto, não discutimos as desvantagens comuns a essas técnicas. Por ser comum que ambos os atributos sejam alcançados por meio da abstração de detalhes e que a abstração leva à adição de novas camadas de implementação, podemos notar que as técnicas mencionadas anteriormente necessitam de mais recursos computacionais para a execução, afetando negativamente o desempenho. No entanto, ao termos processadores e canais de dados cada vez mais rápidos, além de memória e sistemas de armazenamento cada vez mais baratos, o efeito negativo causado por essas técnicas pode ser irrisório comparado ao benefício da compreensibilidade e da modificabilidade no processo de desenvolvimento.

7.2.5 Operabilidade

Por fim, para proporcionar operabilidade ao sistema de software, o arquiteto deve aplicar as seguintes técnicas durante o design da arquitetura.

7.2.5.1 Monitoração e análise do estado do sistema

O operador só é capaz de agir sobre o software, se ele possuir informações sobre seu estado interno. Para isso, é vantajoso que a arquitetura permita a monitoração do estado de seus elementos mais importantes durante a execução. Note que

em um grande sistema, o conjunto de elementos monitorados pode ser grande, gerando assim uma grande massa de dados de monitoração. Portanto, a monitoração pode ser tornar um problema, uma vez que a geração e o consumo dos dados pode necessitar de muitos recursos computacionais (canal de comunicação, caso os dados sejam transferidos entre elementos do sistema, e armazenamento, caso os dados sejam armazenados, e processamento, para extrair informações dos dados). Portanto, a arquitetura deve proporcionar meios de geração e análise dos dados de monitoração, mas deve também implementar meios de agregação e compactação dos dados de forma que poupem o consumo de recursos computacionais.

7.2.5.2 Computação autonômica

Uma forma ainda mais eficiente de proporcionar operabilidade ao software é a de delegar tarefas que antes seriam de responsabilidade do operador ao próprio software. Portanto, permitir que o software seja capaz de pôr ou retirar de execução servidores, realizar *backups*, ou realizar outras atividades para a melhoria da qualidade de serviço. Realizar automaticamente estas e outras atividades baseadas apenas no estado atual do sistema e sem intervenção humana é o que chamamos de computação autonômica. Para permitir a adição de aspectos de computação autonômica ao software, sua arquitetura deve estar preparada de forma que dados sobre o estado atual do sistema não sejam apenas coletados, mas também sejam analisados automaticamente e os resultados dessa análise sejam capaz de ativar automaticamente tarefas de administração do sistema.

7.2.5.3 Desvantagens das técnicas de operabilidade

Como já mencionamos anteriormente, a monitoração e a análise do estado atual do sistema podem consumir muitos re-

cursos computacionais, impactando negativamente no desempenho. Por outro lado, ao possibilitarmos a análise do software em tempo de execução, podemos identificar problemas inicialmente desconhecidos na arquitetura, como gargalos de desempenho ou pontos únicos de falhas. Com estes problemas identificados, o arquiteto pode então corrigi-los na arquitetura, melhorando assim o desempenho e a tolerância a faltas do software.

7.3 Resumo

Este capítulo expôs o que um arquiteto deve saber em relação às técnicas e princípios de design arquitetural. Devemos admitir que seu objetivo é ambicioso, uma vez que que existem muitos livros e artigos de Design de Software sobre o mesmo assunto. No entanto, a maioria dos livros e artigos disponíveis não são explicitamente escritos sobre Arquitetura de Software ou não têm como público-alvo o leitor ainda inexperiente. Daí nossa tentativa de preencher esta lacuna.

Ao final deste capítulo, esperamos que o leitor conheça os seguintes princípios de design arquitetural:

- uso da abstração ou níveis de complexidade;
- separação de preocupações; e
- uso de padrões e estilos arquiteturais.

Mas, além disso, esperamos que o leitor também reconheça algumas táticas que implementam os seguintes atributos de qualidade:

- desempenho e escalabilidade;
- segurança;
- tolerância a faltas;
- compreensibilidade e modificabilidade; e

- operabilidade.

Para informações mais detalhadas sobre os princípios e técnicas apresentados, deixamos uma lista de referências para estudos posteriores.

7.4 Referências

7.4.1 Abstração e separação de preocupações

Sobre os benefícios e aplicação da abstração e separação de preocupações no design de software, recomendamos a leitura do livro *Code Complete*[74], de McConnell. Além dele, podemos citar os seguintes artigos sobre o assunto: *The Structure of The THE-multiprogramming System*[37], de *Dijkstra*, e o *On The Criteria to Be Used in Decomposing Systems Into Modules*[81], de Parnas.

7.4.2 Padrões e estilos arquiteturais

Há diversos padrões e estilos arquiteturais, inclusive catalogados de acordo com seus objetivos. Apesar de termos citado apenas quatro padrões que foram inicialmente descritos por Buschmann, existem muito mais padrões descritos por este autor e outros autores na série de livros *Pattern-Oriented Software Architecture*[27], [95], [24], [25]. Recomendamos também sobre o assunto os livros *Patterns of Enterprise Application Architecture*[44], escrito por Fowler, e o *Software Architecture in Practice*[12], escrito por Bass *et al.*

7.4.3 Técnicas arquiteturais

Sobre técnicas arquiteturais, podemos citar o livro *Beautiful Architecture*[101], editado por Spinellis e Gousios. Ele mostra

na prática a aplicação de diversas técnicas para o alcance de requisitos de qualidade por meio do design arquitetural. Sendo menos prático, porém mais abrangente na exposição de técnicas arquiteturais, podemos citar tanto o livro *Software Architecture: Foundations, Theory, and Practice*[103], de Taylor *et al*, quanto o livro *Software Systems Architecture*[91], de Rozanski e Woods. O livro *The Art of Systems Architecting*[68], de Maier e Rechtin, descreve poucas (porém valiosas) técnicas de arquitetura de software. Neste livro, as técnicas são chamadas de heurísticas.

Podemos ainda mencionar alguns artigos sobre desempenho de software em geral: *Performance Anti-Patterns*[96], de Smaalders; sobre replicação de dados: *Optimistic Replication*[94], de Saito e Shapiro; e sobre segurança: *In Search of Architectural Patterns for Software Security*[93], de Ryoo *et al*.

Por fim, mencionamos dois blogs que contêm muitas descrições de problemas arquiteturais reais e como foram resolvidos na indústria: o *HighScalability.com*[49] e o *Engineering @ Facebook*[106].

Chapter 8

Documentação da Arquitetura[1]

Após entendermos os conceitos e a importância e termos noções de design de arquitetura de software, precisamos saber como capturar a informação do projeto e documentá-lo. Para isso, introduzimos os conceitos de visões e de pontos de vista arquiteturais, que facilitam a documentação por mostrar diferentes dimensões que uma arquitetura apresenta. Este capítulo não dita uma única linguagem ou modelo de documentação de arquitetura, mas apresenta exemplos de como fazê-lo.

Este capítulo tem como objetivo fazer com que o leitor seja capaz de entender que:

- O documento de arquitetura auxilia no processo de design, é uma ferramenta de comunicação entre stakeholders e pode servir de modelo de análise do software;
- Toda informação presente numa arquitetura é uma decisão arquitetural;

[1] This content is available online at <http://cnx.org/content/m17525/1.6/>.

- Decisões arquiteturais podem ser existenciais, descritivas ou executivas;
- Decisões arquiteturais se relacionam, podendo restringir, impedir, facilitar, compor, conflitar, ignorar, depender ou ser alternativa a outras decisões arquiteturais; e
- Um único diagrama não é suficiente para conter a quantidade de informação que deve ser mostrada por um arquiteto. Por isso, a necessidade de múltiplas visões arquiteturais;

8.1 Arquitetura e Documento da Arquitetura

A arquitetura de um software existe independente dela ser projetada ou documentada. No entanto, ao deixarmos a arquitetura simplesmente "emergir" a partir do software, ou seja, evoluir ao longo do processo de desenvolvimento sem projeto ou documentação, corremos o risco de não tirar proveito dos benefícios que ela proporciona.

Por outro lado, apenas realizar o design arquitetural e não documentá-lo (ou documentá-lo de forma precária), pode minimizar as vantagens a serem obtidas pela arquitetura. Isto pode ocorrer porque documentar a arquitetura, além de auxiliar o próprio processo de design, também proporciona:

- uma ferramenta de comunicação entre os stakeholders;
- a integridade conceitual ao sistema e ao processo de desenvolvimento;
- um modelo para análise antecipada do sistema; e
- uma ferramenta de rastreabilidade entre os requisitos e os elementos do sistema.

8.1.1 Auxílio ao Processo de Design

Apesar de dividirmos conceitualmente o processo de design do processo de documentação, é comum que ambos aconteçam em paralelo na prática. Quando isto ocorre, a documentação ajuda no design, principalmente no sentido de separação de preocupações.

Ao documentarmos visões arquiteturais diferentes separadamente, preocupamo-nos separadamente com o design de diferentes aspectos do software. Entre os diversos aspectos de um software, podemos citar os aspectos funcionais, de dados, de concorrência, de desenvolvimento, de implantação e operacionais. Esta separação se torna benéfica porque há diferentes linguagens, que podem ser gráficas ou textuais, que melhor se encaixam à descrição de cada aspecto, ajudando não só numa melhor representação, como também numa melhor modelagem e avaliação em relação aos objetivos.

A seguir, vemos dois exemplos que ilustram a documentação de algumas decisões arquiteturais relacionadas a aspectos diferentes do SASF. No p. 191, podemos observar como o SASF está dividido em grandes módulos funcionais e, assim, podemos inferir quais são suas principais funcionalidades e algumas de suas relações entre si. Podemos dizer que as decisões arquiteturais do exemplo são apresentadas sob o ponto de vista funcional do sistema, constituindo uma visão funcional do software. Note também que esta não é a melhor forma de representar, por exemplo, que o desenvolvimento do cadastro dos usuários será terceirizado ou ainda que o serviço de transmissão de vídeos deve executar em 7 servidores durante dias úteis e em 15 servidores nos finais de semana e feriados, quando demanda aumenta.

Exemplo 8.1
[Decisão Arquitetural 001] O SASF é dividido em cinco

grandes módulos funcionais. Cada módulo é responsável por prover um conjunto de funcionalidades relacionadas entre si. Os grandes módulos do SASF são:

- Locadora de Filmes;
- Transmissor de Filmes;
- Motor de Sugestões;
- Cadastro de Usuários;
- Cadastro de Filmes.

As principais funcionalidades providas por cada módulo e suas relações de uso estão descritas no diagrama representado na Figura 8.1.

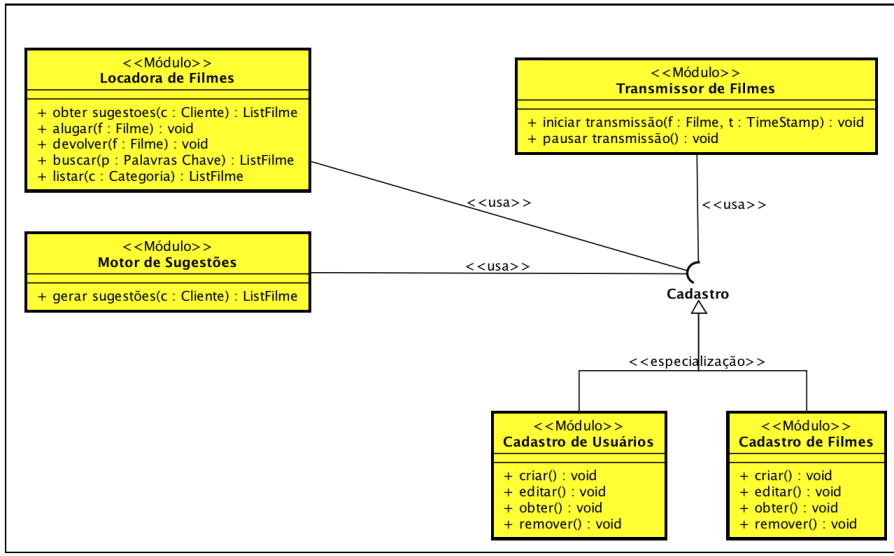

Figura 8.1: Módulos funcionais do SASF. O estereótipo ≪*Módulo*≫ do diagrama indica os módulos funcionais que compõem o sistema. Já o estereótipo ≪*usa*≫ indica relações de uso entre os módulos para que possam implementar suas funções. Por fim, o estereótipo ≪*especialização*≫ indica uma relação de especialização dos dados guardados no elemento responsável pelo cadastro.

Objetivos: A divisão em módulos funcionais possibilita a divisão da implementação entre os times de desenvolvimento de acordo com a especialidade de cada time.

Motivação: As diversas funções a serem providas pelo SASF foram agrupadas em preocupações comuns

(cadastro de dados, locação e transmissão de filmes, e sugestões ao usuário). O cadastro deve ser especializado em dois tipos para dividir o esforço de desenvolvimento: cadastro de filmes e de usuários. O motor de sugestões deve ser alimentado com dados de histórico de locações do usuário e informações sobre a base de filmes no sistema.

No Exemplo 8.2, mostramos o SASF sob um ponto de vista de implantação. Neste exemplo, podemos observar informações de configuração para implantação de serviços para executar o SASF – informações que estavam ausentes no exemplo anterior.

Exemplo 8.2

[Decisão Arquitetural 002] A implantação do módulo que implementa as funcionalidades do serviço de transmissão de filmes deve depender apenas de um parâmetro:

- *endereços.servidores.configuração*: lista de endereços IP dos servidores que constituem o serviço de configuração do SASF.

Os outros parâmetros de configuração devem ser obtidos a partir da comunicação com o serviço de configuração.

Objetivos: Facilitar a operabilidade do sistema.

Motivação: O serviço de configuração do SASF, que é descrito em detalhes na [Decisão Arquitetural 011], é um centralizador da configuração do sistema. Nele, o operador do sistema insere endereços de serviços para que estejam disponíveis para a configuração de uma nova instância. Quando a instância do serviço de transmissão de filmes é iniciada, ela faz requisições ao

serviço de configuração pelos endereços dos serviços de cadastro e dos serviços de armazenamento de filmes, por exemplo.

Exemplo 8.3
[Decisão Arquitetural 002] A implantação do módulo que implementa as funcionalidades do serviço de transmissão de filmes deve depender apenas de um parâmetro:

- *endereços.servidores.configuração*: lista de endereços IP dos servidores que constituem o serviço de configuração do SASF.

Os outros parâmetros de configuração devem ser obtidos a partir da comunicação com o serviço de configuração.

Objetivos: Facilitar a operabilidade do sistema.

Motivação: O serviço de configuração do SASF, que é descrito em detalhes na [Decisão Arquitetural 011], é um centralizador da configuração do sistema. Nele, o operador do sistema insere endereços de serviços para que estejam disponíveis para a configuração de uma nova instância. Quando a instância do serviço de transmissão de filmes é iniciada, ela faz requisições ao serviço de configuração pelos endereços dos serviços de cadastro e dos serviços de armazenamento de filmes, por exemplo.

8.1.2 Ferramenta de Comunicação

Já sabemos que diferentes stakeholders demonstram diferentes preocupações sobre diferentes aspectos do sistema. Como a

documentação da arquitetura versa sobre as muitas preocupações dos stakeholders, ela serve de ferramenta para comunicação, uma vez que provê um vocabulário comum sobre o sistema, além de registrar as relações entre as preocupações e de que forma os eventuais conflitos foram ou devem ser resolvidos.

Note que para servir de ferramenta de comunicação, o documento arquitetural deve ser construído de forma que respeite o conhecimento e as necessidades dos stakeholders. Para que isto seja possível, deve-se conhecer para quem o documento está sendo escrito. Portanto, devemos escrever a arquitetura de forma que possua partes que interessem aos usuários, aos clientes, aos desenvolvedores, aos testadores, ao gerente de projeto ou a outros stakeholders envolvidos no processo.

Por exemplo, os usuários buscam pelas funcionalidades, capacidades e comportamento do sistema, não como ele foi dividido em módulos, como os módulos se comunicam entre si ou se eles foram desenvolvidos do zero ou tiveram partes reutilizadas de outros sistemas. Clientes e gerentes têm alguns interesses semelhantes, como custos de desenvolvimento ou cronograma de lançamento. No entanto, os clientes também se preocupam com o alinhamento do software ao seu negócio, enquanto o gerente procura como minimizar os custos para se adequar ao orçamento, ou como as tarefas de implementação serão divididas entre seu time de desenvolvimento. Finalmente, desenvolvedores e testadores estão interessados em aspectos técnicos do design, por exemplo, qual a divisão em módulos do sistema, quais as alternativas de design disponíveis ou como os atributos de qualidade (desempenho, escalabilidade, tolerância a faltas, etc.) devem ser implementados, cada um motivado pelo seu papel no processo de desenvolvimento.

8.1.3 Integridade Conceitual

Por outro lado, o documento precisa demonstrar consistência entre os diversos aspectos do design da arquitetura para que haja comunicação e integridade conceitual. A consistência é necessária porque, apesar da separação de preocupações ser uma ferramenta poderosa de design, as soluções para as diferentes preocupações trabalham interligadas durante a implementação, ou seja, quando resolvem o problema. Assim, precisamos de integridade em dois níveis: entre os stakeholders e entre os diversos aspectos do documento de arquitetura.

A integridade conceitual entre stakeholders é a defendida por Brooks, em *The Mythical Man-Month*, porque facilita o sucesso no desenvolvimento de sistemas de software. Se os stakeholders – e principalmente os desenvolvedores – não têm em mente o mesmo design que se transformará em produto, são poucas as garantias de que o produto implementado será o projetado. Por isso, no processo, o documento de arquitetura serve para restringir eventuais "deslizes conceituais" em relação ao design arquitetural e prevenir futuras discordâncias entre stakeholders, inclusive de interesses similares. O Exemplo 8.4 ilustra um caso de restrição aos desenvolvedores, mas que é benéfica por proporcionar a integridade conceitual entre times de desenvolvimento. Este caso é a definição das interfaces entre dois serviços presentes no sistema.

Exemplo 8.4

No SASF, se dividirmos os desenvolvedores em mais vários times, é comum que haja uma maior interação entre os membros de um mesmo time do que entre times diferentes. Vamos considerar que delegamos a um time a implementação do serviço responsável pelas funcionalidades do módulo *Cadastro de Filmes*, já apresentado no Exemplo 8.1, e a outro time o módulo *Transmissor de Filmes*. Para que a implementação

dos dois módulos possa ser paralelizada e para que sejam evitadas suposições errôneas ou desnecessárias por parte de cada time sobre outros módulos (e, portanto, seja menos custosa a integração), devemos definir cuidadosamente as interfaces dos módulos, sejam os métodos disponíveis, a forma de comunicação e os tipos de dados usados como parâmetros ou retorno.

A integridade conceitual também se mostra necessária durante a inclusão de novos membros à equipe de desenvolvimento e ao longo da evolução e manutenção do software. Novos membros precisam de uma descrição da arquitetura porque normalmente são inseridos ao time sem qualquer conhecimento prévio do design. Já ao longo da evolução e manutenção do software, o conhecimento das regras de design a serem seguidas se faz necessário para que os requisitos de qualidade permaneçam implementados, mesmo durante mudanças. O exemplo a seguir ilustra uma regra de design para que um software de manipulação de imagens continue exercendo alta portabilidade.

Exemplo 8.5

O software de manipulação de imagens *ImageJ*[2] desempenha bem dois atributos de qualidade: a extensibilidade e a portabilidade. Sua extensibilidade é garantida por ter sua arquitetura ser baseada no uso de *plugins*. Com isso, ele é composto de um núcleo de funcionalidades básicas e provê meios para que novas funcionalidades sejam adicionadas em tempo de execução. Já sua portabilidade é garantida por ele ter seu núcleo e *plugins* implementados usando a linguagem de programação Java, permitindo sua execução em qualquer ambiente que disponha da máquina virtual Java.

Como o código-fonte do *ImageJ* é de domínio público,

[2] *ImageJ - Image Processing and Analysis in Java*: http://rsbweb.nih.gov/ij/ (<http://rsbweb.nih.gov/ij/>)

> qualquer programador pode baixá-lo, usá-lo e escrever novos *plugins* para ele. Inclusive, é possível usar o mecanismo *Java Native Interface* (JNI) para realizar chamadas a bibliotecas escritas em outras linguagens. No entanto, se o programador deseja manter o alto grau de portabilidade do *ImageJ*, ele deve respeitar a regra de design do software que é de nunca realizar chamadas nativas durante a implementação de novas funcionalidades.

Existe também a integridade entre os diversos aspectos da arquitetura no documento ou, em outras palavras, a integridade entre as visões da arquitetura. Este tipo de integridade deve ser mantido para que as partes do design funcionem em conjunto e que, portanto, o design seja passível de implementação. A consistência entre visões se faz necessária por que, apesar da separação de preocupações e de elementos arquiteturais facilitar o design, é em conjunto que eles são construídos e executados. Portanto, se há no documento mais de uma visão sobre os mesmos elementos do sistema, é essencial que na documentação dessas visões exista um mapeamento entre as diferentes representações desses elementos.

O Exemplo 8.6 ilustra a consistência entre visões sobre o armazenamento no SASF. Nele, podemos observar (1) os serviços providos pelo sistema de armazenamento do SASF por meio da visão funcional; (2) que boa parte dos serviços de armazenamento não serão implementados do zero, uma vez que serão obtidos pelo Sistema de Gerência de Banco de Dados adotado, por meio da visão de desenvolvimento; e (3) que o sistema de armazenamento estará executando, no mínimo, em três servidores, por meio da visão de implantação. Note que, se alguma das visões for inconsistente com as outras, podem surgir dúvidas sobre: (1) quais serviços estão disponíveis para quem usa armazenamento, (2) o que será implementado e o que será aproveitado durante a implementação da solução de

armazenamento do SASF e, por fim, (3) que tipo de hardware será necessário para executar a solução de armazenamento.

Exemplo 8.6

Na [Decisão Arquitetural 001], descrita no Exemplo 8.1, apresentamos três módulos que devem lidar diretamente com armazenamento: *Cadastro de Usuários*, *Cadastro de Filmes* e *Transmissor de Filmes*. No entanto, apenas as funções que eles devem implementar foram descritas, não como devem implementar. As decisões a seguir mostram alguns aspectos de como essas funções devem ser implementadas.

(Decisão Arquitetural 002). O armazenamento das informações dos módulos *Cadastro de Usuários* e *Cadastro de Filmes* será realizado usando um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) de modo a permitir criação, edição, obtenção e remoção das entradas.

As informações guardadas no SGBDR são os atributos dos Usuários e Filmes e são essencialmente textuais ou metainformações para localização de arquivos de mídia (fotos ou vídeos). O armazenamento de arquivos de mídia é tratado na [Decisão Arquitetural 003]. Já o armazenamento de arquivos textuais que não são atributos dos Usuários ou Filmes, por exemplo, mensagens para usuários ou comentários sobre filmes é tratado em outra decisão arquitetural que não é descrita aqui.

Objetivo: Permitir a persistência dos atributos dos Usuários e Filmes, facilitando o desenvolvimento.

Motivação: Os atributos de Usuários e Filmes são essencialmente relacionais, se encaixando perfeitamente ao paradigma usado para armazenamento. Além

de ser um paradigma bem conhecido pelo time de desenvolvimento.

(Decisão Arquitetural 003). O armazenamento de arquivos de mídia (fotos de Usuários, fotos de Filmes e arquivos de vídeo) serão armazenados usando uma Rede de Fornecimento de Conteúdo (*Content Delivery Network* ou CDN).

Objetivo: Permitir a persistência de arquivos de mídia, facilitando a implementação e permitindo desempenho e controle de carga.

Motivação: Os arquivos de mídia presentes no SASF são conteúdo estático, que pode ser distribuído por uma CDN e, assim, tirar proveito de replicação, distribuição geográfica e *caching*, sem ter que implementar estas técnicas.

Já as decisões da visão de implantação, devem descrever os servidores que executaram os SGBDR e o serviço que se comunica com a CDN para o envio de arquivos.

8.1.4 Modelo para Análise

A arquitetura é um modelo do sistema, uma vez que descreve suas características. Ao documentar a arquitetura, obtemos um modelo manipulável do sistema que tem utilidade não só ao arquiteto, mas também a outros stakeholders. Com o modelo manipulável, é possível avaliar as decisões arquiteturais registradas e validá-las em relação aos requisitos que o software deve satisfazer. Além disso, o documento pode ainda servir de ferramenta que permita a verificação de se implementação está

de acordo com o design, podendo prevenir eventuais deslizes arquiteturais.

Podemos citar três categorias[3] de validação da arquitetura em relação aos requisitos: análise baseada em inspeções, análise baseada em modelos e análise baseada em simulações. No entanto, a possibilidade de aplicação de uma técnica de uma dada categoria de validação está diretamente ligada à representação usada no documento de arquitetura.

8.1.4.1 Análise baseada em inspeções

Análises baseadas em inspeções são conduzidas por bancas de revisão compostas por vários stakeholders. Entre os stakeholders, podemos encontrar, além do arquiteto e dos desenvolvedores, interessados menos técnicos, como o gerente de projeto e, em alguns casos, o cliente. Durante o processo de inspeção, os stakeholders definem os objetivos da análise e estudam as representações da arquitetura de forma a avaliá-la de acordo com seus objetivos.

Esta categoria de inspeção serve para verificar um conjunto amplo de propriedades da arquitetura e faz uso de múltiplas representações do design, tanto em linguagens formais, quanto informais. Por ser um processo essencialmente manual, é um tipo de análise mais caro do que os de outros, mas que possibilita também a inspeção em busca de qualidades menos formais do software, a exemplo de escalabilidade, manutenibilidade ou operabilidade. Outra vantagem deste tipo de análise é a de permitir o uso de representações informais ou parciais do design da arquitetura, além de possibilitar a análise considerando múltiplos objetivos de múltiplos stakeholders.

[3]Esta divisão foi feita originalmente por Taylor *et al* em *Software Architecture: Foundations, Theory, and Practice* [104].

Como exemplos de análises baseadas em inspeções, podemos citar alguns métodos de avaliação de arquitetura criados pelo *Software Engineering Institute*, da *Carnegie Melon University*: o *Software Architecture Analysis Method* (SAAM), o *Architectural Trade-Off Analysis Method* (ATAM) e o método *Active Reviews for Intermediate Designs* (ARID).[4]

8.1.4.2 Análise baseada em modelos

Análises baseadas em modelos são menos custosas do que as baseadas em inspeções, uma vez que demonstram maior nível de automação. Este tipo de análise utiliza ferramentas que manipulam representações da arquitetura com o objetivo de encontrar algumas de suas propriedades. Para possibilitar a manipulação, as representações devem ser escritas em linguagens formais ou semiformais como ADLs (*architecture description languages* ou linguagens de descrição de arquitetura), como por exemplo, ACME, SADL e Rapide, máquinas de estado finito ou UML.

Esta categoria de inspeção é utilizada na busca de propriedades formais da arquitetura, como corretude sintática ou ausência de *deadlocks* e, apesar de seu alto grau de automação, pode necessitar que o arquiteto guie a ferramenta de inspeção utilizada considerando os resultados parciais. Uma desvantagem desta categoria é seu desempenho na análise de grandes sistemas. Uma vez que as representações da arquitetura podem levar à explosão de estados, a análise de todo o espaço de estados do sistema pode ser inviável. Portanto, é comum que apenas partes da arquitetura sejam analisadas – de preferência partes mais críticas.

Como exemplos de análises baseadas em modelos, podemos

[4]Podemos encontrar a descrição desses métodos no livro *Evaluating Software Architectures*, de Paul Clements *et al* [33].

citar o uso da linguagem Wright para a verificação de ausência de *deadlocks*[5] e o uso da linguagem de modelagem *MetaH* para análise de propriedades confiabilidade e segurança (*safety*)[6].

8.1.4.3 Análise baseada em simulações

Análises baseadas em simulações se utilizam de modelos executáveis da arquitetura para extrair características do software ou de partes dele. Assim como a análise baseada em modelos, esse tipo de análise também se utiliza de ferramentas que automatizam o processo, deixando-o mais barato. No entanto, este tipo de análise produz resultados restritos às propriedades dinâmicas do software e estão sujeitas às imprecisões dos modelos de execução.

Para possibilitar a execução, as representações utilizadas devem ser formais, o que diminui sua aplicação na indústria, mas que proporciona resultados mais precisos em relação às qualidades estruturais, comportamentais e de interação entre as partes do software, como por exemplo qualidades de desempenho e confiabilidade.

Como exemplos de análises baseadas em simulações, podemos citar o uso de simulação de eventos discretos para análise de desempenho ou o uso da ferramenta *XTEAM*[7], que utiliza ADLs e processos de estado finito para diferentes tipos de análises arquiteturais.

[5] Técnica apresentada por Allen e Garlan no artigo *A Formal Basis for Architectural Connection* [7]

[6] Mais informações sobre a linguagem *MetaH* podem ser encontradas no site: http://www.htc.honeywell.com/metah/index.html (<http://www.htc.honeywell.com/metah/index.html>)

[7] A ferramenta *eXtensible Tool-chain for Evaluation of Architectural Models* (XTEAM) é descrita por Edwards *et al* no artigo *Scenario-Driven Dynamic Analysis of Distributed Architectures* [39].

8.1.5 Ferramenta de Rastreabilidade

Por fim, a documentação permite rastreabilidade entre os requisitos e os elementos da arquitetura e implementação que satisfazem esses requisitos. Ao documentar as decisões arquiteturais, registramos (1) seus objetivos, que normalmente são qualidades a serem alcançadas pelo software, e (2) como esses objetivos são alcançados, por meio da descrição dos elementos que compõem o sistema e suas relações e regras de design que devem ser respeitadas durante a implementação. Este registro serve de ponte entre dois extremos do processo de desenvolvimento: os requisitos e a implementação, e assim permite a navegação entre pontos relacionados, sejam eles requisitos, decisões de design ou partes da implementação.

A rastreabilidade nos permite analisar qual o impacto de uma decisão de design, tanto em termos de quais requisitos ela afeta, quanto quais elementos de software ela dita a existência ou, em caso de manutenção, quais elementos são ou devem ser afetados por mudanças nos requisitos ou nas decisões. O exemplo a seguir mostra aspectos de rastreabilidade na documentação da arquitetura do SASF.

Exemplo 8.7

Se observarmos a arquitetura do SASF e procurarmos pelas decisões responsáveis por facilitar a manutenção do sistema, encontraremos entre elas a decisão de divisão do sistema em camadas. Essa decisão sugere uma divisão do sistema em camadas lógicas, mas também influencia na divisão em pacotes, serviços ou mesmo processos. Assim, a satisfação do requisito de manutenibilidade está diretamente ligada à correta divisão das partes do sistema em apresentação, lógica de negócio e persistência.

Da mesma maneira, se partirmos das partes que for-

mam as camadas de apresentação, lógica de negócio e persistência, observaremos que elas estão ligadas à divisão do sistema (e à decisão arquitetural) que se propõe a atender a requisitos de manutenibilidade.

8.2 Decisões Arquiteturais

Em capítulos anteriores, definimos arquitetura de software usando o padrão ISO/IEEE 1471-2000, que diz que ela *é a organização fundamental de um sistema, representada por seus componentes, seus relacionamentos com o ambiente, e pelos princípios que conduzem seu design e evolução.* Após a definição, mencionamos também que a arquitetura é composta de diversas decisões de design (no caso, design de alto-nível ou arquitetural) e que cada decisão contém, ao menos em nível conceitual, uma descrição, objetivos e algum argumento ou motivação. Como a arquitetura é formada por decisões arquiteturais, devemos conhecer os tipos de decisões arquiteturais para então sermos capazes de documentar a arquitetura.

Uma decisão arquitetural, como também já definido anteriormente, *é uma escolha entre as alternativas de design arquitetural, que se propõe a alcançar um ou mais atributos de qualidade do sistema, por meio de estruturas ou regras que ela envolve ou define.* Em outras palavras, podemos dizer que uma decisão arquitetural descreve parte do design, onde essa descrição pode: (1) ditar a existência ou inexistência de partes do sistema, (2) especificar propriedades que, durante a construção, partes do sistema devem satisfazer, ou (3) citar técnicas que devem ser seguidas durante a construção de partes do sistema. Podemos então dividir as decisões arquiteturais em:

- Decisões arquiteturais existenciais (e não-existenciais);
- Decisões arquiteturais de propriedades; e

- Decisões arquiteturais executivas.

8.2.1 Decisões existenciais

Uma decisão existencial é aquela que indica a presença de um ou vários elementos arquiteturais no design e na implementação.

Os elementos arquiteturais já foram apresentados anteriormente, mas vamos relembrá-los aqui. Estes elementos são as partes em que o software é dividido e podem ser classificados em dois tipos: elementos arquiteturais estáticos e elementos arquiteturais dinâmicos. Os elementos estáticos descrevem a estrutura do sistema em tempo de design e são constituídos de elementos de software (por exemplo, classes, pacotes, procedimentos, serviços remotos), elementos de dados (por exemplo, entidades e tabelas de bancos de dados, arquivos ou classes de dados), e elementos de hardware (por exemplo, servidores em que o sistema vai executar ou armazenar dados). Por sua vez, os elementos dinâmicos descrevem o comportamento do sistema em tempo de execução e entre eles podemos incluir processos, módulos, protocolos, ou classes que realizam comportamento. Note que as relações entre os elementos arquiteturais, tanto estáticos quanto dinâmicos, são representadas também como elementos arquiteturais. Estes elementos são chamados de conectores e podem ser associações, composições, generalizações, entre outros.

No Exemplo 8.1, observamos uma decisão arquitetural que divide o SASF em diversos módulos menores, constituindo assim uma decisão existencial que define diversos elementos e as relações entre si.

Já no Exemplo 8.8, observamos uma decisão arquitetural que também dita a presença de elementos na arquitetura do SASF.

No entanto, ao contrário do exemplo citado anteriormente que dita elementos estruturais do software, o Exemplo 8.8 dita elementos comportamentais esperados nele. Assim, podemos encontrar decisões existenciais que sejam decisões estruturais ou comportamentais. As decisões comportamentais são mais relacionadas à implementação dos requisitos de qualidade.

Exemplo 8.8

[Decisão Arquitetural 005] Os dados do *Cadastro de Usuários* e do *Cadastro de Filmes* devem ser particionados horizontalmente.

Objetivo: Distribuir carga, melhorar o desempenho e aumentar o número de pontos de falhas.

Motivação: Ao particionar horizontalmente os dados, permite-se a distribuição da carga de requisições entre vários servidores de armazenamento, que estarão executando instâncias do SGBDR. Com menor carga, o desempenho pode ser melhorado. Além disso, caso uma partição fique inacessível (por falha, por exemplo), parte dos dados ainda estarão acessíveis, não inviabilizando o sistema por completo.

Na prática, é importante observarmos que a divisão entre decisões estruturais e comportamentais não é absoluta. Podemos encontrar decisões que, para descrever um comportamento, necessitem de novas estruturas arquiteturais. Assim, por conveniência, é melhor descrever estas novas estruturas na própria decisão comportamental do que documentar uma nova decisão estrutural. Podemos observar este caso no exemplo anterior, onde descrevemos as partições do conjunto de dados para então descrever o comportamento de particionamento dos dados e assim conseguir algum nível de escalabilidade horizontal.

Por outro lado, há decisões que proíbem a existência de elementos arquiteturais. Essas decisões são chamadas de decisões

não-existenciais e elas servem para restringir as alternativas de design de baixo nível. Alguns padrões arquiteturais, como o 3- *tier* ou mesmo o padrão Camadas, proíbem a comunicação entre alguns dos elementos que eles descrevem, constituindo decisões não-existenciais.

8.2.2 Decisões de propriedades

Decisões de propriedades não determinam a existência de partes do software, mas apresentam alguma qualidade ou característica que uma ou mais partes devem exibir. O papel deste tipo de decisão é o de guiar tanto o design de baixo nível, quanto a implementação, uma vez que descreve os princípios e regras ou restrições de design que devem ser respeitadas ao longo do desenvolvimento.

Os exemplos mais comuns de decisões de propriedades são as decisões sobre preocupações transversais ao software, como por exemplo, decisões de *logging*, decisões de tolerância a faltas ou mesmo decisões sobre a precisão na obtenção dos resultados. Podemos observar uma ilustração mais completa deste tipo de decisão nos exemplos a seguir (Exemplo 8.9 e Exemplo 8.10). Note que, em ambos os exemplos, as decisões não descrevem a existência de elementos que devem estar na arquitetura, mas descrevem as propriedades de elementos arquiteturais que foram descritos em outras decisões.

Exemplo 8.9

[Decisão Arquitetural 008] Os métodos públicos de cada serviço que implementa os módulos descritos na [Decisão Arquitetural 001] devem seguir as seguintes regras de *logging*:

- Deve-se registrar todos os parâmetros das chamadas em nível de *DEBUG*. Este modo deve

poder ser ligado ou desligado em tempo de execução.
- Todas as exceções lançadas devem ser logadas em nível de *ERROR* e registrar os parâmetros usados durante a execução.
- Os tempos de execução de cada chamada ao método devem ser registrados, para possibilitar a monitoração de desempenho do sistema. O canal de *logging* utilizado neste caso deve ser especializado para coleta de métricas de desempenho.

Objetivo: Estas regras facilitam a operabilidade do sistema.

Motivação: O registro dos acontecimentos inesperados no sistema facilita o diagnóstico dos problemas e a possibilidade de aumentar o nível de detalhe dos registros em tempo de execução permite que esse diagnóstico seja mais rápido. Por outro lado, o registro de métricas de desempenho do sistema permite análise de capacidade, podendo indicar se o sistema está precisando de mais recursos computacionais.

Exemplo 8.10
[Decisão Arquitetural 010] Os serviços que implementam os módulos descritos na [Decisão Arquitetural 001] devem ser replicados, evitando assim pontos únicos de falha. Para facilitar a replicação, os módulos não devem manter estado, delegando esta responsabilidade aos serviços de armazenamento.

Objetivo: Replicando instâncias de serviços, elimina-se os pontos únicos de falha, aumentando a confiabilidade do sistema e a tolerância a faltas.

Motivação: Implementando serviços *stateless*, a replicação fica trivial, uma vez que a requisição pode usar

qualquer uma das réplicas ativas. Note que é sempre necessário o registro no balanceador de carga da uma nova réplica em execução. Serviços de armazenamento não podem utilizar esta técnica sem adaptações, uma vez que dados são fundamentalmente *stateful*.

8.2.3 Decisões executivas

A última classe de decisões arquiteturais que apresentamos é a executiva. Este tipo de decisão está mais relacionado ao processo de desenvolvimento do que aos elementos de design. Entre as decisões executivas, podemos encontrar decisões que descrevem: a metodologia utilizada durante desenvolvimento, como o time está dividido durante a implementação do sistema, como o treinamento de novos membros deve ocorrer, ou quais tecnologias e ferramentas devem ser adotadas para auxiliar o processo. Os exemplos a seguir mostram algumas decisões executivas.

Exemplo 8.11
Neste exemplo, apresentamos uma decisão hipotética do software *Vuze*[8].

(Decisão Arquitetural 001). O software será escrito usando a linguagem de programação Java.

Objetivo: Permitir a portabilidade para vários sistemas operacionais.

Motivação: Como um dos objetivos do *Vuze* é alcançar o maior número de usuários possível, não queremos impor a barreira de instalação em um ou outro sistema operacional específico. Uma vez que programas escritos

[8] *Vuze*: http://www.vuze.com/ (<http://www.vuze.com/>)

em Java podem executar em qualquer sistema operacional que seja capaz de executar a Máquina Virtual Java (JVM) e que a maioria dos sistemas para usuários finais já dispõem da JVM, esta linguagem deve ser usada para poupar o trabalho de portar o Vuze para diferentes sistemas.

Exemplo 8.12
[Decisão Arquitetural 011] O time de desenvolvimento será dividido em equipes menores e cada equipe será responsável pela implementação do serviço responsável pelas funcionalidades de módulo descrito na [Decisão Arquitetural 001].

Objetivo: Minimizar o tempo de desenvolvimento.

Motivação: A possibilidade de paralelizar o desenvolvimento pode diminuir o tempo total de construção do software. No entanto, deve-se respeitar as decisões arquiteturais que definem as interfaces entre os módulos, para que sua integração seja facilitada. endexample

8.2.4 Atributos das decisões arquiteturais

No capítulo de fundamentos de arquitetura, mostramos que as decisões arquiteturais devem possuir uma descrição, objetivos e alguma fundamentação. Estes atributos se tornam essenciais ao processo de design das decisões, pois representam, respectivamente, *o que deve ser feito*, *para que deve ser feito* e a *justificativa da solução*. No entanto, há outros atributos que são especialmente úteis quando precisamos documentar as decisões arquiteturais. São eles o escopo, o histórico, o estado atual e as categorias da decisão arquitetural.

Entre as vantagens que eles proporcionam, podemos dizer que esses atributos facilitam a manutenção de um registro histórico das decisões e a rastreabilidade entre requisitos e elementos do software. A seguir, mostramos cada atributo de uma decisão arquitetural separadamente:

8.2.4.1 Descrição

O atributo de descrição, como já mencionamos no capítulo de fundamentos, é simplesmente a descrição da decisão, que mostra o que foi decidido na arquitetura. Na descrição, podemos encontrar (1) quais elementos arquiteturais devem estar presentes, caso seja uma decisão existencial; (2) quais propriedades devem se manifestar nos elementos ou quais regras ou princípios de design devem ser seguidos, caso seja uma decisão de propriedade; ou (3) qual metodologia deve ser seguida, como o time deve ser dividido para a implementação dos módulos ou qual ferramenta deve ser utilizada para integração, caso seja uma decisão executiva.

A descrição pode ser representada usando diversas linguagens, podendo ser textuais ou gráficas e formais ou informais. A escolha da linguagem que será utilizada na descrição depende dos objetivos da decisão e dos stakeholders interessados. Se, entre os seus objetivos, queremos que a decisão permita também geração automática de parte da implementação, análise baseada em modelos ou simulações, ou verificação de conformidade, a descrição deve utilizar linguagens formais ou semiformais que facilitam estas atividades. Por outro lado, se esperamos que a decisão apenas informe que elementos devem estar na arquitetura e suas características, mas não esperamos geração, análise ou verificação automáticas, linguagens semiformais ou mesmo informais podem ser utilizadas, como a língua Portuguesa ou diagramas "caixas e setas", desde que a ambiguidade seja evitada por meio de legendas ou explicações mais detalhadas.

Vale observar que tanto a utilização de linguagens formais, quanto a utilização de linguagens informais na descrição proporcionam vantagens e desvantagens que devem ser consideradas durante o processo de documentação. Ao utilizarmos linguagens formais, permitimos a automatização de processos, que podem poupar bastante trabalho durante o desenvolvimento. Por outro lado, não são todos os stakeholders que as entendem perfeitamente, podendo restringir assim o entendimento da decisão.

As linguagens informais, por sua vez, têm como vantagem a maior facilidade de entendimento por parte dos stakeholders (inclusive não-técnicos, como gerentes, clientes e até usuários). No entanto, o entendimento só é facilitado se a descrição evitar ambiguidades, que são comuns em linguagens informais.

Uma forma de se conseguir mais vantagens nas decisões seria utilizar tanto linguagens formais quanto informais nas descrições das decisões. Nada impede que isso seja feito, obtendo assim precisão na descrição, possibilidade de atividades automatizadas, e entendimento por parte dos stakeholders técnicos e não-técnicos. O problema reside apenas na grande quantidade trabalho empregado ao descrever cada decisão com duas ou mais linguagens e, ainda por cima, ter que manter a consistência da descrição entre elas.

A seguir, mostramos a descrição da decisão arquitetural já apresentada no Exemplo 10 do capítulo de fundamentos (Exemplo 4.10) usando diferentes linguagens diferentes. O primeiro exemplo mostra a decisão escrita em Português.

Exemplo 8.13

[Decisão Arquitetural 001] A arquitetura do SASF é dividida em três camadas lógicas: apresentação, lógica de negócio e persistência de dados. A camada de apresentação se comunica apenas com a lógica de negócio e

apenas a lógica de negócio de comunica com a camada de persistência de dados.

Já o exemplo a seguir mostra a descrição usando também um código que pode ser interpretado pela ferramenta *DesignWizard*[9] e que permite a verificação automática de conformidade do código implementado com a arquitetura.

Exemplo 8.14

[Decisão Arquitetural 001] A arquitetura do SASF é dividida em três camadas lógicas: apresentação, lógica de negócio e persistência de dados, que serão mapeadas respectivamente para os pacotes: *com.sasf.webui*, *com.sasf.service*, *com.sasf.storage*. Os testes presentes na listagem a seguir (p. 215), que podem ser executados usando o *DesignWizard*, descrevem as regras de comunicação entre as camadas.

```
 public class ThreeTierDesignTest extends TestCase {
public void test_communication_web_ui_and_services() {
  String sasfClassesDir =
      System.getProperties("sasf.classes.directory");
  DesignWizard dw = new DesignWizard(sasfClassesDir);
  PackageNode services =
      dw.getPackage("com.sasf.service");
  PackageNode webUI = dw.getPackage("com.sasf.webui");
  Set<PackageNode> callers =
      services.getCallerPackages();
  for (PackageNode caller : callers) {
    assertTrue(caller.equals(webUI));
  }
}
```

[9] O artigo *Design tests: An approach to programmatically check your code against design rules* [21], de Brunet *et al* contém mais informações sobre o *DesignWizard*.

```
  public void test_communication_services_and_storage() {
    String sasfClassesDir =
        System.getProperties("sasf.classes.directory");
    DesignWizard dw = new DesignWizard(sasfClassesDir);
    PackageNode services =
        dw.getPackage("com.sasf.service");
    PackageNode storage =
        dw.getPackage("com.sasf.storage");
    Set<PackageNode> callers =
        storage.getCallerPackages();
    for (PackageNode caller : callers) {
      assertTrue(caller.equals(services));
    }
  }
```

}Testes para comunicação entre tiers.

8.2.4.2 Objetivo

O objetivo da decisão serve para registrarmos o motivo da decisão estar sendo tomada. Como decisões de design são conduzidas por requisitos, sejam eles funcionais ou de qualidade, a identificação dos requisitos deve estar presente neste atributo. Os objetivos das decisões arquiteturais ajudam na rastreabilidade da arquitetura.

No Exemplo 8.15, percebemos duas formas de menção aos requisitos implementados pela decisão. A primeira forma é presença o identificador do requisito de qualidade, RNF-01. Já a outra forma é uma breve descrição do requisito alcançado.

Exemplo 8.15
(continuação da [Decisão Arquitetural 001])

Objetivo: Atendimento ao requisito não-funcional: RNF-01. Esta divisão diminui o acoplamento entre os elementos internos da arquitetura, facilitando o desenvolvimento e a manutenção.

8.2.4.3 Fundamentação

Uma decisão arquitetural deve ser tomada com alguma fundamentação, seja ela uma análise das alternativas de design, baseada na experiência prévia do arquiteto, ou baseada em padrões de design. Esta fundamentação resulta no julgamento das vantagens e desvantagens das alternativas e servem para justificar a solução proposta.

No atributo fundamentação, registramos a justificativa da decisão para que haja um registro histórico das motivações e considerações feitas pelo arquiteto para chegar à solução de design. Este registro é essencial para que este tipo de informação não seja esquecido ao longo do ciclo de vida do software, pois é importante para o seu processo de evolução.

Por exemplo, durante uma atividade de refatoração do código, um novo desenvolvedor pode se interessar pelo motivo de um módulo ter sido criado aparentemente de forma desnecessária. Se não existir algum tipo de registro do motivo para existência do módulo em questão, o novo desenvolvedor pode, simplesmente, modificá-lo ou mesmo removê-lo ignorante dos efeitos que pode causar no design.

A fundamentação é normalmente feita por meio de uma descrição textual, mas deve possuir referências a outros documentos e a outras decisões arquiteturais relacionadas. O exemplo a seguir ilustra a fundamentação de uma decisão arquitetural.

Exemplo 8.16
(continuação da [Decisão Arquitetural 001])

Motivação: Projetar os elementos internos do sistema de modo que cada um pertença a apenas uma camada lógica ajuda a aumentar a coesão e diminuir o acoplamento. A coesão aumenta, pois cada elemento será desenvolvido com o objetivo de ser parte da apresentação, da lógica ou da persistência do sistema. Dessa maneira, cada elemento terá sua responsabilidade bem definida, mesmo que em alto nível. Como a comunicação entre as camadas é pré-definida, a de seus elementos também é: elementos da camada de apresentação não se comunicarão com elementos da camada de persistência, por exemplo. Assim, o acoplamento entre elementos internos será análogo ao acoplamento entre camadas. Com o baixo acoplamento, o desenvolvimento e a manutenção dos elementos também é facilitado, seja por possibilitar o desenvolvimento independente, seja por mudanças em um elemento terem menor impacto nos outros.

É importante observar que uma decisão arquitetural pode estar relacionada a mais de um atributo de qualidade e, como veremos a seguir, a mais de uma categoria. Isso ocorre porque decisões arquiteturais se interrelacionam da mesma forma que os atributos de qualidade e os requisitos impostos pelos stakeholders. Portanto, é na fundamentação que devemos também descrever as relações entre as decisões arquiteturais, ou seja, se uma decisão restringe, proíbe, possibilita, conflita, sobrepõe, compõe ou é composta de, depende de, ou é uma alternativa a outras decisões.

8.2.4.4 Escopo

Nem todas as decisões arquiteturais são válidas durante todo o ciclo de vida do software ou válidas em todos os módulos que o compõem. Por isso, surge a necessidade de registrar o escopo da decisão. Este registro tipo de registro se torna importante em decisões de propriedades, uma vez que normalmente tratam de preocupações transversais e abrangem grandes partes do sistema, e em decisões executivas, que devem, por exemplo, especificar quais etapas do processo de desenvolvimento devem usar tais metodologias.

O Exemplo 8.17, a seguir, descreve o escopo da Decisão Arquitetural 001, que é bem abrangente. Já no Exemplo 8.18, podemos observar que o escopo da decisão de usar JMX[10] como tecnologia de monitoração é mais limitado.

Exemplo 8.17
(continuação da [Decisão Arquitetural 001])

Escopo: Esta decisão é válida para todos os serviços que implementam lógica e que têm interface com o usuário.

Exemplo 8.18
Escopo: A decisão de usar JMX para exposição das métricas de desempenho só é válida para os casos definidos na [Decisão Arquitetural 008].

8.2.4.5 Histórico

A documentação da arquitetura, assim como o que ela representa, evolui ao longo do tempo. Decisões são tomadas, avali-

[10] *Java Management Extensions* (JMX): http://java.sun.com/products/JavaManagement/ (<http://java.sun.com/products/JavaManagement/>)

adas, modificadas ou mesmo contestadas ao longo do ciclo de vida da arquitetura. Portanto, é de se esperar que exista um registro histórico da evolução de cada decisão arquitetural. Por isso consideramos o atributo histórico.

O atributo histórico deve conter, para cada modificação da decisão, uma marca de tempo, o autor da modificação e um resumo da modificação. Se o documento estiver armazenado em um wiki ou outra forma eletrônica, o histórico pode conter links para as versões anteriores da decisão.

O Exemplo 8.19 ilustra o registro histórico da Decisão Arquitetural 001.

Exemplo 8.19
(continuação da [Decisão Arquitetural 001])

Histórico: *sugerida* (G. Germoglio, 2009/07/15); *revisada*, "Escopo modificado." (G. Germoglio, 2009/07/17); *aprovada* (J. Sauvé, 2009/07/18).

8.2.4.6 Estado Atual

O atributo estado atual de uma decisão serve para permitir mais uma dimensão de organização das decisões. Da mesma forma que as decisões evoluem ao longo do tempo, elas podem ter diversos estados que merecem ser registrados. Como o conjunto de estados que podem ser atribuídos a uma decisão arquitetural depende do processo de desenvolvimento adotado, citamos apenas alguns estados mais comuns:

- Sugerida: decisão que ainda não foi avaliada;
- Revisada: decisão sugerida e revisada pelo arquiteto ou time arquitetural;
- Aprovada: decisão sugerida, revisada e aprovada;

- Rejeitada: decisão sugerida, revisada e rejeitada. Ela deve se manter na documentação para referências futuras.

8.2.4.7 Categoria

Assim como o estado atual, o atributo categoria serve para possibilitar a organização das decisões arquiteturais em grupos relacionados. Normalmente, esse atributo é composto por uma lista de palavras-chave associadas às decisões.

Esse atributo permite, por exemplo, que os stakeholders selecionem decisões relacionadas a um atributo de qualidade específico. Portanto, se um membro do grupo de garantia de qualidade do software (*Software Quality Assurance* ou SQA) precisa das decisões arquiteturais necessárias para realizar uma análise de desempenho do projeto do software, ele deve procurar pelas decisões da categoria "desempenho".

8.3 Visões arquiteturais

Como consequência da existência dos diversos interessados na arquitetura e que esses interessados têm diferentes preocupações e diferentes níveis de conhecimento, as decisões arquiteturais não são documentadas da mesma maneira para interessados diferentes. Para resolver este problema, fazemos uso do conceito de múltiplas visões arquiteturais.

Visões arquiteturais são *representações do sistema ou de parte dele da perspectiva de um conjunto de interesses relacionados*. A visões arquiteturais proporcionam vantagens tanto para o processo de design, quanto para a documentação da arquitetura.

Durante o design, o arquiteto pode se focar em diferentes visões separadamente, podendo abstrair os detalhes desnecessários e só se ater às preocupações da visão corrente. Por exemplo, inicialmente, o arquiteto se pode utilizar de uma visão funcional para projetar os serviços primitivos do sistema que constituirão serviços mais complexos e que, por sua vez, servirão de base para as funcionalidades expostas aos usuários. Em seguida, o arquiteto pode se utilizar de uma visão de concorrência para projetar como as funções serão executadas ao longo do tempo: de forma sequencial ou em paralelo, de forma síncrona ou assíncrona. E, por fim, focando-se numa visão informacional, ele pode definir como os dados estão organizados.

Por outro lado, durante o processo de documentação, o arquiteto pode documentar as visões com diferentes níveis de detalhes e utilizar diferentes linguagens, uma vez que diferentes visões interessam a diferentes stakeholders.

As visões são concretizações do que chamamos pontos de vista arquiteturais[11]. Um ponto de vista arquitetural é a especificação dos elementos conceituais que devem ser usados para se construir uma visão. Um ponto de vista apresenta também qual o seu propósito e quem são os stakeholders interessados nas visões criadas a partir dele. Em outras palavras, um ponto de vista arquitetural é definido como:

Definição 8.1: ponto de vista arquitetural

É um arcabouço conceitual que define elementos, conexões e técnicas que compõem uma visão arquitetural, além especificar seu propósito de acordo com seus interessados.

Para documentarmos a arquitetura, devemos definir um conjunto pontos de vista que servirão de base para as visões da

[11] *Viewpoints*, de acordo com o padrão ISO/IEEE 1471-2000, ou *viewtypes* (tipos de visão), de acordo com Clements *et al* em *Documenting Software Architectures: Views and Beyond*.

arquitetura e que estarão presentes no documento. Cada visão terá uma ou mais decisões arquiteturais, que serão descritas a partir dos elementos, conexões e técnicas definidos pelo ponto de vista a que pertence.

Como já existem diversos conjuntos de pontos de vista prontos para uso na literatura, não há motivo para criarmos o nosso próprio conjunto. Portanto, a seguir, apresentamos alguns conjuntos os quais achamos essencial o conhecimento. São eles:

- 4+1 de Kruchten;
- Pontos de vista de Rozanski e Woods.
- Pontos de vista do *Software Engineering Institute*;

8.3.1 4+1 de Kruchten

O conjunto de pontos de vista 4+1 de Kruchten foi descrito inicialmente no artigo *The 4+1 View Model of Architecture* [62] e é um dos primeiros a serem descritos na literatura. Inicialmente, os pontos de vista são chamados pelo autor de visões. No entanto, se analisarmos a definição e o uso das visões empregados pelo autor, percebemos ela são compatíveis com nossas definições e usos dos pontos de vista.

O conjunto é composto por quatro pontos de vista, sendo cada um especializado em um aspecto da arquitetura, e um ponto de vista redundante, que contém cenários de uso. Os pontos de vista mais relevantes desse conjunto são: *Lógico*, *de Processos*, *de Desenvolvimento* e *Físico*. Como o conjunto de Rozanski e Woods é uma evolução do 4+1, ao descrevê-lo na seção a seguir, apresentaremos melhor os pontos de vista de Kruchten.

8.3.2 *Viewpoints* de Rozanski e Woods

Outro conjunto importante de pontos de vista é o descrito por Rozanski e Woods no livro *Software Systems Architecture:*

Working With Stakeholders Using Viewpoints and Perspectives [92]. Ele é uma evolução do conjunto 4+1, pois adiciona dois novos pontos de vista ao conjunto de Kruchten, e provê mais informações que ajudam no design do que na documentação.

Os pontos de vista presentes neste conjunto são:

- *Funcional*: representa o aspecto funcional do software descrito. Visões derivadas deste ponto de vista contêm decisões sobre as funções presentes no software e os módulos e submódulos que implementam essas funções. Este ponto de vista é especializado em mostrar a estrutura estática do software, mostrando suas partes, suas relações e suas interfaces. Seu equivalente no conjunto de Kruchten é o ponto de vista Lógico.
- *de Concorrência*: representa os aspectos dinâmicos e comportamentais do software. Visões derivadas deste ponto de vista contêm decisões sobre concorrência, sincronia ou assincronia de chamadas e aspectos temporais em geral do software e de suas funções. Seu equivalente no conjunto de Kruchten é o ponto de vista de Processo.

- *de Desenvolvimento*: representa os aspectos e relações entre os stakeholders e o processo de desenvolvimento do software. Visões derivadas deste ponto de vista contêm decisões de divisões de módulos, subsistemas, pacotes e classes e decisões sobre a atribuição de tarefas de construção, teste e reuso de partes do sistema aos participantes da equipe de desenvolvimento. Seu equivalente no conjunto de Kruchten é homônimo.
- *de Implantação*: representa os aspectos de implantação do software e suas relações com o ambiente físico. Visões derivadas deste ponto de vista contêm decisões de quantos servidores serão necessários para execução de um

serviço ou como os diferentes serviços são implantados ou atualizados durante o ciclo de vida do software. Seu equivalente no conjunto 4+1 é o ponto de vista Físico.

- *Informacional*: representa os aspectos relacionados aos dados presentes no software. Visões derivadas deste ponto de vista contêm decisões sobre o modelo de dados e sobre o armazenamento, manipulação, gerenciamento e distribuição das informações ao longo da vida do sistema em produção.
- *Operacional*: representa os aspectos operacionais do software. Ou seja, visões derivadas deste ponto de vista contêm decisões com estratégias de execução, administração e suporte do software em produção.

8.3.3 *Viewtypes* do *Software Engineering Institute* (SEI)

O último conjunto de pontos de vista que apresentamos é o descrito por Clements *et al* no livro *Documenting Software Architectures: Views and Beyond* [32]. Este conjunto foi criado com o objetivo de facilitar a documentação, ao contrário da maioria descrita na literatura, que têm seu foco no auxílio do projeto da arquitetura.

O conjunto do SEI possui apenas três pontos de vista, que devem ser especializados por meio dos chamados estilos arquiteturais. Os pontos de vista deste conjunto são:

- *de Componentes e Conectores*: este ponto de vista se preocupa em descrever os aspectos dinâmicos e de comportamento e interações entre os elementos da arquitetura. É nele em que encontramos os estilos arquiteturais: *Pipes-and-filters*, *Publish-Subscribe*, Cliente-Servidor, *Peer-to-Peer* e outros.

- *de Módulos*: este ponto de vista se preocupa em descrever a estrutura estática da arquitetura e em como ela se divide em unidades de código. O estilo arquitetural Camadas é uma especialização desse ponto de vista.
- *de Alocação*: este ponto de vista se preocupa em descrever as relações entre o software e o seu ambiente. O ponto de vista de Alocação se especializa em três aspectos diferentes: aspectos de implantação, que descreve as relações entre as partes do software e os recursos físicos utilizados (como servidores ou roteadores); aspectos de implementação, que descreve o mapeamento das partes do software e as partes do código (como pacotes, classes ou estrutura de diretórios); e aspectos de atribuição de trabalho, relacionados à distribuição de responsabilidades do projeto entre os membros do time de desenvolvimento.

8.4 Documentando a Arquitetura

A partir dos conceitos de decisões, visões e pontos de vista arquiteturais, estamos aptos a registrar o design da arquitetura em um documento. O primeiro passo para sermos capazes de escrever um bom documento arquitetural é conhecer os interessados na arquitetura. Esse conhecimento é um parâmetro fundamental para o processo de escolha dos pontos de vista a serem usados. Depois de definir os pontos de vista relevantes aos stakeholders da arquitetura, devemos então registrar as decisões arquiteturais que descrevem o design em visões derivadas a partir dos pontos de vista escolhidos.

Devemos observar que os processos de definição dos stakeholders, de escolha dos pontos de vista arquiteturais e de descrição das decisões em visões são dependentes do processo de desenvolvimento seguido pelo time de desenvolvimento. Além disso, apesar de descrevermos separadamente o processo de documen-

tação do processo de design, é possível (e bastante comum) que ocorram em paralelo, uma vez que a documentação e o design se ajudam mutuamente para alcançar seus objetivos.

Praticamos os conceitos apresentados neste capítulo no Apêndice onde ilustramos o documento de arquitetura do SASF.

8.4.1 Dificuldades da Documentação

Apesar dos benefícios proporcionados pela documentação da arquitetura, documentá-la não é fácil. A dificuldade de documentar a arquitetura reside, principalmente, em três características que descrevemos a seguir:

- o documento reflete o tamanho da solução;
- o documento reflete a complexidade do design da solução;

- é custoso manter o documento consistente com o design atual ao longo do ciclo de vida do software.

8.4.1.1 O tamanho do documento

Projetos de grandes sistemas de software são os que mais se beneficiam com o design e com a documentação da arquitetura. Isto ocorre porque o design e a documentação proporcionam orientação na implementação dos requisitos de qualidade, ajuda no controle intelectual sobre a complexidade da solução e servem de ferramenta que promove a integridade conceitual entre os stakeholders.

No entanto, um grande sistema de software implica numa grande solução de design, que deve conter muitas decisões arquiteturais e que devem ser apresentadas a muitos stakeholders, que demandam visões diferentes. A consequência disso é

que a arquitetura de um grande sistema deverá ser apresentada em um documento extenso.

Documentos muito extensos podem ser fonte de alguns problemas durante o processo de desenvolvimento. Entre estes problemas, podemos citar que eles levam muito tempo para serem construídos e que, em geral, geram uma certa resistência para serem lidos ou atualizados durante o desenvolvimento. Se o documento não é lido ou não é atualizado durante o desenvolvimento (e isso pode ocorrer porque a arquitetura pode evoluir ao longo do tempo, seja a evolução planejada ou não), ele corre o risco de ficar inconsistente com a realidade do software, tornando-se uma fonte de informação inútil e, portanto, deixando de proporcionar as vantagens de se projetar e documentar a arquitetura.

8.4.1.2 A complexidade do documento

Tanto o tamanho do documento quanto a complexidade do design influenciam na complexidade do documento da arquitetura. Um documento muito complexo, que usa muitas visões e diferentes linguagens para descrever diferentes aspectos do software, é difícil de se construir e de se manter consistente em caso de mudanças durante a evolução da arquitetura.

Como já mencionamos anteriormente, existem técnicas de verificação de conformidade entre o documento de arquitetura e o software implementado a partir dele, que podem ajudar na manutenção da consistência do documento com a realidade. No entanto, devemos nos lembrar que há também o esforço de se manter as diferentes visões arquiteturais consistentes entre si. Isso pode até ser facilitado se usarmos algumas linguagens específicas e ferramentas para descrever as decisões arquiteturais, como a ferramenta SAVE[12]. Por outro lado, estas lingua-

[12]Software Architecture Visualization and Evaluation

gens não são capazes de descrever todos os tipos necessários de decisões arquiteturais e isso limita o processo de automação na manutenção de consistência entre visões, tornando-o custoso.

8.4.1.3 Consistência entre o design atual e o documento

A consistência entre a implementação e o documento é condição fundamental para que o processo de desenvolvimento se beneficie da arquitetura. Por isso, deve existir um esforço para a manutenção desta consistência, tanto durante a evolução da arquitetura, quanto durante a implementação do software. Se a manutenção da consistência não é realizada, temos o chamado "deslize arquitetural" (*architectural drift*).

O deslize arquitetural é a inconsistência entre implementação do software e o design planejado. Esta inconsistência tem dois efeitos. O primeiro é que se a implementação não está seguindo o que foi planejado, ela pode também não estar alcançando os objetivos propostos. Já o segundo, como foi mencionado anteriormente, é que a inconsistência do documento com a realidade do software, inutiliza o documento, pois o transforma numa fonte de informação inútil. Assim, considerando que é custoso o processo de criação do documento de arquitetura, todo o trabalho realizado para tanto pode ter sido em vão.

Para a evitar o deslize arquitetural, recomenda-se que sejam periodicamente utilizadas durante todo o processo de desenvolvimento técnicas de verificação de conformidade. Essas técnicas, quando aplicadas, indicam se a implementação está de acordo com o design. Há basicamente dois tipos de técnicas de verificação de conformidade: as técnicas manuais, que são baseadas em inspeções do código; e as técnicas automáticas, que só podem ser realizadas se a descrição da arquitetura uti-

(SAVE): http://fc-md.umd.edu/save/default.asp (<http://fc-md.umd.edu/save/default.asp>)

lizar linguagens que permitam esse tipo de verificação. Assim como os tipos de análise arquitetural, as técnicas de verificação manuais são mais custosas, mas têm um maior alcance, podendo verificar aspectos do software que não são formalizados. Já as técnicas de verificação automáticas, se beneficiam do baixo custo, mas são limitadas aos aspectos que podem ser descritos pelas linguagens formais que utilizam.

8.5 Resumo

O objetivo deste livro não é só fazer com que o leitor saiba projetar arquiteturas, mas também que tenha noções de como documentar o projeto. Dessa maneira, o objetivo deste capítulo foi fazer com que o leitor conheça a importância e a técnica primordial da documentação da arquitetura, que é representá-la por meio de múltiplas visões. Assim, esperamos que a partir de agora o leitor, além de conhecer alguns conjuntos de pontos de vista arquiteturais de referência, seja capaz de entender que:

- O documento de arquitetura auxilia no processo de design, é uma ferramenta de comunicação entre stakeholders e pode servir de modelo de análise do software;
- Toda informação presente numa arquitetura é uma decisão arquitetural;
- Decisões arquiteturais podem ser existenciais, descritivas ou executivas;
- Decisões arquiteturais se relacionam, podendo restringir, impedir, facilitar, compor, conflitar, ignorar, depender ou ser alternativa a outras decisões arquiteturais; e
- Um único diagrama não é suficiente para conter a quantidade de informação que deve ser mostrada por um arquiteto. Por isso, a necessidade de múltiplas visões arquiteturais;

8.6 Referências

8.6.1 Benefícios da documentação

Muitos autores afirmam que a documentação do design arquitetural é benéfica para o processo de desenvolvimento do software. Além dos trabalhos já referenciados ao longo do texto, citamos: o livro *The Art of Systems Architecting* [69] de Maier e Rechtin, que descreve os benefícios proporcionados quando o documento de arquitetura é consistente entre suas múltiplas visões; o artigo de Wirfs-Brock, *Connecting Design with Code* [108], que cita a importância de documentar as decisões de design em diversos níveis de detalhe, inclusive o arquitetural; o livro *Software Architecture: Foundations, Theory, and Practice* [104], de Taylor *et al*, que mostra este assunto de forma bem abrangente; e o livro *Code Complete* [75], de McConnell, que afirma a arquitetura é uma ferramenta para o controle da complexidade do software.

8.6.2 Arquitetura como conjunto de decisões

O documento de arquitetura descrito como um conjunto de decisões arquiteturais é o tema de diversos artigos. Entre eles, podemos citar: *Building Up and Reasoning About Architectural Knowledge* [64], *An Ontology of Architectural Design Decisions in Software Intensive Systems* [65] e *The Decision View's Role in Software Architecture Practice* [63], de Kruchten *et al* e que são focados em descrever a arquitetura como decisões arquiteturais e em montar um arcabouço conceitual de como descrever decisões arquiteturais, propondo inclusive um ponto de vista neste sentido; *Architecture Knowledge Management: Challenges, Approaches, and Tools* [9], de Babar e Gorton, que mostram ferramentas para organizar e documentar as decisões arquiteturais; e *The Decision View of Software Architecture* [38], de Dueñas e Capilla, e *Software*

Architecture as a Set of Architectural Design Decisions [55], de Jansen e Bosch, que mostram a importância das decisões na arquitetura.

8.6.3 Visões e pontos de vista

Além dos trabalhos que apresentam conjuntos de pontos de vista e que já foram referenciados na Seção "Visões arquiteturais" (Section 8.3: Visões arquiteturais), podemos citar o livro *Software Design* [22], de Budgen, que afirma que diferentes representações lidam com diferentes qualidades e interesses, além de mostrar alguns benefícios de se documentar a arquitetura por meio de visões. Já o artigo *Four Metaphors of Architecture in Software Organizations: Finding Out the Meaning of Architecture in Practice* [98], de Smolander, descreve como alguns stakeholders percebem a arquitetura na prática e assim justifica o uso de visões arquiteturais. Por fim, citamos o livro *Applied Software Architecture* [50], de Hofmeister *et al*, que apresenta mais um conjunto de pontos de vista.

8.6.4 Ferramentas para análise

Em relação a análises baseadas em inspeções, citamos o livro da série do SEI, *Evaluating Software Architectures* [33], de Clements *et al*, que descreve os métodos SAAM, ATAM e ARID, inclusive com estudos de casos.

Já sobre ADLs, citamos dois *surveys* sobre o assunto: um conduzido por Clements, *A Survey of Architecture Description Languages* [29], e outro por Taylor e Medvidovic, *A Classification and Comparison Framework for Software Architecture Description Languages* [77].

E, finalmente, apresentamos alguns trabalhos sobre verificação de conformidade: *Software Reflexion Models: Bridging The*

Gap Between Design and Implementation [78] e *Software Reflexion Models: Bridging The Gap Between Source and High-Level Models* [79], por Murphy *et al*; *Bridging the Software Architecture Gap* [66], de Lindvall e Muthig; e *Design tests: An approach to programmatically check your code against design rules* [21], de Brunet *et al*.

Glossary

A arquitetura de software

Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução.

atributo de qualidade

É uma propriedade de qualidade do software ou de seu ciclo de desenvolvimento, podendo se manifestar como características, capacidades ou restrições de uma função específica ou de um conjunto de funções do software.

D decisão arquitetural

Uma escolha entre as alternativas de design arquitetural. Essa escolha se propõe a alcançar um ou mais atributos de qualidade do sistema, por meio da(s) estrutura(s) arquiteturais que ela envolve ou define.

M modelo de qualidade

Modelo que define e organiza os atributos do software importantes para a avaliação de sua qualidade.

P

ponto de vista arquitetural

É um arcabouço conceitual que define elementos, conexões e técnicas que compõem uma visão arquitetural, além especificar seu propósito de acordo com seus interessados.

R

rastreamento de requisitos

É o processo/capacidade de ligar requisitos do sistema a estruturas arquiteturais.

requisito funcional

É a declaração de uma função ou comportamento providos pelo sistema sob condições específicas.

requisito não-funcional de processo

Requisito que restringe o processo de desenvolvimento do software.

requisito não-funcional de produto

Requisito que especifica as características que um sistema ou subsistema deve possuir.

requisito não-funcional externo

Requisito derivado do ambiente em que o sistema é desenvolvido, que pode ser tanto do produto quanto do processo.

requisito não-funcional

É a descrição de propriedades, características ou restrições que o software apresenta exibidas por suas funcionalidades.

S stakeholder

"Um stakeholder em uma arquitetura de software é uma pessoa, grupo ou entidade com um interesse ou preocupações sobre a realização da arquitetura." [13]

V visão arquitetural

É a representação do sistema ou de parte dele da perspectiva de um conjunto de interesses relacionados.

A alternativa de design

Uma possibilidade de solução representada em nível de conhecimento.

D design arquitetural

Descreve a arquitetura do software ou, em poucas palavras, como o software é decomposto e organizado em módulos e suas relações.

design de software

"É tanto o processo de definição da arquitetura, módulos, interfaces e outras características de um sistema quanto o resultado desse processo."[14]

design detalhado

Descreve o comportamento específico e em detalhes dos módulos que compõem o design arquitetural.

[13] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley Professional 2005.

[14] Freny Katki *et al*, editors. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers Inc., 1991.

O

objetivo de design

Aquilo que se pretende alcançar para resolver as necessidades do cliente.

R

representação de design

A linguagem do processo de design que representa o produto do design para sua construção e também dá suporte ao processo de design como um todo.

requisito funcional

É a declaração de uma função ou comportamento providos pelo sistema sob condições específicas.

requisito não-funcional

É a descrição de propriedades, características ou restrições que o software apresenta exibidas por suas funcionalidades.

restrição de design

A regra, requisito, relação, convenção, ou princípio que define o texto do processo de design.

S

solução do design

A descrição do design que permite a construção do sistema de software que alcança os objetivos do design.

Bibliography

[1] *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers Inc., The, 1991.

[2] *Software Evolution.* Springer, March 2008.

[3] IEEE Std 754-2008. *IEEE Standard for Floating-Point Arithmetic.* Institute of Electrical and Electronics Engineers, 2008.

[4] ISO 9126-1:2001. *Software engineering 8211; Product quality 8211; Part 1: Quality model.* International Organization for Standardization, Geneva, Switzerland.

[5] Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp. *Guide to the Software Engineering Body of Knowledge (SWEBOK).* IEEE, 2004.

[6] Defense Information Systems Agency. *Department of Defense Joint Technical Architecture, Version 6.0. Volume 2.* U.S. Department of Defense, October 2003.

[7] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):2138211;249, July 1997.

[8] Andy and Kai Qian. *Component-Oriented Programming.* Wiley-Interscience, March 2005.

[9] Muhammad A. Babar and Ian Gorton. Architecture knowledge management: Challenges, approaches, and tools. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, page 1708211;171, 2007.

[10] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2 edition, April 2003.

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2 edition, April 2003.

[13] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[14] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[15] L. Belady. Foreword. In *Software Design: Methods and Techniques (L.J. Peters, author)*. Yourdon Press, 1981.

[16] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *International Conference on Software Engineering*, page 5928211;605, San Francisco, 1976. IEEE Computer Society Press.

[17] G. Booch. Goodness of fit. *Software, IEEE*, 23(6):148211;15, 2006.

[18] Grady Booch. The irrelevance of architecture. *Software, IEEE*, 24(3):108211;11, 2007.

[19] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison-Wesley Professional, April 2007.

[20] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[21] J. Brunet, D. Guerrero, and J. Figueiredo. Design tests: An approach to programmatically check your code against design rules. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, page 2558211;258, 2009.

[22] David Budgen. *Software Design*. Addison Wesley, 2 edition, May 2003.

[23] David Budgen. *Software Design (2nd Edition)*. Addison Wesley, May 2003.

[24] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[25] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, June 2007.

[26] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[27] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[28] J. N. Buxton and B. Randell. Software engineering techniques. Technical report, NATO Science Committee, Rome, Italy, April 1970.

[29] P. C. Clements. A survey of architecture description languages. In *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, page 168211;25, 1996.

[30] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, September 2002.

[31] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, September 2002.

[32] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, September 2002.

[33] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, January 2002.

[34] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *6th Symposium on Operating Systems Design & Implementation (OSDI8217;04)*, 2004.

[35] Chris Dibona, Mark Stone, and Danese Cooper. *Open Sources 2.0 : The Continuing Evolution*. O'Reilly Media, Inc., October 2005.

[36] Edsger W. Dijkstra. The structure of the the-multiprogramming system. *Commun. ACM*, 11(5):3418211;346, 1968.

[37] Edsger W. Dijkstra. The structure of the the-multiprogramming system. *Commun. ACM*, 11(5):3418211;346, 1968.

[38] Juan C. Due[U+FFFD]nd Rafael Capilla. The decision view of software architecture. page 2228211;230. 2005.

[39] George Edwards, Sam Malek, and Nenad Medvidovic. Scenario-driven dynamic analysis of distributed architectures. page 1258211;139. 2007.

[40] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):18211;12, 2001.

[41] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. d. thesis, University of California, Irvine, 2000.

[42] Brian Foote and Joseph W. Yoder. Big ball of mud. In *Pattern Languages of Program Design*, volume 4, page 6548211;692. Addison Wesley, 2000.

[43] M. Fowler. Design - who needs an architect? *Software, IEEE*, 20(5):118211;13, 2003.

[44] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[45] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[46] David Garlan and Mary Shaw. An introduction to software architecture. Technical report CMU-CS-94-166, Carnegie Mellon University, Pittsburgh, PA 15213-3890, January 1994.

[47] Ian Gorton. *Essential Software Architecture*. Springer, June 2006.

[48] Ian Gorton. *Essential Software Architecture*. Springer, June 2006.

[49] Todd Hoff. High scalability: Building bigger, faster, more reliable websites. http://highscalability.com.

[50] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Professional, November 1999.

[51] Luke Hohmann. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Addison-Wesley Professional, January 2003.

[52] Luke Hohmann. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Addison-Wesley Professional, January 2003.

[53] IEEE and ISO/IEC. Systems and software engineering - recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, page c18211;24, July 2007.

[54] IEEE and ISO/IEC. Systems and software engineering - recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, page c18211;24, July 2007.

[55] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, page 1098211;120, Washington, DC, USA, 2005. IEEE Computer Society.

[56] D. Kalinsky and J. Ready. Distinctions between requirements specification and design of real-time systems. In *Software Engineering for Real Time Systems, 1989., Second International Conference on*, page 268211;30, 1989.

[57] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, April 1997.

[58] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, September 1998.

[59] P. Kruchten. The software architect 8211; and the software architecture team. *Software Architecture; TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 2:5658211;583.

[60] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *Software, IEEE*, 23(2):228211;30, 2006.

[61] P. B. Kruchten. The 4+1 view model of architecture. *Software, IEEE*, 12(6):428211;50, 1995.

[62] P. B. Kruchten. The 4+1 view model of architecture. *Software, IEEE*, 12(6):428211;50, 1995.

[63] Philippe Kruchten, Rafael Capilla, and Juan C. Duex00f1;as. The decision view's role in software architecture practice. *IEEE Software*, 26(2):368211;42, 2009.

[64] Philippe Kruchten, Patricia Lago, Hans van Vliet, and Timo Wolf. Building up and exploiting architectural knowledge. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, page 2918211;292, Washington, DC, USA, 2005. IEEE Computer Society.

[65] Philippe Krutchen. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen Workshop Software Variability*, page 548211;61, October 2004.

[66] M. Lindvall and D. Muthig. Bridging the software architecture gap. *Computer*, 41(6):988211;101, June 2008.

[67] John D. C. Little. A proof for the queuing formula: L= w. *Operations Research*, 9(3):3838211;387, 1961.

[68] Mark W. Maier and Eberhardt Rechtin. *The Art of Systems Architecting*. CRC, 2 edition, June 2000.

[69] Mark W. Maier and Eberhardt Rechtin. *The Art of Systems Architecting*. CRC, 2 edition, June 2000.

[70] Ruth Malan and Dana Bredemeyer. Defining non-functional requirements. Online at: http://www.bredemeyer.com/pdf_files/NonFunctReq.PDF, August 2001.

[71] Matthew R. McBride. The software architect: Essence, intuition, and guiding principles. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, page 2308211;235. ACM Press, 2004.

[72] J. McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*, volume 1-3. General Electric, November 1977.

[73] Steve McConnell. *Code Complete.* Microsoft Press, 2 edition, June 2004.

[74] Steve McConnell. *Code Complete.* Microsoft Press, second edition, June 2004.

[75] Steve McConnell. *Code Complete.* Microsoft Press, 2 edition, June 2004.

[76] Steve Mcconnell. *Code Complete, Second Edition.* Microsoft Press, June 2004.

[77] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):708211;93, 2000.

[78] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *Software Engineering, IEEE Transactions on*, 27(4):3648211;380, April 2001.

[79] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, page 188211;28, New York, NY, USA, 1995. ACM.

[80] Inc. Object Management Group. Unified modeling language. http://www.uml.org, September 2008.

[81] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Classics in Software Engineering*, page 1398211;150.

[82] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Classics in Software Engineering*, page 1398211;150, 1979.

[83] David L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, page 2798211;287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[84] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):408211;52, October 1992.

[85] Andy Powell, Mikael Nilsson, Ambjrn Naeve, Pete Johnston, and Thomas Baker. Dcmi abstract model. DCMI Recommendation, June 2007.

[86] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 6 edition, April 2004.

[87] Jack W. Reeves. What is software design? *C++ Journal*, 1992.

[88] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[89] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[90] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[91] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[92] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[93] Jungwoo Ryoo, Phil Laplante, and Rick Kazman. In search of architectural patterns for software security. *Computer*, 42(6):988211;100, 2009.

[94] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):428211;81, March 2005.

[95] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, September 2000.

[96] Bart Smaalders. Performance anti-patterns. *Queue*, 4(1):448211;50, 2006.

[97] G. F. Smith and G. J. Browne. Conceptual foundations of design problem solving. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(5):12098211;1219, 1993.

[98] Kari Smolander. Four metaphors of architecture in software organizations: Finding out the meaning of architecture in practice. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering*, Washington, DC, USA, 2002. IEEE Computer Society.

[99] Ian Sommerville. *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley, May 2004.

[100] Ian Sommerville. *Software Engineering*. Addison Wesley, 8 edition, June 2006.

[101] Diomidis Spinellis and Georgios Gousios. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*. O'Reilly Media, Inc., January 2009.

[102] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[103] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[104] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[105] Richard N. Taylor and Andre van der Hoek. Software design and architecture 8211; the once and future focus of software engineering. In *FOSE '07: 2007 Future of Software Engineering*, page 2268211;243, Washington, DC, USA, 2007. IEEE Computer Society.

[106] Facebook Team. Engineering @ facebook. http://www.facebook.com/notes.php?id=9445547199.

[107] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):1058211;119, March 2002.

[108] Rebecca J. Wirfs-Brock. Connecting design with code. *Software, IEEE*, 25(2):208211;21, 2008.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Arquitetura de Software*
Edited by: Guilherme Germoglio
URL: http://cnx.org/content/col10722/1.9/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Mensagens do Livro"
By: Guilherme Germoglio
URL: http://cnx.org/content/m17526/1.7/
Pages: 1-8
Copyright: Guilherme Germoglio
License: http://creativecommons.org/licenses/by/2.0/

Module: "Introdução a Design de Software"
By: Guilherme Germoglio
URL: http://cnx.org/content/m17494/1.26/
Pages: 9-39
Copyright: Guilherme Germoglio
License: http://creativecommons.org/licenses/by/2.0/

Module: "Estudo de Caso: SASF"
By: Guilherme Germoglio
URL: http://cnx.org/content/m23389/1.5/
Pages: 41-53
Copyright: Guilherme Germoglio
License: http://creativecommons.org/licenses/by/3.0/

Module: "Fundamentos de Arquitetura de Software"
By: Guilherme Germoglio
URL: http://cnx.org/content/m17524/1.21/
Pages: 55-103
Copyright: Guilherme Germoglio
License: http://creativecommons.org/licenses/by/2.0/

Arquitetura de Software
Um livro de Arquitetura de Software para alunos de graduação.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.