Técnicas de Design Arquitetural*

Guilherme Germoglio

Abstract

Capítulo onde descrevemos técnicas de design arquitetural. Este capítulo ainda está em construção. Ele precisa de exemplos e estudos de caso para as técnicas citadas.

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; 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.

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.

*Version 1.5: Aug 19, 2009 5:48 pm +0000
†http://creativecommons.org/licenses/by/2.0/

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[1].

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*.

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 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.

[1] 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.

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.

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[2] 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*[3], 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 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*[4], 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

[2] Também chamado de estilo arquitetural.
[3] McConnell, S. *Code Complete*. Microsoft Press, Segunda edição, Junho 2004.
[4] Buschmann, F. *et al*, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Agosto 1996.

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 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*[5], *Struts*[6] e *Spring MVC*[7].

***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*[8], o ambiente de desenvolvimento *Eclipse*[9] e diversos sistemas de manipulação de imagens que são extensíveis por meio de *plugins*, como o *GIMP*[10] e o *ImageJ*[11].

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 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.

[5] *JavaServer Faces Technology (JSF)*: http://java.sun.com/javaee/javaserverfaces/ (<http://java.sun.com/javaee/javaserverfaces/>)
[6] *Apache Struts*: http://struts.apache.org/ (<http://struts.apache.org/>)
[7] *Spring Framework*: http://www.springsource.org/ (<http://www.springsource.org/>)
[8] *MINIX*: http://www.minix3.org/ (<http://www.minix3.org/>)
[9] *Eclipse*: http://www.eclipse.org (<http://www.eclipse.org/>)
[10] *The GNU Image Manipulation Program* (GIMP): http://www.gimp.org (<http://www.gimp.org>)
[11] *ImageJ - Image Processing and Analysis in Java*: http://rsbweb.nih.gov/ij/ (<http://rsbweb.nih.gov/ij/>)

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.

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.

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.

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, executando 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.

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 desempenho. 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.

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*[12] (ou de sua implementação *open source*, o *Hadoop*[13]), 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

[12] A arquitetura do *MapReduce* é brevemente apresentada por Dean e Ghemawat no artigo *MapReduce: Simplified Data Processing on Large Clusters*[5].

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

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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).

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 elementos 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.

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.

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.

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.

2.5 Operabilidade

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

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.

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.

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 recursos 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.

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.

4 Referências

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*[10], de McConnell. Além dele, podemos citar os seguintes artigos sobre o assunto: *The Structure of The THE-multiprogramming System*[6], de *Dijkstra*, e o *On The Criteria to Be Used in Decomposing Systems Into Modules*[11], de Parnas.

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*[4], [15], [2], [3]. Recomendamos também sobre o assunto os livros *Patterns of Enterprise Application Architecture*[7], escrito por Fowler, e o *Software Architecture in Practice*[1], escrito por Bass *et al*.

4.3 Técnicas arquiteturais

Sobre técnicas arquiteturais, podemos citar o livro *Beautiful Architecture*[17], 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*[18], de Taylor *et al*, quanto o livro *Software Systems Architecture*[12], de Rozanski e Woods. O livro *The Art of Systems Architecting*[9], 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*[16], de Smaalders; sobre replicação de dados: *Optimistic Replication*[14], de Saito e Shapiro; e sobre segurança: *In Search of Architectural Patterns for Software Security*[13], 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*[8] e o *Engineering @ Facebook*[19].

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2 edition, April 2003.

[2] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[3] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, June 2007.

[4] 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.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *6th Symposium on Operating Systems Design & Implementation (OSDI8217;04)*, 2004.

[6] Edsger W. Dijkstra. The structure of the the-multiprogramming system. *Commun. ACM*, 11(5):3418211;346, 1968.

[7] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[8] Todd Hoff. High scalability: Building bigger, faster, more reliable websites. http://highscalability.com.

[9] Mark W. Maier and Eberhardt Rechtin. *The Art of Systems Architecting*. CRC, 2 edition, June 2000.

[10] Steve McConnell. *Code Complete*. Microsoft Press, second edition, June 2004.

[11] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Classics in Software Engineering*, page 1398211;150.

[12] Nick Rozanski and E[U+FFFD]oods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[13] Jungwoo Ryoo, Phil Laplante, and Rick Kazman. In search of architectural patterns for software security. *Computer*, 42(6):988211;100, 2009.

[14] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):428211;81, March 2005.

[15] 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.

[16] Bart Smaalders. Performance anti-patterns. *Queue*, 4(1):448211;50, 2006.

[17] Diomidis Spinellis and Georgios Gousios. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*. O'Reilly Media, Inc., January 2009.

[18] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[19] Facebook Team. Engineering @ facebook. http://www.facebook.com/notes.php?id=9445547199.