UFES - Universidade Federal do Espírito Santo

Projeto de Sistemas de Software

Notas de Aula

Ricardo de Almeida Falbo

E-mail: falbo@inf.ufes.br

2018

Índice

Capítulo 1 – Introdução

O desenvolvimento de software é um processo que envolve atividades técnicas, gerenciais e de garantia da qualidade. No que concerne às atividades técnicas, tipicamente o processo de software inicia-se com o Levantamento de Requisitos, quando os requisitos do sistema a ser desenvolvido são preliminarmente capturados e organizados. Uma vez capturados, os requisitos devem ser modelados, avaliados e documentados. A Modelagem Conceitual é uma atividade essencial no processo de Engenharia de Requisitos e cuida da elaboração de modelos descrevendo *o quê* o software tem de fazer (e não *como* fazê-lo). Até este momento, a ênfase está sobre o domínio do problema e não se deve pensar na solução técnica, computacional, a ser adotada.

Requisitos incluem especificações dos serviços/funções que o sistema deve prover, restrições sob as quais ele deve operar, propriedades gerais do sistema e restrições que devem ser satisfeitas no seu processo de desenvolvimento. Os requisitos podem ser funcionais ou não funcionais. Requisitos funcionais, como o próprio nome indica, apontam as funções que o sistema deve prover e como o sistema deve se comportar em determinadas situações. Já os requisitos não funcionais descrevem restrições sobre as funções a serem providas, restrições essas que limitam as opções para criar uma solução para o problema.

Com os requisitos pelo menos parcialmente capturados e especificados na forma de modelos, pode-se começar a trabalhar no domínio da solução. Muitas soluções são possíveis para o mesmo conjunto de requisitos e elas são intrinsecamente ligadas a uma dada plataforma de implementação (linguagem de programação, mecanismo de persistência a ser adotado, dispositivos onde o sistema deverá rodar e tipos de interface associados etc.). Em especial, requisitos não funcionais são, muitas vezes, conflitantes e a escolha de quais atributos de qualidade devem ser mais atentamente considerados tem também um forte impacto na escolha da solução. Assim, ao se considerar alternativas de solução, todos esses aspectos têm de ser levados em conta.

A fase de projeto tem por objetivo definir e especificar uma solução a ser implementada. É uma fase de tomada de decisão, tendo em vista que muitas soluções são possíveis. Além disso, o projeto é um processo de refinamento. Partindo dos modelos conceituais do sistema, a fase de projeto inicia-se com o projeto da arquitetura do sistema, que visa descrever a estrutura de nível mais alto da aplicação, identificando seus principais elementos ou componentes[1] e como eles se relacionam uns com os outros. Uma vez definida a arquitetura, o projeto passa a se concentrar no detalhamento

[1] Grande parte dos trabalhos na literatura trata os blocos primários de construção de uma arquitetura de software como "componentes". Contudo, conforme apontam Bass, Clements e Kazman (2003), este termo vem sendo cada vez mais estreitamente associado ao movimento de Desenvolvimento Baseado em Componentes – DBC (GIMENES; HUZITA, 2005), onde assume uma conotação mais restrita. Assim, neste texto procura-se utilizar o termo "elemento" para atribuir um caráter mais geral. Quando usado, o termo "componente" tem também essa concepção mais geral, tendo em vista que este texto não aborda diretamente o DBC.

de cada um desses elementos, até atingir o nível de unidades de implementação (p.ex., classes no desenvolvimento orientado a objetos).

Uma vez especificado o projeto dos elementos da arquitetura, pode dar-se início à implementação, quando as unidades de software do projeto detalhado são implementadas e testadas individualmente (teste de unidade). Gradativamente, os elementos vão sendo integrados e testados (teste de integração), até se obter o sistema, quando o todo deve ser testado (teste de sistema).

Por fim, uma vez testado no ambiente de desenvolvimento, o software pode ser colocado em produção. Usuários devem ser treinados, o ambiente de produção deve ser configurado e o sistema deve ser instalado e testado, agora pelos usuários no ambiente de produção (testes de homologação ou aceitação). Caso o software demonstre prover as capacidades requeridas, ele pode ser aceito e a operação iniciada.

Este texto aborda a fase de projeto, concentrando-se no projeto de software.

1.1 – A Fase de Projeto

O objetivo da fase de projeto (ou design) é produzir uma solução para o problema identificado e modelado nas fases de levantamento e análise de requisitos, incorporando a tecnologia aos requisitos e projetando o que será construído na implementação. Sendo assim, é necessário conhecer a tecnologia disponível e os ambientes de hardware e software onde o sistema será desenvolvido e implantado. Durante o projeto, deve-se decidir como o problema será resolvido, começando em um alto nível de abstração, próximo da análise, e progredindo sucessivamente para níveis mais detalhados até se chegar a um nível de abstração próximo da implementação.

O projeto de software encontra-se no núcleo técnico do processo de desenvolvimento de software e é aplicado independentemente do modelo de ciclo de vida e paradigma adotados. É iniciado assim que os requisitos do software tiverem sido modelados e especificados pelo menos parcialmente e é a última atividade de modelagem. Por outro lado, corresponde à primeira atividade que leva em conta aspectos tecnológicos (PRESSMAN, 2006).

Enquanto a fase de análise pressupõe que a tecnologia é perfeita (capacidade ilimitada de processamento, com velocidade instantânea, capacidade ilimitada de armazenamento, custo zero e não passível de falha), a fase de projeto envolve a modelagem de como o sistema será implementado, agora considerando também os requisitos não funcionais de caráter tecnológico. Assim, conforme aponta Mitch Kapor, citado por Pressman (2006), o projeto é "onde você se instala com um pé em dois mundos – o mundo da tecnologia e o mundo das pessoas e objetivos humanos – e você tenta juntar os dois". Ou, de outra maneira, o projeto de software é a ponte que conecta esses dois mundos: o mundo real e o mundo computacional. A Figura 1.1 procura ilustrar esta situação. O projeto é, portanto, a fase do processo de software na qual os requisitos, as necessidades do negócio e as considerações técnicas se juntam na formulação de um produto ou sistema de software (PRESSMAN, 2006).

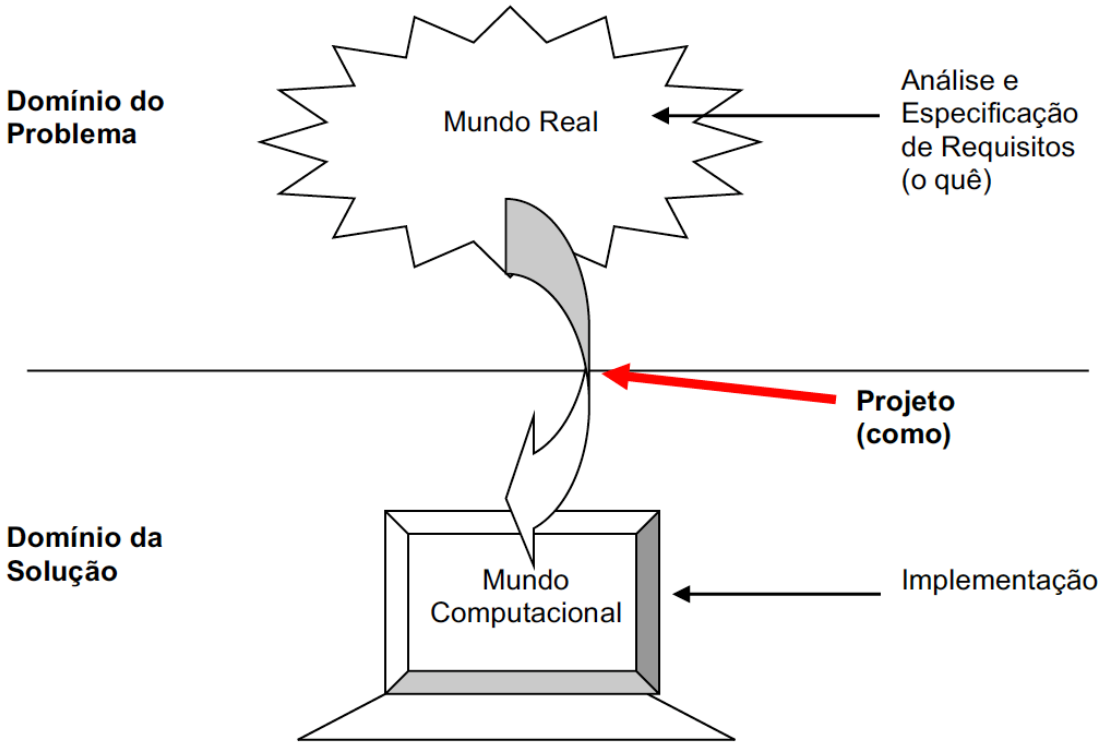


Figura 1.1 – A Fase de Projeto

Conforme mencionado anteriormente, o projeto é um processo de refinamento. Inicialmente, o projeto é representado em um nível alto de abstração, enfocando a estrutura geral do sistema. Definida a arquitetura, o projeto passa a tratar do detalhamento de seus elementos. Esses refinamentos conduzem a representações de menores níveis de abstração, até se chegar ao projeto de algoritmos e estruturas de dados. Assim, independentemente do paradigma adotado, o processo de projeto envolve as seguintes atividades:

- Projeto da Arquitetura do Software: visa definir os elementos estruturais do software e seus relacionamentos.
- Projeto dos Elementos da Arquitetura: visa projetar em um maior nível de detalhes cada um dos elementos estruturais definidos na arquitetura, o que envolve a decomposição de módulos em outros módulos menores.
- Projeto Detalhado: tem por objetivo refinar e detalhar os elementos mais básicos da arquitetura do software, i.e., as interfaces, os procedimentos e as estruturas de dados. Deve-se descrever como se dará a comunicação entre os elementos da arquitetura (interfaces internas), a comunicação do sistema em desenvolvimento com outros sistemas (interfaces externas) e com as pessoas que vão utilizá-lo (interface com o usuário), bem como se devem projetar detalhes de algoritmos e estruturas de dados.

Tendo em vista que a orientação a objetos é um dos paradigmas mais utilizados atualmente no desenvolvimento de sistemas, este texto aborda o projeto de software orientado a objetos. Além disso, o foco deste texto são os sistemas de informação.

Considerando essa classe de sistemas, de maneira geral, os seguintes elementos estão presentes na arquitetura de um sistema:

- Lógica de Negócio: é o elemento da arquitetura que trata da lógica de negócio apoiada pelo sistema, englobando tanto aspectos estruturais (classes de domínio derivadas dos modelos conceituais estruturais da fase de análise), quanto comportamentais (classes de lógica de aplicação, que tratam das funcionalidades descritas pelos casos de uso).
- Interface com o Usuário: é o elemento da arquitetura que trata da interação humano-computador. Envolve tanto as interfaces propriamente ditas (objetos gráficos responsáveis por receber dados e comandos do usuário e apresentar resultados), quanto o controle da interação, abrindo e fechando janelas, habilitando ou desabilitando botões etc. (WAZLAWICK, 2004).
- Persistência: é o elemento da arquitetura responsável pelo armazenamento e recuperação de dados em memória secundária (classes que representam e isolam os depósitos de dados do restante do sistema).

Esses elementos são os principais elementos discutidos neste texto.

1.2 – A Organização deste Texto

Este texto procura oferecer uma visão geral do Projeto de Software, discutindo as principais atividades desse processo e como realizá-las. Nos capítulos que se seguem, os seguintes temas são abordados:

- Capítulo 2 – *O Projeto de Software*: discute princípios gerais de projeto e como eles se aplicam ao projeto de software, características de um bom projeto de software, atributos de qualidade de software que devem ser considerados no projeto de software, reutilização no projeto de software por meio de padrões (*patterns*) e a documentação do projeto de software.
- Capítulo 3 – *Arquitetura de Software*: define o que é arquitetura de software, apresenta alguns padrões arquitetônicos, discute o impacto de atributos de qualidade no projeto da arquitetura e táticas para tratá-los, bem como aborda a documentação da arquitetura de software.
- Capítulo 4 – *Projeto da Lógica de Negócio*: concerne ao projeto dos elementos da arquitetura que tratam da lógica do negócio a ser apoiado pelo sistema. Dois componentes principais são abordados neste capítulo: o Componente de Domínio do Problema, que se refere aos elementos responsáveis por tratar diretamente as informações relevantes, capturadas na modelagem estrutural da fase de análise, e o Componente de Gerência de Tarefas, que concerne aos elementos responsáveis por tratar as funcionalidades descritas pelos casos de uso, modelados e descritos na fase de especificação de requisitos.
- Capítulo 5 – *Projeto da Interface com o Usuário*: aborda o projeto da interface do sistema computacional com seus usuários. Dois componentes principais são discutidos neste capítulo: o Componente de Apresentação (ou Visão), que se refere às interfaces propriamente ditas (objetos gráficos

responsáveis por receber dados e comandos do usuário e apresentar resultados), e o Componente de Controle de Interação, que diz respeito ao controle da interação com o usuário, envolvendo aspectos relacionados à ativação de comandos, controle e sequência da interação.

- Capítulo 6 – *Projeto da Persistência*: trata do armazenamento e recuperação de dados em um mecanismo de persistência. Como os bancos de dados relacionais são atualmente o principal mecanismo de persistência utilizado no desenvolvimento de sistemas de informação, este capítulo apresenta brevemente o Modelo Relacional, discutindo questões relativas ao mapeamento objeto-relacional. Além disso, é abordado também o uso de *frameworks* de mapeamento objeto-relacional no projeto da persistência. O capítulo encerra com o projeto do Componente de Gerência de Dados, responsável por isolar o mecanismo de persistência dos demais elementos da arquitetura do sistema.
- Capítulo 7 – *Projeto de Classes e Avaliação da Qualidade do Projeto de Software*: discute o projeto detalhado de classes, seus atributos, associações e métodos, bem como aspectos relacionados à avaliação da qualidade do projeto de software.

Além dos capítulos anteriormente citados, este texto contém, ainda, um anexo:

- Anexo A – *Padrões de Projeto*: apresenta alguns padrões de projeto (*design patterns*) propostos por Gamma et al. (1995).

Capítulo 2 – Requisitos e o Projeto de Software

O projeto é o processo criativo de transformar uma especificação de um problema em uma especificação de uma solução. O projeto de software utiliza-se da especificação de requisitos e dos modelos conceituais gerados na fase de levantamento e análise de requisitos. A partir dos requisitos, muitas soluções são possíveis e, portanto, muitos projetos diferentes podem ser produzidos. Uma solução é considerada adequada ao problema, se ela satisfizer os requisitos especificados (PFLEEGER, 2004). Assim, o projeto é também uma atividade de tomada de decisão. Em suma, após ter analisado o problema, pode-se decidir como projetar uma solução.

Em função das limitações da tecnologia e outras restrições, várias decisões devem ser tomadas de modo a tratar, sobretudo, requisitos não funcionais. A tecnologia é passível de falhas e muitos são os impactos de sua imperfeição, tais como necessidade de uso de diferentes processadores, necessidade de distribuição e comunicação, necessidade de redundância (i.e., repetição de dados e atividades, e inclusão de dados derivados, tais como totalizadores) e necessidade de inclusão de novas atividades e funções, acrescidas em função de requisitos não funcionais (p.ex., funções de autenticação e autorização requeridas para dar segurança contra acessos indevidos).

Este capítulo discute princípios gerais de projeto e a importância dos requisitos não funcionais nessa fase do processo de desenvolvimento de software. A Seção 2.1 discute princípios gerais de projeto e sua aplicação ao projeto de software. A Seção 2.2 aborda a qualidade do projeto de software. A Seção 2.3 trata da estreita relação entre o projeto e os requisitos não funcionais que definem atributos de qualidade para o sistema em desenvolvimento. A Seção 2.4 aborda a especificação de requisitos não funcionais. A Seção 2.5 discute táticas para tratar requisitos não funcionais, as quais podem ser usadas para incorporar atributos de qualidade ao software. A Seção 2.6 discute a revisão da especificação de requisitos, com vistas a considerar as táticas definidas para tratar requisitos não funcionais. A Seção 2.7 introduz o tema reutilização no projeto de software, com destaque para os padrões (*patterns*). Finalmente, a Seção 2.8 trata da documentação das atividades do projeto de software.

2.1 – Princípios de Projeto

Seja o exemplo do projeto de uma casa. Obviamente, a construção de uma casa começa com o levantamento dos requisitos do dono da casa, o que inclui, dentre outros, a definição do número e tipo dos cômodos (quartos, salas, banheiros etc), tipos de serviços a serem providos (p.ex., haverá um sistema central de ar condicionado? Haverá um sistema de aquecimento solar?), estilo da casa (rústico, moderno) etc. Restrições também devem ser levantadas, dentre elas: custos e prazos, área disponível para a construção, acessibilidade (p.ex., a casa pode ter mais de um pavimento?), restrições legais (p.ex., legislação vigente do Plano Diretor Urbano) etc.

Projetar uma casa é prover uma solução para o problema colocado, procurando satisfazer os requisitos do dono da casa (o cliente) e as restrições levantadas. Muitos projetos são possíveis. Arquitetos diferentes (ou até o mesmo arquiteto) darão soluções diferentes e o cliente escolherá aquela que melhor satisfizer a todos os requisitos especificados, incluindo as restrições.

Assim, de maneira geral, um projeto deve:

- considerar abordagens alternativas com base nos requisitos do problema, restrições e conceitos de projeto;
- ser rastreável à sua especificação;
- reutilizar soluções bem sucedidas previamente adotadas;
- exibir uniformidade (estilo) e integração (interfaces bem definidas entre componentes da coisa a ser construída);
- ser estruturado para acomodar mudanças;
- ser passível de avaliação da qualidade;
- ser revisado para minimizar erros.

Além disso, em geral, um modelo de projeto deve:

- prover uma visão da totalidade da coisa a ser construída;
- decompor o todo em partes e prover diferentes visões da coisa;
- refinar e descrever com mais detalhes cada parte ou visão da coisa, de modo a prover orientação para a construção de cada detalhe.

No exemplo do projeto de uma casa, plantas baixas e maquetes (ou desenhos em três dimensões) podem ser usadas para prover uma visão geral da casa, segundo perspectivas diferentes, interna e externa, respectivamente. O todo pode ser decomposto em partes e modelos específicos podem ser construídos, como plantas baixas para o primeiro piso e para o segundo piso. Diferentes visões podem ser trabalhadas, tais como um projeto tratando apenas do sistema elétrico da casa e outro tratando do sistema hidráulico. Por fim, informações mais detalhadas devem ser providas para cada parte ou visão da casa, tal como o cálculo estrutural para a fundação (lajes, colunas etc) ou o detalhamento do projeto elétrico, contendo informações sobre a fiação, dutos etc.

As características citadas anteriormente valem tanto para o projeto de uma casa, quanto para o projeto de um sistema de software. Colocando-os de forma mais específica, o projeto de software deve:

- considerar abordagens alternativas com base nos requisitos (funcionais e não funcionais) e conceitos de projeto de software;
- estar relacionado aos modelos de análise e à especificação de requisitos e deve ser a eles rastreado;
- reutilizar padrões de projeto, componentes, *frameworks* e outras soluções que se mostraram eficazes em outros projetos, sobretudo aqueles similares ao sistema em desenvolvimento;
- exibir uniformidade (estilo) e integração (interfaces entre componentes);
- ser estruturado para acomodar mudanças (alterabilidade);
- ser passível de avaliação da qualidade;
- ser revisado para minimizar erros.

Além disso, o projeto de software deve:

- minimizar a distância conceitual e semântica entre o software e o mundo real. Os modelos de projeto devem ser facilmente compreensíveis, tendo em vista que seu propósito é comunicar informações para profissionais responsáveis pela codificação, teste e manutenção;
- acomodar circunstâncias não usuais. Se necessário abortar o processamento, fazê-lo de modo elegante;
- apresentar nível de abstração superior ao código fonte, afinal, projeto não é codificação.

Por fim, modelos de projeto também devem ser construídos com o objetivo de prover uma visão geral do sistema a ser construído, bem como uma variedade de visões mais específicas de seus elementos, de modo a guiar a implementação. Um modelo da arquitetura do sistema pode ser usado para prover uma visão geral da organização do sistema. Modelos específicos podem detalhar os diversos elementos da arquitetura. Diferentes diagramas podem ser usados para prover diferentes visões desses elementos, tais como diagramas de classes para uma visão estrutural e diagramas de sequência para uma visão comportamental. Por fim, devem-se projetar as classes que compõem cada um dos elementos da arquitetura, definindo detalhes de como implementar atributos, associações e métodos.

Além dos princípios gerais de projeto, Hooker (1996, apud PRESSMAN, 2006) enumera sete princípios gerais da Engenharia de Software que se aplicam também ao projeto de software. São eles:

- Um sistema de software existe para fornecer valor aos clientes e usuários. Todas as decisões, inclusive as de projeto, devem ser tomadas tendo isso em mente.
- Todo projeto de software deve ser tão simples quanto possível, sem, no entanto, descartar características de qualidade importantes em nome da simplicidade.
- O comprometimento com a visão arquitetural do sistema é essencial para o sucesso do projeto de software.
- Os modelos elaborados na fase de projeto serão usados posteriormente por desenvolvedores responsáveis pela implementação, teste e manutenção do sistema. Assim, esses modelos devem ser claros, não ambíguos e fáceis de entender.
- Um sistema com um longo tempo de vida tem mais valor. Contudo, para ter vida longa, um sistema deve ser projetado para acomodar mudanças.
- A reutilização pode ajudar a poupar tempo e esforço, bem como aumentar a qualidade do sistema em desenvolvimento. Para conseguir um bom nível de reutilização, é necessário planejar o reúso com antecedência. Para a fase de projeto, existem muitos padrões arquitetônicos e padrões de projeto detalhado (*design patterns*) bastante maduros e documentados. Conhecer padrões e comunicar essas e outras oportunidades de reúso para os membros da organização é vital.
- Raciocinar clara e completamente antes de realizar uma ação quase sempre produz melhores resultados. Aprender com os erros também é importante. Assim, ao raciocinar sobre uma decisão de projeto, soluções anteriores devem ser pesquisadas.

Por fim, uma vez que a fase de projeto é essencialmente uma atividade de modelagem, princípios da modelagem ágil (AMBLER, 2004) também se aplicam. Dentre eles, merecem destaque:

- Seja econômico. Não crie mais modelos do que você precisa. Seja capaz de declarar um objetivo para cada modelo criado.
- Procure produzir modelos mais simples.
- Construa modelos de modo que sejam passíveis de mudanças.
- Obtenha *feedback* tão logo quanto possível.

2.2 – Qualidade do Projeto de Software

Um bom projeto de software de qualidade deve apresentar determinadas características de qualidade, tais como facilidade de entendimento, facilidade de implementação, facilidade de realização de testes, facilidade de modificação e tradução correta das especificações de requisitos e de análise (PFLEEGER, 2004). Para se obter bons projetos, é necessário considerar alguns aspectos intimamente relacionados com a qualidade de projeto (design), dentre eles (PRESSMAN, 2006):

- **Níveis de Abstração**: a abstração é um dos modos fundamentais pelos quais os seres humanos enfrentam a complexidade. Assim, um bom projeto deve considerar vários níveis de abstração, começando com em um nível mais alto, próximo da fase de análise. À medida que se avança no processo de projeto, o nível de abstração deve ser reduzido. Dito de outra maneira, o projeto deve ser um processo de refinamento, no qual o projeto vai sendo conduzido de níveis mais altos para níveis mais baixos de abstração.

- **Modularidade**: um bom projeto deve estruturar um sistema como módulos ou componentes coesos e fracamente acoplados. A modularidade permite a um projeto de sistema ser intelectualmente gerenciável. A estratégia de "dividir para conquistar" é reconhecidamente útil no projeto de software, pois é mais fácil resolver um problema complexo quando o mesmo é dividido em partes menores e, por conseguinte, mais facilmente gerenciáveis.

- **Ocultação de Informações**: o conceito de modularidade leva o projetista a uma questão fundamental: até que nível a decomposição deve ser aplicada? Em outras palavras, quão modular deve ser o software? O princípio da ocultação de informações sugere que os módulos / componentes sejam caracterizados pelas decisões de projeto que cada um deles esconde dos demais. Módulos devem ser projetados e especificados de modo que as informações neles contidas (dados e algoritmos) sejam inacessíveis a outros módulos, sendo necessário conhecer apenas a sua interface. Ou seja, a ocultação de informação trabalha encapsulando detalhes que provavelmente serão alterados de forma independente, em módulos distintos. A interface de um módulo revela apenas aqueles aspectos considerados improváveis de mudar (BASS; CLEMENTS; KAZMAN, 2003).

- **Independência Funcional**: a independência funcional é uma decorrência direta dos conceitos de abstração, modularidade e ocultação de informações. Ela é obtida pelo desenvolvimento de módulos com finalidade única e pequena interação com outros módulos. Em outras palavras, módulos devem cumprir uma função bem estabelecida, minimizando interações com outros módulos. Módulos funcionalmente independentes são mais fáceis de entender, codificar, testar e alterar. Efeitos colaterais causados pela modificação de um módulo são

limitados e, por conseguinte, a propagação de erros é reduzida. A independência funcional pode ser avaliada usando dois critérios de qualidade: coesão e acoplamento. A coesão se refere às responsabilidades atribuídas a um módulo. Uma classe, p.ex., é dita coesa quando tem um conjunto pequeno e focado de responsabilidades e aplica seus atributos e métodos especificamente para implementar essas responsabilidades. Já o acoplamento diz respeito ao grau de interdependência entre dois módulos. O objetivo é minimizar o acoplamento, isto é, tornar os módulos tão independentes quanto possível. Idealmente, classes de projeto em um subsistema deveriam ter conhecimento limitado de classes de outros subsistemas. Coesão e acoplamento são interdependentes e, portanto, uma boa coesão deve levar a um baixo acoplamento. A Figura 2.1 procura ilustrar este fato em um cenário envolvendo a distribuição de edificações e fábricas em duas cidades.

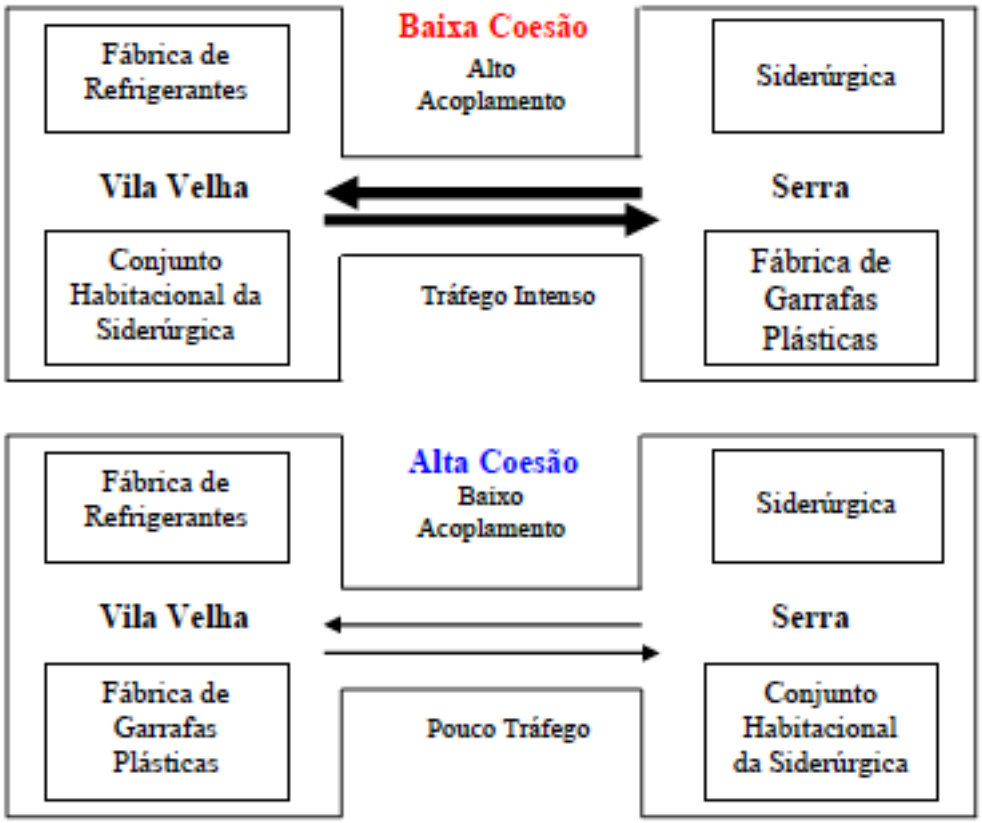


Figura 2.1 – Coesão e Acoplamento

2.3 – Projeto e Atributos de Qualidade

Conforme citado anteriormente, a fase de projeto é responsável por incorporar requisitos tecnológicos aos requisitos essenciais. Assim, o projetista deve estar atento aos critérios de qualidade que o sistema deve atender. As considerações de negócio determinam as características de qualidade que devem ser acomodadas em um sistema. Essas características de qualidade vão além das funcionalidades, ainda que estejam fortemente relacionadas a elas. De fato, funcionalidades e atributos de qualidade são ortogonais. Muitos sistemas são reconstruídos não porque são funcionalmente deficientes (os substitutos são frequentemente idênticos funcionalmente aos sistemas antigos), mas sim porque são difíceis de manter, portar, escalar ou porque são muito lentos ou inseguros (BASS; CLEMENTS; KAZMAN, 2003). Isso mostra a importância de considerar atentamente os requisitos não funcionais durante a fase de projeto.

São muitos os atributos de qualidade que potencialmente podem ser importantes para um sistema. Por exemplo, o modelo de qualidade de produtos de software definido na norma ISO/IEC 25010, utilizado como referência para a avaliação de produtos de software, define oito características de qualidade para produtos de software, desdobradas em subcaracterísticas (ISO/IEC, 2011):

- Adequação Funcional: grau em que o produto provê funções que satisfazem às necessidades explícitas e implícitas, quando usado em condições especificadas. Inclui subcaracterísticas que evidenciam a existência de um conjunto de funções e suas propriedades específicas, a saber:
  - Completude Funcional: capacidade do produto de software de prover um conjunto apropriado de funções para tarefas e objetivos do usuário especificados. Refere-se às necessidades declaradas. Um produto no qual falte alguma função requerida não apresenta este atributo de qualidade.
  - Correção Funcional (ou Acurácia): grau em que o produto de software fornece resultados corretos e precisos, conforme acordado. Um produto que apresente dados incorretos, ou com a precisão abaixo dos limites definidos como toleráveis, não apresenta este atributo de qualidade.
  - Aptidão Funcional: capacidade do produto de software de facilitar a realização das tarefas e objetivos do usuário. Refere-se às necessidades implícitas.
- Confiabilidade: grau em que o produto executa as funções especificadas com um comportamento consistente com o esperado, por um período de tempo. A confiabilidade está relacionada com as falhas que um produto apresenta e como este produto se comporta em situações consideradas fora do normal. Suas subcaracterísticas são:
  - Maturidade: é uma medida da frequência com que o produto de software apresenta falhas ao longo de um período estabelecido de tempo. Refere-se, portanto, à capacidade do produto de software de evitar falhas decorrentes de defeitos no software, mantendo sua operação normal ao longo do tempo.
  - Disponibilidade: capacidade do produto de software de estar operacional e acessível quando seu uso for requerido.

  - Tolerância a falhas: capacidade do produto de software de operar em um nível de desempenho especificado em casos de falha no software ou no hardware. Esta subcaracterística tem a ver com a forma como o software reage quando ocorre uma situação externa fora do normal (p.ex., conexão com a Internet interrompida).
  - Recuperabilidade: capacidade do produto de software de se colocar novamente em operação, restabelecendo seu nível de desempenho especificado, e recuperar os dados diretamente afetados no caso de uma falha.

- Usabilidade: grau em que o produto apresenta atributos que permitem que o mesmo seja entendido, aprendido e usado, e que o tornem atrativo para o usuário. Tem como subcaracterísticas:
  - Reconhecimento da Adequação: grau em que os usuários reconhecem que o produto de software é adequado para suas necessidades.
  - Inteligibilidade: refere-se à facilidade de o usuário entender os conceitos-chave do produto e aprender a utilizá-lo, tornando-se competente em seu uso.
  - Operabilidade: refere-se à facilidade do usuário operar e controlar o produto de software.
  - Proteção contra Erros do Usuário: capacidade do produto de software de evitar que o usuário cometa erros.
  - Estética da Interface com o Usuário: capacidade do produto de software de ser atraente ao usuário, lhe oferecendo uma interface com interação agradável.
  - Acessibilidade: capacidade do produto de software ser utilizado por um amplo espectro de pessoas, incluindo portadores de necessidades especiais e com limitações associadas à idade.
- Eficiência de Desempenho: capacidade de o produto manter um nível de desempenho apropriado em relação aos recursos utilizados em condições estabelecidas. Inclui subcaracterísticas que evidenciam o relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizados, a saber:
  - Comportamento em Relação ao Tempo: capacidade do produto de software de fornecer tempos de resposta e de processamento apropriados, quando o software executa suas funções, sob condições estabelecidas.
  - Utilização de Recursos: capacidade do produto de software de usar tipos e quantidades apropriados de recursos, quando executa suas funções, sob condições estabelecidas.
  - Capacidade: refere-se ao grau em que os limites máximos dos parâmetros do produto de software (p.ex., itens que podem ser armazenados, número de usuários concorrentes etc.) atendem às condições especificadas.
- Segurança: grau em que informações e dados são protegidos contra o acesso por pessoas ou sistemas não autorizados, bem como grau em que essas informações

e dados são disponibilizados para pessoas ou sistemas com acesso autorizado. Tem como subcaracterísticas:

- Confidencialidade: garantir acesso a quem tem autorização para tal.
- Integridade: impedir ao acesso a quem não tem autorização para tal.
- Não Repúdio: garantia de que a ocorrência de ações ou eventos possa ser provada, evitando-se questionamentos futuros.
- Responsabilização: ações realizadas por uma pessoa ou sistema devem poder ser rastreadas de modo a comprovar por quem foram feitas.
- Autenticidade: capacidade de o sistema avaliar a identidade de um usuário ou recurso.

- Compatibilidade: capacidade do produto de software de trocar informações com outras aplicações e/ou compartilhar o mesmo ambiente de hardware ou software. Suas subcaracterísticas são:
  - Coexistência: capacidade do produto de software de coexistir com outros produtos de software independentes, em um ambiente compartilhando recursos comuns.
  - Interoperabilidade: capacidade do produto de software de interagir com outros sistemas especificados, trocando e usando as informações trocadas.
- Manutenibilidade: capacidade do produto de software de ser modificado. Tem como subcaracterísticas:
  - Modularidade: o produto de software deve possuir componentes coesos e fracamente acoplados, de modo que uma modificação em um componente tenha impacto mínimo em outros componentes.
  - Reusabilidade: capacidade dos componentes do produto de software serem utilizados na construção de outros componentes ou sistemas.
  - Analisabilidade: capacidade do produto de software de permitir o diagnóstico de deficiências ou de causas de falhas, bem como a identificação das partes a serem modificadas.
  - Modificabilidade: grau em que o produto de software pode ser modificado sem introduzir novos defeitos ou degradar a qualidade do produto, permitindo que uma modificação seja eficazmente implementada.
  - Testabilidade: capacidade do produto de software de permitir que, quando modificado, seja testado para determinar se as alterações foram propriamente realizadas e não introduziram efeitos colaterais.

- Portabilidade: refere-se à capacidade do software ser transferido de um ambiente de hardware ou software para outro. Tem como subcaracterísticas:
  - Adaptabilidade: capacidade do produto de software de ser adaptado para diferentes ambientes especificados, sem necessidade de aplicação de outras ações ou meios além daqueles fornecidos para essa finalidade pelo próprio software.
  - Capacidade para ser instalado (instalabilidade): avalia a facilidade de se instalar (e desinstalar) o produto de software em um ambiente especificado.
  - Capacidade de substituição (substituibilidade): capacidade do produto de software de ser usado em substituição a outro produto de software especificado, com o mesmo propósito e no mesmo ambiente. Considera, também, a facilidade de atualizar uma nova versão.

Além das características de qualidade que se aplicam diretamente ao sistema, ditas atributos de qualidade de produto, Bass, Clements e Kazman (2003) listam outras características relacionadas a metas de negócio, dentre elas: tempo para chegar ao mercado (*time to market*), custo-benefício, tempo de vida projetado para o sistema, mercado alvo, cronograma e orçamento do projeto, e integração com sistemas legados.

Um problema chave para o projeto é definir prioridades para tratar requisitos não funcionais conflitantes. Por exemplo, normalmente ao se melhorar o desempenho de uma porção de um sistema diminui-se a sua capacidade de acomodar mudanças. Ou seja, torna-se mais difícil alterar o sistema. Assim, uma importante atividade do projeto de sistemas é avaliar os requisitos não funcionais e resolver requisitos conflitantes. A base para essa decisão deve ser a importância relativa das várias características levantadas para o sistema em questão (BLAHA; RUMBAUGH, 2006).

Deve-se observar que, embora os requisitos não funcionais tenham cunho tecnológico, eles, assim como os requisitos funcionais, devem ser levantados junto aos clientes e usuários. Dentre outras, as seguintes informações devem ser levantadas:

- Qual a localização geográfica dos usuários? Há necessidade de transporte de dados?
- Quais são problemas operacionais existentes nas atividades dos usuários? Qual será o ambiente de hardware e software de produção? Há restrições técnicas (novo ambiente?) ou ambientais (p.ex., temperatura)?
- Qual a frequência de disparo das operações do sistema? Qual o tempo de resposta esperado para cada uma delas?
- Qual o volume de dados esperado (inicial, estimativa de crescimento e política de esvaziamento)?
- Há restrições de confiabilidade (tempo mínimo entre falhas)?
- Há restrições de segurança (classes de usuários e acesso)?
- Quais as características desejadas para a interface com o usuário?

2.4 – Especificação de Requisitos Não Funcionais

Assim como os requisitos funcionais, os requisitos não funcionais (RNFs) precisam ser especificados. Essa especificação deve ser tal que o RNF seja passível de avaliação. Não basta dizer coisas como "o sistema deve ser fácil de manter" ou "o sistema deve ter uma interface com o usuário amigável". O que significa "fácil de manter" ou "interface amigável"? Os RNFs têm de ser especificados de forma tal que seja possível posteriormente avaliar se os mesmos foram atendidos ou não.

Uma forma de especificar requisitos não funcionais é prover uma descrição associada a um critério de aceitação. Este último deve ser testável e, para tal, medidas objetivas devem ser providas.

A ISO/IEC 9126[2] pode ser uma boa fonte de medidas. As partes 2 (Medidas Externas) (ISO/IEC, 2003a) e 3 (Medidas Internas) (ISO/IEC, 2003b) dessa norma apresentam diversas medidas que podem ser usadas para especificar objetivamente os RNFs. Nessas partes da norma, medidas são sugeridas para as diversas sub-características descritas na Parte 1, indicando, dentre outros nome e propósito da medida, método de aplicação e fórmula, e como interpretar os valores da medida.

Seja o exemplo em que um sistema de locação de carros que, associado ao requisito funcional "Efetuar Locação de Carro", tem como requisito não funcional: "ser fácil de aprender". Este RNF poderia ser especificado usando a medida "Facilidade de aprender a realizar uma tarefa em uso", definida conforme mostrado na Tabela 2.1. A especificação do RNF é mostrada na Tabela 2.2, indicando seu critério de aceitação.

Tabela 2.1 – Medida "Facilidade de aprender a realizar uma tarefa em uso".

Medida	Facilidade de aprender a realizar uma tarefa em uso
Propósito	Quanto tempo o usuário leva para aprender a realizar uma tarefa especificada eficientemente?
Método de Aplicação	Observar o comportamento do usuário desde quando ele começa a aprender até quando ele começa a operar eficientemente a funcionalidade.
Medição	T = soma do tempo de operação do usuário até que ele consiga realizar a tarefa em um tempo especificado.

Tabela 2.2 – Especificação de Requisito Não Funcional.

RNF01 – A funcionalidade "Efetuar Locação de Carro" deve ser fácil de aprender.	
Medida:	Facilidade de aprender a realizar uma tarefa em uso
Critério de Aceitação:	T <= 15 minutos, considerando que o usuário está operando o sistema eficientemente quando a tarefa "Efetuar Locação de Carro" é realizada em um tempo inferior a 2 minutos.

A Tabela 2.3 mostra exemplos de medidas que podem ser usadas para a especificação e avaliação de requisitos não funcionais, algumas delas extraídas e adaptadas de (ISO/IEC, 2003a).

[2] A família de normas ISO/IEC 9126 encontra-se em fase de transição para a nova família de normas, a ISO/IEC 25000 - *Software Product Quality Requirement and Evaluation* (SQuaRE). Em especial, medidas para a qualidade de produtos e sistemas são objeto da futura norma ISO/IEC 25023.

Tabela 2.3 – Exemplos de Medidas para a Especificação e Avaliação de RNFs

Subcaracterística de Qualidade	Medida	Propósito da Medida	Procedimento de Medição	Fórmula de Medição	Interpretação do valor medido	Fonte de Entrada para Medição
Interoperabilidade	**Consistência de interface (protocolo)**	Quão corretamente os protocolos de interface foram implementados?	Contar o número de protocolos de interface corretamente implementados (NPIC) conforme definido nas especificações e comparar com o número total de protocolos de interface a serem implementados (NPIT) como definido nas especificações.	X=NPIC/NPIT	0 <= X <= 1. Quanto mais próximo de 1, mais consistente.	Especificação de requisitos, Design, Código fonte
Responsabilização	**Auditabilidade de acesso**	Quão auditável é o login de acesso?	Contar o número de tipos de acesso que estão sendo logados corretamente (NALC) conforme definido nas especificações e comparar com o número total de tipos de acesso que são requeridos (NART) como definido nas especificações.	X=NALC/NART	0 <= X <= 1. Quanto mais próximo de 1, mais auditável.	Especificação de requisitos, Design, Código fonte
Maturidade	**Tempo médio entre falhas**	Quanto tempo o produto de software opera sem apresentar falhas?	Calcular o tempo total de funcionamento correto em um período (TTFC) e dividir pelo número de falhas no período (NF), obtendo o tempo médio entre falhas.	X=TTFC / NF	0 <= X. Quanto maior o valor, mais maduro.	Registros de falhas.
Disponibilidade	**Probabilidade de estar disponível**	Qual a probabilidade do produto de software estar disponível?	Calcular o tempo médio entre falhas (TMF) e o tempo médio de reparo (TMR), derivando a probabilidade conforme a fórmula de medição dada.	X= TMF / (TMF+TMR)	0 <= X <= 1. Quanto mais próximo de 1, mais disponível.	Registros de falhas e de controle de alterações.
Reconhecimento da Adequação	**Funções Evidentes**	Qual a proporção de funções do produto de software que estão evidentes para os usuários?	Contar o número de funções evidentes para o usuário (NFE) e comparar com o número de funções total (NFT).	X=NFE/NFT	0 <= X <= 1. Quanto mais próximo de 1, melhor.	Especificação de requisitos, Projeto de IU
Inteligibilidade	**Inteligibilidade de funções**	Qual a proporção de funções do produto de software corretamente entendidas pelos usuários?	Contar o número de funções de interface com o usuário facilmente compreensíveis pelo usuário (NFIC) e comparar com o número de funções de interface com o usuário total (NFIT).	X=NFIC/NFIT	0 <= X <= 1. Quanto mais próximo de 1, melhor.	Projeto de IU
Operabilidade	**Checagem de validade de entrada**	Qual a proporção de itens de entrada que checa se os dados são válidos?	Contar o número de itens de entrada com checagem de validade (NIEC) e comparar com o número de itens de entrada passíveis de serem checados (NIEPC).	X=NIE/NIEPC	0 <= X <= 1. Quanto mais próximo de 1, melhor.	Especificação de requisitos, Projeto de IU
Operabilidade	**Habilidade de desfazer**	Qual a proporção de funções que podem ser desfeitas?	Contar o número funções que podem ser desfeitas pelo usuário após completadas (NFU) e comparar com o número total de funções (NTF).	X=NFU/NTF	0 <= X <= 1. Quanto mais próximo de 1, melhor.	Especificação de requisitos, Projeto de IU

2.5 – Táticas para Tratar Atributos de Qualidade

Entender os atributos de qualidade e definir os níveis em que eles devem ser atingidos é muito importante. Contudo, é necessário definir como atingir esses níveis. Assim, deve-se definir que estratégias serão aplicadas para atingir essas metas.

Bass, Clements e Kazman (2003) enumeram um conjunto de táticas que podem ser aplicadas para tratar certos atributos de qualidade. Uma tática, neste contexto, é uma decisão de projeto que influencia o controle de certo atributo de qualidade. Uma coleção de táticas define uma estratégia de projeto arquitetônico. A seguir, algumas dessas táticas são brevemente discutidas, bem como abordagens propostas por outros autores para tratar requisitos não funcionais.

2.5.1 – Confiabilidade

Um defeito (*fault*) é uma imperfeição do sistema e, em geral, refere-se a algo que está implementado de maneira incorreta. Uma falha (*failure*) é um resultado incorreto, visível, provocado por um defeito (KOSCIANSKI; SOARES, 2006). Uma falha ocorre quando o sistema não entrega mais um serviço consistente com sua especificação e essa falha é perceptível ao usuário (BASS; CLEMENTS; KAZMAN, 2003). Assim, defeitos (ou uma combinação de defeitos) podem provocar a ocorrência de falhas. Contudo, os defeitos podem existir sem provocarem falhas.

Recuperação e reparo são importantes aspectos para a confiabilidade. Todas as abordagens de confiabilidade envolvem alguma combinação de redundância, monitoração da saúde do sistema para detectar falhas e alguma forma de recuperação a ser aplicada quando uma falha é detectada (BASS; CLEMENTS; KAZMAN, 2003). Assim, as táticas de confiabilidade apresentadas em (BASS; CLEMENTS; KAZMAN, 2003) são divididas em três grupos: táticas de detecção de falhas, táticas de recuperação de falhas e táticas de prevenção de falhas. A Tabela 2.4 apresenta as táticas de confiabilidade propostas por esses autores.

Tabela 2.4 – Táticas de Confiabilidade (BASS; CLEMENTS; KAZMAN, 2003).

Detecção de Falha	
Silvo/Eco (*Ping/echo*)	Um componente envia um silvo (som agudo prolongado) e espera receber um eco de volta, dentro de um tempo predefinido, do componente sendo examinado. Ex.: clientes enviam um silvo para verificar se o servidor e o caminho de comunicação até ele estão operando dentro do limites de desempenho esperados.
Batida de Coração (*Heartbeat*)	Um componente emite mensagens periódicas (batida de coração) e outro componente as escuta. Se a batida de coração falhar, assume-se que o componente de origem falhou e um componente de recuperação de falha é notificado.
Exceções (*Exceptions*)	Uma exceção é gerada quando uma falha é detectada. Um manipulador de exceção é então acionado para tratá-la. O manipulador de exceção opera dentro do mesmo processo que gerou a exceção e geralmente realiza uma transformação da falha em uma forma que possa ser processada.
Votação (*Voting*)	Processos rodando em processadores redundantes recebem entradas equivalentes e computam a saída que é enviada para um votante (*voter*). Se o votante detectar um comportamento desviante de um dado processador, ele gera uma falha no correspondente componente.

Tabela 2.4 – Táticas de Confiabilidade (BASS; CLEMENTS; KAZMAN, 2003) (continuação)

Recuperação de Falha	
Redundância Ativa (*Active Redundancy*)	Componentes redundantes respondem a eventos em paralelo. Apenas a resposta de um deles é utilizada (geralmente o primeiro a responder) e o restante é descartado. Em sistemas distribuídos que precisam de alta disponibilidade, a redundância pode ocorrer nos caminhos de comunicação.
Redundância Passiva (*Passive Redundancy*)	Um componente (o principal) responde aos eventos e informa os demais (os componentes em espera – *standbys*) sobre as atualizações de estado que ele fez. Quando uma falha ocorre, o sistema precisa primeiro garantir que o estado de backup é suficientemente recente para prosseguir.
Sobressalente (*Spare*)	Uma plataforma de computação sobressalente em espera é configurada para substituir componentes com falha. Essa plataforma tem de ser iniciada com a configuração apropriada e o estado recuperado. Para tal, periodicamente o estado do sistema deve ser armazenado, bem como se devem registrar as alterações feitas no dispositivo de persistência.
Operação em Sombra (*Shadow Operation*)	Um componente que falhou anteriormente, para voltar a executar normalmente, precisa primeiro rodar em "modo sombra" por um pequeno período de tempo para garantir que ele tem o mesmo comportamento do componente que está funcionando correntemente.
Ressincronização de Estado (*State Resynchronization*)	As táticas de redundância ativa e passiva requerem que um componente sendo restaurado tenha seu estado atualizado antes de retornar ao serviço.
Ponto de Verificação / Reversão (*Checkpoint / Rollback*)	Um ponto de verificação, criado periodicamente ou em resposta a eventos específicos, registra um estado consistente. O sistema, após a ocorrência de uma falha, deve ser restaurado usando um ponto de verificação de um estado consistente e um registro (*log*) das transações que ocorreram após o ponto de verificação.
Prevenção de Falha	
Retirada de Serviço (*Removal from service*)	Um componente é removido do sistema em operação para ser submetido a atividades que previnam falhas antecipadamente. Ex.: reiniciar um componente para evitar, por exemplo, estouro de pilha.
Transações (*Transactions*)	Uma transação é a agregação de diversos passos sequenciais de modo que o pacote como um todo possa ser desfeito de uma só vez. Transações são usadas para prevenir que dados sejam afetados caso um dos passos do processo falhe, bem como para prevenir colisões entre *threads* simultâneas acessando os mesmos dados.
Monitor de processo (*Process monitor*)	Uma vez que uma falha é detectada em um processo, um processo de monitoramento pode excluir o processo que não está executando e criar uma nova instância do mesmo.

2.5.2 – Desempenho

Desempenho está dentre os mais importantes requisitos não funcionais. Ele depende fortemente da interação (e do tipo da interação) existente entre os componentes de um sistema e, portanto, está muito associado à arquitetura do mesmo. Contudo, é geralmente difícil lidar com este atributo de qualidade, uma vez que ele conflita com vários outros, tais como confiabilidade e manutenibilidade. De maneira geral, requisitos

de desempenho impõem restrições relativas à velocidade de operação de um sistema. Isso pode ser especificado em termos de (MENDES, 2002):

- Requisitos de tempo de resposta: especificam o tempo de resposta aceitável para certas funcionalidades do sistema;
- Requisitos de processamento (*throughput*): especificam restrições relativas à quantidade de processamento (p.ex., número de transações) realizada em um determinado período de tempo;
- Requisitos de temporização: especificam quão rapidamente o sistema deve coletar dados de entrada de sensores, antes que novas leituras sejam feitas, sobrescrevendo os dados anteriores;
- Requisitos de recursos: estabelecem condições (memórias principal e secundária, velocidade do processador etc.) para o bom funcionamento do sistema.

Bass, Clements e Kazman (2003) listam diversas táticas para tratar requisitos de desempenho. O propósito dessas táticas é gerar uma resposta a um evento que chega ao sistema dentro de alguma restrição de tempo. Após a chegada do evento, o sistema pode estar processando sobre esse evento ou o processamento pode estar bloqueado por alguma razão. Assim, há dois fatores principais que influenciam o tempo de resposta: consumo de recursos e tempo de bloqueio. Recursos incluem processadores, bases de dados, redes de comunicação e memória. Um processamento pode estar impedido de usar um recurso devido à disputa pelo mesmo, devido ao fato do recurso estar indisponível ou porque a computação depende do resultado de outros componentes que não estão disponíveis. Com base nessas considerações, as táticas de desempenho propostas são agrupadas em:

- Táticas relativas à demanda de recursos: visam tentar diminuir a demanda por recursos. Como os fluxos de eventos são a fonte da demanda por recursos, essas táticas se preocupam em diminuir a frequência da ocorrência de eventos ou a quantidade de recursos consumidos por cada requisição.
- Táticas relativas à gerência de recursos: visam gerenciar os recursos, normalmente tornando mais recursos disponíveis, de modo a melhorar o desempenho.
- Táticas relativas à arbitragem de recursos: sempre que houver disputa por recursos, é necessário escaloná-los. Processadores, *buffers* e redes são escalonados. Assim, essas táticas referem-se à escolha da política de escalonamento compatível com cada recurso, visando melhorar o desempenho.

A Tabela 2.5 apresenta algumas das táticas de desempenho propostas em (BASS; CLEMENTS; KAZMAN, 2003).

Tabela 2.5 – Táticas de Desempenho (BASS; CLEMENTS; KAZMAN, 2003).

Demanda de Recursos	
Aumentar a eficiência computacional	Uma forma de diminuir o tempo de resposta consiste em aperfeiçoar os algoritmos usados em partes críticas do software.
Reduzir *overhead* computacional	O uso de intermediários, tão importantes para a manutenibilidade, aumenta o consumo de recursos no processamento de um evento e, portanto, removê-los pode ajudar a diminuir o tempo de resposta. Considerações análogas valem para camadas, sobretudo quando utilizada uma arquitetura em camadas fechada.
Reduzir o número de eventos processados	Se for possível reduzir a frequência de amostragem na qual variáveis do ambiente são monitoradas, então a demanda pode ser reduzida. Quando não se tem controle sobre a chegada de eventos gerados externamente, as solicitações podem ser tratadas com uma frequência menor, com perda de algumas requisições.
Gerência de Recursos	
Introduzir concorrência	Concorrência pode ser introduzida, processando diferentes conjuntos de eventos em diferentes linhas de controle (*threads*) ou criando linhas de controle adicionais para diferentes conjuntos de atividades. Uma vez introduzida concorrência, balancear a carga é importante para explorá-la de maneira ótima. Contudo, essa tática só pode ser aplicada se as requisições puderem ser processadas em paralelo.
Manter múltiplas réplicas de dados e serviços	O propósito de uma réplica é reduzir a contenção que ocorreria se todas as computações ocorressem em um computador central. O uso de *cache* e fragmentação de bases de dados são exemplos dessa tática. *Caching* é uma tática na qual dados são replicados para reduzir contenção. Uma vez que, neste contexto, os dados colocados em *cache* são cópias de dados existentes, manter as cópias consistentes e sincronizadas torna-se uma responsabilidade do sistema.
Aumentar a disponibilidade de recursos.	Processadores mais rápidos, processadores adicionais, memória adicional e redes mais rápidas são táticas para aumentar a disponibilidade de recursos.
Arbitragem de Recursos	
Escolher política de escalonamento	Todas as políticas de escalonamento atribuem prioridades. Além disso, para que um evento de mais alta prioridade seja disparado, é necessário interromper o processo usuário corrente do recurso (preempção). Algumas políticas comuns de escalonamento são: FIFO (*first-in first out*), prioridades fixas e prioridades dinâmicas.

2.5.3 – Segurança

A segurança tem como objetivo não permitir que pessoas ou sistemas não autorizados tenham acesso a eventos ou dados do sistema. Bass, Clements e Kazman (2003) listam algumas táticas para prover segurança, agrupadas em táticas para resistir a ataques, táticas para detectar ataques e táticas para recuperar o sistema de ataques. A Tabela 2.6 apresenta as táticas de segurança propostas por esses autores.

Para controlar o acesso (táticas para resistir a ataques), é necessário identificar, autenticar e autorizar usuários. A identificação se dá através de uma forma unívoca, ou seja, que identifique apenas um usuário. A autenticação consiste em comprovar que o usuário é mesmo quem ele diz ser na identificação, sendo feita, por exemplo, por meio

de uma senha. Finalmente, a autorização é dada ao usuário, ou a uma classe de usuários, dando acesso a determinadas funcionalidades do sistema.

Tabela 2.6 – Táticas de Segurança (BASS; CLEMENTS; KAZMAN, 2003).

Resistir a Ataques	
Autenticar usuários	Autenticação diz respeito a garantir que um usuário ou computador remoto é realmente quem diz ser. Senhas, certificados digitais e identificação biométrica são alguns meios de se prover autenticação.
Autorizar usuários	Autorização diz respeito a garantir que um usuário autenticado tenha o direito de acessar e modificar tanto dados quanto serviços. Isso é feito normalmente provendo alguns padrões de controle de acesso dentro do sistema. O controle de acesso pode ser por usuário ou classe de usuário. Classes de usuários podem ser definidas por grupos de usuários, por papéis de usuário ou por listas de indivíduos.
Manter confidencialidade dos dados	Dados devem ser protegidos contra acesso não autorizado. Confidencialidade é normalmente atingida aplicando alguma forma de criptografia a dados e links de comunicação. Criptografia provê uma proteção extra para dados mantidos em bases de dados, além da autorização. Já links de comunicação tipicamente não têm controles de autorização e, portanto, a criptografia é a única proteção neste caso.
Manter integridade dos dados	A integridade dos dados também deve ser garantida. Para verificar a integridade, informação redundante, tal como soma de verificação, pode ser codificada junto aos dados.
Limitar exposição	A alocação de serviços a servidores pode ser feita de modo que apenas serviços limitados estejam disponíveis em cada servidor.
Limitar acesso	Firewalls podem ser usados para restringir o acesso com base em fonte de mensagens ou porta de destinação. Mensagens de fontes desconhecidas podem ser uma forma de ataque. Contudo, nem sempre é possível limitar o acesso a fontes conhecidas. Um *site* web público, por exemplo, pode esperar receber solicitações de fontes desconhecidas.
Detectar Ataques	
Sistema de detecção de intrusão	Sistemas de detecção de intrusão funcionam comparando padrões de tráfego de rede. No caso de detecção de mau uso, o padrão de tráfego é comparado com padrões históricos de ataques conhecidos. Detectores de intrusão têm de ter, dentre outros, algum tipo de sensor para detectar ataques, bases de dados para registrar eventos para posterior análise, ferramentas para análise e um console de controle que permita ao analista modificar ações de detecção de intrusão.
Recuperar-se de Ataques	
Restaurar estado	As técnicas de recuperação de falhas sugeridas nas táticas de confiabilidade podem ser aplicadas, já que elas se propõem a restaurar um estado consistente a partir de um estado inconsistente. Atenção especial deve ser dada à manutenção de cópias redundantes de dados administrativos do sistema, tais como senhas, listas de controle de acesso, serviços de nomes de domínio e dados de perfis de usuários.
Manter uma trilha de auditoria	Uma trilha de auditoria é um registro das transações aplicadas aos dados do sistema junto com informações de identificação. Essas informações podem ser usadas para trilhar as ações do agressor, apoiar reconhecimento (provendo evidência de que uma particular requisição foi feita) e apoiar a recuperação do sistema. Trilhas de auditoria são frequentemente alvo de ataques e, portanto, devem ser mantidas de modo confiável.

2.5.4 – Usabilidade

A usabilidade está preocupada com o quão fácil é para o usuário realizar uma tarefa desejada e o tipo de apoio provido pelo sistema ao usuário. Envolve, dentre outros, aspectos relativos à facilidade de entender, aprender e operar o sistema.

Bass, Clements e Kazman (2003) definem táticas para apoiar a usabilidade, as quais são organizadas em dois grandes grupos: táticas de tempo de execução e táticas de tempo de projeto. As táticas de tempo de execução, como o próprio nome indica, se referem ao apoio ao usuário durante a execução do sistema. Uma vez que o sistema estiver executando, a usabilidade é reforçada dando feedback ao usuário sobre o quê o sistema está fazendo e provendo ao usuário a capacidade de entrar com comandos que permitam operar o sistema de modo mais eficiente ou corrigir erros, tais como cancelar e desfazer (BASS; CLEMENTS; KAZMAN, 2003).

As táticas de usabilidade em tempo de execução são agrupadas em táticas para apoiar iniciativas do usuário e táticas para apoiar iniciativas do sistema. Quando um usuário toma uma iniciativa, o projetista projeta a resposta tomando por base a funcionalidade do sistema. Quando o sistema toma a iniciativa, o projetista precisa apoiar-se em alguma informação sobre o usuário, a tarefa sendo executada pelo usuário ou sobre o estado do sistema. Assim, são táticas de apoio a iniciativas do sistema (BASS; CLEMENTS; KAZMAN, 2003):

- Manter um modelo da tarefa: o modelo da tarefa é usado para determinar o contexto, de modo que o sistema pode inferir o que o usuário está tentando fazer e prover assistência. Ex.: sabendo que frases começam com letras maiúsculas, um editor de texto pode corrigir um texto sendo digitado.
- Manter um modelo do usuário: esse modelo mantém informações sobre o conhecimento que o usuário tem do sistema, sobre o comportamento do usuário etc. Esse modelo é usado para prover assistência.
- Manter um modelo do sistema: esse modelo mantém informações sobre o comportamento esperado do sistema, de modo a dar um feedback para o usuário. Ex.: prever o tempo necessário para completar uma atividade.

As táticas de usabilidade de tempo de projeto estão fortemente relacionadas às táticas de manutenibilidade. São táticas dessa natureza:

- Separar a interface do restante da aplicação. Diversos padrões arquitetônicos foram desenvolvidos para implementar essa tática, dentre eles o padrão Modelo-Visão-Controlador (MVC) (BASS; CLEMENTS; KAZMAN, 2003).
- Adotar padrões de interface (*design patterns* de interface com o usuário) amplamente utilizados. Para tal, é necessário considerar a(s) plataforma(s) de computação na(s) qual(is) o sistema vai rodar. Ex.: Padrões de navegação, busca, tabelas e listas para aplicativos Android e iOS.
- Prover ao usuário a capacidade de operar o sistema de modo mais eficiente. De fato, há várias táticas que se enquadram nesta categoria, tais como: (i) Permitir, sempre que possível, a entrada por meio de seleção ao invés da digitação de campos; (ii) Prover acesso mais fácil a funcionalidades importantes para o usuário; (iii) Não disponibilizar funcionalidades ou dados que não possam ser

usados pelo usuário; (iv) Prover teclas de atalho (ou outros mecanismos de interação rápida) quando pertinente; (v) Apresentar mensagens de erro e avisos significativas, dentre outras. Para maiores detalhes, vide Capítulo 5.

2.5.5 – Manutenibilidade

Para se obter sistemas fáceis de manter, deve-se, dentre outros, facilitar a localização das alterações (analisabilidade), a realização das alterações (modificabilidade) e os testes (testabilidade). Bass, Clements e Kazman (2003) definem diversas táticas para apoiar a manutenibilidade, organizadas de acordo com suas metas, dentre elas:

- Localização de Alterações: tem como objetivo reduzir o número de módulos que são diretamente afetados por uma alteração. Uma tática desse grupo consiste em manter a coerência semântica, procurando garantir que um módulo trabalha em conjunto com outros, mas sem depender excessivamente deles (independência funcional, alta coesão, baixo acoplamento).
- Prevenção de efeito cascata: tem como objetivo limitar a abrangência de uma modificação somente aos módulos localizados, evitando afetar outros módulos não diretamente envolvidos. São exemplos de táticas desse grupo: (i) ocultação de informação; (ii) garantia das interfaces existentes; e (iii) uso de intermediários, tais como repositórios para dados e padrões de projeto (*design patterns*) para intermediar serviços (p.ex., padrão fachada, mediador etc.).
- Táticas de Testabilidade: visam facilitar os testes de uma porção de software. Dentro desse grupo, merecem destaque táticas relacionadas às entradas e saídas, tais como: (i) separar interface da implementação, o que permite a substituição da implementação para vários propósitos de teste, tal como o uso de *stubs*; (ii) (ii) registro e reprodução, que diz respeito a capturar a informação cruzando uma interface (entradas e saídas) durante a operação normal em algum repositório e utilizá-la para testes.

2.5.6 – Portabilidade

Para se obter sistemas fáceis de portar, deve-se, dentre outros, facilitar a instalação, a substituição de partes do sistema e a adaptação a diferentes plataformas. As táticas de portabilidade estão bastante relacionadas às táticas de manutenibilidade. De fato, algumas delas são as mesmas, tal como garantir interfaces existentes, usar intermediários e separar a interface da aplicação. Além disso, o uso de linguagens, bibliotecas e mecanismos de persistência capazes de rodar em diferentes plataformas operacionais é uma importante tática para tratar a portabilidade.

2.6 – Revisitando a Especificação de Requisitos Funcionais

Em decorrência de requisitos tecnológicos, novos casos de uso podem ser necessários, tais como casos de uso para apoiar atividades de segurança, limpeza, reorganização, cópia e restauração de arquivos, e facilidades de ajuda (*help*).

Seja o caso de segurança. Novos casos de uso e classes devem ser adicionados aos modelos conceituais do sistema produzidos durante a Análise de Requisitos. P.ex., ao se utilizar táticas de autenticação e autorização para controle de acesso, é necessário introduzir casos de uso para gerenciar informações de usuários e seus perfis (inclusão, exclusão alteração e consulta de usuários e perfis), além, é claro, das atividades de autenticação e autorização de usuários. Neste contexto, há duas opções a seguir. Primeiro, pode-se criar uma versão de projeto do modelo de casos de uso, na qual são introduzidos os casos de uso para tratar as táticas selecionadas, bem como para realizar alterações nos casos de uso de análise visando adequá-los para as decisões de projeto tomadas. Diagramas de classes também precisariam ser alterados para acomodar tais mudanças. A segunda opção é não alterar o modelo de casos de uso originalmente produzido na fase de análise de requisitos, e criar um outro modelo de casos de uso contendo apenas os casos de uso tecnológicos, tipicamente agrupados por atributo de qualidade e táticas utilizadas. Neste caso, diagramas de classes correspondentes seriam desenvolvidos também separadamente. Neste texto, sugere-se adotar a segunda opção. Isso porque abre-se espaço para o desenvolvimento de novos pacotes dedicados a certas preocupações (ditos utilitários) que podem ser reutilizados em outros sistemas que decidam aplicar as mesmas táticas.

Seja o caso de controle de acesso anteriormente citado. Poder-se-ia criar um utilitário Controle de Acesso, incluindo os casos de uso e classes mostrados na Figura 2.2. Esse utilitário poderia ser utilizado por vários sistemas. Para capturar aspectos específicos do sistema em desenvolvimento, p.ex., no que tange à definição de perfis, uma matriz *Caso de Uso x Perfil de Usuário* poderia ser utilizada para documentar o nível de autorização adotado no projeto, como ilustra a Figura 2.3.

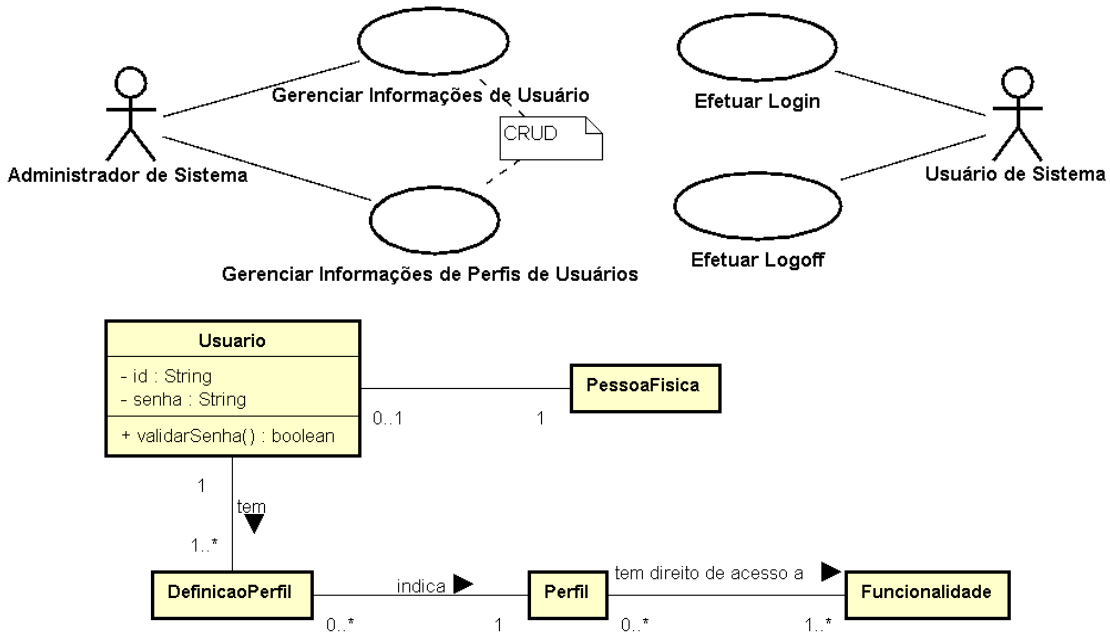


Figura 2.2 – Diagramas de Caso de Uso e de Classes para Utilitário Controle de Acesso.

	Perfis de Usuários		
Casos de Uso	Cliente	Gerente	Funcionário
Efetuar pedido	X	X	X
Cancelar pedido		X	X
Alterar pedido	X	X	X
Solicitar relatório de pedidos		X	

Figura 2.3 – Exemplo de uma Matriz *Caso de Uso x Perfil de Usuário*.

Note que são várias táticas que requerem novos casos de uso. A auditoria da ocorrência de violações também requer novas funcionalidades. Quando for necessário um nível maior de segurança, é importante criar um procedimento de monitoração que registre os usuários que acessaram o sistema e os casos de uso por eles realizados. Um arquivo de histórico (*log*) deve ser mantido e deve ser passível de consulta. O mesmo ocorre em relação à restauração do estado do sistema; ou seja, funcionalidades devem ser providas para tal.

É bom lembrar que todo caso de uso tecnológico projetado é decorrente de um requisito tecnológico acrescido ao sistema, mas nem todo requisito tecnológico necessariamente implica na criação de um caso de uso. Alguns atributos de qualidade, tais como desempenho e usabilidade, são tipicamente incorporados a casos de uso do sistema já existentes.

Além disso, é útil definir tarefas ou unidades de execução, isto é, conjuntos de casos de uso agrupados em uma unidade, do ponto de vista de execução. Alguns critérios que podem ser usados para agrupar casos de uso em unidades de execução incluem:

- Quando existir uma classe de usuários específica, cujos casos de uso autorizados sejam de seu único acesso e interesse.
- Em situações de dispersão geográfica em que diferentes localidades têm demandas diferentes de casos de uso.
- Não há memória disponível suficiente para que o sistema como um todo execute eficientemente.
- Casos de uso são realizados uma única vez (ou muito poucas vezes), tais como conversão de dados e instalação do sistema.

A principal estratégia para agrupar casos de uso em unidades de execução consiste em fazer um levantamento sobre o modo de utilização do sistema, o tempo de resposta esperado, o tamanho e número de casos de uso, aspectos de segurança e facilidade de uso. Além disso, deve-se levar em conta que essas unidades de execução podem ser completamente separadas, dando origem a diferentes programas executáveis, ou que elas podem apenas ter sua ligação sendo feita em tempo de execução.

Por fim, a escolha de táticas para tratar RNFs também ajuda a melhorar a especificação desses requisitos. Assim, é importante adicionar à especificação de RNFs informações sobre como cada RNF será tratado, como ilustra a Tabela 2.7.

Tabela 2.7 – Especificação de Requisito Não Funcional.

RNF01 – O sistema deve disponibilizar a venda de livros pela Web, a partir dos principais navegadores disponíveis no mercado, considerando, inclusive, dispositivos móveis (tablets e smartphones).	
Categoria:	Portabilidade
Tratamento:	Os navegadores a serem considerados são: Google Chrome, Microsoft Edge / Internet Explorer, Mozilla Firefox e Safari. Deverão ser considerados os correlatos ao Chrome e Safari nas plataformas Android e iOS para tablets e smartphones, respectivamente. O desempenho do usuário na realização de tarefas deve ser equivalente. Para manter a independência dos demais componentes, a interface com o usuário deverá ser separada do restante da aplicação.
Medida:	Percentual de tarefas não completadas corretamente por um usuário quando usando um novo navegador.
Critério de Aceitação:	O sistema deverá funcionar essencialmente da mesma maneira nos navegadores Google Chrome, Microsoft Edge e Mozilla Firefox. Nesses três navegadores não poderá haver variações superiores a 1%. Serão admitidas variações em até 5% no Safari e no Internet Explorer. Para dispositivos móveis, serão admitidas variações de até 10%.

2.7 – Projeto de Software e Padrões (*Patterns*)

Todo projeto de desenvolvimento é, de alguma maneira, novo, na medida em que se quer desenvolver um novo sistema, seja porque ainda não existe um sistema para resolver o problema que está sendo tratado, seja porque há aspectos indesejáveis nos sistemas existentes. Isso não quer dizer que o projeto tenha que ser desenvolvido a partir do zero. Muito pelo contrário. A reutilização é um aspecto fundamental no desenvolvimento de software e, em especial, na fase de projeto. Muitos sistemas previamente desenvolvidos são similares ao sistema em desenvolvimento e há muito conhecimento que pode ser reaplicado para solucionar questões recorrentes no projeto de software. Os padrões (*patterns*) visam capturar esse conhecimento, procurando torná-lo mais geral e amplamente aplicável, desvinculando-o das especificidades de um determinado projeto ou sistema.

Um padrão é uma solução testada e aprovada para um problema geral. Diferentes padrões se destinam a diferentes fases do ciclo de vida: análise, projeto da arquitetura, projeto detalhado e implementação. Um padrão vem com diretrizes sobre quando usá-lo, bem como vantagens e desvantagens de seu uso. Um padrão já foi cuidadosamente considerado por outras pessoas e aplicado diversas vezes na solução de problemas anteriores de natureza similar. Assim, tende a ser uma solução de qualidade, com maiores chances de estar correto e estável do que uma solução nova, específica, ainda não testada (BLAHA; RUMBAUGH, 2006). Um padrão normalmente tem o formato de um par nomeado problema/solução, que pode ser utilizado em novos contextos, com orientações sobre como utilizá-lo nessas novas situações (LARMAN, 2007).

O objetivo de um padrão de projeto é registrar uma experiência no projeto de software que possa ser efetivamente utilizada por projetistas. Cada padrão sistematicamente nomeia, explica e avalia uma importante situação de projeto que ocorre repetidamente em sistemas (GAMMA et al., 1995).

Um projetista familiarizado com padrões pode aplicá-los diretamente a problemas sem ter que redescobrir as abstrações e os objetos que as capturam. Uma vez que um padrão é aplicado, muitas decisões de projeto decorrem automaticamente.

Em geral, um padrão tem os seguintes elementos (GAMMA et al., 1995) (BUSCHMANN et al., 1996):

- **Nome**: identificação de uma ou duas palavras, utilizada para nomear o padrão.
- **Contexto**: uma situação que dá origem a um problema.
- **Problema**: explica o problema que surge repetidamente no dado contexto.
- **Solução**: descreve uma solução comprovada para o problema, incluindo os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. É importante observar que um padrão não descreve um particular projeto concreto ou implementação. Um padrão provê uma descrição abstrata de um problema de projeto e como uma organização geral de elementos resolve esse problema.
- **Consequências**: são os resultados e os comprometimentos feitos ao se aplicar o padrão.

No que concerne aos padrões relacionados à fase de projeto, há três grandes categorias a serem consideradas:

- Padrões Arquitetônicos: definem uma estrutura global do sistema. Um padrão arquitetônico indica um conjunto pré-definido de elementos, especifica as suas responsabilidades e inclui regras e diretrizes para estabelecer relacionamentos entre eles. São aplicados na atividade de projeto da arquitetura de software e de seus elementos, e podem ser vistos como modelos (*templates*) para arquiteturas de software concretas (BUSCHMANN et al., 1996).
- Padrões de Projeto (*Design Patterns*): atendem a uma situação específica de projeto, mostrando classes e relacionamentos, seus papéis e suas colaborações e também a distribuição de responsabilidades. Um padrão de projeto descreve uma estrutura recorrente de classes que se comunicam, a qual resolve um problema de projeto geral dentro de um particular contexto (GAMMA et al., 1995).
- Idiomas: representam o nível mais baixo de padrões, endereçando aspectos tanto de projeto quanto de implementação. Um idioma é um padrão de baixo nível, específico de uma linguagem de programação, descrevendo como codificar aspectos particulares de componentes ou os relacionamentos entre eles, usando as características de uma dada linguagem de programação (BUSCHMANN et al., 1996).

2.8 – Documentação de Projeto

Uma vez que o projeto de software encontra-se no núcleo técnico do processo de desenvolvimento, sua documentação tem grande importância para o sucesso de um projeto e para a manutenção futura do sistema. Diferentes interessados vão requerer diferentes informações e a documentação de projeto é crucial para a comunicação. Analistas, projetistas e clientes vão precisar negociar prioridades entre requisitos conflitantes; programadores e testadores vão utilizar a documentação para implementar/ testar o sistema; gerentes de projeto vão usar informações da decomposição do sistema para definir e alocar equipes de trabalho; mantenedores vão recorrer a essa documentação na hora de avaliar e realizar uma alteração, dentre outros usos.

Uma vez que o projeto é um processo de refinamento, a sua documentação também deve prover representações em diferentes níveis de abstração. Além disso, o projeto de um sistema é uma entidade complexa que não pode ser descrita em uma única perspectiva. Ao contrário, múltiplas visões são essenciais e a documentação deve abranger aquelas visões consideradas relevantes. De fato, como muitas visões são possíveis, a documentação é uma atividade que envolve a escolha das visões relevantes e a documentação das visões selecionadas (BASS; CLEMENTS; KAZMAN, 2003). A escolha das visões é dependente de vários fatores, dentre eles, do tipo de sistema sendo desenvolvido, dos atributos de qualidade considerados e do público alvo da documentação de projeto. Diferentes visões realçam diferentes elementos de um sistema. De maneira geral, o documento de projeto deve conter (BASS; CLEMENTS; KAZMAN, 2003):

- Informações gerenciais, tais como versão, responsáveis, histórico de alterações;
- Uma descrição geral do sistema;
- Uma lista das visões consideradas e informações acerca do mapeamento entre elas;
- Para cada visão, deve-se ter:
    - Uma representação básica da visão, que pode ser gráfica, tabular ou textual, sendo a primeira a mais usual, sobretudo na forma de um diagrama UML. Se for usada uma representação gráfica não padronizada, deve-se ter uma legenda explicando a notação ou simbologia usada.
    - Uma descrição sucinta dos elementos presentes na visão.

Ainda que não seja possível advogar em favor de uma visão ou um conjunto de visões, Bass, Clements e Kazman (2003) consideram três grupos de visões que tipicamente devem ser levadas em consideração, a saber:

- Visão de Módulos: os elementos considerados nessa visão são módulos. A um módulo é tipicamente atribuída uma responsabilidade funcional. Essa visão inclui, dentre outras, estruturas de decomposição (módulos decompostos em submódulos) e uso (um módulo usa outro módulo);
- Visão de Componente e Conector (C&C): os elementos considerados nessa visão são componentes de tempo de execução (unidades de computação) e conectores (veículos de comunicação entre componentes).

- Visão de Alocação: mostra o relacionamento entre elementos de software e elementos do ambiente externo no qual o software está sendo criado ou executado. Estruturas de alocação incluem aspectos relacionados com implantação (mostrando como componentes de tempo de execução são alocados a unidades de hardware), implementação (mostrando como módulos são mapeados para estruturas de arquivos) e designação de trabalho (mostrando as equipes responsáveis por implementar e integrar módulos).

Além das informações anteriormente relacionadas, uma especificação de projeto deve:

- contemplar os requisitos contidos na especificação de requisitos, sendo que, muitas vezes, podem ser levantados novos requisitos, sobretudo requisitos não funcionais, durante a fase de projeto;
- ser um guia legível e compreensível para aqueles que vão codificar, testar e manter o software.
- prover um quadro completo do software, segundo uma perspectiva de implementação.

Capítulo 3 – Arquitetura de Software

À medida que os sistemas computacionais crescem em tamanho e complexidade, as técnicas relacionadas ao projeto de estruturas de dados e algoritmos, tais como decomposição modular de programas, tipos abstratos de dados e programação orientada a objetos, não são mais suficientes para lidar com os problemas envolvendo o projeto de software no nível de sistema. Passa a ser necessário considerar um nível de organização relativo à arquitetura do software (MENDES, 2002). A arquitetura de software refere-se às estruturas de grandes sistemas de software. A visão arquitetônica de um sistema é abstrata, retirando de cena detalhes de implementação, algorítmicos e de representação de dados e procurando se concentrar no comportamento e interação entre elementos considerados caixas pretas (BASS; CLEMENTS; KAZMAN, 2003).

O projeto da arquitetura envolve, dentre outras, questões relativas à organização e estrutura geral do sistema, seleção de alternativas de projeto, atribuição de funcionalidades a elementos de projeto e atendimento a atributos de qualidade (requisitos não funcionais) (MENDES, 2002).

Este capítulo trata do projeto da arquitetura de software. A Seção 3.1 discute o que é arquitetura de software, os fatores que influenciam o seu projeto, os interessados na arquitetura e a importância de sua representação. A Seção 3.2 apresenta algumas classes de sistemas, as quais podem ser associadas a certos estilos arquitetônicos. A Seção 3.3 apresenta alguns estilos arquitetônicos. A Seção 3.4 discute aspectos específicos do projeto de sistemas de informação e enumera alguns padrões arquitetônicos que podem ser usados no projeto dessa classe de sistemas. A Seção 3.5 trata da relação entre atributos de qualidade (requisitos não funcionais) e a arquitetura e apresenta algumas táticas que podem ser usadas para incorporar esses atributos a sistemas de software. A Seção 3.6 apresenta um processo para conduzir o projeto da arquitetura de software. Finalmente, a Seção 3.7 comenta brevemente sobre o projeto de componentes da arquitetura, apontando como o restante deste texto se posiciona em relação a essa questão.

3.1 – O que é Arquitetura de Software?

De acordo com Bass, Clements e Kazman (2003), a arquitetura de software de um sistema computacional refere-se à sua estrutura, consistindo de elementos de software, propriedades externamente visíveis desses elementos e os relacionamentos entre eles. A arquitetura define elementos de software (ou módulos) e envolve informações sobre como eles se relacionam uns com os outros. Uma arquitetura pode envolver mais de um tipo de estrutura, com diferentes tipos de elementos e de relacionamentos entre eles. A arquitetura omite as informações sobre os elementos que não são relativas às suas interações. As propriedades externamente visíveis indicam as suposições que os demais elementos podem fazer sobre um elemento, tais como serviços providos e características de qualidade esperadas. Assim, uma arquitetura é antes de tudo uma abstração que suprime detalhes dos elementos que não afetam como eles são usados, como se

relacionam, como interagem e como usam outros elementos. Na maioria das vezes, a arquitetura é usada para descrever aspectos estruturais de um sistema.

Em quase todos os sistemas modernos, elementos interagem com outros por meio de interfaces que dividem detalhes sobre um elemento em partes pública e privada. A arquitetura está preocupada com a parte pública dessa divisão. Detalhes privados, i.e., aqueles que têm a ver somente com a implementação interna do elemento, não são arquiteturais (BASS; CLEMENTS; KAZMAN, 2003).

Até o projeto arquitetônico, aspectos relacionados ao hardware e à plataforma de implementação ainda não foram tratados, já que a fase de análise não se preocupa com a tecnologia a ser usada na solução, mas somente com o domínio do problema (visão de mundo real). Este é o momento para resolver como um modelo ideal vai executar em uma plataforma restrita. É importante realçar que não existe a solução perfeita. O projeto da arquitetura é uma tarefa de negociação, onde se faz compromissos entre soluções sub-ótimas. O modelo de arquitetura mapeia os requisitos essenciais da fase de análise em uma arquitetura técnica. Uma vez que muitas arquiteturas diferentes são possíveis, o propósito do projeto arquitetônico é escolher a configuração mais adequada.

Normalmente, o projeto da arquitetura é discutido à luz dos requisitos do sistema, ou seja, é necessário conhecer minimamente os requisitos do sistema para se poder projetar a sua arquitetura. Contudo, deve-se considerar o projeto arquitetônico como algo mais abrangente, envolvendo aspectos técnicos, sociais e de negócio. Todos esses fatores (e não somente os requisitos) influenciam a arquitetura de software. A arquitetura, por sua vez, afeta o ambiente da organização (incluindo ambientes técnico, social e de negócio), o qual vai influenciar arquiteturas futuras, criando um ciclo de realimentação contínua. Por exemplo, se os projetistas encarregados do projeto de um novo sistema obtiveram bons resultados em projetos de sistemas anteriores usando uma particular abordagem de arquitetura, então é natural que eles tentem a mesma abordagem em um novo projeto. Por outro lado, se suas experiências anteriores com essa abordagem foram desastrosas, os projetistas vão relutar em tentá-la outra vez, mesmo que ela se apresente como uma solução adequada. Assim, as escolhas são guiadas, também, pela formação e experiência dos projetistas (BASS; CLEMENTS; KAZMAN, 2003).

Outro fator que afeta a escolha da arquitetura é o ambiente técnico (ou plataforma de implementação) corrente. Muitas vezes, há para esse ambiente um conjunto dominante de padrões, práticas e técnicas que é aceito pela comunidade de arquitetos ou pela organização de desenvolvimento. Por fim, a arquitetura é influenciada também pela estrutura e natureza da organização de desenvolvimento, bem como pela gerência do projeto em questão (BASS; CLEMENTS; KAZMAN, 2003).

Assim, no projeto da arquitetura de software, projetistas são influenciados por requisitos para o sistema, estrutura e metas da organização de desenvolvimento, ambiente técnico disponível e por sua própria experiência e formação. Além disso, os relacionamentos entre metas de negócio, requisitos de sistemas, experiência dos projetistas, arquiteturas e sistemas implantados geram diversos laços de realimentação que podem ser gerenciados pela organização. Esse ciclo de realimentação, ilustrado na Figura 3.1, atua, dentre outros, das seguintes maneiras (BASS; CLEMENTS; KAZMAN, 2003):

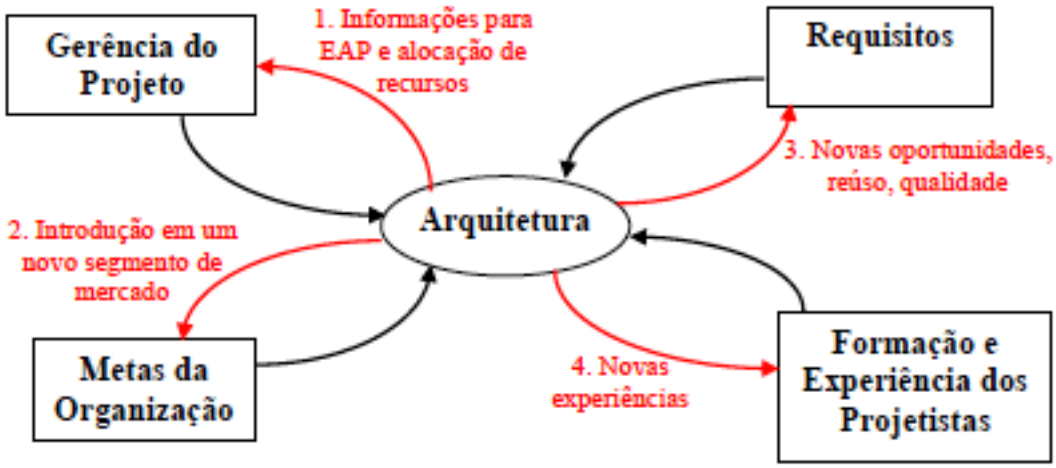


Figura 3.1 – A Arquitetura de Software e suas Influências.

1. A arquitetura afeta a gerência do projeto de desenvolvimento. Ao prescrever uma estrutura para um sistema, a arquitetura prescreve também as unidades de software a serem implementadas (ou obtidas) e integradas para formar o sistema. Essas unidades formam a base para o desenvolvimento da Estrutura Analítica do Projeto (EAP) e para a alocação de equipes e pessoas às unidades da EAP.
2. A arquitetura pode afetar as metas da organização de desenvolvimento, uma vez que um sistema bem sucedido, construído a partir dela, pode habilitar a organização a estabelecer uma base em um particular nicho de mercado.
3. A arquitetura pode afetar os requisitos do cliente para novos sistemas, ao dar ao cliente a oportunidade de receber um sistema baseado na mesma arquitetura de maneira mais econômica, rápida e confiável do que se o mesmo sistema tivesse que ser desenvolvido a partir do zero.
4. O processo de construir o sistema baseado na arquitetura afeta a experiência do projetista.

Muitas pessoas têm interesse na arquitetura de software, tais como clientes, usuários finais, desenvolvedores, gerentes de projeto e mantenedores. Alguns desses interesses são conflitantes e o projetista frequentemente tem de mediar conflitos até chegar à configuração que atenda de forma mais adequada a todos os interesses. Neste contexto, a arquitetura de software é importante principalmente porque (BASS; CLEMENTS; KAZMAN, 2003):

- Representa uma abstração do sistema que pode ser usada para compreensão mútua, negociação, consenso e comunicação entre os interessados. A arquitetura provê uma linguagem comum na qual diferentes preocupações podem ser expressas, negociadas e resolvidas em um nível que seja intelectualmente gerenciável.
- Manifesta as primeiras decisões de projeto. Essas decisões definem restrições sobre a implementação e a estrutura do projeto. A implementação tem de ser feita considerando a divisão de elementos prescrita pela arquitetura. Os elementos têm de interagir conforme o prescrito e cada elemento tem de cumprir sua responsabilidade conforme definido pela arquitetura. Também a

estrutura do projeto, e às vezes até a estrutura da organização como um todo, torna-se amarrada à estrutura proposta pela arquitetura. Neste sentido, a arquitetura pode ajudar a obter estimativas e cronogramas mais precisos, bem como pode ajudar na prototipagem do sistema. Além disso, a extensão na qual o sistema vai ser capaz de satisfazer os atributos de qualidade (requisitos não funcionais de produto) requeridos é substancialmente determinada pela arquitetura. Particularmente a manutenibilidade é fortemente afetada pela arquitetura. A arquitetura divide possíveis alterações em três categorias: locais (confinadas em um único elemento), não locais (requerem a alteração de vários elementos, mas mantêm intacta a abordagem arquitetônica subjacente) e arquitetônicas (afetam a estrutura do sistema e podem requerer alterações ao longo de todo o sistema). Obviamente, alterações locais são as mais desejáveis e, portanto, uma arquitetura efetiva deve propiciar que as alterações mais prováveis sejam as mais fáceis de fazer.

- Constitui um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus elementos trabalham em conjunto. Além disso, esse modelo é transferível para outros sistemas, em especial para aqueles que exibem requisitos funcionais e não funcionais similares, promovendo reúso em larga escala. Um desenvolvimento baseado na arquitetura frequentemente enfoca a composição ou montagem de elementos que provavelmente foram desenvolvidos separadamente, ou até mesmo de forma independente. Essa composição é possível porque a arquitetura define os elementos que devem ser incorporados ao sistema. Além disso, a arquitetura restringe possíveis substituições de elementos, tomando por base a forma como eles interagem com o ambiente, como eles recebem e entregam o controle, que dados consomem e produzem, como acessam esses dados e quais protocolos usam para se comunicar e compartilhar recursos.

É importante que o projetista seja capaz de reconhecer estruturas comuns utilizadas em sistemas já desenvolvidos, de modo a compreender as relações existentes e desenvolver novos sistemas como variações dos sistemas existentes. O entendimento de arquiteturas de software existentes permite que os projetistas avaliem alternativas de projeto. Neste contexto, uma representação da arquitetura é essencial para permitir descrever propriedades de um sistema complexo, bem como uma análise da arquitetura proposta (MENDES, 2002).

Muitas vezes, arquiteturas são representadas na forma de diagramas contendo caixas (representando elementos) e linhas (representando relacionamentos). Entretanto, tais diagramas não dizem muita coisa sobre o que são os elementos e como eles cooperam para realizar o propósito do sistema e, portanto, não capturam importantes informações de arquitetura (BASS; CLEMENTS; KAZMAN, 2003). Por exemplo, em uma visão de decomposição de módulos, é importante distinguir quando um módulo é decomposto em outros módulos e quando um módulo simplesmente usa outros módulos. Já em uma arquitetura cliente-servidor, é importante apontar quando um módulo é considerado cliente e quando ele é considerado servidor. Assim, idealmente, a representação de uma arquitetura deve incorporar informações acerca dos tipos dos elementos e dos relacionamentos. Além disso, conforme discutido no Capítulo 2, múltiplas visões podem ser usadas para representar diferentes aspectos da arquitetura.

3.2 – Classes de Sistemas

Sistemas similares, muito provavelmente, terão arquiteturas similares. Se há soluções bem estabelecidas para certas classes de sistemas, não há razão para reinventar a roda. O melhor é procurar reutilizar alguma dessas soluções.

Sistemas podem ser classificados de muitas formas diferentes. Blaha e Rumbaugh (2006) listam seis tipos de sistemas, sendo que um dado sistema pode se enquadrar em mais de uma dessas categorias. As classes de sistemas listadas são:

- Sistemas de Transformação (ou Processamento) em Lote: são organizados como uma série de módulos conectados sequencialmente. O sistema recebe as entradas (todas de uma só vez) e realiza uma série de transformações sobre os dados, até produzir uma resposta, sem haver qualquer interação com o mundo exterior durante todo o processamento. O aspecto mais importante do projeto de um sistema desta natureza é a definição de uma série lógica de etapas de processamento. Ex.: compiladores, sistema de folha de pagamento.
- Sistemas de Transformação Contínua: similares aos sistemas de transformação em lote no que se refere ao fato de serem organizados como uma série de módulos conectados sequencialmente, os sistemas de transformação contínua recebem entradas continuamente e calculam saídas de maneira incremental. Ex.: sistemas de processamento de sinais, sistemas de monitoramento de processos.
- Sistemas de Interface Interativa: são dominados por interações com agentes externos, tais como pessoas e dispositivos. Como os agentes externos são independentes do sistema, este não pode controlá-los, ainda que possa solicitar respostas deles. Para sistemas desta natureza, recomenda-se isolar as classes de interface das classes de aplicação, bem como se recomenda o uso de classes predefinidas (p.ex., fornecidas por sistemas de gerenciamento de interfaces com o usuário) para o projeto da interação.
- Sistemas de Simulação Dinâmica: modelam ou controlam objetos do mundo real e, portanto, o desempenho pode ser um fator crítico. Em um mundo ideal, um número arbitrário de processadores paralelos executaria a simulação em uma analogia exata à situação do mundo real. Na prática, é preciso adequar-se aos recursos disponíveis. Etapas discretas são usadas para aproximar processos contínuos. Ex.: simulação de fenômenos atmosféricos, jogos.
- Sistemas de Tempo Real: são sistemas interativos com fortes restrições de tempo, frequentemente projetados para operarem próximos de seus limites. O projeto de um sistema de tempo real é um processo complexo e envolve aspectos relacionados à priorização e coordenação de tarefas, bem como tratamento de interrupções. Desempenho é um fator determinante e os fatores portabilidade e manutenibilidade normalmente são sacrificados em detrimento do primeiro. Ex.: sistemas de controle de processos industriais, sistemas de monitoramento de pacientes em hospitais etc.
- Sistemas Gerenciadores de Transações: são sistemas cuja função principal é armazenar e recuperar dados. Normalmente lidam com vários usuários realizando operações ao mesmo tempo e precisam proteger seus dados contra acesso não autorizado (segurança) e perda acidental (recuperabilidade). São geralmente construídos sobre um sistema gerenciador de banco de dados. Neste

tipo de sistema, determinar a unidade de transação (i.e., o conjunto de recursos que precisa ser trabalhado como uma transação) é um importante aspecto de projeto, já que transações são completamente bem-sucedidas ou completamente fracassadas. Ex.: sistemas de venda de passagens aéreas, sistemas de controle de estoque, sistemas bancários etc.

Uma classe de sistemas muito importante, que combina algumas das classes descritas anteriormente, é a classe dos Sistemas de Informação. O principal objetivo dessa classe de sistemas é o gerenciamento de informações (MENDES, 2002). Sistemas de Informação (SIs) são responsáveis por coletar, manipular e preservar grandes quantidades de informações complexas (SHAW; GARLAN, 1996). Fowler (2003) enumera as seguintes características para SIs:

- SIs geralmente envolvem grandes quantidades de dados e a sua gerência é uma parte importante do sistema. Assim, bancos de dados são frequentemente utilizados. Além disso, esses dados precisam ser armazenados por vários anos e durante esse tempo, muitas alterações nos programas que os manipulam vão ocorrer. É muito provável que haja diversas alterações, inclusive, na estrutura dos dados para acomodar novas porções de informação.
- Usuários tipicamente acessam os dados concorrentemente.
- Há uma grande quantidade de interfaces com o usuário para tratar os dados. O perfil dos usuários varia de ocasional a regular e normalmente eles não dominam profundamente a tecnologia. Assim, os dados têm de ser apresentados em muitos diferentes meios para diferentes propósitos.
- Um SI geralmente precisa estar integrado com outros SIs da organização. Os vários sistemas são construídos em diferentes momentos, por equipes distintas, usando diferentes tecnologias. Esta situação é agravada quando os SIs precisam estar integrados com sistemas de organizações parceiras. Mesmo unificando a tecnologia de integração, há problemas advindos de diferenças nos processos de negócio e na semântica dos dados (dissonância conceitual).
- Regras de negócio são impostas e é necessário lidar com diversas condições que, muitas vezes, estabelecem relações umas com as outras, de modos até surpreendentes. A diversidade de casos específicos torna um SI muito complexo. Além disso, essas regras certamente mudarão ao longo do tempo.

Dadas essas características, pode-se perceber que SIs enquadram-se em algumas das classes anteriormente apresentadas. A maior parte dos SIs são sistemas gerenciadores de transação com interface interativa e um componente vital para esse tipo de sistema é o banco de dados no qual transações são realizadas (MENDES, 2002). Atributos de segurança, confiabilidade, usabilidade, eficiência e compatibilidade devem ser considerados atentamente no projeto desse tipo de sistema. Como estão associados a processos de negócio de organizações, SIs estão sujeitos também a modificações frequentes e a manutenibilidade é também um atributo de qualidade importante para essa classe de sistemas.

Quanto à plataforma em que executam, sistemas podem ser classificados em aplicações *desktop*, aplicações web e aplicações móveis. Aplicações *desktop* executam em estações de trabalho (computadores, *notebooks*) e podem utilizar os recursos dessas máquinas. Como consequência, podem explorar uma rica variedade de controles de

interface com o usuário. Aplicações *desktop* podem executar em uma única máquina (*standalone*) ou em várias máquinas (aplicações distribuídas). Uma aplicação web, por sua vez, é uma aplicação de software que utiliza a Web, por meio de um navegador (*browser*), como ambiente de execução. O escopo e a complexidade das aplicações Web variam muito: de pequena escala e tempo de desenvolvimento curto (algumas semanas), a aplicativos corporativos de grande escala, distribuídos através da Internet ou via intranets e extranets (MURUGESAN; GINIGE, 2008). Já aplicações móveis são aplicativos de software desenvolvidos para dispositivos móveis de menor capacidade de processamento, tais como *tablets* e *smartphones*. Podem executar via Web (e, portanto, neste caso são também aplicações web) ou como clientes específicos de uma certa plataforma móvel (p.ex., iOS, Android, Windows Mobile). Neste último caso, uma aplicação cliente desenvolvida para um sistema operacional pode não ser portável para outros.

Uma característica marcante da plataforma *desktop* em relação às demais é a maior riqueza das interfaces com o usuário. As aplicações web tradicionais são baseadas na apresentação de páginas HTML para o usuário, via um navegador, o que permite uma interação um pouco menos flexível. Contudo, mais recentemente, com o surgimento das chamadas Aplicações Ricas para Internet (*Rich Internet Applications* - RIAs), possibilitado por tecnologias de desenvolvimento como AJAX (*Asynchronous Javascript and XML*), aplicações Web também podem exibir uma interação com o usuário avançada e sofisticada (CASTELEYN et al., 2009).

Aplicações Web podem ser classificadas de diversas maneiras, não havendo uma classificação única e amplamente aceita. Murgesan e Ginige (2008) propõem a seguinte categorização das aplicações Web com base em sua funcionalidade:

- Informativas: visam apresentar informação. Ex.: Jornais *online*, catálogos de produtos, classificados *online*, *website* de um museu ou uma enciclopédia *online*.
- Interativas: envolvem uma maior interação com o usuário, tais como formulários de inscrição, apresentação de informações personalizadas, jogos online etc.
- Transacional: envolvem transações, tais como comércio eletrônico (solicitação de bens e serviços), *homebaking*, reservas de passagens aéreas etc.
- Orientadas ao Fluxo de Trabalho (*Workflow*): seguem um fluxo de trabalho bem definido, normalmente definido por um processo de negócio. Exemplos incluem gestão de estoque, gerenciamento da cadeia de fornecimento etc.
- Ambientes de Trabalho Colaborativo: incluem sistemas de autoria distribuída, ferramentas de design colaborativo etc.
- Comunidades e mercados online: incluem grupos de discussão, sistemas de recomendação etc.

Pressman e Lowe (2009), por sua vez, propõem outras categorias de aplicações web, que incluem aplicações informativas, de *download*, orientadas a transações, orientadas a serviços, de acesso a bases de dados, *data warehousing*, portais, dentre outras.

3.3 – Estilos Arquitetônicos

Conforme discutido no Capítulo 2, a reutilização é um aspecto-chave para se obter um bom projeto de software. No que refere ao reúso no projeto da arquitetura de software, estilos e padrões arquitetônicos são importantes ferramentas. Muitos autores consideram estilos e padrões como diferentes termos para designar o mesmo conceito, tal como Gorton (2006). Outros consideram estilos e padrões conceitos diferentes, como é o caso de Albin (2003). Neste texto, vamos tratá-los como conceitos distintos, ainda que ambos se refiram à captura de conhecimento relevante relativo a uma solução genérica para um problema relacionado à arquitetura de software.

Um estilo arquitetônico define um vocabulário de tipos de elementos e tipos de relacionamentos, e um conjunto de restrições sobre como eles podem ser combinados (SHAW; GARLAN, 1996). Padrões arquitetônicos também estabelecem tipos de elementos e de relacionamentos entre eles, mas sua apresentação segue uma forma bem definida, indicando nome, contexto, problema, solução e consequências. Especialmente a solução define estratégias para tratar o problema, o que não acontece com os estilos arquitetônicos. Assim, normalmente, estilos têm uma apresentação mais livre e são descritos de maneira mais abstrata que padrões arquitetônicos.

Shaw e Garlan (1996) propuseram uma taxonomia de estilos arquitetônicos que considera cinco categorias principais, contendo algum grau de sobreposição entre elas:

- Sistemas de Fluxo de Dados: são caracterizados pelo modo como dados se movem através do sistema (ALBIN, 2003). O estilo "dutos e filtros" (*pipes and filters*) é classificado nesta categoria.
- Sistemas de Chamada e Retorno: são caracterizados por um modelo de ativação que envolve a linha principal de controle que realiza invocações explícitas de operações (ALBIN, 2003), tal como ocorre com chamadas de sub-rotinas na programação estruturada ou a invocação de operações em sistemas orientados a objetos. O estilo "camadas" (*layers*) é classificado nesta categoria, bem como o estilo "partições" (*partitions*).
- Componentes Independentes: este estilo fia-se na invocação implícita de operações, ou seja, a invocação de uma operação é desacoplada de sua execução, de modo que os componentes chamador e chamado podem existir em processos separados e possivelmente distribuídos em processadores distintos (ALBIN, 2003). O estilo "invocação implícita" é classificado nesta categoria.
- Máquinas Virtuais: uma máquina virtual é desenvolvida em software, contendo uma máquina de interpretação, uma memória contendo o código a ser interpretado, uma representação do estado da máquina de interpretação e uma representação do estado corrente do programa (SHAW e GARLAN, 1996). Interpretadores e sistemas baseados em regras são estilos que se enquadram nesta categoria.
- Repositórios: esta categoria envolve um repositório de dados compartilhado para a passagem de informações. Diferentes estilos baseados em repositório variam em termos de estilos de ativação e comunicação (ALBIN, 2003). Exemplos de estilos nessa categoria incluem os estilos baseado em bancos de dados e quadro-negro.

Estilos arquitetônicos podem ser combinados de várias maneiras. Por exemplo, um componente de um sistema organizado em um estilo pode ter sua estrutura interna desenvolvida em um estilo completamente diferente. Outra opção é permitir que um único componente use uma mistura de conectores arquitetônicos diferentes. Por exemplo, um componente pode acessar uma base de dados como parte de sua interface, mas interagir através de dutos com outros componentes do sistema (SHAW; GARLAN, 1996). Vale ressaltar que é sempre uma boa opção considerar a combinação de estilos durante o projeto de uma arquitetura. Assim, uma arquitetura pode combinar diferentes estilos em partes distintas de um sistema, de modo a exibir diferentes atributos de qualidade (ALBIN, 2003). A seguir, alguns estilos arquitetônicos são apresentados.

Dutos e Filtros

O estilo "dutos e filtros" (*pipes and filters*) considera a existência de uma rede pela qual dados fluem de uma extremidade (origem) até a outra (destino). O fluxo de dados se dá através dos dutos e os dados sofrem transformações nos filtros. Um duto provê uma forma unidirecional de fluxo de dados, atuando como um condutor entre dois componentes: do componente origem para o componente destino (MENDES, 2002). Assim, os componentes são denominados filtros e os conectores, dutos (SHAW; GARLAN, 1996). A canalização de programas no sistema operacional Unix, o modelo clássico de compiladores e sistemas de folha de pagamento são exemplos de uso do estilo dutos e filtros. A Figura 3.2 ilustra esse estilo arquitetônico.

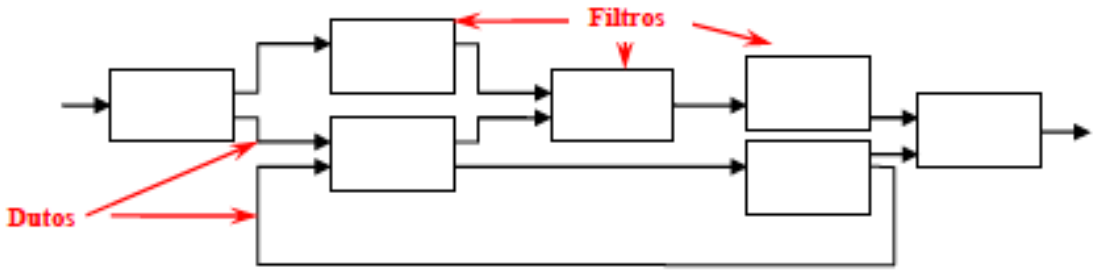


Figura 3.2 – Estilo Dutos e Filtros (adaptado de (SHAW; GARLAN, 1996)).

No estilo dutos e filtros, cada componente (filtro) possui um conjunto de entradas e um conjunto de saídas. Um componente lê dados de suas entradas e produz dados em suas saídas, realizando alguma transformação local. Os filtros têm de ser entidades independentes e não podem compartilhar estado (informações internas) com outros filtros. Além disso, um filtro não conhece os filtros anteriores e posteriores no fluxo. A especificação de um filtro define apenas o que é requerido nos dutos de entrada e garantir o que vai ser produzido nos dutos de saída, sem, no entanto, identificar os componentes nas extremidades desses dutos. Uma especialização comum desse estilo são os chamados *pipelines*, que restringem a topologia para sequências lineares de filtros. Quando em um *pipeline*, cada filtro processa a sua entrada como um todo, tem-se um estilo de transformação em lote sequencial (*batch sequential*) (SHAW; GARLAN, 1996).

O estilo "dutos e filtros" possui diversas propriedades interessantes, dentre elas (SHAW; GARLAN, 1996):

- Permite que o projetista compreenda o comportamento geral de um sistema como uma composição simples dos comportamentos dos filtros individuais.

- Facilita o reúso. Quaisquer dois filtros podem ser conectados, desde que eles estejam de acordo em relação aos dados sendo transmitidos entre eles.
- Facilita a manutenção e o crescimento. Novos filtros podem ser adicionados a um sistema existente, bem como filtros podem ser substituídos por outros melhores ou atualizados.
- Suporta execução concorrente. Cada filtro pode ser implementado como uma tarefa separada e potencialmente executada em paralelo com outros filtros.

Entretanto, há também desvantagens, tais como (SHAW; GARLAN, 1996):

- Apesar dos filtros poderem processar dados de forma incremental, eles são inerentemente independentes e, portanto, o projetista deve pensar cada filtro como provendo uma transformação completa dos dados de entrada para dados de saída.
- Devido a seu caráter transformacional, este estilo não cai bem para tratar aplicações interativas.

Camadas

No estilo em camadas, um sistema é organizado hierarquicamente, como um conjunto ordenado de camadas, cada uma delas construída em função das camadas inferiores e fornecendo serviços para as camadas superiores. O conhecimento é unidirecional, i.e., uma camada conhece as camadas inferiores, mas não tem qualquer conhecimento sobre as camadas superiores. Há uma relação cliente-servidor entre uma camada usuária de serviços e suas camadas inferiores, provedoras de serviços (BLAHA; RUMBAUGH, 2006). Cada camada acrescenta um nível de abstração sobre a camada inferior (MENDES, 2002). Arquiteturas de redes, tal como a arquitetura de Internet TCP/IP, são bons exemplos de sistemas em camadas.

Uma arquitetura em camadas pode ser fechada, quando uma camada é construída apenas em função da camada imediatamente inferior, ou aberta, quando uma camada pode usar recursos de quaisquer camadas inferiores. A arquitetura fechada reduz a dependência entre as camadas, permitindo que alterações sejam feitas mais facilmente. Já a arquitetura aberta reduz a necessidade de redefinir operações em vários níveis, o que tende a resultar em melhor desempenho. Por outro lado, mudanças em uma camada podem afetar qualquer camada acima, tornando a arquitetura menos robusta e comprometendo a manutenibilidade do sistema (BLAHA; RUMBAUGH, 2006). A Figura 3.3 ilustra arquiteturas em camadas fechada e aberta.

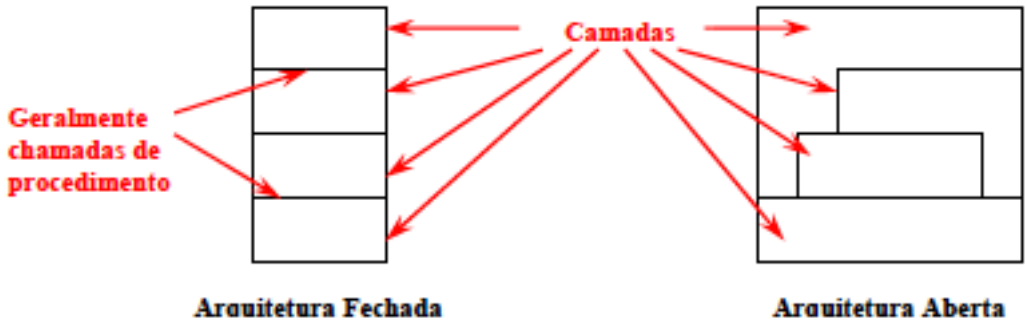


Figura 3.3 – Estilo em Camadas.

Arquiteturas em camadas apresentam diversas propriedades interessantes, dentre elas (SHAW; GARLAN, 1996):

- Apoiam o projeto baseado em níveis crescentes de abstração, permitindo dividir um problema complexo em camadas, tratando de diferentes preocupações.
- Tratam bem a manutenibilidade e a possibilidade de crescimento, especialmente as arquiteturas fechadas, pois alterações em uma camada afetam apenas a camada adjacente superior.
- Apoiam o reúso e diferentes implementações da mesma camada podem ser usadas alternativamente, desde que provejam as mesmas interfaces para as camadas superiores.

Como qualquer estilo, a arquitetura em camadas apresenta também algumas desvantagens, dentre elas:

- Nem todos os sistemas são facilmente estruturados em camadas, sendo muitas vezes difícil encontrar os níveis de abstração corretos. Mesmo quando isso é possível, considerações de desempenho podem colocar obstáculos, sobretudo para o uso de uma arquitetura fechada (SHAW; GARLAN, 1996). Neste caso, o desempenho poderá ficar comprometido face à necessidade de uma solicitação externa ter de passar por diversas camadas para ser tratada (MENDES, 2002).
- Pode não ser fácil definir o número adequado de camadas. Esse número dependerá não só da funcionalidade a ser provida pelo sistema, mas também dos requisitos não funcionais desejados (MENDES, 2002).

Partições

Outra forma de estruturar sistemas é por meio de partições. Partições dividem um sistema verticalmente em subsistemas fracamente acoplados, cada um fornecendo, normalmente, um tipo de serviço. Este é o caso, por exemplo, de sistemas operacionais, que possuem partições para controle de arquivos, gerência de memória, controle de processo etc. As partições podem ter algum conhecimento acerca das outras, mas esse conhecimento não é profundo e evita maiores dependências de projeto. Ao contrário das camadas, que variam em seu nível de abstração, as partições simplesmente dividem um sistema em partes, todas tendo um nível de abstração semelhante. Outra diferença entre camadas e partições é que as camadas dependem umas das outras, enquanto as partições podem ser tanto independentes como dependentes e eventualmente mutuamente dependentes (BLAHA; RUMBAUGH, 2006). A Figura 3.4 ilustra o estilo em partições. Nesta figura, as partições 1 e 2 são mutuamente dependentes, enquanto a Partição 2 é dependente da Partição 3. Já as Partições 3 e 4 são partições independentes.

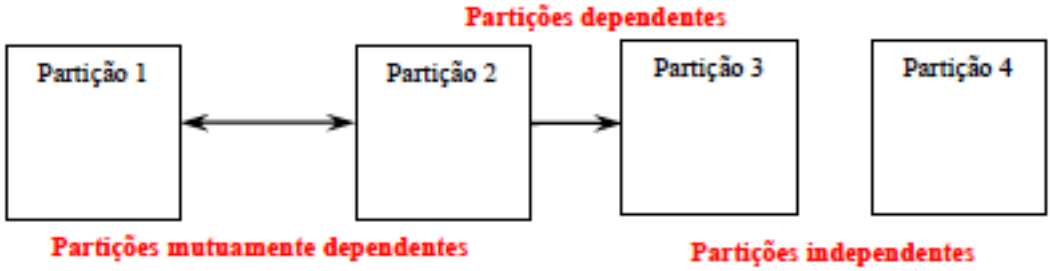


Figura 3.4 – Partições.

Normalmente, o uso de partições simplesmente não é suficiente para estruturar um sistema. Uma boa opção é combinar partições e camadas: as camadas podem ser divididas em partições ou, como é mais comum, as partições podem ser divididas em camadas.

Invocação Implícita

O estilo arquitetônico invocação implícita (ou baseado em eventos) requer que os componentes interessados em um evento registrem-se a fim de recebê-lo. Componentes podem tanto registrar interesse em receber eventos quanto em divulgar eventos (MENDES, 2002). A ideia por trás da invocação implícita é a de que um componente pode anunciar um ou mais eventos, enquanto outros componentes podem registrar interesse por um evento, associando um procedimento a ele. Quando um evento é anunciado, o próprio sistema invoca todos os procedimentos registrados para o evento. Assim, o anúncio de um evento implicitamente provoca a invocação de procedimentos em outros componentes (SHAW; GARLAN, 1996).

A Figura 3.5 ilustra o estilo invocação implícita. Como mostra a figura, um componente registrado como anunciante de um evento anuncia o evento, o qual é capturado pelo mecanismo de difusão de eventos. Esse mecanismo é responsável por difundir o evento para todos os componentes que registraram interesse no mesmo (componentes ouvintes), invocando os procedimentos associados. Deste modo, o componente anunciante não invoca diretamente os procedimentos dos componentes ouvintes, tornado esses componentes dissociados uns dos outros. A invocação de procedimentos é feita implicitamente pelo mecanismo de difusão de eventos.

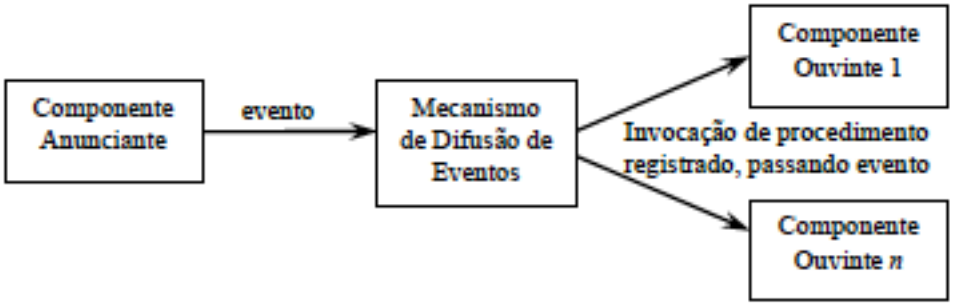


Figura 3.5 – Estilo Invocação Implícita.

Um bom exemplo de aplicação do estilo invocação implícita ocorre no tratamento de exceções, i.e., tratamento de falhas que acontecem em tempo de execução. Se durante a execução do sistema uma falha ocorre, uma exceção (evento) é gerada com a informação do erro. Essa exceção é encaminhada para o mecanismo de difusão de eventos que a encaminha para os componentes responsáveis por tratá-la. Deste modo, o mecanismo de difusão de eventos pode ser usado em todo o sistema, a fim de lidar com a ocorrência de exceções, oferecendo suporte à confiabilidade (MENDES, 2002).

Os eventos são assíncronos, o que permite que o componente anunciante continue realizando alguma outra computação após o envio do evento. Além disso, o componente anunciante desconhece a identidade dos componentes ouvintes. Assim, este estilo arquitetônico provê suporte à manutenibilidade, desde que a inserção e a remoção de componentes sejam fáceis de serem realizadas (MENDES, 2002). Componentes podem ser introduzidos no sistema simplesmente registrando-os como ouvintes de

eventos do sistema. Componentes podem, também, ser substituídos por outros componentes, sem afetar as interfaces de outros componentes do sistema (SHAW; GARLAN, 1996).

Contudo, uma vez que o componente anunciante desconhece os componentes ouvintes, eles não podem fazer suposições acerca da ordem de processamento, nem mesmo sobre que processamento vai ocorrer como resultado de seus eventos. Esta é a principal desvantagem da invocação implícita. Devido a isto, a maioria dos sistemas de invocação implícita também inclui invocação explícita como uma forma de interação complementar (SHAW; GARLAN, 1996).

Sistemas Baseados em Regras

Sistemas baseados em regras são sistemas de apoio à decisão que procuram representar o modo de raciocínio e o conhecimento utilizado por especialistas na resolução de problemas no seu âmbito de especialidade. Existe, portanto, um paralelismo entre esses sistemas e a forma como os especialistas humanos atingem um alto nível de desempenho na resolução de problemas, na medida em que estes conhecem muito bem as suas áreas de especialização (CARDOSO; MARQUES, 2007).

Os problemas solucionados por um sistema baseado em regras são, tipicamente, aqueles resolvidos por um especialista humano e que podem ser expressos mediante um conjunto de regras bem definido para analisar o problema, tais como planejamento, diagnóstico e interpretação de dados (CARDOSO; MARQUES, 2007).

A principal diferença entre um sistema baseado em regras e um sistema desenvolvido usando uma abordagem tradicional (procedimental ou funcional) está na maneira como o conhecimento sobre o domínio do problema é codificado. Em aplicações tradicionais, o conhecimento sobre o domínio do problema é codificado tanto nas instruções propriamente ditas quanto nas estruturas de dados. Já na abordagem de regras, todo o conhecimento relativo ao domínio do problema é codificado exclusivamente nas estruturas de dados. Nenhum conhecimento é armazenado nas instruções ou nos programas propriamente ditos.

Um sistema baseado em regras utiliza regras explícitas para expressar o conhecimento do domínio de um problema e permite, através da confrontação do conhecimento existente em fatos conhecidos sobre um determinado problema, inferir novos fatos a partir dos existentes.

A arquitetura geral de um sistema baseado em regras compreende dois componentes principais: um conjunto de declarações totalmente dependentes do domínio do problema, chamado de base de conhecimento, e um programa independente do domínio do problema (apesar de altamente dependente das estruturas de dados), chamado máquina de inferência.

A base de conhecimento é, tipicamente, divida em base de regras e base de fatos. A base de regras contém regras que definem como derivar informações a partir dos fatos, enquanto os fatos são informações acerca do estado do domínio. As regras assumem a forma: Se *condição* então *ação*, em que a condição descreve uma determinada situação que, se verdadeira, desencadeia a ação como consequência.

A máquina (ou motor ou mecanismo) de inferência é responsável por efetuar o processamento, sendo ela independente do conhecimento do domínio do problema.

Além desses componentes principais, um sistema baseado em regras possui, ainda, uma memória, que armazena temporariamente fatos iniciais e conclusões intermediárias, e uma interface com o usuário, a partir da qual se introduzem novos fatos do problema e se recebem os resultados ou conclusões retiradas pelo sistema (CARDOSO; MARQUES, 2007). A Figura 3.6 ilustra a arquitetura de sistemas baseados em regras.

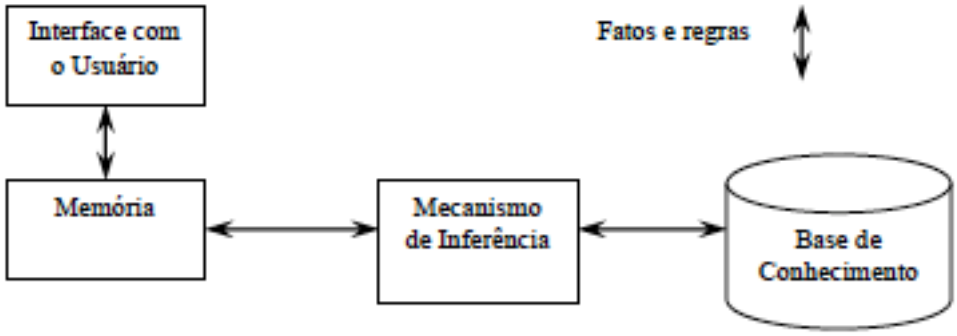


Figura 3.6 – Estilo Baseado em Regras.

3.4 – Padrões Arquitetônicos para Projeto de Sistemas de Informação

Conforme discutido no Capítulo 2, um padrão descreve um problema que ocorre diversas vezes e o núcleo central de uma solução para esse problema. Padrões estão enraizados na prática. Ou seja, eles não são ideias originais, mas sim observações do que funciona na prática. Assim, padrões não são inventados, mas descobertos (FOWLER, 2003).

Padrões são um ponto de partida e não um fim em si. De início, não é necessário saber detalhes sobre os diversos padrões. É importante ter uma visão geral dos mesmos para saber que problemas eles resolvem e como eles resolvem, de modo a selecioná-los quando forem aplicáveis. Uma vez reconhecida a necessidade de se aplicar o padrão, tem-se de descobrir como aplicá-lo ao problema específico que se tem em mãos. Muito dificilmente será possível aplicar um padrão cegamente. Padrões são artefatos semi-prontos, o que implica na necessidade de completar a solução no contexto específico do projeto. Padrões são relativamente independentes, mas não são isolados uns dos outros, sobretudo os padrões arquitetônicos. Muitas vezes, um padrão leva a outros ou um padrão só é aplicável se outro estiver envolvido (FOWLER, 2003).

Em (FOWLER, 2003), é apresentado um conjunto de padrões para o projeto de Sistemas de Informação (SIs) que, conforme aponta o próprio autor, podem ser razoavelmente bem caracterizados como arquitetônicos, na medida em que representam decisões relativas à estrutura dos sistemas.

Conforme discutido anteriormente, Sistemas de Informação (SIs) são sistemas usados para prover informações para apoiar processos de negócio das organizações e apresentam como características marcantes: grandes quantidades de dados que precisam ser armazenados por longos períodos de tempo, usuários acessando o sistema de maneira concorrente por meio de interfaces com o usuário e regras de negócio que devem ser consideradas para que o sistema satisfaça as necessidades de seus usuários

O estilo subjacente aos padrões apresentados por Fowler é o estilo em camadas. Assim, a maior parte dos padrões apresentados refere-se a como decompor SIs em camadas e como essas camadas trabalham em conjunto. Contudo, há também padrões de projeto (*design patterns*) utilizados para realizar a arquitetura. Assim, não há em (FOWLER, 2003) uma clara separação entre padrões arquitetônicos e *design patterns*.

No que se refere à organização de SIs em camadas, componentes podem ser agrupados em três camadas lógicas principais, ilustradas na Figura 3.7:

- **Camada de Apresentação ou de Interface com o Usuário**: sua função é tratar a interação entre o usuário e o sistema. As responsabilidades principais dessa camada são exibir informações para os usuários e interpretar comandos do usuário em ações da lógica de negócio e da persistência de dados.
- **Camada de Lógica de Negócio**: contém as funcionalidades que apoiam os processos de negócio. Conceitos do domínio, regras de negócio, processamentos e cálculos são encontrados nesta camada.
- **Camada de Persistência ou de Gerência de Dados**: provê acesso a dados corporativos. É sua responsabilidade gerenciar requisições concorrentes de acesso às bases de dados, assim como a sincronização de elementos de dados distribuídos.

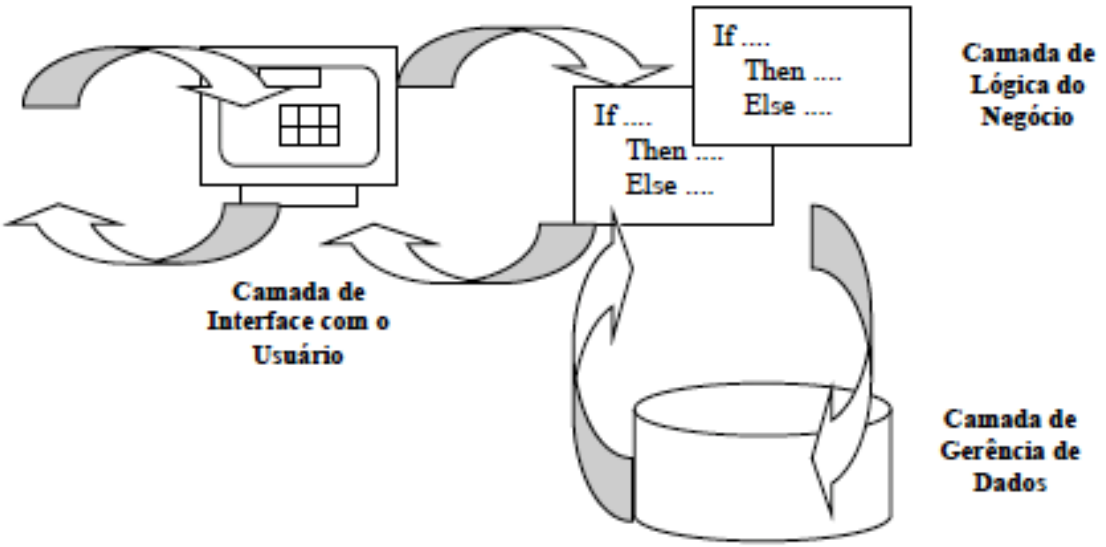


Figura 3.7 – Camadas Lógicas Típicas em Sistemas de Informação

Algumas vezes as camadas são dispostas de modo que a camada de lógica de negócio oculta completamente a camada de gerência de dados da interface com o usuário (arquitetura fechada). Mais frequentemente, contudo, a camada de apresentação acessa a gerência de dados diretamente (arquitetura aberta). Ainda que menos pura, a segunda abordagem tende a funcionar melhor na prática (FOWLER, 2003).

Deve-se realçar, ainda, que geralmente um sistema específico pode possuir vários pacotes de cada um desses três grandes tipos. Uma aplicação pode ser projetada para ser manipulada pelos usuários tanto por meio de uma interface com o usuário rica, quanto por uma interface de linha de comando, tendo, portanto, diferentes pacotes de interface com o usuário para cada um dos propósitos (FOWLER, 2003).

Ainda que seja possível identificar os três tipos de camadas discutidos anteriormente, a questão de como separá-las depende de quão complexa é a aplicação. Via de regra, a seguinte regra sobre dependências deve ser respeitada: as camadas de negócio e de gerência de dados não devem ser dependentes da camada de apresentação. Isso torna mais fácil substituir e modificar uma camada de apresentação (FOWLER, 2003).

Os padrões apresentados em (FOWLER, 2003) são organizados de acordo com essas camadas. Assim, há grupos de padrões relativos à:

- Camada da Lógica de Negócio: padrões nesse grupo tratam da organização das funcionalidades na camada de lógica de negócio. São eles: Script de Transação (*Transaction Script*), Modelo de Domínio (*Domain Model*), Módulo Tabela (*Table Module*) e Camada de Serviço (*Service Layer*). Alguns desses padrões são discutidos no Capítulo 4, que aborda o projeto da lógica de negócio.
- Camada de Apresentação: padrões nesse grupo enfocam a separação dos objetos de interfaces gráficas (visão) do controle da interação com o usuário. Exemplos são os padrões Modelo Visão Controlador (*Model View Controller*) e Controlador de Aplicação (*Application Controller*). Além disso, há padrões mais focados na visão (p.ex., *Template View*) e outros mais voltados para o projeto de controladores (p.ex., Controlador Frontal ou, em inglês, *Front Controller*). Alguns desses padrões são discutidos no Capítulo 5, que trata do projeto da interação com o usuário.
- Camada de Gerência de Dados: padrões nesse grupo se referem ao mapeamento objeto relacional, já que a tecnologia dominante de banco de dados para Sistemas de Informação é a dos bancos de dados relacionais. Há, dentre outros, padrões relativos a como a lógica de negócio conversa com os bancos de dados, bem como padrões relacionados a como mapear estruturas do mundo de objetos (p.ex., associações e herança) para bancos de dados relacionais. Alguns desses padrões são discutidos no Capítulo 6, que trata do projeto da gerência de dados.

Além desses três grupos, Fowler (2003) discute, ainda, outros dois grupos, um relativo a padrões que buscam tratar problemas de concorrência e outro envolvendo padrões enfocando o problema de estado de sessões.

Os padrões apresentados em (FOWLER, 2003) têm a seguinte estrutura: nome, intenção (que busca resumir um padrão em uma ou duas sentenças), esboço (uma representação visual do padrão, geralmente na forma de um diagrama UML), problema, solução, quando usar o padrão e leitura adicional.

Além do catálogo de padrões apresentado em (FOWLER, 2003), que enfoca o projeto de sistemas de informação em camadas, outro catálogo bastante interessante é o conhecido como POSA (*Pattern Oriented Software Architecture*) (BUSCHMANN et al., 1996). Este é, na verdade, um conjunto de catálogos de propósito mais geral, enfocando diversos aspectos e envolvendo tanto padrões arquitetônicos como padrões de projeto, idiomas e linguagens de padrões.

3.5 – Projeto de Sistemas de Informação Distribuídos

Tipicamente, SIs, sobretudo de médio a grande porte, são aplicações distribuídas e utilizam arquiteturas ditas cliente-servidor. Nessa arquitetura, componentes assumem os papéis de clientes e servidores de serviços. Um servidor é um componente que fica em estado de espera, aguardando a solicitação de um serviço por um ou mais clientes. O servidor pode trabalhar de forma síncrona ou assíncrona. Os clientes, por sua vez, podem ser vistos como processos independentes, i.e., a execução de seu processo não interfere em outros (MENDES, 2002).

O estilo em camadas é a base para as arquiteturas cliente-servidor. De fato, a noção de camadas tornou-se mais aparente nos anos 1990 com o surgimento dos sistemas cliente-servidor, ainda sob a marca do paradigma estruturado. Eles eram sistemas em duas camadas, em que o cliente tratava a interface com o usuário e a lógica de negócio, e o servidor era usualmente um banco de dados relacional. Contudo, à medida que a lógica de negócio foi ficando mais complexa, essa solução foi sendo objeto de muitas críticas. Uma alternativa passou a ser colocar a lógica de negócio no servidor, tipicamente na forma de *stored procedures*. Entretanto, *stored procedures* tinham mecanismos limitados de estruturação, levando também a muitos problemas. Com o surgimento da orientação a objetos, passou-se a considerar sistemas em três camadas (FOWLER, 2003).

Ao longo do tempo, a arquitetura cliente-servidor tem evoluído, existindo atualmente uma gama de variações. Todas elas, contudo, compartilham a ideia de haver funcionalidades compartilhadas entre clientes e servidores. Geralmente, clientes tratam as solicitações dos usuários, transformando-as em algum protocolo e as transmitindo para um servidor. O servidor processa as solicitações e retorna respostas para o cliente. Por fim, o cliente envia as respostas para o usuário (MENDES, 2002).

Primordialmente, o termo cliente-servidor é usado para descrever software que executa em mais de um hardware, de modo a realizar uma tarefa de negócio. A separação de hardware é uma característica marcante em aplicações cliente-servidor. A distância entre processadores remotos varia desde computadores localizados na mesma sala ou prédio, até aqueles localizados em diferentes prédios, cidades ou países. Entretanto, em relação à arquitetura de software, o termo pode ser usado para descrever diferentes componentes de software se comunicando uns com os outros, ainda que rodando em uma mesma máquina (RUBLE, 1997). No projeto da arquitetura de software, está-se falando de camadas lógicas, cujo objetivo é dividir o sistema em pacotes separados para reduzir o acoplamento entre diferentes partes do sistema. A separação em camadas lógicas é útil mesmo se essas camadas estiverem todas rodando na mesma máquina física. Contudo, há situações em que a estrutura física do sistema faz a diferença (FOWLER, 2003).

Considerando o mapeamento de camadas lógicas em camadas físicas, um uso bastante comum de arquiteturas cliente-servidor consiste em explorar o poder dos computadores pessoais para gerenciar interfaces gráficas com o usuário, mantendo os serviços e dados do negócio em servidores. Em sua forma mais simples, a arquitetura cliente-servidor envolve múltiplos clientes fazendo requisições para um único servidor, como ilustra a Figura 3.8, o que caracteriza uma arquitetura em duas camadas (*two-tier*).

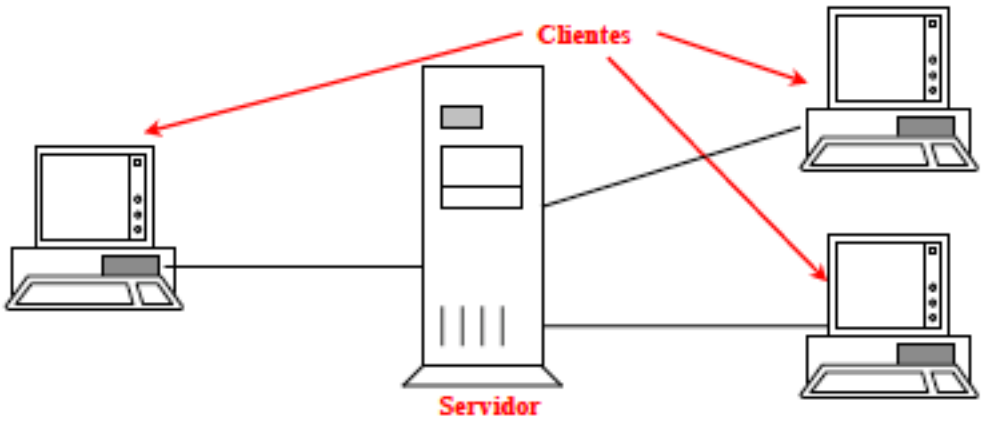


Figura 3.8 – Arquitetura de hardware cliente-servidor em duas camadas.

A Figura 3.9 mostra uma arquitetura cliente-servidor em três camadas, na qual máquinas-cliente estão conectadas via rede a um servidor de aplicação que, por sua vez, se comunica com um servidor de dados. Essa arquitetura pode ser, ainda, estendida para *n* camadas (*n-tier*).

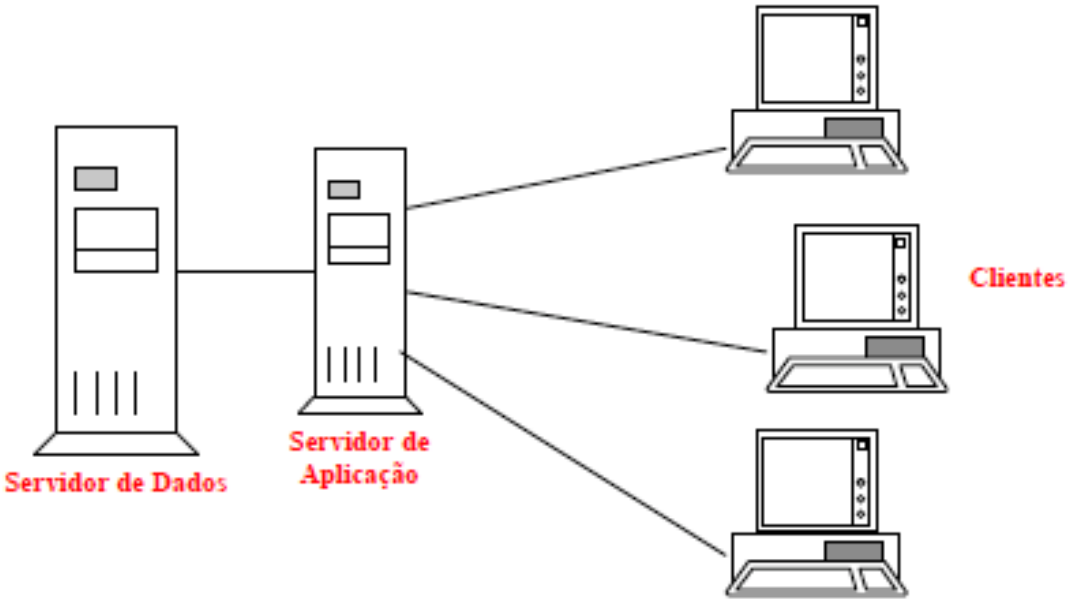


Figura 3.9 – Arquitetura de hardware cliente-servidor em três camadas.

A primeira geração de ferramentas populares de desenvolvimento de aplicações cliente-servidor com interfaces gráficas assumia uma arquitetura de hardware de duas camadas e encorajava uma abordagem de projeto arquitetônico de software do tipo cliente pesado, onde a lógica do negócio era totalmente amarrada à camada de apresentação da aplicação, o que provocava sérios problemas de manutenibilidade. Essas mesmas ferramentas evoluíram e, em um segundo momento, passaram a ficar mais em linha com uma filosofia do tipo servidor pesado, onde a lógica de negócio é fortemente mapeada no servidor de dados. Hoje, contudo, reconhece-se a necessidade de separar a camada de lógica do negócio das demais camadas. Essa separação traz várias vantagens, dentre elas (RUBLE, 1997):

- Reusabilidade: Classes de negócio tendem a ser complexas e podem desempenhar diferentes papéis dentro dos sistemas corporativos. A meta é criar classes que garantam as regras de negócio para uma particular classe de negócio e reutilizá-las em todos os contextos que lidem com a mesma.
- Portabilidade: ao se mover a lógica de negócio para uma camada separada, permite-se tirar proveito de diferentes plataformas de hardware e software, sem ter que reescrever grandes porções do código.
- Manutenibilidade: o fato da lógica de negócio estar concentrada em uma camada única da arquitetura de software facilita a localização das alterações e evoluções a serem feitas no sistema. O mesmo pode se dizer em relação à interface com o usuário e à gerência de dados.

É preciso, portanto, escolher onde cada uma das camadas lógicas vai rodar (camadas físicas). De maneira geral, a gerência de dados ficará no servidor. A exceção ficará por conta de situações em que é necessário duplicar dados para permitir operação desconectada (FOWLER, 2003).

No que se refere à camada de apresentação, a decisão depende principalmente do tipo desejado de interface com o usuário. Com a difusão do uso da Web como plataforma para o desenvolvimento de SIs, a opção mais simples é rodar tudo no servidor, cabendo ao cliente apenas a apresentação de uma interface HTML (*front end*) rodando em um navegador Web. Essa solução facilita a manutenção e a implantação de novas versões, mas não permite operação desconectada e compromete a rapidez com que o sistema reconhece uma solicitação do usuário (*responsiveness*), uma vez que a solicitação tem de trafegar, ida e volta, para o servidor (FOWLER, 2003). Além disso, interfaces mais ricas podem não ser possíveis. Para tratar esses problemas, surgiram as Aplicações Ricas para Internet (*Rich Internet Applications* – RIAs), discutidas na Seção 3.6.

Por fim, em relação à camada de domínio, pode-se rodar tudo no cliente (cliente pesado), tudo no servidor (servidor pesado) ou dividi-la. Rodar tudo no servidor é a melhor opção do ponto de vista da manutenibilidade. Dividir a lógica de negócio entre o servidor e o cliente traz uma dificuldade adicional na manutenibilidade do sistema: saber onde certa porção da aplicação está rodando. Neste caso, devem-se isolar as porções que vão rodar no cliente em módulos autocontidos que sejam independentes das outras partes do sistema.

Levando-se em consideração os aspectos discutidos anteriormente, o projeto de sistemas de informação distribuídos envolve questões adicionais, dentre elas (RUBLE, 1997):

- a definição de requisitos de distribuição geográfica do sistema,
- a definição das máquinas clientes, servidoras e da configuração e número de camadas de hardware cliente-servidor,
- a localização dos processos,
- a localização dos dados físicos e as estratégias de sincronização para dados distribuídos geograficamente.

Para responder a essas (e outras) questões, é importante levantar algumas informações a cerca de (RUBLE, 1997):

- volumes de dados, frequência de disparo de casos de uso e expectativas de tempos de resposta para os mesmos;
- topologia geográfica do negócio e necessidades de distribuição geográfica da computação, incluindo a necessidade de distribuição de dados e de processos nos diferentes locais.

A seguir, alguns aspectos importantes para o projeto de sistemas de informação distribuídos são discutidos.

3.5.1 – Efetuando Estimativas

Determinar a arquitetura mais apropriada para um sistema de informação distribuído envolve a quantificação da capacidade de computação necessária para o problema. O modelo de análise provê a base necessária para a realização de estimativas dos requisitos de computação do sistema final. Assim, uma atividade do projeto de SIs consiste em completar as informações de análise, fazendo estimativas importantes acerca do modelo estrutural e do modelo de casos de uso.

Estimativas sobre o Modelo Estrutural

A realização de estimativas sobre o modelo conceitual estrutural (diagrama de classes) é importante para a definição da arquitetura, sobretudo no que concerne à escolha de táticas a serem aplicadas (vide Seção 3.7). Uma importante estimativa é o tamanho da base de dados do sistema. Essa informação é útil para apoiar a definição de quais estratégias adotar em relação ao armazenamento e a comunicação de dados.

Para se estimar o tamanho da base de dados de um sistema, duas informações são essenciais: o tamanho de uma instância de uma classe (para cada classe) e o número estimado de instâncias que poderão ser acumuladas ao longo do tempo. Para determinar o tamanho de um objeto, devem-se observar os tipos de dados de atributo da classe. Já a estimativa do número de instâncias esperadas não é tão simples. Algumas classes, tais como *Pedido* e *Itens de Pedido*, têm potencial de crescimento ilimitado. Para essas classes, o modelo de casos de uso pode dizer quais casos de uso criam instâncias das mesmas. Para estimar o tamanho das bases de dados relativas a essas classes, é necessário saber quão frequentemente o caso de uso ocorre e o período de retenção da informação na base de dados.

Em suma, as seguintes informações para cada classe do modelo estrutural devem ser coletadas:

- tamanho estimado de uma instância, calculado pelo somatório dos tipos de dados de cada atributo,
- taxa de ocorrência dos casos de uso que criam novas instâncias da classe,
- período de retenção.

Essas estimativas são usadas para estimar os recursos necessários para armazenar adequadamente os dados. Mesmo se espaço em disco não for um problema para o projeto, elas são úteis para tratar outros aspectos, como aspectos relacionados à comunicação de dados e à distribuição geográfica de dados.

Uma vez que é importante saber que tipo de operação um caso de uso pode realizar em uma classe, é útil produzir uma matriz CRUD (*Create*, *Retrieve*, *Update*, *Delete* - Criar, Recuperar, Atualizar, Eliminar) *Caso de Uso x Classe*, mostrando se, por ocasião da ocorrência do caso de uso, o sistema vai criar, atualizar, recuperar ou eliminar instâncias da classe. A Figura 3.10 mostra um exemplo de matriz CRUD *Caso de Uso x Classe*.

	Classes		
Casos de Uso	Cliente	Pedido	Item de Pedido
Efetuar pedido	R	C	C
Cancelar pedido	R	RD	RD
Alterar pedido	R	RU	CRUD

Figura 3.10 – Exemplo de uma Matriz CRUD *Caso de Uso x Classe*.

Estimativas sobre o Modelo de Casos de Uso

Um modelo de casos de uso captura as funcionalidades de um sistema segundo uma perspectiva externa, isto é, dos usuários. Em um modelo de casos de uso, alguns casos de uso são mais críticos do que outros em termos do negócio da organização. Além disso, cada caso de uso pode ter requisitos não funcionais específicos.

No que se refere a desempenho, pode ser importante levantar informações acerca da frequência de ocorrência (ou disparo) e o tempo de resposta desejado para um caso de uso. O objetivo é identificar os casos de uso críticos, já que eles serão objeto de atenção especial. Para esses, devem ser levantadas informações sobre a frequência média de ocorrência, a taxa de pico e o período de tempo do pico.

Se a ocorrência dos eventos for uniforme ao longo do tempo, a frequência média de disparo aponta para o nível normal de operação do sistema. Para eventos irregulares, contudo, é necessário conhecer as taxas de pico e o período em que esses picos ocorrem. Além disso, uma importante questão se coloca: o sistema tem de ser dimensionado para tratar picos, de modo a sempre atender satisfatoriamente às necessidades dos usuários? Se o sistema for dimensionado pela capacidade de pico, provavelmente ficará ocioso durante grande parte do tempo. Por outro lado, se for dimensionado pela média, nos momentos de pico não atenderá totalmente às necessidades dos usuários. Esta não é uma decisão fácil e não deve ser tomada pelo projetista. Ao contrário, essa decisão cabe, principalmente, ao cliente. De maneira geral, a menos que o custo seja proibitivo, a maioria das organizações prefere gastar mais, dimensionando o sistema pelo pico.

Não é necessário analisar todos os casos de uso de um sistema. É suficiente concentrar a atenção nos principais casos de uso do negócio, aqueles com os maiores impactos sobre o cliente, com os maiores volumes de dados e com as localizações mais remotas geograficamente.

3.5.2 – Determinando Requisitos de Distribuição Geográfica

Outro passo importante do projeto arquitetônico consiste em examinar a distribuição geográfica dos casos de uso, o que conduz naturalmente à distribuição necessária dos dados. Juntos, a frequência de disparo dos casos de uso, volume de dados, restrições de tempo de resposta e a distribuição geográfica do negócio formarão a base para se determinar uma arquitetura aceitável para o sistema em questão.

Algumas matrizes podem ser usadas para mapear o modelo de análise na topologia de localizações do negócio. Uma matriz *Caso de Uso x Localização do Negócio*, como a mostrada na Figura 3.11, juntamente com uma matriz *Caso de Uso x Classe*, mostrada anteriormente na Figura 3.10, permite mapear as necessidades de distribuição geográfica de funcionalidades e dados de um sistema.

	Localização		
Casos de Uso	Internet	Matriz	Filiais
Efetuar pedido		X	X
Cancelar pedido	X	X	
Alterar pedido	X	X	

Figura 3.11 – Exemplo de uma Matriz *Caso de Uso x Localização do Negócio*.

Uma vez levantadas as necessidades de computação de cada localização do negócio, pode-se avaliar qual a melhor estratégia para a distribuição de dados. Algumas opções são manter todos os dados em uma única base de dados centralizada, descentralizar completamente os dados, ou usar uma abordagem de fragmentação.

Quando a estratégia de uma única base de dados centralizada é adotada, os dados são mantidos em um servidor de dados central e todas as aplicações que necessitem acessá-los devem fazer suas consultas e atualizações no servidor central. Os benefícios são numerosos, tais como: (i) os dados estão sempre atualizados e não há redundância; (ii) o projeto é mais simples, tendo em vista que a segurança pode ser mantida centralmente e nenhuma rotina de sincronização é requerida; e (iii) é fácil fazer cópias de segurança dos dados. Entretanto, há também desvantagens, tais como: (i) não permite operação desconectada e, portanto, a disponibilidade do sistema é fortemente afetada pela comunicação de dados e pela disponibilidade do servidor de dados; (ii) o desempenho do sistema também é muito dependente da velocidade de comunicação de dados.

Uma solução diametralmente oposta consiste em ter os dados totalmente descentralizados e replicados. Neste caso a base de dados é completamente replicada em todos os locais que dela necessitem. Atualizações em um local podem ser irradiadas (*broadcast*) para outros locais em tempo real. São benefícios dessa estratégia: (i) o tempo de resposta não é onerado pelo tráfego em rede; e (ii) a disponibilidade também tem sua dependência diminuída em relação à comunicação de dados. Contudo, há diversos problemas, dentre eles: (i) o tráfego global na rede aumenta devido à replicação de dados em todos os locais; (ii) rotinas complexas de sincronização são necessárias para manter as várias cópias da base de dados atualizadas; (iii) podem surgir problemas se o mesmo registro for atualizado em dois locais; (iv) se um dos servidores cair ou o

software de replicação falhar, pode ser difícil reconstruir o conjunto dos dados e reaplicar atualizações na ordem correta; (v) procedimentos de backup tornam-se mais complexos, uma vez que não se tem uma visão clara de qual é a base de dados principal; e (vi) dados completamente replicados conduzem a uma redundância desnecessária de dados.

Uma abordagem intermediária entre a centralização e a replicação total é a fragmentação de dados, na qual apenas os dados necessários para cada localidade são mantidos localmente. A fragmentação pode ser:

- Vertical: ocorre quando apenas certos elementos (classes / atributos das classes) são fisicamente distribuídos a locais remotos. Cada localidade possui apenas aqueles elementos que são requeridos pelos casos de uso que lá ocorrem. Isto reduz tráfego em rede, pois apenas os elementos de dados necessários precisam ser sincronizados com outros locais. Entretanto, essa estratégia pode ser bastante complexa para gerenciar. Os procedimentos de replicação devem ser capazes de sincronizar atualizações atributo-a-atributo em diferentes locais.
- Horizontal: ocorre quando apenas algumas instâncias de uma classe são fisicamente distribuídas a locais remotos. Esta estratégia é empregada tipicamente quando localidades têm seus próprios dados que não são manipulados em outras localidades. Cada localidade tem sua própria cópia do banco de dados. Os esquemas são idênticos, mas os dados que povoam as bases de dados são diferentes. Geralmente, há uma base de dados principal que contém todos os registros. Assim como a fragmentação vertical, a horizontal diminui o tráfego na rede, eliminando transferências de dados desnecessárias. Contudo, o processo de sincronização é também de alguma forma complicado, principalmente quando diferentes locais compartilham registros.
- Híbrida: bases de dados distribuídas compartilham os mesmos tipos de entidades lógicas, mas possuem diferentes atributos e instâncias. Cada local possui apenas os atributos e instâncias que são realmente necessários para os eventos que ali ocorrem. É fácil perceber que tal estratégia pode ser muito difícil de gerenciar.

3.6 – Aplicações Web e Tecnologias Relacionadas

No início, a *World Wide Web*, ou simplesmente Web, era um ambiente que hospedava documentos hipermídia estáticos, que eram entregues diretamente a um navegador do cliente como resposta a requisições HTTP (*HyperText Transfer Protocol*, protocolo utilizado na Web). Com o surgimento da tecnologia CGI (*Common Gateway Interface*) e de linguagens de programação para a *Web*, tais como PHP, ASP e Java Servlets / JSP, o ambiente para desenvolvimento de aplicações para Web se tornou mais poderoso, permitindo o desenvolvimento de aplicações de negócio. Em pouco tempo, grandes sistemas começaram a ser desenvolvidos para a Web. Surgia, então, o conceito de aplicação Web (SOUZA, 2007) (CASTELEYN et al., 2009).

De maneira simplista, uma aplicação Web consiste em um conjunto de páginas Web para interação com o usuário, armazenando, processando e provendo informações.

Hoje existe uma variedade de aplicações Web, indo desde coleções simples de páginas estáticas HTML até aplicações corporativas distribuídas usando a Web como plataforma de execução (CASTELEYN et al., 2009).

3.6.1 – As Bases das Aplicações Web

O protocolo HTTP (*HiperText Transfer Protocol*) é o ingrediente mais básico sobre o qual a Web está fundamentada. Ele é um protocolo de aplicações cliente-servidor que define um formato padrão para especificar a requisição de recursos na Web. Através dele, um usuário utilizando uma aplicação cliente (p.ex., um navegador) pode requisitar recursos disponíveis em um servidor remoto (p.ex., o servidor Web), como ilustra a Figura 3.12. Tipicamente os recursos passados via HTTP são páginas HTML, mas, de maneira mais geral, uma requisição pode estar relacionada a um arquivo em qualquer formato armazenado no servidor Web ou à invocação de um programa a ser executado no lado do servidor. Uma vez que tais recursos estão distribuídos ao longo da Internet, é necessário um mecanismo de identificação que permita localizá-los e acessá-los. O identificador usado para referenciar esses recursos é uma URL (*Uniform Resource Locator*) (CASTELEYN et al., 2009).

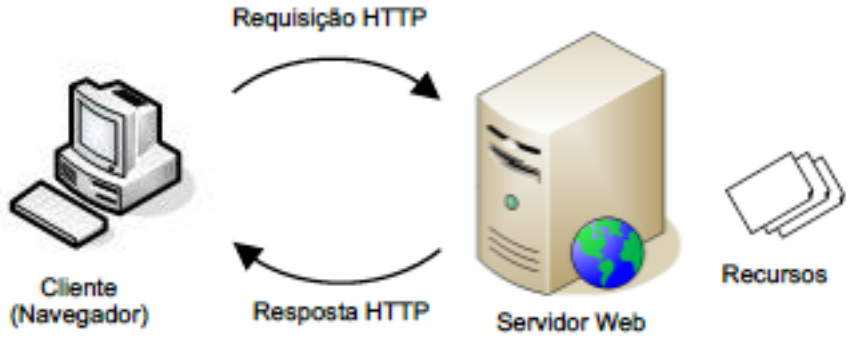


Figura 3.12 – Ciclo Requisição–Resposta de HTTP (adaptado de (CASTELEYN et al., 2009))

Além de gerenciar a requisição e a transferência de recursos através do protocolo HTTP, um navegador Web também trata da apresentação visual dos recursos. A linguagem HTML (*HyperText Markup Language*) é comumente usada para expressar o conteúdo e a formatação visual de páginas Web. O navegador recebe como entrada o conteúdo marcado, interpreta o significado das *tags* e transforma o conteúdo recebido em um documento "renderizado" (CASTELEYN et al., 2009).

As primeiras páginas HTML eram caracterizadas por parcos recursos de apresentação e interação. Entretanto, com a expansão e difusão da Web, documentos simples logo se tornaram inadequados e novos requisitos de apresentação começaram a surgir. Como avanços ao longo da linha de evolução da Web podem ser citados o uso de folhas de estilo para separar conteúdo e apresentação, e tecnologias para tornar clientes Web dinâmicos e capazes de suportar um conjunto mais rico de características de apresentação e interatividade. Tais características englobam, dentre outros, a capacidade de alterar dinamicamente as propriedades de apresentação de páginas Web e a capacidade de executar pedaços da lógica de negócio no lado do cliente, usando linguagens de script.

3.6.2 – Aplicações Web Dinâmicas

Linguagens de script, tal como JavaScript, abriram espaço para uma abordagem de HTML dinâmico. Um script de lado do cliente é um programa interpretado e executado pelo navegador quando uma página é carregada ou quando o usuário invoca algum evento sobre elementos de uma página. Eles representam a parte "ativa" dessa abordagem, enquanto a linguagem de marcação HTML representa a parte estática, a qual está sujeita a alterações dinâmicas pela lógica do script. A combinação de scripts, HTML e DOM (*Document Object Model*, uma plataforma e modelo independentes de linguagem para representar documentos HTML e XML) ofereceu uma maneira eficaz de implementar comportamentos dinâmicos no navegador (CASTELEYN et al., 2009).

O estabelecimento de tecnologias XML e algumas extensões às arquiteturas cliente-servidor tradicionais pavimentaram o caminho para o desenvolvimento de páginas Web dinâmicas, compostas em tempo de execução a partir de conteúdo extraído dinamicamente de uma fonte de dados, o qual é usado para produzir a apresentação final da página "durante o voo" (CASTELEYN et al., 2009).

A maneira mais simples de se construir dinamicamente uma página Web em resposta a uma requisição HTTP é deixar que o servidor HTTP delegue a recuperação de dados dinâmicos e a construção da página para um programa externo, que recebe parâmetros de entrada e gera uma página como saída. A invocação para o programa externo ocorre quando a requisição HTTP vinda do navegador inclui uma URL apontando para um programa ao invés de apontar para um documento estático.

Para que o servidor se comunique com um programa externo, é necessário usar uma interface padrão, chamada *Common Gateway Interface* – CGI. CGI permite que um servidor Web invoque programas, chamados programas CGI, que são executados durante o voo para dinamicamente construir a página. Um programa CGI pode fazer consultas em uma base de dados para extrair os dados (que são usados, então, para montar a página HTML) e pode armazenar entradas de dados do usuário na base de dados, inserindo ou atualizando dados nessa base. A Figura 3.13 ilustra os passos que caracterizam a geração dinâmica de páginas com CGI (CASTELEYN et al., 2009).

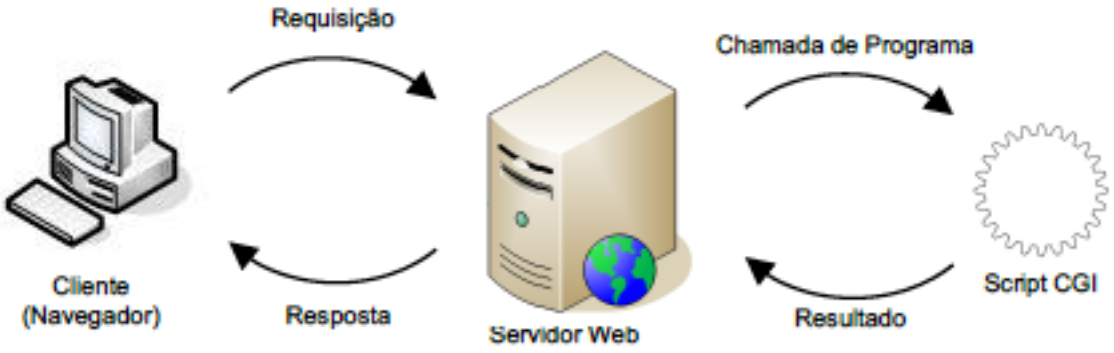


Figura 3.13 – Construção dinâmica de páginas Web em resposta a uma requisição HTTP (adaptado de (CASTELEYN et al., 2009))

Primeiro, o cliente envia uma requisição HTTP para o servidor, requisitando a execução de um programa CGI. A requisição pode incluir dados de entrada. Usando uma interface CGI, o servidor HTTP invoca o programa externo (i.e., o script CGI), passando os parâmetros de entrada. O programa externo, então, constrói a página e a

retorna como resultado para o servidor HTTP. O servidor HTTP monta a resposta HTTP, a qual embute a página construída pelo programa externo, e a envia de volta ao cliente (CASTELEYN et al., 2009).

Entretanto, a arquitetura CGI tem algumas desvantagens significativas, que a tornam impraticável na maioria das situações. O principal problema é o desempenho: para cada requisição HTTP para um script CGI, o servidor Web inicia um novo processo, o qual é terminado no fim da execução do script. A criação e a finalização de processos são atividades muito custosas, as quais se tornam rapidamente um gargalo. Além disso, o fato de terminar o processo após a execução do script CGI, após cada requisição, impede que as informações sobre a interação com o usuário sejam mantidas entre requisições de usuário consecutivas (a menos que elas sejam armazenadas em uma base de dados), o que tem impacto novamente no desempenho (CASTELEYN et al., 2009).

3.6.3 – Extensões de Servidores Web

As limitações da arquitetura CGI podem ser superadas pela extensão das capacidades do servidor Web com uma máquina capaz de executar aplicações, na qual os programas para computar as respostas HTTP podem ser executados de maneira eficiente, sem serem terminados após cada requisição, e os recursos compartilhados podem ser associados a uma ou mais aplicações e concorrentemente acessados por múltiplos usuários. Tal arquitetura estendida, ilustrada na Figura 3.14, tipicamente oferece, ainda, uma memória para armazenar dados da sessão, cuja duração atravessa múltiplas requisições HTTP (CASTELEYN et al., 2009).

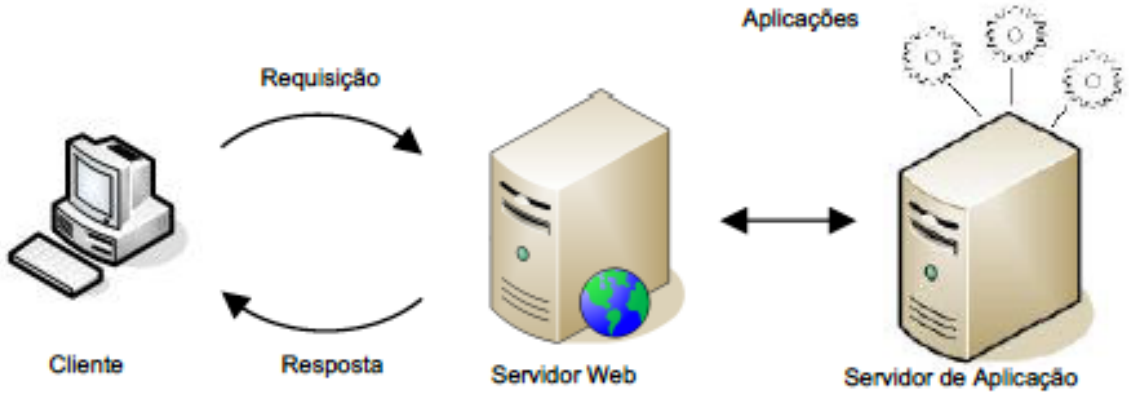


Figura 3.14 – Arquitetura de Servidor Web estendida (adaptado de (CASTELEYN et al., 2009))

Um exemplo de arquitetura de servidor Web estendida é a API Java Servlet, que associa o servidor Web com uma máquina virtual Java (*Java Virtual Machine* – JVM). A JVM suporta a execução de um programa Java especial, o container servlet, que por sua vez é responsável por gerenciar dados de sessão e executar servlets Java. Servlets Java são programas Java que podem ser invocados por requisições HTTP para dinamicamente construir páginas. O container servlet é responsável por receber a requisição HTTP do servidor Web, criar a sessão de usuário quando necessário, invocar o servlet associado à requisição e passar para ele os parâmetros da requisição na forma de objetos Java (CASTELEYN et al., 2009).

O uso de scripts de lado do cliente permite construir páginas dinâmicas contendo instruções de programação dentro de *templates* de página HTML. Essas instruções são executadas por um programa do servidor. Assim, quando uma requisição HTTP refere-se a um *template*, o seguinte processo ocorre: (i) o servidor Web encaminha o *template* para a máquina de script; (ii) a máquina de script processa as instruções embutidas, determina as partes dinâmicas da página e as inclui no resultado, uma página HTML plana; (iii) a página gerada é retornada para o servidor Web; (iv) este, por sua vez, encaminha a página para o cliente (navegador), onde é apresentada (renderizada). Esse processo é completamente transparente para o navegador que recebe a página HTML resultante, cujo código é idêntico ao de uma página estática. A Figura 3.15 ilustra o processo descrito acima (CASTELEYN et al., 2009).

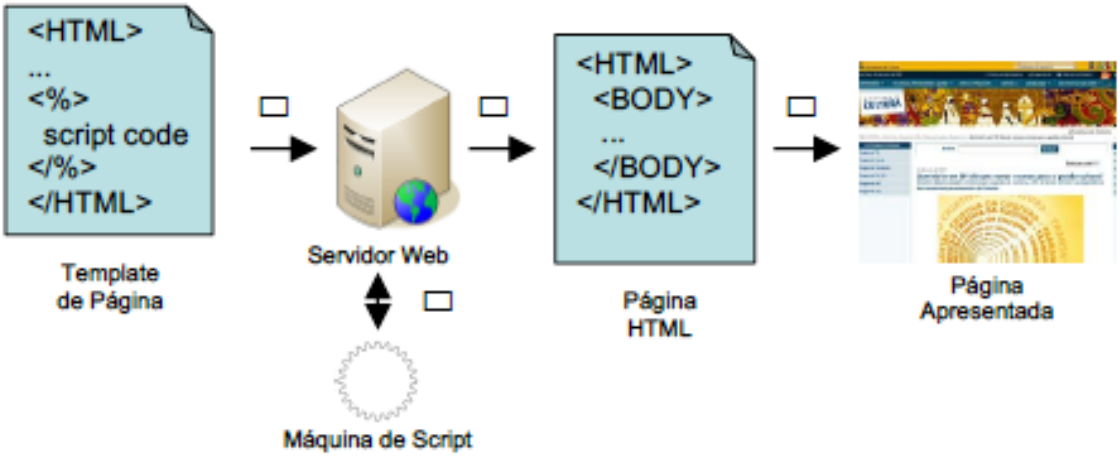


Figura 3.15 – Execução de um script em *template* de página (adaptado de (CASTELEYN et al., 2009))

A despeito da similaridade, o estilo de codificação de scripts de lado do cliente é completamente diferente do estilo de codificação de servlets. Um servlet contém instruções para gerar a página inteira, enquanto um *template* de página contém HTML regular junto com instruções de programação, as quais se limitam a computar a parte variável da página. Atualmente, existem diversas linguagens de script de lado do cliente disponíveis no mercado, dentre elas PHP, ASP e JSP. Uma página JSP, por exemplo, é composta por blocos de código estático e por porções de código Java. Vale ressaltar, contudo, que JSP é, na verdade, uma extensão de servlets Java: na primeira vez que um servidor Web (um container JSP) recebe uma requisição para uma página JSP, o *template* JSP é traduzido para um servlet, o qual é compilado, armazenado em memória principal e executado (CASTELEYN et al., 2009).

Ainda que os scripts de lado de cliente facilitem o desenvolvimento de aplicações Web dinâmicas, é necessário misturar programação com conteúdo e marcação. Assim, o programador e o designer gráfico precisam trabalhar no mesmo arquivo, o que limita a separação de preocupações entre diferentes aspectos do desenvolvimento Web: conteúdo estático, apresentação (*look and feel*) e lógica de programação (CASTELEYN et al., 2009).

As bibliotecas de *tags* de lado do cliente dão um passo à frente na separação de conteúdo e marcação da programação de um *template* de página dinâmica. A ideia por trás dessa abordagem está em camuflar o código de programação usando *tags* XML, as quais podem ser inseridas como elementos de marcação regulares, mas que são

executadas por um interpretador em tempo de execução. Em JSP, por exemplo, *tags* customizadas são usadas para invocar JavaBeans, componentes Java. O código do JavaBean é ocultado do desenvolvedor da página. Ele vai apenas invocar os métodos providos pelas correspondentes classes Java (CASTELEYN et al., 2009).

Existem outras tecnologias que também permitem a implementação de lógica de negócio executando no lado do cliente, tais como Java applets e ActiveX. Entretanto, enquanto as linguagens de script são integradas aos navegadores, essas tecnologias não são integradas a eles e, portanto, não permitem HTML dinâmico. Ao contrário, aplicações dessa natureza representam pequenas aplicações isoladas que são embutidas na marcação HTML de uma página Web, baixadas quando a página é acessada, e executadas localmente quando a página é exibida. A execução de Java applets, contudo, requer a instalação da máquina virtual Java na máquina cliente, enquanto os controles ActiveX são suportados basicamente pelo navegador Internet Explorer (CASTELEYN et al., 2009).

As extensões de servidor Web descritas anteriormente oferecem um eficiente meio de implementar aplicações Web com estado, i.e., aplicações baseadas em HTTP capazes de reter o estado da interação com o usuário. Informações de estado podem ser armazenadas no lado do cliente, na forma de cookies, ou do lado do servidor, na forma de dados da sessão (CASTELEYN et al., 2009).

3.6.4 – Arquiteturas Multicamadas de Aplicações Web

Aplicações Web de grande escala, tais como portais e aplicações de comércio eletrônico, tipicamente estão expostas a um grande número de requisições concorrentes e devem exibir um alto nível de disponibilidade. Para tal, são necessárias arquiteturas de software modulares e multicamadas, nas quais os diferentes componentes possam ser facilmente replicados para aumentar o desempenho e evitar pontos de falha (CASTELEYN et al., 2009). Assim, uma aplicação Web pode ser considerada um tipo de sistema distribuído, com uma arquitetura cliente-servidor com as seguintes características (DI LUCCA; FASOLINO, 2006 apud MARINHO; RESENDE, 2011):

- Grande número de usuários distribuídos;
- Ambiente de execução heterogêneo composto de vários tipos de hardware, conexões de rede, sistemas operacionais, servidores e navegadores;
- Natureza extremamente heterogênea, que depende da grande variedade de componentes que a constituem. Esses componentes podem ser escritos em diferentes linguagens de programação e serem de natureza diferente, tais como componentes legados e componentes de prateleira (COTS);
- Habilidade de gerar componentes em tempo de execução, de acordo com a entrada do usuário e status do servidor.

Arquiteturas de software de aplicações Web dessa natureza são normalmente organizadas em três camadas de software (CASTELEYN et al., 2009):

- Camada de Apresentação: responsável por processar as requisições vindas do cliente e construir as páginas HTML. É desenvolvida usando as extensões de servidores Web discutidas anteriormente, as quais são capazes de construir

dinamicamente as páginas HTML a serem enviadas como resposta para o cliente. Essas páginas são geradas tomando por base os dados produzidos pela execução de componentes de negócio, que estão na camada abaixo.

- Camada de Lógica de Negócio: responsável por executar componentes que realizam a lógica de negócio da aplicação. Para tal, comunica-se com a camada de gerência de recursos para acessar bases de dados persistentes, sistemas legados ou para invocar serviços externos.
- Camada de Gerência de Recursos: representa o conjunto de serviços oferecidos por diferentes sistemas, tais como aqueles suportando o acesso a bases de dados, a outros sistemas ou, de maneira geral, a serviços Web externos.

Para tornar real essa arquitetura de software de três camadas, é necessário um ambiente de execução suportando comunicação em camadas. A Figura 3.16 mostra um ambiente dessa natureza. O servidor Web recebe a requisição HTTP oriunda do cliente e a transforma em uma requisição para o motor de script. Este executa o programa associado com a URL requisitada, o qual pode incluir chamadas para componentes de negócio hospedados no servidor de aplicação. Os componentes de negócio, por sua vez, enviam consultas para servidores de dados, operam sobre seus resultados e geram uma resposta para o motor de script. Os dados são integrados pelo motor de script a uma página HTML, a qual é enviada de volta ao cliente pelo servidor Web. O ambiente de execução ilustrado na Figura 3.16 tem muitas implementações comerciais, dentre elas as plataformas Java EE (*Java Enterprise Edition*) e .NET (CASTELEYN et al., 2009).

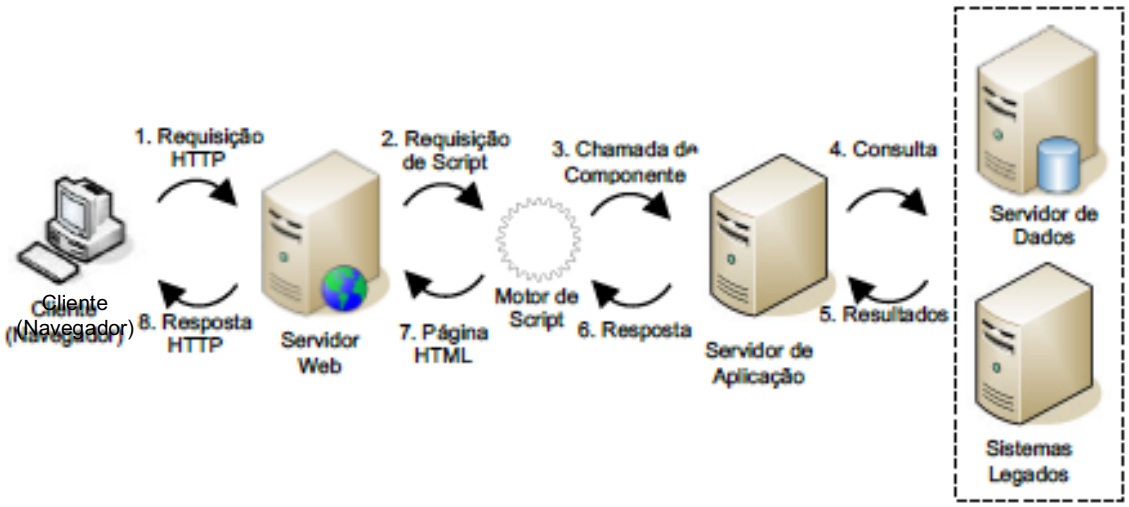


Figura 3.16 – Ambiente de Execução para Arquitetura Multicamadas (adaptado de (CASTELEYN et al., 2009))

A maior parte das aplicações Web adota uma filosofia de cliente leve, na qual a maior parte da lógica de negócio reside nos servidores. As aplicações Web tradicionais foram classificadas por Tilley e Huang (2001, apud MARINHO; RESENDE, 2011) em três classes com complexidade crescente:

- A classe 1 é constituída por aplicações estáticas implementadas em HTML e que não têm interatividade com o usuário.

- A classe 2 engloba as aplicações Web que proveem interação do usuário com páginas HTML dinâmicas e que associam ações implementadas em scripts a eventos gerados pelo usuário.
- A classe 3 reúne as aplicações de conteúdo dinâmico, cujas páginas podem ser criadas em tempo de execução, de acordo com a interação do usuário com a aplicação.

Contudo, essas aplicações ainda permitem uma interação pouco flexível. Mais recentemente, com a implementação da API XMLHttpRequest dentro da maioria dos navegadores, scripts de lado do cliente passaram a poder fazer, também, requisições HTTP para um servidor Web de maneira transparente em background e independentemente da interação com o usuário. O uso combinado de JavaScript, XMLHttpRequest, XML e DOM é conhecido como AJAX (*Asynchronous JavaScript and XML*). Atualmente AJAX é quase um sinônimo de aplicações Web distribuídas entre cliente e servidor. Essa tecnologia permitiu que desenvolvedores tornassem a maior parte do HTML dinâmico, já que passou a ser possível carregar conteúdo do servidor Web e adicioná-lo dinamicamente à página sendo exibida no navegador sem que este último tenha que dar um "*refresh*" ou recarregar a página inteira. Isso finalmente levou a aplicações mais ergonômicas e responsivas, ditas Aplicações Ricas para Internet (*Rich Internet Applications* - RIAs) (CASTELEYN et al., 2009).

3.6.5 – Aplicações Ricas para Internet - RIAs

O termo RIA designa uma classe de soluções caracterizada pelo propósito comum de adicionar novas características às aplicações Web tradicionais (MARINHO; RESENDE, 2011). Essas características incluem (CASTELEYN et al., 2009):

- Interfaces com o usuário sofisticadas, incluindo facilidades do tipo arrastar e soltar (*drag & drop*), animações e sincronização multimídia, buscando dar às aplicações Web uma interatividade semelhante à das aplicações *desktop*.
- Mecanismos para minimizar a transferência de dados entre cliente e servidor, movendo a gerência das camadas de apresentação e interação do servidor para o cliente.
- Armazenamento e processamento de dados tanto no lado do cliente quanto no lado do servidor, buscando melhor utilizar a capacidade de computação do cliente.

Do ponto de vista da distribuição de dados, nas RIAs, os dados da aplicação podem ser armazenados tanto no cliente quanto no servidor. No caso dos dados serem armazenados no cliente, estes são enviados ao servidor ao final de cada operação. Do ponto de vista da distribuição da lógica de negócio, nas RIAs tanto o cliente quanto o servidor podem realizar operações complexas. Do ponto de vista da comunicação cliente-servidor, as RIAs permitem tanto a comunicação síncrona quanto a comunicação assíncrona (MARINHO; RESENDE, 2011).

A distribuição de dados e funcionalidades entre cliente e servidor amplia as características dos eventos produzidos. Estes podem ser originados, detectados e processados de diferentes formas. Isso permite atualização parcial de página (comunicação delta), alterações em elementos individuais da página (*Page*

Rearrangement) e alterações na estrutura da página (*Display Morphing*). Contudo, há um aumento do esforço de desenvolvimento, já que, conforme discutido na Seção 3.5.2, há necessidade de replicação de dados, políticas de alocação e sincronização, bem como rotinas para manter a consistência dos dados, uma vez que os clientes precisam manter dados (MARINHO; RESENDE, 2011).

Finalmente, do ponto de vista de aprimoramento da interface com o usuário, as RIAs possibilitam apresentação e interações com o usuário avançadas e menor tempo de resposta. A aplicação pode operar como aplicação de página única, evitando atualizações da página inteira e permitindo carregamento progressivo de elementos da página. Contudo, pode gerar problemas de desempenho e de compatibilidade com navegadores (MARINHO; RESENDE, 2011).

A estrutura de página única, em contraposição à estrutura de múltiplas páginas das aplicações Web tradicionais, segue um estilo arquitetônico denominado SPIAR (*Single Page Internet Application aRchitecture*). Esse estilo abstrai as características comuns de vários *frameworks* e enfatiza o desenvolvimento baseado em componentes de interfaces com o usuário, a comunicação delta entre componentes cliente/servidor e a notificação de mudanças de estado através dos componentes. A comunicação delta é aquela em que apenas informações sobre estado são trocadas entre o cliente e o servidor, evitando o fluxo de informações redundantes. Segundo Mesbah e van Deursen (MESBAH; VAN DEURSEN, 2008 apud MARINHO; RESENDE, 2011), essas características do estilo arquitetônico SPIAR têm por objetivo a melhoria da interatividade do usuário, a diminuição da latência percebida pelo usuário, o aumento da coerência dos dados e maior facilidade de desenvolvimento.

RIAs foram concebidas primeiramente pela Adobe, quando da introdução de produtos para autoria multimídia, como Flash (CASTELEYN et al., 2009). Atualmente, existem várias tecnologias (AJAX, Silverlight, Flex, AIR, Ruby on Rails, JavaFX) e diversos *patterns* (p.ex., AjaxPatterns - http://ajaxpatterns.org/) disponíveis para esse tipo de aplicação (MARINHO; RESENDE, 2011).

Em suma, as aplicações Web passaram a poder ter partes da lógica de aplicação no lado do cliente, mudando o paradigma cliente-servidor estrito das primeiras aplicações Web em direção a um paradigma de sistemas distribuídos em que a distinção tradicional entre clientes e servidores é cada vez mais turva. A lógica de aplicação do lado do cliente não é mais apenas usada para enriquecer a camada de apresentação com características de interfaces com o usuário dinâmicas; ao contrário, ela é parte da lógica de negócio global da aplicação (CASTELEYN et al., 2009).

Conforme discutido na Seção 3.2, o termo aplicações web envolve diversas subcategorias de aplicações, tais como *websites* informativos, aplicações transacionais, e portais. Neste texto, no entanto, o foco está em uma categoria específica de aplicações *Web*: os Sistemas de Informação Baseados na Web.

3.7 – O Processo de Projeto de Software

Projetar a arquitetura de um software requer o levantamento de informações relativas à plataforma de implementação do sistema, as quais se somarão ao conhecimento acerca dos requisitos funcionais e não funcionais, para embasar as decisões relativas à arquitetura do sistema que está sendo projetado. De maneira geral, o processo de projetar envolve os seguintes passos:

1. Levantar informações acerca da plataforma de implementação do sistema, incluindo linguagem de programação a ser adotada, mecanismo de persistência e necessidades de distribuição geográfica.
2. Com base nos requisitos, iniciar a decomposição do sistema em subsistemas, considerando preferencialmente uma decomposição pelo domínio do problema. Se na fase de análise já tiver sido estabelecida uma decomposição inicial em subsistemas, esta deverá ser utilizada. Neste momento, deve-se escolher um estilo arquitetônico (ou uma combinação adequada de estilos arquitetônicos) para organizar a estrutura geral do sistema (Seção 3.3). Um bom ponto de partida para essa escolha pode ser a classe de sistemas à qual pertence o sistema que está sendo projetado (Seção 3.2).
3. Priorizar os atributos de qualidade e definir quais deles guiarão o projeto da arquitetura, ditos condutores da arquitetura (Seção 2.3).
4. Para cada atributo de qualidade, identificar táticas a serem aplicadas. Uma vez que essas táticas podem levar a conflitos, procurar balanceá-las, respeitando as prioridades estabelecidas no passo anterior (Seção 3.7).
5. Estabelecer uma arquitetura base, identificando tipos de módulos e tipos de relacionamentos entre eles, os quais são dados, essencialmente, pela combinação dos estilos arquitetônicos escolhidos.
6. Alocar requisitos funcionais (casos de uso) e não funcionais aos componentes da arquitetura.
7. Avaliar a arquitetura, procurando identificar se ela acomoda tanto os requisitos funcionais, quanto os atributos de qualidade, dando atenção especial aos atributos considerados "condutores da arquitetura", ou seja, aqueles de maior prioridade.
8. Uma vez definida a arquitetura em seu nível mais alto, passar ao projeto de seus elementos. Padrões arquitetônicos são instrumentos muito valiosos para o projeto dos componentes da arquitetura.

É importante frisar que o processo descrito acima, ainda que descrito de forma sequencial, é um processo essencialmente iterativo. Por exemplo, a priorização dos atributos de qualidade pode levar a alterações na combinação de estilos arquitetônicos escolhido no passo anterior. O mesmo pode ocorrer em relação às táticas selecionadas. Ainda, as táticas podem demandar o levantamento de novas informações (passo 1). Ao se avaliar a arquitetura (passo 7), pode-se concluir que ela não está satisfatória, sendo necessário retomar do início.

Uma vez que neste texto o foco recai sobre sistemas de informação, algumas diretrizes mais específicas podem ser providas:

- Passo 1: atenção especial deve ser dada à questão da distribuição geográfica, bem como para a topologia da arquitetura de hardware a ser adotada. Aspectos discutidos nas seções 3.5 e 3.6 devem ser observados aqui.
- Passo 2: uma combinação de partições e camadas tende a ser um bom ponto de partida para a estruturação global de sistemas de informação. Partições podem ser derivadas a partir do domínio do problema, levando-se em conta funcionalidades coesas que levem à criação de subsistemas fracamente acoplados. Idealmente, alguma divisão em subsistemas deve ter sido feita na fase de análise e, se feita da maneira correta, a mesma deve ser preservada. As partições podem ser estruturadas em camadas e novamente um bom ponto de partida é considerar camadas de Interface com o Usuário, Domínio do Problema e Gerência de Dados, conforme discutido na Seção 3.4.

3.8 – Detalhando os Componentes da Arquitetura de Software

Uma vez estabelecida a arquitetura global do sistema, os passos subsequentes referem-se ao detalhamento de seus elementos. Ainda não se trata efetivamente de um projeto detalhado (projeto de classes), mas está-se em um nível intermediário onde muitas decisões são ainda de cunho arquitetônico. Assim, o uso de padrões arquitetônicos é muito importante neste contexto.

Como o foco deste texto é o projeto de Sistemas de Informação, os capítulos que se seguem discutem o projeto de componentes de Lógica de Negócio (Capítulo 4), Interface com o Usuário (Capítulo 5) e Gerência de Dados (Capítulo 6). Alguns padrões arquitetônicos e de projeto (*design patterns*) úteis para o projeto desses componentes são discutidos nesses capítulos.

Capítulo 4 – Projeto da Lógica de Negócio

A camada de lógica de negócio engloba o conjunto de classes que vai realizar toda a lógica do sistema de informação. As demais camadas são derivadas ou dependentes dessa camada (WAZLAWICK, 2004) e, portanto, é interessante iniciar o projeto dos componentes da arquitetura do sistema por ela.

Os modelos construídos na fase de análise são os principais insumos para o projeto dessa camada, em especial o modelo conceitual estrutural e o modelo de casos de uso. Os diagramas de classes da fase de análise são a base para a construção dos diagramas de classes da fase de projeto. De fato, a versão inicial do modelo estrutural de projeto (diagramas de classes de projeto) da lógica de negócio é uma cópia do modelo conceitual estrutural. Durante o projeto, esse modelo será objeto de refinamento, visando incorporar informações importantes para a implementação, tais como distribuição de responsabilidades entre as classes (definição de métodos das classes) e definição de navegabilidades e visibilidades. Além disso, alterações na estrutura do diagrama de classes podem ser necessárias para tratar requisitos não funcionais, tais como usabilidade e desempenho.

Para organizar a lógica de negócio, um bom ponto de partida são os padrões arquitetônicos relativos a essa camada. Fowler (2003) apresenta quatro desses padrões, a saber: Script de Transação (*Transaction Script*), Modelo de Domínio (*Domain Model*), Módulo de Tabelas (*Table Module*) e Camada de Serviço (*Service Layer*). No contexto do desenvolvimento de Sistemas de Informação Orientados a Objetos, merecem destaque os padrões Modelo de Domínio e Camada de Serviço.

Uma questão bastante importante tratada por esses padrões é a distribuição de responsabilidades ao longo das classes que compõem o sistema. No mundo de objetos, uma funcionalidade é realizada através de uma rede de objetos interconectados, colaborando entre si. Objetos encapsulam dados e comportamento. O posicionamento correto do comportamento na rede de objetos é um dos principais problemas a serem enfrentados durante o projeto da lógica de negócio.

Neste contexto, pode-se observar que a lógica de negócio é, na verdade, composta por dois tipos de lógica: a lógica de domínio do problema, que tem a ver puramente com as classes previamente identificadas na fase de análise; e lógica da aplicação, que se refere às funcionalidades descritas pelos casos de uso. Neste contexto, um desafio a mais se coloca: que classes vão comportar as funcionalidades descritas pelos casos de uso (lógica de aplicação)? Duas formas básicas são comumente adotadas:

- Distribuir as responsabilidades para a execução dos casos de uso ao longo dos objetos do domínio do problema: essa abordagem é refletida no padrão Modelo de Domínio e considera que as funcionalidades relativas aos casos de uso do sistema (lógica de aplicação) estarão distribuídas nas classes previamente identificadas na fase de análise.

- Considerar que a lógica de negócio é, na verdade, composta por dois componentes: o Componente de Domínio do Problema, que tem a ver puramente com as classes previamente identificadas na fase de análise e que trata apenas da lógica de domínio; e o Componente de Gerência de Tarefas, que se refere às funcionalidades descritas pelos casos de uso e trata, portanto, da lógica de aplicação. Esta segunda opção é a essência do padrão Camada de Serviço.

Seja qual for a opção escolhida, para apoiar a definição dos métodos das classes, pode ser útil elaborar diagramas de interação.

Este capítulo discute o projeto da camada de lógica de negócio. A Seção 4.1 discute brevemente os diagramas de interação, dando destaque aos diagramas de sequência, que serão usados posteriormente ao longo do capítulo para apoiar a definição de métodos das classes. A Seção 4.2 apresenta sucintamente os padrões arquitetônicos Modelo de Domínio e Camada de Serviço. A Seção 4.3 trata do projeto da lógica de domínio do problema, discutindo as alterações e refinamentos a serem feitos nos diagramas de classes da análise para incorporar as decisões de projeto. Finalmente, a Seção 4.4 trata do projeto da lógica de aplicação, que lida com a lógica dos casos de uso.

4.1 – Diagramas de Interação

Os diagramas de interação da Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) (BOOCH; RUMBAUGH; JACOBSON, 2006) são utilizados para modelar aspectos dinâmicos dos sistemas, com ênfase na distribuição de responsabilidades entre classes e no fluxo de controle ao longo das classes.

A modelagem de aspectos dinâmicos de sistemas pode ser feita construindo-se roteiros de cenários (p.ex., fluxos de eventos de casos de uso, fragmentos deles ou operações) que envolvem a interação entre certos objetos de interesse e as mensagens trocadas entre eles. Na UML, a modelagem desses roteiros é feita com o uso dos diagramas de interação (BOOCH; RUMBAUGH; JACOBSON, 2006).

Tipicamente, um diagrama de interação ilustra o comportamento de um grupo de objetos colaborando para a realização de um dado caso de uso, mostrando um número de instâncias concretas ou prototípicas de classes e as mensagens que são trocadas entre elas no contexto do caso de uso. Há dois tipos de diagramas de interação: diagramas de sequência e diagramas de comunicação. Os diagramas de sequência dão ênfase à ordenação temporal das mensagens, enquanto os diagramas de comunicação dão ênfase à organização estrutural dos objetos. Esses dois tipos de diagramas são semanticamente equivalentes. Ou seja, é possível converter um no outro sem perda de informação (BOOCH; RUMBAUGH; JACOBSON, 2006). Neste texto, são abordados somente os diagramas de sequência, discutidos a seguir.

4.1.1 - Diagramas de Sequência

Um diagrama de sequência é um diagrama de interação que dá ênfase à ordenação temporal das mensagens. Ele é organizado ao longo de dois eixos. Os objetos que participam da interação são colocados no eixo horizontal, na parte superior do diagrama. O objeto que inicia a interação é colocado à esquerda e os objetos

subordinados vão aparecendo à direita. Quando um diagrama de sequência modela um cenário de execução de um caso de uso iniciado por um ator, tipicamente uma instância desse ator é o objeto mais à esquerda no diagrama. As mensagens que os objetos enviam e recebem são colocadas ao longo do eixo vertical, na ordem cronológica de envio, de cima para baixo. Isso proporciona ao leitor uma clara identificação do fluxo de controle ao longo do tempo. A Figura 4.1 mostra um exemplo de diagrama de sequência, ilustrando seus principais elementos de modelagem.

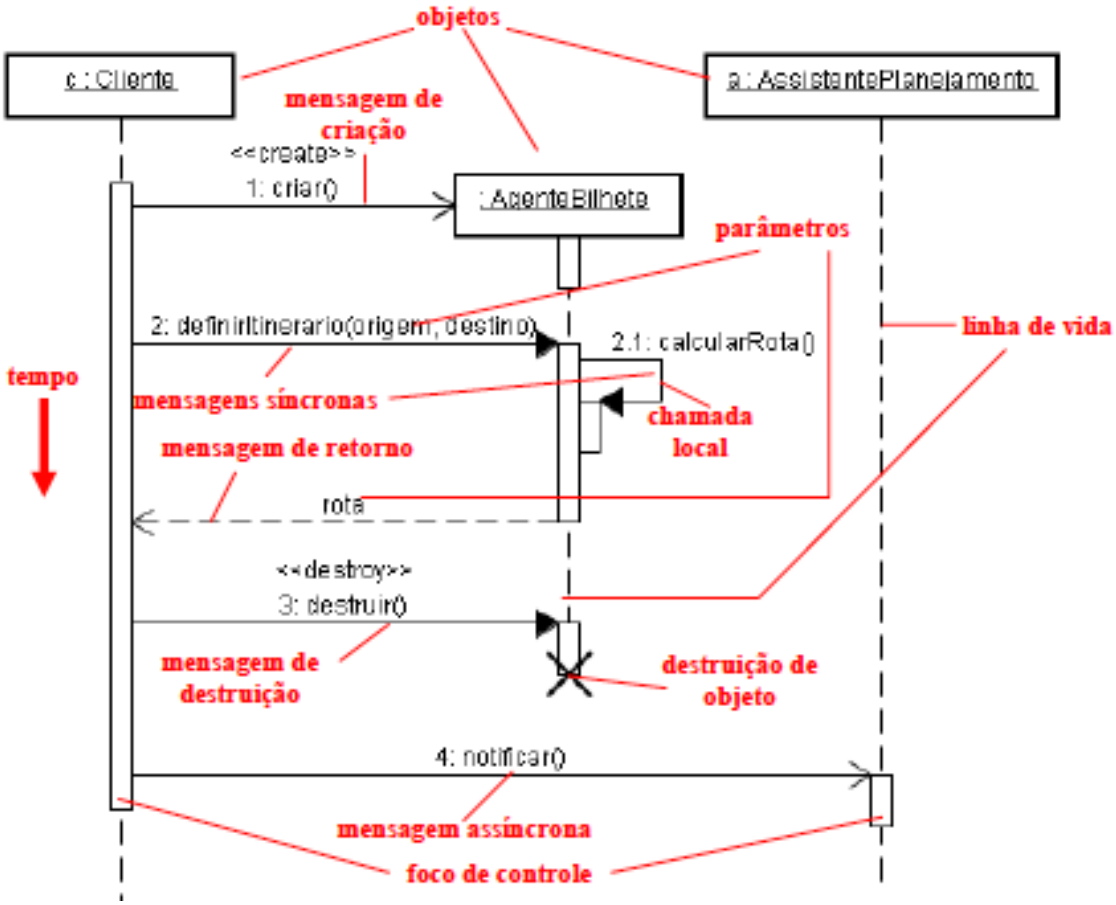


Figura 4.1 – Diagrama de Sequência: Elementos de Modelagem (adaptado de (BOOCH; RUMBAUGH; JACOBSON, 2006)).

Como ilustra a figura acima, um objeto é mostrado como um retângulo com uma linha vertical pontilhada anexada. Essa linha é chamada de *linha de vida* do objeto e representa a existência do objeto durante a interação. Cada mensagem é representada por uma seta direcionada entre as linhas de vida dos objetos emissor e receptor da mensagem. A ordem temporal na qual as mensagens são trocadas é mostrada de cima para baixo. Opcionalmente, números antes das mensagens são usados para indicar a sua ordem. Cada mensagem é rotulada com pelo menos o nome da mensagem. Adicionalmente, pode incluir também parâmetros e alguma informação de controle, tal como uma condição de guarda ([condição]), indicando que a mensagem só é enviada se a condição for verdadeira.

Há diferentes tipos de mensagens (BOOCH; RUMBAUGH; JACOBSON, 2006):

- Uma *mensagem de criação* indica que o objeto receptor está sendo criado naquele momento. Assim, ao contrário dos demais objetos, ele aparece nivelado na altura do envio da mensagem e não na parte superior do diagrama.
- Uma *mensagem síncrona de chamada* invoca uma operação no objeto receptor, passando o controle para esse objeto. Um objeto pode enviar uma mensagem para ele mesmo, resultando em uma chamada local de uma operação.
- Uma *mensagem assíncrona* envia um sinal para o objeto receptor, mas o objeto emissor continua sua própria execução. O objeto receptor recebe o sinal e decide de forma independente o que fazer. A representação de mensagens assíncronas é ligeiramente diferente da representação de mensagens síncronas. A primeira tem uma seta fina, enquanto a segunda tem uma seta grossa.
- Uma *mensagem de retorno* indica uma resposta a uma mensagem síncrona (retorno de chamada) e não uma nova mensagem. Para diferenciar, mensagens de retorno são simbolizadas por uma linha tracejada com uma seta fina. A mensagem de retorno pode ser omitida, já que há um retorno implícito após qualquer chamada. Contudo, muitas vezes é útil mostrar os valores de retorno.
- Por fim, uma *mensagem de destruição* elimina um objeto, deixando o mesmo de existir. Assim, mensagens de destruição são acompanhadas do símbolo de destruição de objeto.

O foco de controle mostra o período durante o qual um objeto está desempenhando uma ação, diretamente ou por meio de um procedimento subordinado. Ele é mostrado como um retângulo estreito sobre a linha de vida do objeto. A parte superior do retângulo é alinhada com o início da ação; a parte inferior é alinhada com a sua conclusão e pode ser delimitada por uma mensagem de retorno (BOOCH; RUMBAUGH; JACOBSON, 2006).

Durante a modelagem do comportamento de um sistema, muitas vezes é importante mostrar fluxos condicionais, laços e a execução concorrente de sequências. Para modelar essas situações, operadores de controle podem ser utilizados. Um operador de controle é apresentado como uma região retangular no diagrama de sequência, contendo um rótulo no canto superior esquerdo, informando o tipo de operador de controle. Os tipos mais comuns são (BOOCH; RUMBAUGH; JACOBSON, 2006):

- Execução opcional (rótulo *opt*): o corpo do operador de controle é executado somente se a condição de guarda na entrada do operador for verdadeira.
- Execução alternativa (rótulo *alt*): o corpo do operador de controle é dividido em sub-regiões por linhas horizontais tracejadas, sendo que cada sub-região representa um ramo condicional e é executada somente se sua condição de guarda for verdadeira.

- Execução paralela (rótulo *par*): o corpo do operador de controle é dividido em sub-regiões por linhas horizontais tracejadas, sendo que cada sub-região representa uma computação paralela (concorrente).
- Execução iterativa (rótulo *loop*): o corpo do operador de controle é executado repetidamente enquanto a condição de guarda na entrada do operador for verdadeira. A condição de guarda é testada no início de cada iteração.

Mensagens trocadas entre objetos em um diagrama de sequência devem ser mapeadas como operações na classe do objeto receptor da mensagem. Assim, toda mensagem que chega a um objeto aponta para a necessidade de um método com mesma assinatura na classe desse objeto, contribuindo de maneira decisiva para a identificação de métodos nas classes.

4.2 – Padrões Arquitetônicos para o Projeto da Lógica de Negócio

Um importante aspecto do projeto da Camada de Lógica de Negócio diz respeito à organização das classes e distribuição de responsabilidades entre elas, o que vai definir, em última instância, os métodos de cada classe dessa camada.

Os diagramas de classes da fase de análise são modelos conceituais estruturais e, como tal, trazem importantes informações sobre os objetos que abstraem entidades do mundo real, os quais são elementos chave da lógica de negócio. Os modelos de caso de uso descrevem as funcionalidades que o sistema deverá prover e, portanto, têm informações igualmente relevantes sobre a lógica de negócio, mas sob um prisma funcional. Em essência, os casos de uso deverão dar origem a operações e consultas do sistema, que geralmente vão estar disponíveis para acesso a partir da interface do sistema com o mundo externo. Assim, pode-se dividir a lógica de negócio em dois tipos principais: a **lógica de domínio do problema**, que tem a ver puramente com as classes previamente identificadas na fase de análise; e **lógica de aplicação**, que se refere às funcionalidades descritas pelos casos de uso. Como consequência, é importante definir onde posicionar os métodos que vão cumprir esses diferentes tipos de responsabilidades em um modelo de classes de projeto.

Fowler (2003) apresenta alguns padrões para organizar a lógica de negócio, a saber: Script de Transação (*Transaction Script*), Modelo de Domínio (*Domain Model*), Módulo de Tabelas (*Table Module*) e Camada de Serviço (*Service Layer*). Neste texto, cujo foco é o desenvolvimento de Sistemas de Informação Orientados a Objetos, merecem destaque os padrões Modelo de Domínio e Camada de Serviço.

Em ambos os casos, é útil ter uma classe que vai representar o sistema como um todo, dita uma classe controladora do sistema[3] (WAZLAWICK, 2004). Essa classe servirá como uma fachada para receber requisições da interface com o usuário e direcioná-las para os objetos capazes de tratá-las. Contudo, dependendo do padrão arquitetônico adotado, esse tratamento das requisições será ligeiramente diferente.

O padrão **Modelo de Domínio** preconiza que o modelo de classes do domínio incorpore tanto dados quanto comportamento. As classes de domínio identificadas na

[3] Uma classe controladora de sistema é um controlador no sentido usado no padrão Modelo-Visão-Controlador (MVC) discutido no Capítulo 5.

Análise de Requisitos são as responsáveis por tratar tanto a lógica de domínio quanto a lógica de aplicação (casos de uso). Assim, dados e processos são combinados (FOWLER, 2003) e pode-se dizer que o padrão Modelo de Domínio captura a essência da orientação a objetos.

De forma resumida, no padrão Modelo de Domínio, tanto a lógica de domínio quanto a lógica de aplicação são atribuídas às classes do domínio do problema, sendo que diferentes classes podem ter partes da lógica de aplicação que são relevantes a elas. Vale ressaltar que esta pode não ser uma boa solução sob uma perspectiva de manutenibilidade, já que uma alteração em uma funcionalidade (caso de uso) pode afetar diversas classes e, assim, pode ser difícil de ser incorporada.

Já na abordagem do padrão **Camada de Serviço**, um conjunto de classes gerenciadoras de casos de uso[4] fica responsável por tratar a lógica de aplicação, controlando o fluxo de eventos dentro do caso de uso. Grande parte da lógica de negócio ainda fica a cargo das classes do domínio do problema, cabendo aos gerenciadores de casos de uso apenas centralizar o controle sobre a execução do caso de uso. Assim, a camada de serviço é construída, de fato, sobre a camada de domínio.

A motivação principal para esse padrão é o fato de algumas funcionalidades (casos de uso) não serem facilmente distribuídas nas classes de domínio do problema, principalmente aquelas que operam sobre várias classes. Assim, criam-se classes gerenciadoras de casos de uso (gerenciadores ou coordenadores de tarefas), responsáveis pela realização de tarefas (casos de uso). Tipicamente, esses gerenciadores de tarefas agem como aglutinadores, unindo outros objetos para dar forma a um caso de uso. Consequentemente, gerenciadores de tarefa são normalmente encontrados diretamente a partir dos casos de uso. Os tipos de funcionalidade tipicamente atribuídos a gerenciadores de tarefa incluem comportamento relacionado a transações e sequências de controle específicas de um caso de uso.

O padrão Camada de Serviço (FOWLER, 2003) define uma fronteira da lógica de negócio usando uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena as respostas do sistema para cada uma das operações. A camada de serviço encapsula a lógica de negócio do sistema, controlando transações e coordenando respostas na implementação de suas operações. A argumentação em favor desse padrão é que misturar lógica de domínio e lógica de aplicação nas mesmas classes torna essas classes menos reutilizáveis transversalmente em diferentes aplicações, bem como pode dificultar a manutenção da lógica de aplicação, uma vez que a lógica dos casos de uso não é diretamente perceptível em nenhuma classe.

A identificação das operações necessárias na camada de serviço é fortemente apoiada nos casos de uso do sistema. Uma opção é considerar que cada caso de uso vai dar origem a uma classe de serviços, dita classe gerenciadora de caso de uso. Por exemplo, um caso de uso de cadastro, envolvendo funcionalidades de inclusão, alteração, consulta e exclusão, pode ser mapeado em uma classe com operações para tratar essas funcionalidades. Contudo, não há uma prescrição clara, apenas heurísticas. Para uma aplicação relativamente pequena, pode ser suficiente ter uma única classe provendo todas as operações. Para sistemas maiores, compostos de vários subsistemas, pode-se ter uma classe por subsistema (FOWLER, 2003).

[4] Classes gerenciadoras de caso de uso são classes que centralizam as interações no contexto de casos de uso específicos e não são consideradas controladores no sentido usado no padrão MVC.

Não se deve confundir uma abordagem de Camada de Serviços com uma abordagem de Modelo de Domínio Anêmico (*Anemic Domain Model*) (FOWLER, 2003a), na qual os objetos do domínio do problema apresentam comportamento vazio. Nessa abordagem, as classes de análise são divididas em classes de dados (ditos objetos de valor – *Value Objects* – VOs) e classes de lógica (ditos objetos de negócio – *Business Objects* – BOs), que separam o comportamento do estado dos objetos. Os VOs têm apenas o comportamento básico para alterar e manipular seu estado (métodos construtor e destrutor e métodos *get* e *set*). Os BOs ficam com os outros comportamentos, tais como cálculos, validações e regras de negócio. De maneira geral, a abordagem de Modelo de Domínio Anêmico deve ser evitada, sendo, por isso, considerada um anti-padrão. Essa abordagem tem diversos problemas. Primeiro, não há encapsulamento, já que dificilmente um VO vai ser utilizado apenas por um BO. Segundo, a vantagem de se ter um modelo de domínio rico é anulada, já que a proximidade com as abstrações do mundo real é destruída. No mundo real não existe lógica de um lado e dados de outro, mas sim ambos combinados em um mesmo conceito. Outro problema é a manutenção de um sistema construído desta maneira. Os BOs possuem um acoplamento muito alto com os VOs e a mudança em um afeta drasticamente o outro (FRAGMENTAL, 2007).

Uma maneira de se implementar o padrão Camada de Serviço consiste em ter uma ou mais classes gerenciadoras de casos de uso (veja discussão na Seção 4.4), as quais encapsulam a lógica da aplicação. Para realizar um caso de uso, a classe gerenciadora de caso de uso invoca métodos da camada de domínio do problema, tal como ocorre no padrão Modelo de Domínio. A diferença entre os dois padrões reside, neste caso, no fato da classe gerenciadora de tarefa centralizar o controle do caso de uso, evitando delegar responsabilidades a classes que não têm como tratá-las.

4.3 – Projeto da Lógica de Domínio do Problema

No projeto orientado a objetos, os modelos conceituais estruturais (diagramas de classes) produzidos na fase de análise estão diretamente relacionados à lógica de domínio do problema e podem ser incorporados a um Componente de Domínio do Problema (CDP). Como ponto de partida para a elaboração do diagrama de classes do CDP, deve-se utilizar uma cópia do diagrama de classes de análise. A partir dessa cópia, alterações serão feitas para incorporar as decisões de projeto. Vale ressaltar que o trabalho deve ser efetuado em uma cópia, mantendo o modelo conceitual original intacto para efeito de documentação e manutenção do sistema.

Para se poder conduzir o projeto do CDP de maneira satisfatória, algumas informações acerca da plataforma de implementação são essenciais, dentre elas a linguagem de programação e o mecanismo de persistência de objetos a serem adotados. Além disso, informações relativas aos requisitos não funcionais e suas prioridades são igualmente vitais para se tomar decisões importantes relativas ao projeto do CDP.

As alterações básicas a serem incorporadas em um diagrama de classes do CDP são:

- *Alteração de informações relativas a tipos de dados de atributos*: Na fase de análise, é comum especificar tipos de dados gerais para atributos. Na fase de projeto, contudo, os atributos devem ser mapeados em variáveis de um tipo de dados provido pela linguagem de implementação. Além disso, muitas vezes, atributos podem dar

origem a novas classes (ou tipos de dados enumerados) para atender a requisitos de qualidade, tais como usabilidade, manutenibilidade e reusabilidade.

- *Adição de navegabilidades nas associações*: Na fase de análise, as associações são consideradas navegáveis em todas as direções. O mesmo não ocorre na fase de projeto, quando se pode definir que certas associações são navegáveis apenas em um sentido, indicando que apenas um dos objetos terá uma referência para o outro (ou para coleções de objetos, no caso de associações com multiplicidade *). Esta decisão deve ser feita, sobretudo, considerando os usos frequentes das funcionalidades do sistema e requisitos não funcionais, com destaque para desempenho, mesmo que ele não seja um atributo condutor da arquitetura. Além disso, essa definição pode ser influenciada, dentre outros, pelo mecanismo de persistência de objetos a ser adotado.

- *Adição de informações de visibilidade de atributos e associações*: De maneira geral, na fase de análise não se especifica a visibilidade de atributos e associações. Como discutido anteriormente, as associações são tipicamente consideradas navegáveis e visíveis em todas as direções. Já os atributos são considerados públicos. Porém essa não é uma boa estratégia para a fase de projeto. Conforme discutido no Capítulo 2, ocultar informações é um importante princípio de projeto. Assim, atributos só devem poder ser acessados pela própria classe ou por suas subclasses. Uma classe não deve ter acesso aos atributos de uma classe a ela associada. Como consequência disso, cada classe deve ter operações para consultar (tipicamente nomeadas como *get*) e atribuir / alterar valor (normalmente nomeada como *set*) de cada um de seus atributos e associações navegáveis. Essas operações, contudo, não precisam ser mostradas no diagrama de classes, visto que elas podem ser deduzidas pela própria existência dos atributos e associações (WAZLAWICK, 2004).

- *Adição de métodos às classes*: Muitas vezes, as classes de um diagrama de classes de análise não têm informação acerca das suas operações. Mesmo quando elas têm essa informação, ela pode ser insuficiente, tendo em vista que é no projeto que se decide efetivamente como abordar a distribuição de responsabilidades para a realização de funcionalidades. Assim, durante o projeto do CDP atenção especial deve ser dada à definição de métodos nas classes. Para apoiar esta etapa, diagramas de sequência podem ser utilizados para modelar a interação entre objetos na realização de funcionalidades do sistema. A escolha de um padrão arquitetônico para o projeto da lógica de negócio também tem influência na distribuição de responsabilidades, conforme discutido na Seção 4.2. Vale ressaltar que já se assume que algumas operações, consideradas básicas, existem e, portanto, não precisam ser representadas no diagrama de classes. Essas operações são as operações de criação e destruição de instâncias, além das operações de consulta e atribuição / alteração de valores de atributos e associações, conforme discutido no item anterior. No diagrama de classes devem aparecer apenas os métodos que não podem ser deduzidos (WAZLAWICK, 2004).

- *Eliminação de classes associativas*: Caso o diagrama de classes de análise contenha classes associativas, recomenda-se substituí-las por classes normais, criando novas associações. Isso é importante, pois as linguagens de programação não têm construtores capazes de implementar diretamente esses elementos de modelo.

Além das alterações básicas a que todos os diagramas de classes do CDP estarão sujeitos, outras fontes de alteração incluem:

- *Reutilizar projetos anteriores e classes já programadas*: é importante que na fase de projeto seja levada em conta a possibilidade de se reutilizar classes já projetadas e programadas (desenvolvimento com reúso), bem como a possibilidade de se desenvolver classes para reutilização futura (desenvolvimento para reúso). Tipicamente, ajustes feitos para incorporar tais classes envolvem alterações na estrutura do modelo, podendo atingir hierarquias de generalização-especialização, de modo a tratar as classes do domínio do problema como subclasses de classes de bibliotecas pré-existentes. Também ao incorporar um padrão de projeto (*design pattern*), muito provavelmente a estrutura do diagrama de classes de projeto sofrerá alterações.

- *Ajustar hierarquias de generalização-especialização*: muitas vezes, as hierarquias de herança da fase de análise não são adequadas para a fase de projeto. Dentre os fatores que podem provocar mudanças na hierarquia de herança destacam-se:

    - *Ajustar hierarquias de generalização-especialização para adequação ao mecanismo de herança suportado pela linguagem de programação a ser usada na implementação*: se, por exemplo, o modelo de análise envolve herança múltipla e a linguagem de implementação não oferece tal recurso, alterações no modelo são necessárias. Quando se estiver avaliando hierarquias de classes para eliminar relações de herança múltipla, deve-se considerar se uma abordagem de delegação não é mais adequada do que o estabelecimento de uma relação de herança.

    - *Ajustar hierarquias de generalização-especialização para aproveitar oportunidades decorrentes da definição de operações*: as definições de operações nas classes podem também conduzir a alterações na hierarquia de generalização-especialização. De fato, pode ser que durante a fase de análise não sejam exploradas todas as oportunidades de herança. É útil reexaminar o diagrama de projeto procurando observar se determinadas classes têm comportamento parcialmente comum, abrindo-se espaço para a criação de uma superclasse encapsulando as propriedades (atributos e operações) compartilhadas, abstraindo o comportamento comum. A criação de interfaces também pode ser interessante para garantir uma separação da interface contratual de uma classe de sua implementação. Conforme discutido anteriormente, a reutilização pode ser um fator motivador para a criação de novas superclasses. Contudo, deve-se tomar cuidado com a refatoração da hierarquia de classes. Criar uma nova classe para abstrair comportamento comum somente se justifica quando há, de fato, uma relação de subtipo entre as classes existentes e a nova classe criada; ou seja, pode-se dizer que a subclasse é semanticamente um subtipo da superclasse. Não se deve alterar a hierarquia de classes simplesmente para herdar uma parte do comportamento, quando as classes envolvidas não guardam entre si uma relação efetivamente de subtipo, em uma abordagem dita herança de implementação (BLAHA; RUMBAUGH, 2006).

- *Ajustar o modelo para melhorar o desempenho*: Visando melhorar o desempenho do sistema, o projetista pode alterar o diagrama de classes do CDP para melhor acomodar os ajustes necessários. Atributos e associações redundantes podem ser adicionados para evitar recomputação, bem como podem ser criadas novas classes para registrar estados intermediários de um processo.

- *Ajustar o modelo para facilitar o projeto de interfaces com o usuário amigáveis*: com o objetivo de incorporar o atributo de qualidade usabilidade, pode ser importante considerar novas classes (ou tipos enumerados de dados) que facilitem a apresentação de listas para seleção do usuário.

- *Ajustar o modelo para incorporar aspectos relacionados à segurança*: táticas como autenticação e autorização requerem novas funcionalidades que, por sua, vez, requerem novas classes do CDP. Em casos como esse, pode ser útil separar as classes relativas a essas funcionalidades em um novo pacote, visando ao reúso.

Além dos ajustes discutidos anteriormente, vários deles relacionados a atributos de qualidade (a saber, reusabilidade, desempenho, usabilidade e segurança), o CDP pode ser alterado, ainda, para comportar outros requisitos não funcionais, tais como testabilidade, confiabilidade etc.

O CDP é um componente obrigatório, tanto quando se adota o padrão Modelo de Domínio quanto quando se adota o padrão Camada de Serviço. No padrão Modelo de Domínio, o CDP é a própria camada de lógica de negócio, tendo em vista que não há classes gerenciadoras de tarefas (gerenciadoras de casos de uso). No caso do padrão Camada de Serviço, além do CDP, a camada de lógica de negócio tem outro componente, o Componente de Gerência de Tarefas.

4.4 – Projeto da Lógica de Aplicação

Conforme discutido anteriormente, dependendo do padrão arquitetônico adotado, a lógica de aplicação é projetada de maneira distinta. No padrão Modelo de Domínio, a lógica de aplicação é distribuída nos objetos do domínio do problema; no padrão Camada de Serviço, um novo componente, o Componente de Gerência de Tarefas (CGT), fica responsável por tratar a lógica de aplicação, controlando os fluxos de eventos dos casos de uso. Em ambos os casos, o projeto da lógica de aplicação está intimamente ligado ao modelo de casos de uso.

4.4.1 – Projeto da Lógica de Aplicação no Padrão Modelo de Domínio

Wazlawick (2004) propõe uma maneira interessante de projetar a lógica de aplicação usando o padrão Modelo de Domínio. Essa abordagem consiste em considerar uma classe controladora do sistema no modelo de domínio do problema, relacionando-a a todos os conceitos independentes, correspondentes aos cadastros do sistema, ou seja, os elementos a serem cadastrados para a operação do sistema[5]. O fluxo de controle sempre inicia em uma instância da classe controladora. Essa classe recebe as requisições da interface e, para tratá-las, o controlador invoca métodos que tratam a lógica de aplicação posicionados nas classes do domínio do problema, em uma abordagem chamada de delegação.

A delegação consiste em capturar uma operação que trata uma mensagem em um objeto e reenviar essa mensagem para um outro objeto associado ao primeiro que seja

[5] Vale ressaltar que, embora a classe controladora de sistema seja modelada no diagrama de classes do domínio do problema, ela corresponde, de fato, ao pacote de interface com o usuário, tendo em vista que ela é um controlador no sentido adotado pelo padrão MVC.

capaz de tratar a requisição. Neste caso, a execução é delegada para objetos do domínio do problema, procurando efetuar uma cadeia de delegação sobre as linhas de visibilidade das associações já existentes, até se atingir um objeto capaz de atender à requisição. Novas linhas de visibilidade só devem ser criadas quando isso for estritamente necessário. Deste modo, mantém-se fraco o acoplamento entre as classes, conforme preconiza o padrão de projeto *Acoplamento Fraco* (LARMAN, 2004).

Assim, partindo-se do controlador do sistema, deve-se identificar qual o objeto a ele relacionado que melhor pode tratar a requisição e delegar essa responsabilidade a ele. Esse objeto, por sua vez, vai colaborar com outros objetos para a realização da funcionalidade.

De maneira geral, um objeto só deve mandar mensagens para outros objetos que estejam a ele associados ou que foram passados como parâmetro no método que está sendo executado. Nessa abordagem, deve-se evitar obter um objeto como retorno de um método (visibilidade local) para mandar mensagens a ele (WAZLAWICK, 2004), conforme apontado pelo padrão "não fale com estranhos" (LARMAN, 2004).

Seja o exemplo de uma locadora de vídeo, cujo modelo de projeto do domínio do problema é parcialmente apresentado na Figura 4.2.

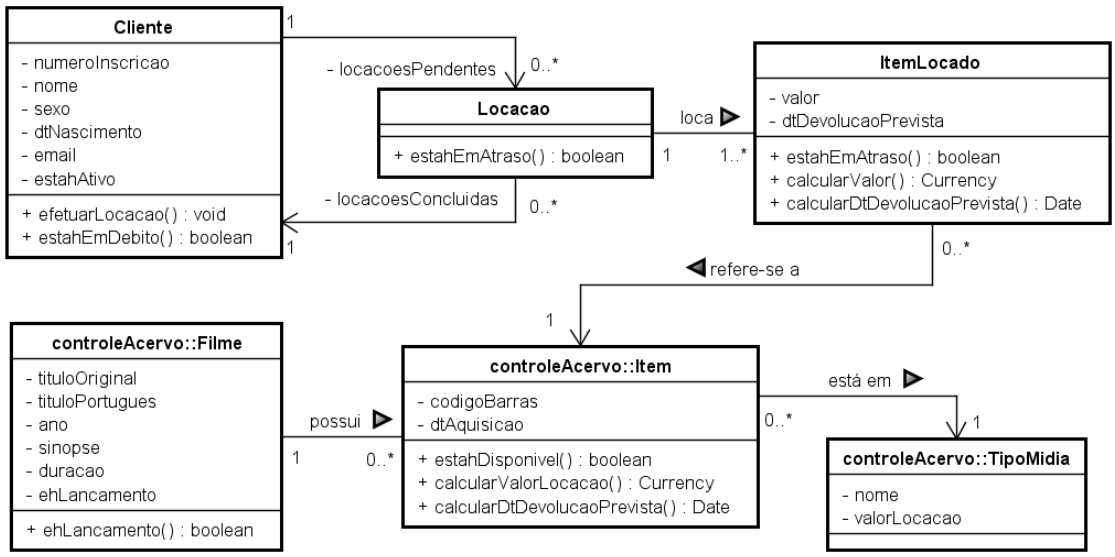


Figura 4.2 –Diagrama de Classes parcial do CDP de um sistema de videolocadora.

Neste exemplo, o modelo de casos de uso possui o caso de uso *Efetuar Locação*, cuja descrição do fluxo de eventos principal é a seguinte:

1. O atendente informa o cliente que deseja efetuar a locação.
2. Para cada item a ser locado
   2.1 - O atendente informa o item a ser locado.
   2.2 - O sistema calcula o valor de locação do item. O valor da locação de um item é dado pelo tipo de mídia do item. Cada tipo de mídia tem um valor de locação associado. Um acréscimo de 50% do valor da locação do tipo de mídia deve ser aplicado no caso do filme do item ser um lançamento.
   2.3 - O sistema calcula a data de devolução prevista. A data de devolução prevista é definida em função do filme do item ser lançamento ou não.

Lançamentos têm prazo de um dia; filmes do catálogo têm três dias de prazo.

2.4 - Caso deseje, o atendente poderá alterar a data de devolução prevista e o valor de locação de um item locado.

2.5 - O sistema adiciona o valor de locação do item locado ao valor da locação.

3. A locação é registrada com a data corrente como data de locação.

Este caso de uso possui, ainda, um fluxo alternativo para o passo 1, a saber:

- Cliente está em débito: Uma mensagem de erro é exibida, informando que há itens locados pelo cliente em atraso e apresentando dados desses itens. O fluxo de eventos é abortado.

No diagrama da Figura 4.2, apenas as classes *Cliente*, *Filme* e *TipoMidia* são conceitos independentes. Assim, a classe controladora de sistema (*Videolocadora*) deve ser relacionada a essas classes e deve possuir um método *efetuarLocacao* que simplesmente vai delegar a responsabilidade de realizar o caso de uso *Efetuar Locação* para uma delas. Dentre essas três classes, *Cliente* aparece como a opção natural para comportar o método *efetuarLocacao*, uma vez que nesse caso de uso são informados o cliente e os itens a serem locados.

O fluxo de controle inicia na instância da classe controladora *Videolocadora*, a qual recebe a requisição da interface para efetuar uma locação, informando o cliente e os itens a serem locados. Para tratá-la, o controlador invoca o método *efetuarLocacao* da classe Cliente, que efetivamente realiza a lógica de aplicação. Para tal, o objeto *Cliente* colabora com outros objetos para a realização da funcionalidade, como mostra o diagrama de sequência da Figura 4.3. Vale ressaltar que as colaborações nesse diagrama ocorrem sobre as linhas de visibilidade das associações já existentes (veja Figura 4.2).

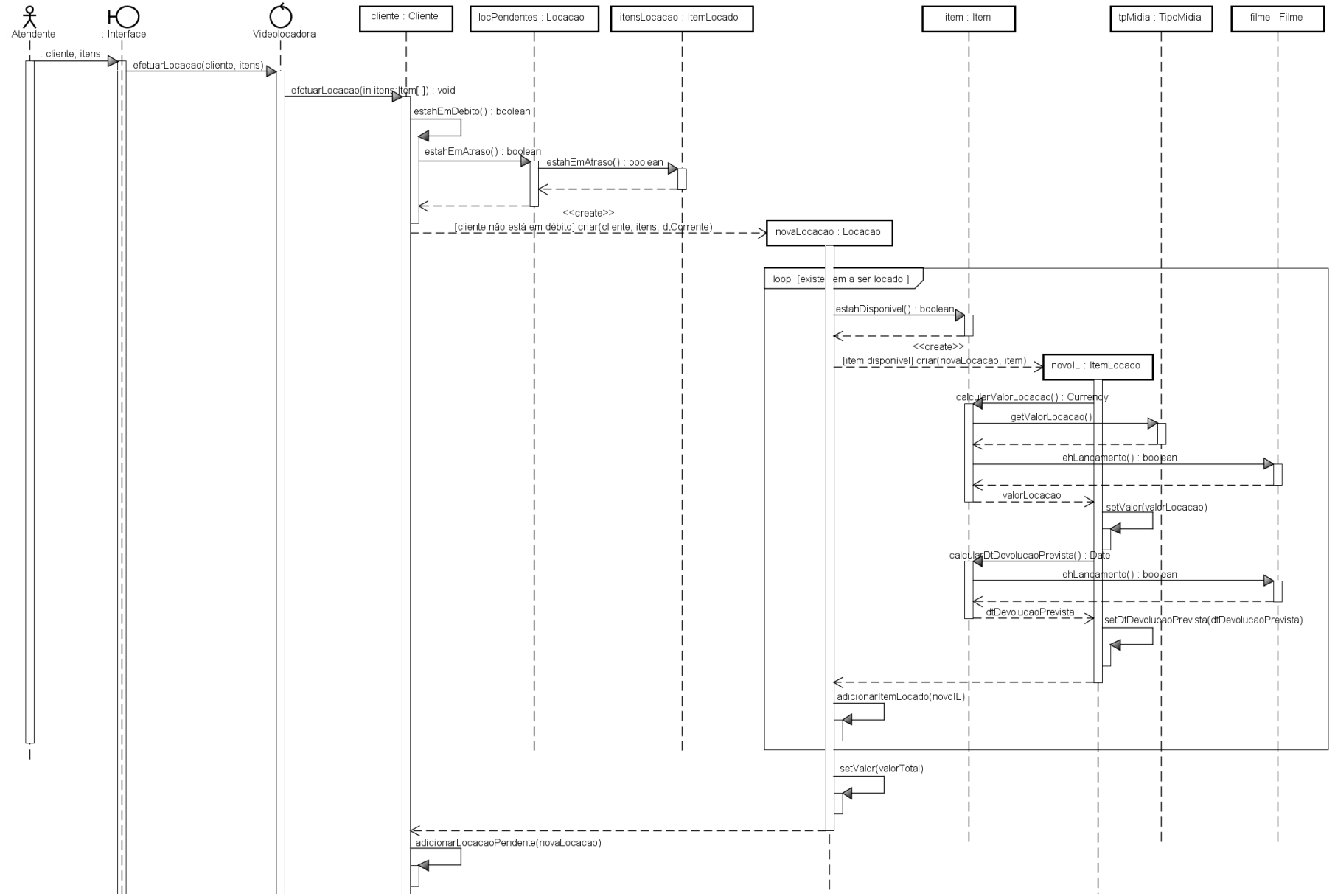


Figura 4.3 – Diagrama de Sequência – Caso de Uso Efetuar Locação – Padrão Modelo de Domínio

4.4.2 – Projeto da Lógica de Aplicação no Padrão Camada de Serviço

No padrão Camada de Serviço, um novo componente, o Componente de Gerência de Tarefas (CGT), fica responsável por tratar a lógica de aplicação, controlando os fluxos de eventos dos casos de uso. Esse componente define classes gerenciadoras de tarefas (ou gerenciadoras de casos de uso).

Quando o padrão Camada de Serviço é aplicado, a camada de lógica de negócio é dividida em duas outras camadas. A Camada de Domínio do Problema (CDP) e, sobre ela, a Camada de Gerência de Tarefa.

Em um esboço preliminar, pode-se atribuir um gerenciador de tarefas para cada caso de uso, sendo que os seus fluxos de eventos principais dão origem a operações da classe que representa o caso de uso (classe gerenciadora de caso de uso ou classe de aplicação). Deste modo, a manutenibilidade pode ser facilitada, uma vez que, detectado um problema em um caso de uso, é fácil identificar a classe que trata do mesmo.

Uma solução diametralmente oposta consiste em definir uma única classe de aplicação para todo o sistema. Neste caso, os fluxos de eventos de todos os casos de uso dão origem a operações dessa classe. Fica evidente que, exceto para sistemas muito pequenos, essa classe tende a ter muitas operações e será extremamente complexa e, portanto, essa opção tende a não ser prática.

Normalmente, uma solução intermediária entre as duas anteriormente apresentadas conduz a melhores resultados. Nessa abordagem, casos de uso complexos são designados a classes de gerência de tarefas específicas. Casos de uso mais simples e de alguma forma relacionados são tratados por uma mesma classe de gerência de tarefas.

No caso da aplicação do padrão Camada de Serviço, não há restrições de que a classe gerenciadora de tarefa obtenha um objeto como retorno de um método (visibilidade local) e mande mensagens a ele. Assim, a classe gerenciadora de tarefa pode ter referência a diversos objetos do domínio, tipicamente aqueles envolvidos na realização do caso de uso correspondente. A Figura 4.4 mostra o exemplo discutido anteriormente, projetado agora usando o padrão Camada de Serviço.

Da mesma forma que no Padrão Modelo de Domínio, o fluxo de controle inicia na instância da classe controladora *Videolocadora*, a qual recebe a requisição da interface para efetuar uma locação, informando o cliente e os itens a serem locados. Para tratá-la, o controlador invoca o método *efetuarLocacao* da classe controladora de caso de uso AplLocacao, que trata da lógica de aplicação relacionada ao caso de uso Efetuar Locação. Essa classe controla a sequência do caso de uso, colaborando com os objetos do Componente de Domínio do Problema para a realização da funcionalidade, como mostra o diagrama de sequência da Figura 4.4.

O conjunto de tarefas a serem apoiadas pelo sistema oferece um recurso bastante útil para a definição das janelas, menus e outros componentes de interface com o usuário necessários para cada uma dessas tarefas. Assim os projetos dos componentes de gerência de tarefa e de apresentação (veja Capítulo 5) estão bastante relacionados e devem ser realizados conjuntamente, uma vez que, muitas vezes, são as tarefas que determinam a necessidade de elementos de interface com o usuário para sua execução.

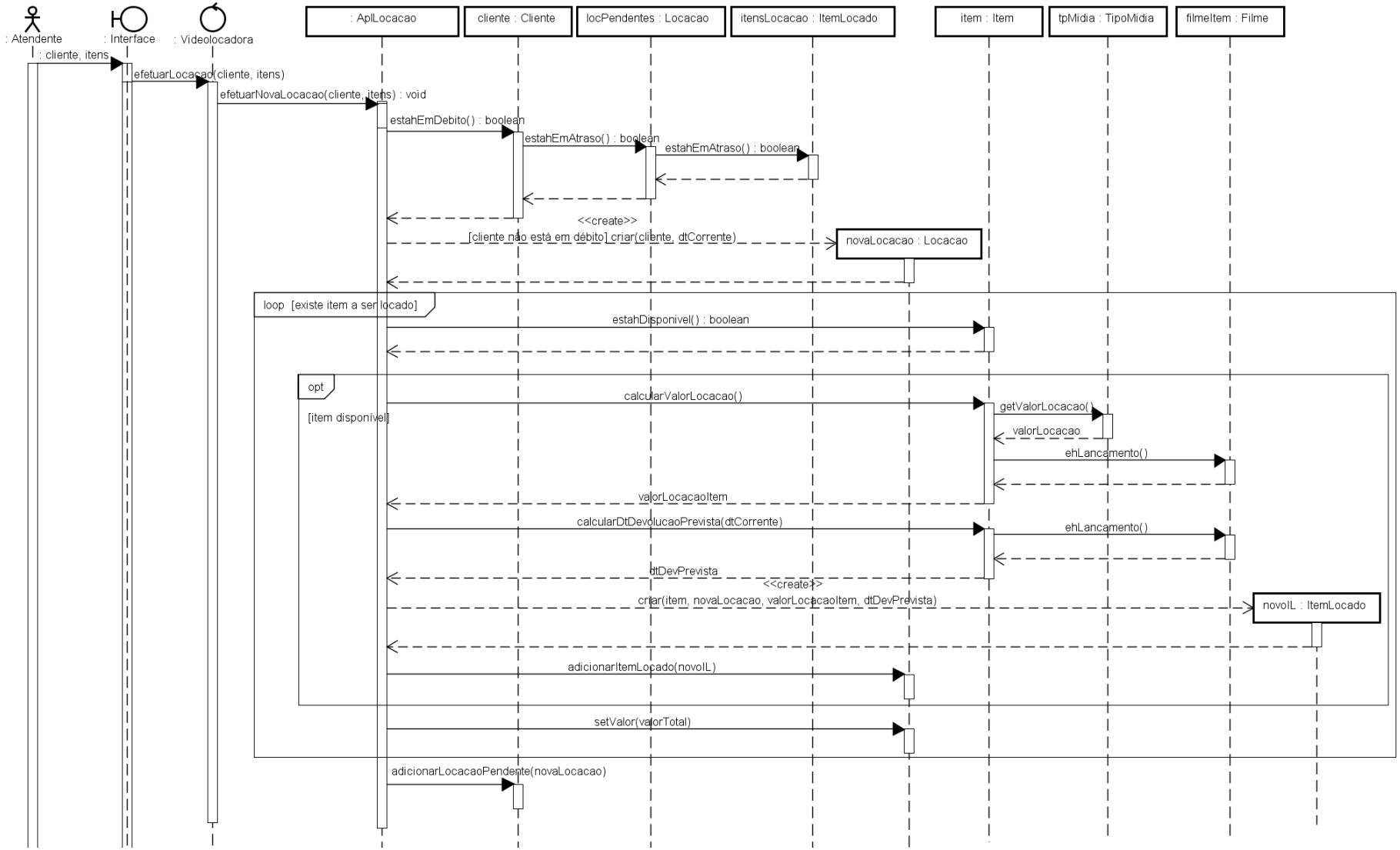


Figura 4.4 – Diagrama de Sequência – Caso de Uso Efetuar Locação – Padrão Camada de Serviço

Capítulo 5 – Projeto da Interação Humano-Computador

Sistemas, em especial os sistemas de informação, são desenvolvidos para serem utilizados por pessoas. Assim, um aspecto fundamental no projeto de sistemas é a interface com o usuário (IU). O projeto da IU estabelece uma forma de comunicação entre as pessoas e o sistema computacional. A IU define como um usuário comandará o sistema e como o sistema apresentará as informações a ele.

Um dos princípios fundamentais para um bom projeto de software é a separação da apresentação (camada de IU) da lógica de negócio (camada de LN). Essa separação é importante por diversas razões, dentre elas (FOWLER, 2003):

- O projeto de IU e o projeto da LN tratam de diferentes preocupações. No primeiro, o foco está nos mecanismos de interação e em como dispor uma boa IU. O segundo concentra-se em conceitos e processos do negócio.
- Usuários podem querer ver as mesmas informações de diferentes maneiras (p.ex., usando diferentes interfaces, tais como interfaces ricas de sistemas *desktop*, interfaces de aplicações *Web* tradicionais, interfaces de linha de comando etc.). Neste contexto, separar a IU da LN permite o desenvolvimento de múltiplas apresentações.
- Objetos não visuais são geralmente mais fáceis de testar do que objetos visuais. Ao separar objetos da LN de objetos de IU, é possível testar os primeiros sem envolver os últimos.

Dada a importância dessa separação, é importante usar algum padrão arquitetônico que trabalhe essa separação, tal como o padrão Modelo-Visão-Controlador (MVC) (FOWLER, 2003).

A camada de IU envolve dois tipos de funcionalidades:

- Visão: refere-se aos objetos gráficos usados na interação com o usuário;
- Controle de Interação: diz respeito ao controle da lógica da interface, envolvendo a ativação dos objetos gráficos (p.ex., abrir ou fechar uma janela, habilitar ou desabilitar um item de menu etc.) e o disparo de ações.

Este capítulo discute o projeto da camada de interface com o usuário. Deve-se realçar, contudo, que o foco deste texto recai sobre aspectos arquitetônicos do projeto da camada de IU. Apenas considerações de cunho geral são feitas em relação ao projeto da interação humano-computador, tema que, por si só, requer outra disciplina. Assim, a Seção 5.1 apresenta o padrão MVC. A Seção 5.2 dá uma visão geral do processo de projeto de IU. A Seção 5.3 trata do Projeto da Visão, discutindo brevemente táticas relacionadas ao projeto dos objetos gráficos de interação humano-computador. A Seção 5.4 trata do Projeto do Controle de Interação, que lida com a lógica de interface.

Finalmente, a Seção 5.5 discute como alguns padrões de projeto (*design patterns*) podem ser empregados no projeto da IU.

5.1 – O Padrão Modelo-Visão-Controlador

O padrão Modelo-Visão-Controlador (MVC) considera três papéis relacionados à interação humano-computador. O *modelo* refere-se aos objetos que representam alguma informação sobre o negócio e corresponde, de fato, a objetos da camada de Lógica de Negócio. A *visão* refere-se à entrada e à exibição de informações na IU. Qualquer requisição é tratada pelo terceiro papel: o controlador. Este pega a entrada do usuário, envia uma requisição para a camada de lógica de negócio, recebe sua resposta e solicita que a visão se atualize conforme apropriado. Assim, a IU é uma combinação de visão e controlador (FOWLER, 2003). Em outras palavras, elementos da visão representam informações de modelo e as exibem ao usuário, que pode enviar, por meio da visão, requisições ao sistema. Essas requisições são tratadas pelo controlador, que as repassa para classes do modelo. Uma vez alterado o estado dos elementos do modelo, o controlador pode, se apropriado, alterar elementos de visão a serem exibidos ao usuário. Assim, o controlador situa-se entre o modelo e a visão, isolando-os um do outro.

Neste ponto é importante distinguir os controladores do padrão MVC das classes gerenciadoras de caso de uso do Componente de Gerência de Tarefas (*cgt*). Estas últimas representam classes da lógica de negócio (lógica de aplicação) que encapsulam e centralizam o tratamento de casos de uso. Já um controlador do padrão MVC é um controlador de interação, ou seja, ele controla a lógica de interface, abrindo e fechando janelas, habilitando ou desabilitando botões, enviando requisições etc.

O padrão MVC trabalha dois tipos de separação. Primeiro, separa a apresentação (visão) da lógica de negócio (modelo), conforme advogado pelas boas práticas de projeto. Segundo, mantém também separados o controlador e a visão. Essa segunda separação (entre a visão e o controlador) é menos importante que a primeira (entre a visão e a lógica de negócio). Vários sistemas têm um único controlador por visão e, por isso, a separação entre a visão e o controlador muitas vezes não é feita. Em sistemas de interfaces ricas *desktop* ela é muitas vezes desprezada. Contudo, em interfaces Web, essa separação é comum, já que a parte de visão *front end* é naturalmente separada do controlador. De fato, a maioria dos padrões de projeto de interfaces Web é baseada nesse princípio (FOWLER, 2003). A Figura 5.1 mostra um diagrama de pacotes ilustrando o padrão MVC.

A separação entre visão e controlador dá origem a dois tipos de classes que podem ser organizados em dois pacotes na camada de interface com o usuário: o Componente de Interação Humana (*cih*), que é responsável pelas interfaces com o usuário propriamente ditas (janelas, painéis, botões, menus etc.) e representa a visão no modelo MVC; e o Componente de Controle de Interação (*cci*), que é responsável por controlar a interação, recebendo requisições da interface, disparando operações da lógica de negócio e atualizando a visão com base no retorno dessas operações. O *cci* é, portanto, o controlador do modelo MVC.

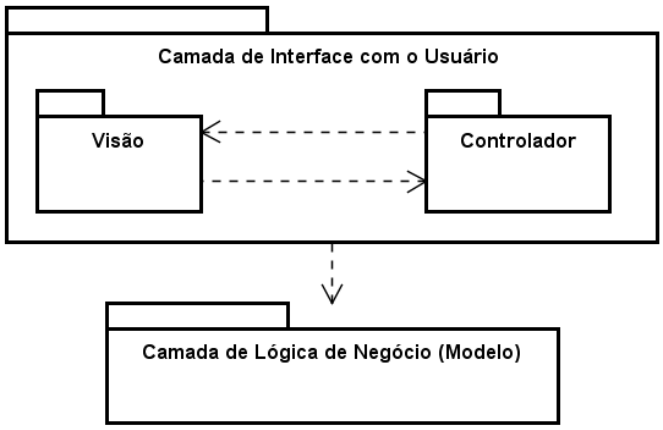


Figura 5.1 – O Padrão MVC.

É importante frisar que, mesmo quando se opta por não fazer a separação física em pacotes de visão e controlador, é útil ter classes distintas para desempenhar esses papéis. As classes controladoras de interação devem ser marcadas com o estereótipo <<control>> para diferenciá-las das classes de visão, que devem ser marcadas com o estereótipo <<boundary>>. A Figura 5.2 mostra a notação específica da UML para esses dois estereótipos.

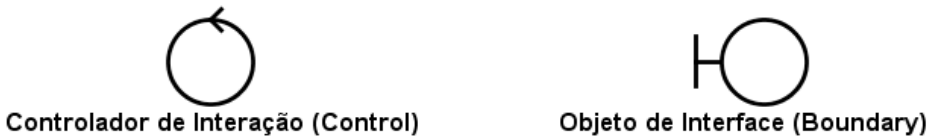


Figura 5.2 – Notação da UML para os Objetos da Camada de IU.

No que se refere à interação entre as camadas de IU e Lógica de Negócio (modelo), ela se dá de maneiras distintas em função do padrão arquitetônico adotado nesta última. Quando o padrão Modelo de Domínio é adotado, os controladores de interação enviam as requisições diretamente para os objetos do domínio do problema (*cdp*), uma vez que, neste caso, não existem objetos gerenciadores de tarefa (*cgt*). Quando o padrão Camada de Serviço é adotado, as requisições dos controladores de interação são enviadas para os objetos gerenciadores de tarefas (*cgt*). A Figura 5.3 ilustra o modo de interação entre essas camadas nos dois padrões.

Embora a Figura 5.3 mostre os pacotes de visão (Componente de Interação Humana – *cih*) e de controle de interação (Componente de Controle de Interação – *cci*) fisicamente separados, essa separação física muitas vezes não é empregada, conforme discutido anteriormente, havendo um único pacote *ciu* (Componente de Interface com o Usuário). O que essa figura procura ressaltar é que, mesmo habitando o mesmo pacote, são as classes controladoras de interação que requisitam serviços da camada de lógica de negócio. Em outras palavras, são as classes controladoras de interação que disparam a lógica de aplicação.

Outro ponto a ser destacado acerca da Figura 5.3 é que ela considera que apenas os objetos controladores de interação se comunicam com objetos da lógica de negócio. Contudo, essa abordagem pode ser flexibilizada. É bastante comum que os próprios objetos de visão (*cih*) se comuniquem com objetos da lógica de negócio, mas apenas para montar os objetos gráficos e não para requisitar serviços.

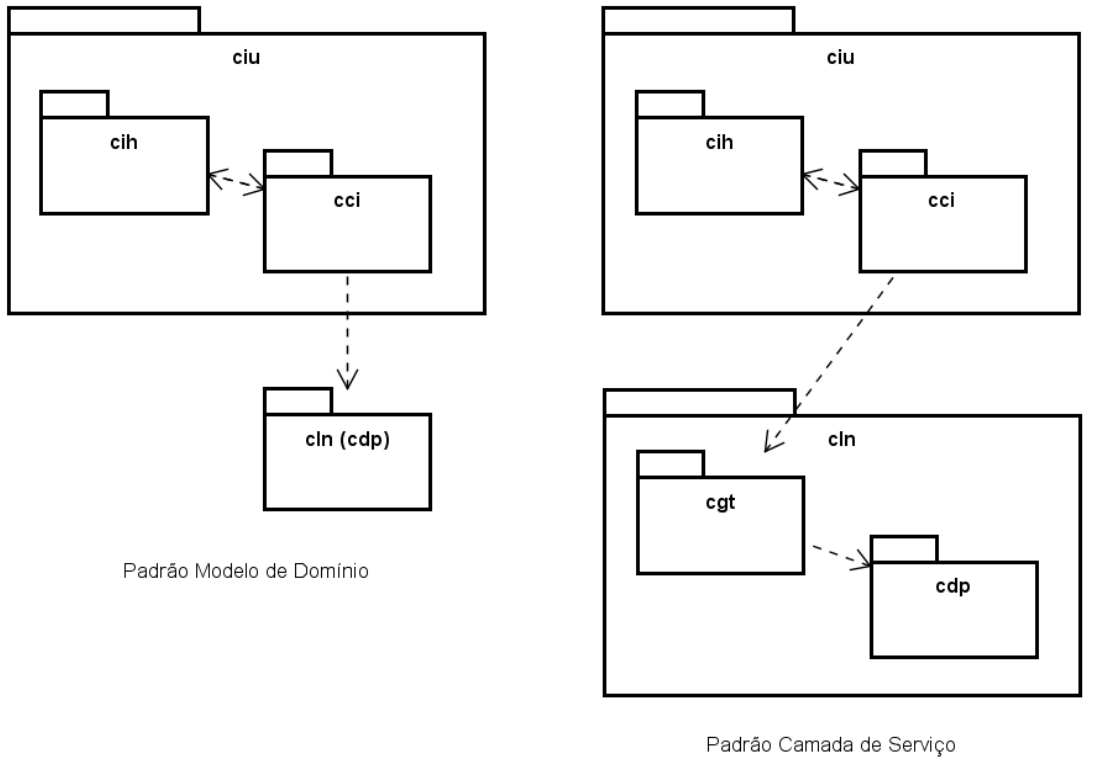


Figura 5.3 – Interação entre Camadas de IU e LN.

O projeto da camada de IU é fortemente relacionado ao projeto da lógica de aplicação e ambos são apoiados pelo modelo de casos de uso. Assim, sobretudo quando o padrão Camada de Serviço é adotado, é uma boa estratégia elaborar um único diagrama de classes envolvendo as classes do *cgt* e as classes do *ciu*. Quando essa estratégia é adotada, é bastante importante usar as notações especializadas da UML para classes controladoras de interação e classes de interface, mostradas na Figura 5.2, de modo a destacar os tipos das diferentes classes no diagrama.

5.2 – O Processo de Projeto da Interface com o Usuário

O projeto de interface com o usuário envolve não apenas aspectos de tecnologia (facilidades para interfaces gráficas, multimídia, etc.), mas principalmente o estudo das pessoas. Quem é o usuário? Como ele aprende a interagir com um novo sistema? Como ele interpreta uma informação produzida pelo sistema? O que ele espera do sistema? Estas são apenas algumas das muitas questões que devem ser levantadas durante o projeto da interface com o usuário (PRESSMAN, 2006).

O princípio básico para o projeto de IU é o seguinte: *Conheça o usuário e as tarefas* (PRESSMAN, 2006). Assim, é importante ter modelos tanto do usuário quanto das tarefas que os mesmos vão desempenhar no sistema. Os modelos de casos de uso têm precisamente essas informações. As tarefas são os casos de uso; os usuários são agrupados em atores. Assim, o modelo de casos de uso é a base principal para o projeto da IU. De maneira geral, o projeto de interfaces com o usuário envolve os seguintes passos:

1. *Definir as funcionalidades acessíveis a partir da IU do sistema*: este passo visa estabelecer como as tarefas que as pessoas fazem normalmente no contexto do sistema (casos de uso) podem ser mapeadas em um conjunto similar (mas não necessariamente idêntico) de tarefas a serem implementadas no contexto da interface humano-computador. Deve-se definir, também, o fluxo global da interação, aglutinando os diversos casos de uso na forma de um ou mais aplicativos.
2. *Estabelecer o perfil dos usuários*: A interface do sistema deve ser adequada ao nível de habilidade dos seus futuros usuários. Assim, é necessário estabelecer o perfil desses potenciais usuários e classificá-los segundo aspectos como nível de habilidade, nível na organização e membros em diferentes grupos. Uma classificação possível considera os seguintes grupos (PRESSMAN, 2006):
    - *Usuário Novato*: não conhece a dinâmica de interação requerida para utilizar a interface eficientemente (conhecimento sintático; p.ex., não sabe como atingir uma funcionalidade desejada) e conhece pouco a semântica da aplicação, isto é, entende pouco as funções e objetivos do sistema, ou não sabe bem como usar computadores em geral;
    - *Usuário conhecedor, mas esporádico*: possui um conhecimento razoável da semântica da aplicação, mas tem relativamente pouca lembrança das informações sintáticas necessárias para utilizar a interface;
    - *Usuário conhecedor e frequente*: possui bom conhecimento tanto sintático quanto semântico e busca atalhos e modos abreviados de interação.
3. *Considerar princípios gerais de projeto de IU*, tais como facilidades de ajuda, mensagens de erro, tipos de comandos, entre outros, de modo a prover uma IU adequada para os perfis de usuários estabelecidos. Este passo pode ser visto como a definição de quais táticas de usabilidade devem ser aplicadas no projeto da IU de um sistema.
4. *Construir protótipos* e, em última instância, implementar as interfaces do sistema, usando ferramentas apropriadas. A prototipagem abre espaço para uma abordagem iterativa de projeto de interface com o usuário, como mostra a Figura 5.4. Nessa abordagem, utilizando diversos protótipos construídos iterativamente, o usuário avalia a adequação da IU para o uso em seus processos de negócio. Em uma abordagem de prototipagem, é imprescindível o uso de ferramentas para a construção de interfaces, provendo facilidades para manipulação de janelas, menus, botões, comandos etc.
5. *Avaliar o resultado*: A construção de protótipos é feita com a participação de apenas uns poucos usuários. Uma vez que se obtém o projeto considerado completo da IU, idealmente, deve-se avaliar seu resultado para o conjunto (ou uma amostra significativa) de usuários. Para tal, pode ser útil coletar dados qualitativos e quantitativos por meio, p.ex., de questionários distribuídos a uma amostra significativa de usuários.

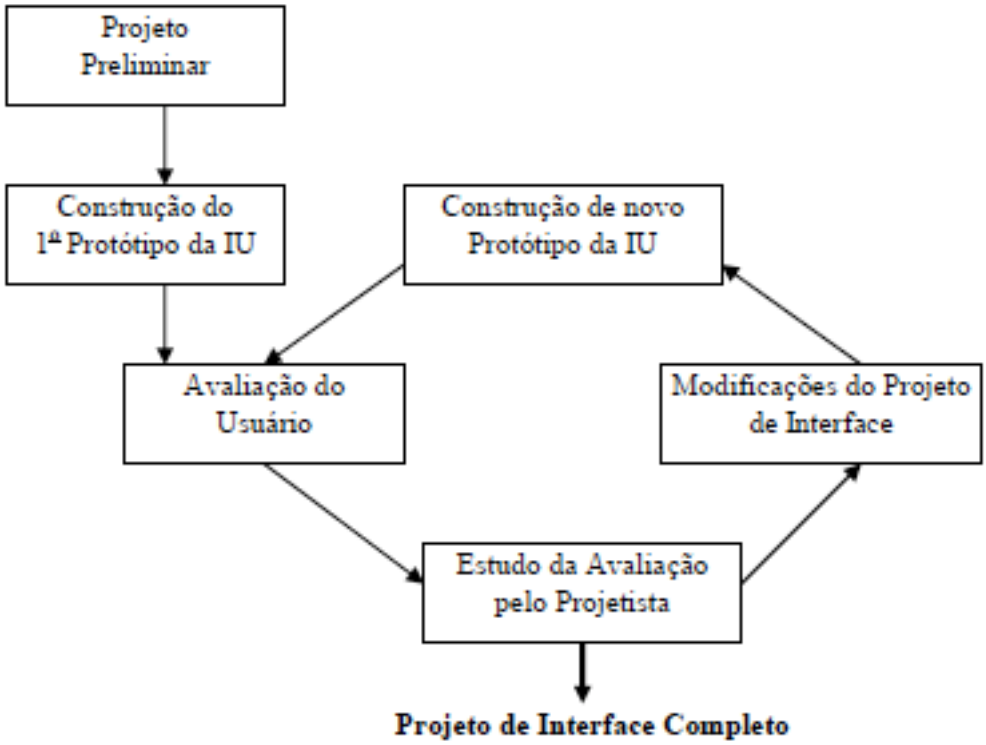


Figura 5.4 - Abordagem Iterativa para o Projeto de Interface com o Usuário (adaptado de (PRESSMAN, 2006)).

5.3 – Projeto da Visão

A porção do sistema que lida com a visão da interface com o usuário deve ser mantida tão independente e separada do restante da arquitetura do software quanto possível. Aspectos de interface com o usuário provavelmente serão alvo de alterações ao longo da vida do sistema e essas alterações devem ter um impacto mínimo nas demais partes do sistema.

A visão trata do projeto da interação humano-computador, definindo formato de janelas, formulários, relatórios, entre outros. Conforme discutido anteriormente, durante o projeto da visão, é muito útil construir protótipos, de modo a apoiar a escolha e o desenvolvimento dos mecanismos de interação a serem usados.

O ponto de partida para o projeto da visão é o modelo de casos de uso, incluindo as descrições de atores e casos de uso. Com base nos casos de uso, deve-se projetar uma hierarquia de comandos, definindo barras de menus, menus *pull-down*, ícones etc., que levem à execução dos casos de uso, quando acionados pelo usuário. A hierarquia de comandos deve respeitar convenções e estilos existentes com os quais o usuário já esteja familiarizado. Note que a hierarquia de comandos é, de fato, um meio de apresentar ao usuário as várias funcionalidades disponíveis no sistema. Assim, a hierarquia de comandos deve permitir o acesso aos casos de uso do sistema.

Uma vez definida a hierarquia de comandos, as interações detalhadas entre o usuário e o sistema devem ser projetadas. Neste momento, é útil observar atentamente táticas de usabilidade, discutidas mais à frente na Seção 5.3.1.

Normalmente, não é necessário projetar as classes básicas de interfaces gráficas com o usuário. Existem vários ambientes de desenvolvimento de interfaces oferecendo classes reutilizáveis (janelas, ícones, botões etc.) e, portanto, basta especializar essas classes e instanciar os objetos que possuem as características apropriadas para o problema em questão. Ainda assim, é muito útil desenvolver classes gerais de visão visando à uniformidade da apresentação e ao reúso. Essas classes podem ser organizadas em hierarquias de classes, de modo que, no projeto de um sistema específico, o projeto da sua visão seja realizado por meio da especialização de classes de visão já existentes ao invés de ter de se compor todas as classes de visão a partir de componentes básicos de interface providos pelo ambiente de desenvolvimento.

O projeto detalhado das classes de visão do sistema pode ser mais facilmente compreendido pela visualização das próprias telas, sendo pouco útil indicar os atributos de uma classe de visão em um diagrama de classes. Para compreender a estrutura interna de uma classe de visão, é mais indicado prover o layout correspondente. Entretanto, ainda é útil mostrar em um diagrama de classes as classes de visão e as suas relações de especialização com outras classes de visão e, sobretudo, as suas associações com as classes controladoras de interação, o que permite capturar como se dará efetivamente o tratamento da interação humano-computador do sistema.

5.3.1 – Táticas de Usabilidade

Diversas táticas relativas à usabilidade podem ser aplicadas durante o projeto de IU. Algumas dessas táticas incluem: (i) provisão de facilidades de ajuda, (ii) apresentação de mensagens de aviso e de erro significativas, (iii) oferta de diferentes tipos de comandos, adequados para diferentes perfis de usuário e (iv) visão do estado do sistema quando em processamento.

Facilidade de Ajuda

Ajuda é fundamental para os usuários, sobretudo para os novatos ou para aqueles conhecedores do problema, mas usuários esporádicos do sistema. Para projetar adequadamente facilidades de ajuda, é necessário definir vários aspectos, dentre eles: (i) quando a ajuda estará disponível e para que funções do sistema ou campos da IU; (ii) como ativar (botão, tecla de função, menu etc.); (iii) onde apresentar (janela separada, local fixo da tela etc.); (iv) como retornar à interação normal (botão, tecla de função); (v) como estruturar a informação (estrutura plana, hierárquica, hipertexto).

Em relação às funções do sistema, é importante prover ajuda ao usuário para que este esclareça o objetivo de uma funcionalidade do sistema, qual a sua sequência de passos (fluxo do caso de uso) e em que passo ele está no momento (e, por conseguinte, como chegou até ali). P.ex., como ilustra a Figura 5.5, em um sistema de compra de passagens aéreas, é importante identificar para o usuário quais os passos a serem realizados e em que passo o usuário se encontra em um dado momento.

Figura 5.5 – Exemplo de ajuda relativa à sequência de passos de uma função.

Em relação aos campos da IU, é importante informar o significado de campos não óbvios para o usuário, como preenchê-los e, eventualmente, onde obter a informação para seu preenchimento. P.ex., em um sistema de registro de empregados domésticos para emissão de guia de recolhimento de impostos, é importante dizer o que significa o campo NIT, como preenchê-lo e onde obter essa informação. Além disso, para campos que são apresentados no mundo real geralmente formatados (p.ex., datas, CPF etc.) é importante dizer como os mesmos devem ser preenchidos. P.ex., em sistemas que requerem o preenchimento de CPF, deve-se informar a forma de preenchimento (apenas números ou com uso de separadores).

Em relação a como ativar as facilidades de ajuda, há diferentes maneiras de se prover acesso às facilidades de ajuda. Quando a ajuda é pontual (p.ex., o significado de um campo ou a sua formatação), a informação pode estar prontamente disponível na própria interface ou pode-se disponibilizar um recurso de ajuda (p.ex., um botão ou um campo sensível à presença do mouse) próximo ao elemento para o qual se deseja prover a ajuda. Para facilidades de ajuda mais sofisticadas, que envolvem muitas informações (tal como um help completo do sistema) normalmente usam-se botões, itens de menu e teclas de função (p.ex., F1) ou uma combinação desses recursos. A Figura 5.6 ilustra alguns exemplos de formas de se ativar facilidades de ajuda.

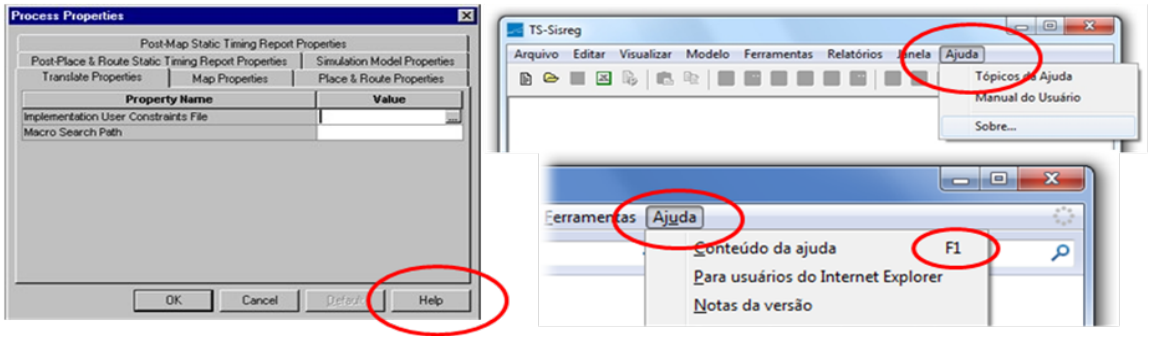


Figura 5.6 – Exemplos de formas de ativar facilidades de ajuda

No que se refere a onde apresentar, facilidades de ajuda mais complexas (p.ex., ajuda para todo sistema) tipicamente são apresentadas em janelas separadas, ativadas por itens de menu, botões ou teclas de função, como ilustra a Figura 5.7. Para informação mais simples, pode-se usar uma janela *pop-up* ou apresentar a ajuda em um local fixo na tela. Para ajuda sobre campos, pode-se prover instrução próxima ao campo, disponibilizar a informação quando da passagem do mouse ou prover um botão de ajuda ao lado do campo. No caso de ajuda sobre a formatação de campos, além das opções anteriores, pode-se prover a informação na própria descrição do campo, ou até mesmo já prover a formação. A Figura 5.8 ilustra algumas dessas possibilidades.

Quando uma facilidade de ajuda é provida em outro objeto de interface (p.ex., janela separada ou janela *pop-up*), é preciso definir como retornar à interação normal. Janelas *pop-up* tipicamente têm botões para fechá-las. Janelas separadas podem usar os mecanismos típicos de encerramento (veja Figura 5.7). Outra opção é fazer uso de teclas de função (p.ex., ESC para retornar à interação normal).

Por fim, para facilidades de ajuda com conteúdo complexo, é importante definir como estruturar a informação. Algumas opções são estruturas planas, hierárquicas e hipertextos, as quais podem ser combinadas, como no caso da Figura 5.7.

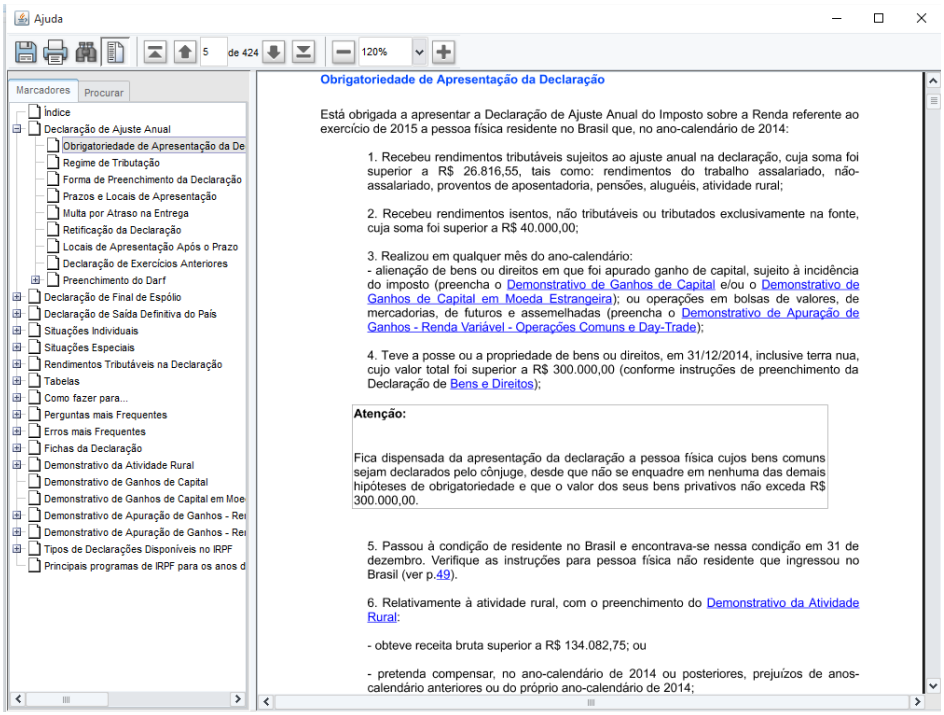


Figura 5.7 – Exemplo de ajuda de todo sistema.

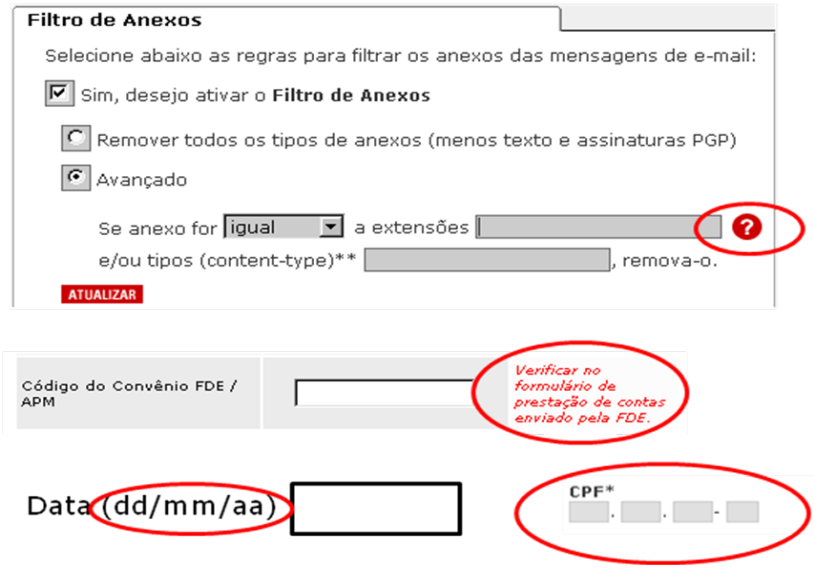


Figura 5.8 – Exemplos de formas de apresentação de ajuda para campos

Mensagens de Erro e Avisos

Mais até do que a ajuda, as mensagens de erro e avisos são fundamentais para uma boa interação humano-computador. Ao definir mensagens de erro e avisos considere as seguintes diretrizes:

- Descreva o problema com um vocabulário passível de entendimento pelo usuário.
- Sempre que possível, proveja assistência para recuperar um erro.
- Quando for o caso, indique as consequências negativas de uma ação.
- Para facilitar a percepção da mensagem por parte do usuário, pode ser útil que a mesma seja acompanhada de uma dica visual (p.ex., destacar o campo em que o preenchimento apresentou problema) ou sonora.

Tipos de Comandos

Diferentes grupos de usuários têm diferentes necessidades de interação. Em muitas situações é útil prover aos usuários mais de uma forma de interação. Nestes casos, é necessário definir e avaliar:

- Quais funções terão mais de uma forma de interação (p.ex., um item de menu e um comando correspondente);
- Qual será a forma do comando. Controle de sequência (p.ex., ^Q), teclas de função (p.ex., F1) e comandos digitados são opções tipicamente combinadas com itens de menu;
- Quão difícil é aprender e lembrar o comando;
- Os padrões a serem adotados: Devem ser adotados padrões uniformes para todo sistema. Esses padrões devem estar em conformidade com outros padrões, tais como o definido pelo sistema operacional e por produtos de software tipicamente utilizados pelos usuários.

Progresso do Processamento

É importante mostrar o progresso do processamento para os usuários, principalmente para eventos com tempo de resposta longo ou com grande variação de tempos de resposta. A Figura 5.9 ilustra uma janela de progresso de processamento.

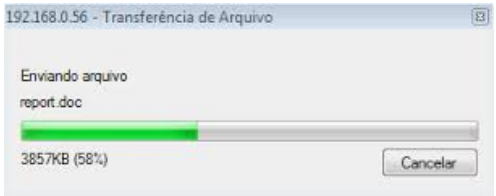


Figura 5.9 – Exemplos de interface informando o progresso do processamento

5.3.2 – Diretrizes para o Projeto da Visão

Levando-se em conta princípios gerais de projeto de IU, algumas orientações adicionais devem ser consideradas, dentre elas:

- Seja consistente. Use formatos consistentes para seleção de menus, entrada de comandos, apresentação de dados etc.
- Ofereça retorno significativo ao usuário.
- Peça confirmação para ações destrutivas, tais como ações para apagar ou sobrepor informações ou para terminar a seção corrente do aplicativo.
- Permita reversão da maioria das ações (função *Desfazer*).
- Reduza a quantidade de informação que precisa ser memorizada entre ações.
- Busque eficiência no diálogo (movimentação, teclas a serem apertadas).
- Trate possíveis erros do usuário. O sistema deve se proteger de erros, casuais ou não, provocados pelo usuário.
- Classifique atividades por função e organize geograficamente a tela de acordo. Menus do tipo *pull-down* são uma boa opção.
- Proveja facilidades de ajuda sensíveis ao contexto.
- Use verbos de ação simples ou frases curtas para nomear funções e comandos.

No que se refere à apresentação de informações, considere as seguintes diretrizes:

- Mostre apenas informações relevantes ao contexto corrente.
- Use formatos de apresentação que permitam assimilação rápida da informação, tais como gráficos e figuras.
- Use rótulos consistentes, abreviaturas padrão e cores previsíveis.
- Produza mensagens de erro significativas.
- Projete adequadamente o layout de informações textuais. Leve em consideração o bom uso de letras maiúsculas e minúsculas, identação, agrupamento de informações etc.
- Separe diferentes tipos de informação. Painéis podem ser usados para este fim.
- Use formas de representação análogas às do mundo real para facilitar a assimilação da informação. Para tal considere o uso de figuras, cores etc.

No que se refere à entrada de dados, considere as seguintes diretrizes:

- Minimize o número de ações de entrada requeridas e possíveis erros. Para tal considere a seleção de dados a partir de um conjunto pré-definido de valores de entrada, o uso de valores *default* etc.

- Mantenha consistência entre apresentação e entrada de dados; ou seja, mantenha as mesmas características visuais, dentre elas tamanho do texto, cor e localização.
- Permita ao usuário customizar a entrada para seu uso, quando possível, dando-lhe liberdade para definir comandos customizados, dispensar algumas mensagens de aviso e verificações de ações, dentre outros.
- Flexibilize a interação, permitindo afiná-la ao modo de entrada preferido do usuário (comandos, botões, *plug-and-play*, digitação etc.).
- Desative comandos inapropriados para o contexto das ações correntes.
- Proveja ajuda significativa para assistir as ações de entrada de dados.
- Nunca requeira que o usuário entre com uma informação que possa ser adquirida automaticamente pelo sistema ou computada por ele.

5.4 – Projeto do Controle de Interação

O projeto do Controle de Interação visa definir as classes responsáveis por controlar a interação (ativar/desativar objetos de visão) e enviar requisições para os objetos da Lógica de Negócio.

Em sistemas rodando em plataforma *desktop*, deve haver pelo menos uma classe controladora, dita classe controladora de sistema, representando o sistema como um todo. Os objetos dessa classe representam as várias sessões (execuções) do sistema. Neste contexto, é necessário levar em conta quantos executáveis devem ser gerados para o sistema. Se mais do que um executável for necessário, cada executável terá de dar origem a uma classe controladora. Esta, contudo, é apenas uma abordagem possível.

Conforme discutido na Seção 5.1, a interação entre controladores e objetos da lógica de negócio se dá de maneiras distintas em função do padrão arquitetônico adotado no projeto da lógica de negócio. Quando o padrão Modelo de Domínio é adotado, os controladores de interação estão associados diretamente com os objetos do domínio do problema (*cdp*). Conforme discutido na Seção 4.4.1, uma abordagem interessante consiste em relacionar a classe controladora do sistema (ou uma classe controladora responsável por parte da funcionalidade do sistema) com os objetos independentes do *cdp*. Vale destacar que o conceito de objeto independente pode ser analisado no contexto do subsistema, i.e., se um objeto não depende de outros objetos no contexto do subsistema em questão, então ele pode ser considerado independente neste subsistema e estar ligado ao controlador.

De fato, seguindo a estratégia geral apontada acima, algumas opções podem ser exploradas. A Figura 5.10 ilustra o caso do projeto do subsistema Controle de Frota de um sistema de locação de carros. Há a necessidade de se ter um executável para esta porção desktop do sistema e, portanto, há uma classe controladora de sistema para ela (CtrlControleFrota), a qual controla todos os objetos gráficos deste subsistema (a janela principal e os vários formulários para cadastro das entidades consideradas) e faz a ligação com a Camada de Lógica de Negócio, via objetos de domínio, já que o padrão Modelo de Domínio está sendo utilizado nesse exemplo.

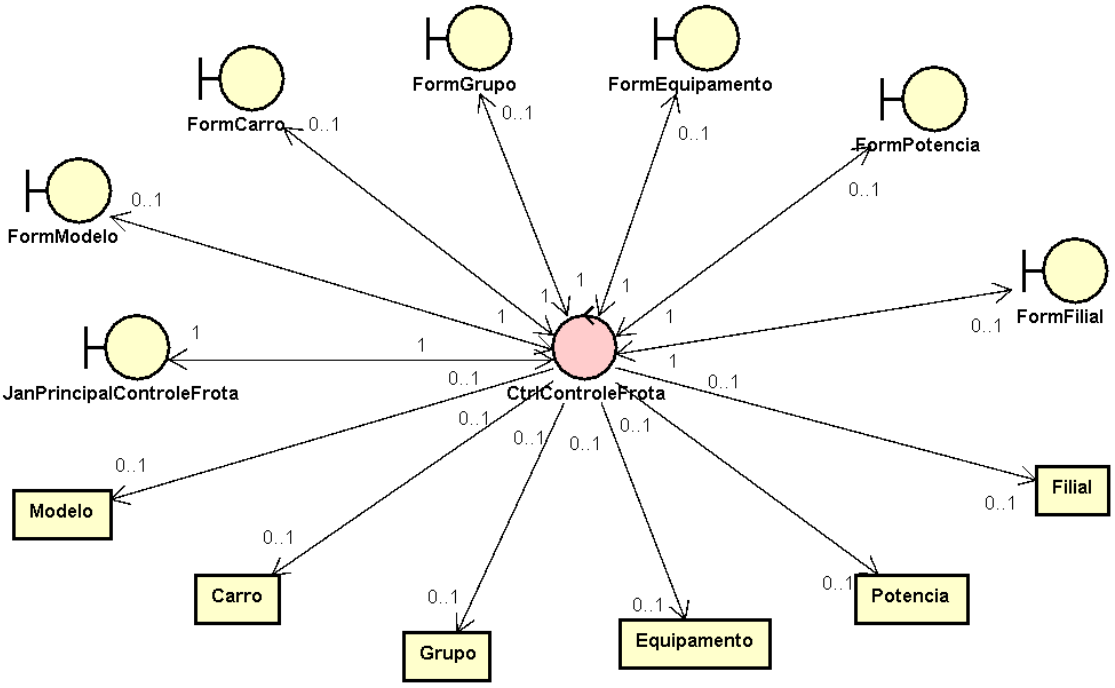


Figura 5.10 – Projeto da IU considerando um único controlador de interação e a aplicação do Padrão Modelo de Domínio na Camada de Lógica de Negócio.

A solução acima é uma opção extrema, na medida em que concentra todo o controle de interação em uma única classe, o que, de maneira geral, traz problemas para a manutenibilidade. Na prática, é comum considerar a criação de mais do que uma classe controladora de interação. Neste caso, além da classe controladora de (sub)sistema, pode-se criar, ainda, um controlador de interação por caso de uso, como ilustra parcialmente a Figura 5.11.

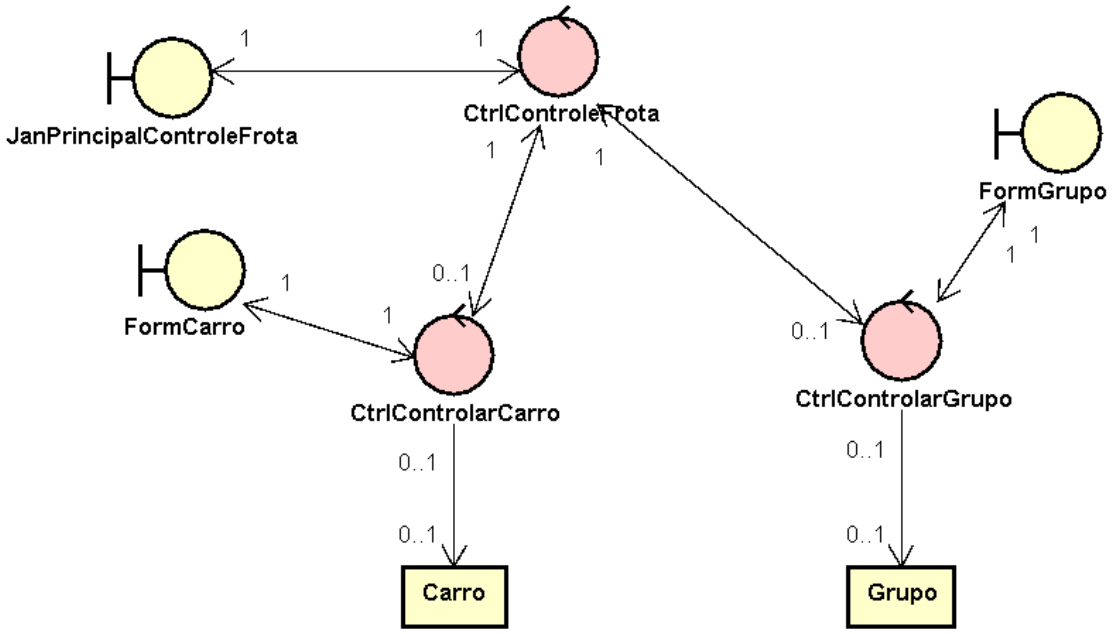


Figura 5.11 – Projeto da IU considerando vários controladores de interação e a aplicação do Padrão Modelo de Domínio na Camada de Lógica de Negócio.

Quando o padrão Camada de Serviço é adotado, os controladores estão associados a objetos gerenciadores de tarefas (*cgt*). Analogamente ao projeto do Componente de Gerência de Tarefas (*cgt*) (ver Seção 4.4.2), é possível definir um número arbitrário de controladores de interação. Uma opção é definir um único controlador para todo o sistema (controlador de sistema), o qual fica responsável por gerenciar a interação de toda aplicação. A Figura 5.12 mostra o mesmo caso da Figura 5.10, agora apenas considerando a aplicação do Padrão Camada de Serviço. Como se pode perceber, a única diferença entre os diagramas das figuras 5.10 e 5.12 é que na Figura 5.10, o controlador de sistema comunica-se com as classes de domínio (Padrão Modelo de Domínio), enquanto na Figura 5.12 o controlador de sistema comunica-se com as classes de lógica de aplicação (Padrão Camada de Serviço).

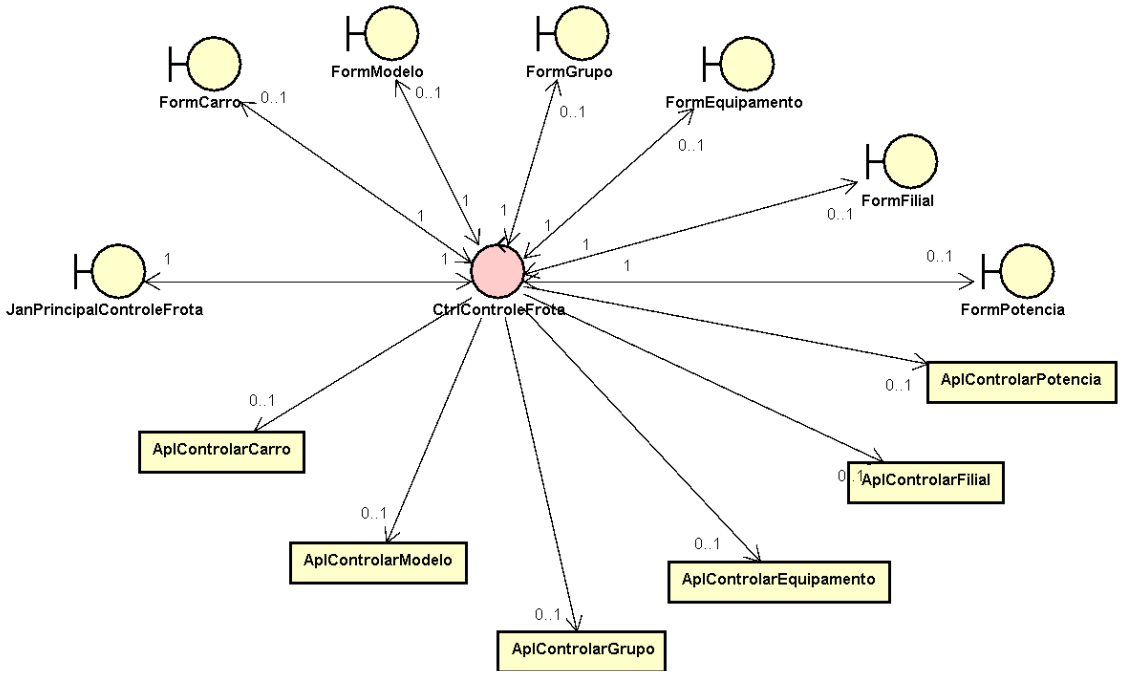


Figura 5.12 – Projeto da IU considerando um único controlador de interação e a aplicação do Padrão Camada de Serviço na Camada de Lógica de Negócio.

Assim como ocorre no projeto do *cgt*, essa opção tende a ser inadequada, pois a classe controladora de sistema pode ficar muito complexa. Uma opção no sentido de favorecer a manutenibilidade consiste em definir um controlador de interação para cada caso de uso. Em aplicações *desktop*, adicionalmente, é necessário ter ainda, pelo menos, uma classe controladora de sistema (ou uma classe controladora para cada executável). A Figura 5.13 mostra o mesmo caso da Figura 5.11, agora considerando a aplicação do Padrão Camada de Serviço. Como se pode perceber, a diferença básica entre os diagramas das figuras 5.11 e 5.13 é que na Figura 5.11, os controladores de interação relativos a casos de uso específicos comunicam-se com as classes de domínio (Padrão Modelo de Domínio), enquanto na Figura 5.13 esses controladores de interação comunicam-se com as classes de lógica de aplicação (Padrão Camada de Serviço).

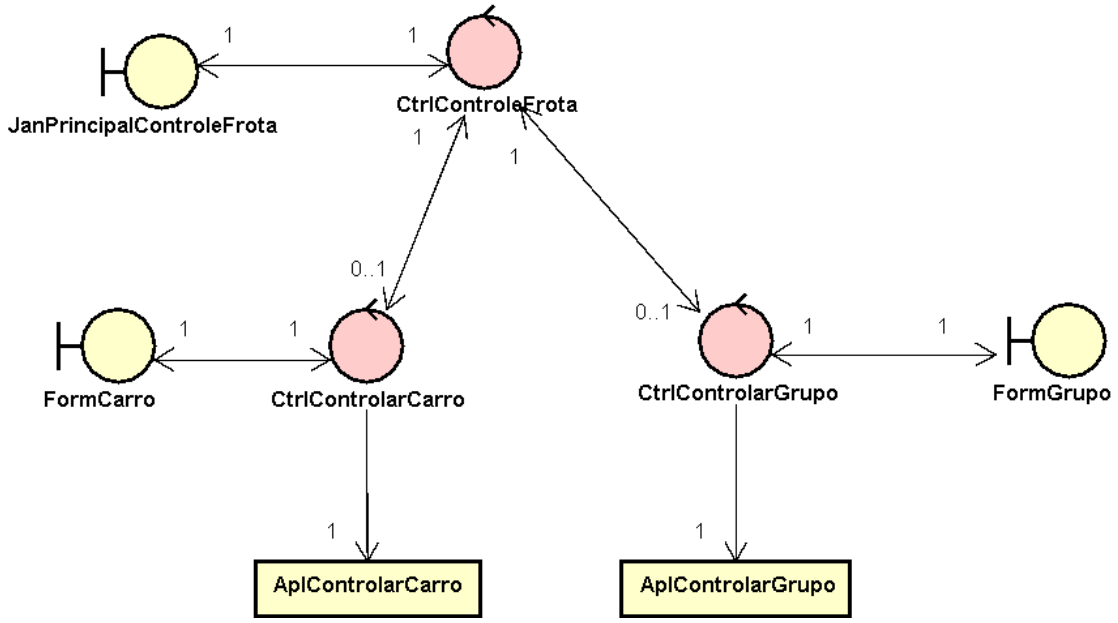


Figura 5.13 – Projeto da IU considerando vários controladores de interação e a aplicação do Padrão Camada de Serviço na Camada de Lógica de Negócio.

Vale ressaltar que uma solução intermediária entre as duas anteriormente apresentadas pode ser considerada. Uma vez que as classes controladoras de interação tendem a ser mais simples que as classes do *cgt*, não é necessário adotar a mesma estratégia usada na *cgt*. Assim, ainda que haja uma analogia entre o projeto do controle de interação e o projeto do *cgt*, as motivações são diferentes e a escolha dos controladores de interação pode ser diferente da escolha dos gerenciadores de tarefas.

5.5 – *Design Patterns* no Projeto da Interface com o Usuário

Alguns padrões de projeto (*design patterns*) são bastante utilizados para tratar problemas recorrentes no projeto da IU, dentre eles os padrões Decorador, Observador e Comando.

O padrão Decorador (*Decorator*) anexa responsabilidades adicionais a um objeto dinamicamente, permitindo estender sua funcionalidade (GAMA et al., 1995). Ele é utilizado por *frameworks* decoradores de interface para automatizar a tarefa de manter uma aplicação *Web* tradicional[6] com a mesma aparência, ou seja, cabeçalho, rodapé, barra de navegação, esquema de cores e demais elementos gráficos de layout integrados num mesmo projeto de apresentação. Esse tipo de *framework* funciona segundo o padrão de projeto Decorador, se posicionando como um filtro entre uma requisição do cliente e um servidor *Web*, como ilustra a Figura 5.14 (SOUZA, 2005).

[6] i.e., uma aplicação Web que não se enquadra no conceito de Aplicação Rica para Internet, conforme discutido no Capítulo 3.

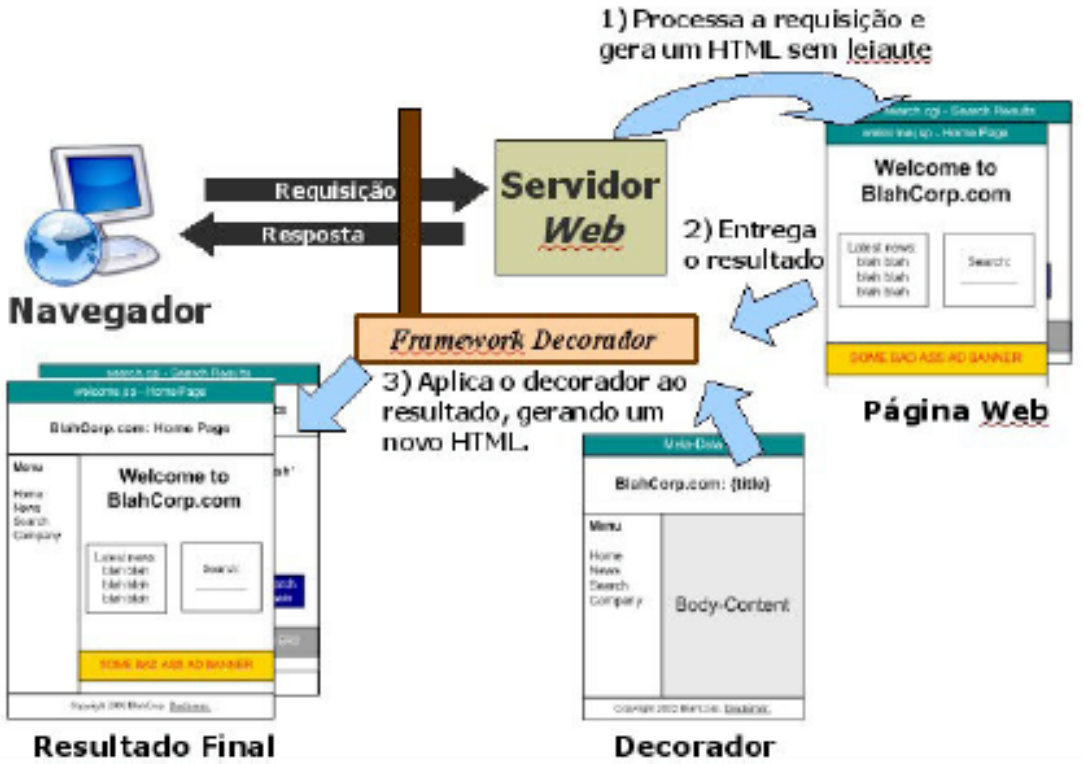


Figura 5.14 – Funcionamento de um *framework* Decorador (SOUZA, 2005).

O padrão Observador (*Observer*) define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente (GAMMA et al., 1995). Ele é uma boa opção para separar aspectos de apresentação dos respectivos dados da aplicação, permitindo o uso de múltiplas representações. Por exemplo, os mesmos dados estatísticos podem ser apresentados em formato de um gráfico de barras ou em uma planilha, usando apresentações diferentes. O gráfico de barras e a planilha devem ser independentes. Contudo, eles têm de se comportar consistentemente, isto é, quando um usuário alterar a informação na planilha, o gráfico de barras deve refletir a troca imediatamente e vice-versa, como ilustra a Figura 5.15 (GAMMA et al., 1995). Na solução proposta pelo padrão observador, a apresentação atua como um observador do domínio do problema: sempre que o domínio do problema é alterado, ele envia um evento e a apresentação atualiza a informação (FOWLER, 2003).

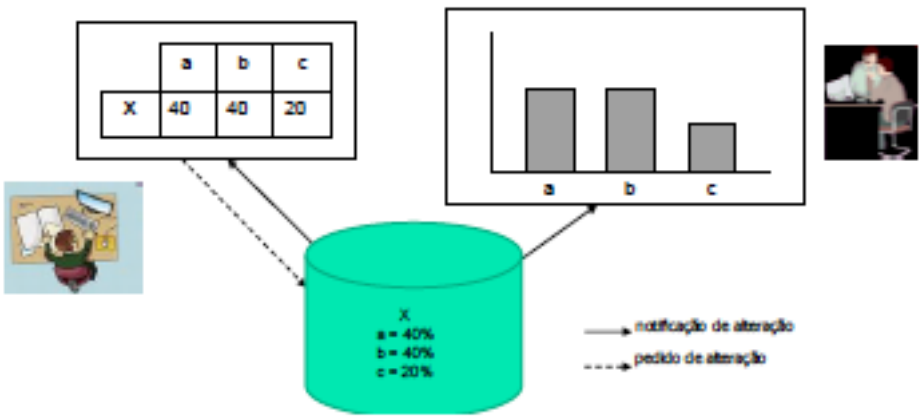


Figura 5.15 – Aplicação do padrão Observador (adaptado de (GAMA et al., 1995)).

Finalmente, o padrão Comando (*Command*) encapsula uma requisição como um objeto, permitindo parametrizar clientes com diferentes requisições e desfazer operações. Em interfaces com o usuário, objetos gráficos (p.ex., botões e menus), quando acionados, devem disparam requisições para a execução de funcionalidades do sistema. Entretanto, essas requisições não podem ser implementadas diretamente nos objetos gráficos, pois somente a aplicação deve saber efetivamente o que deve ser feito. O padrão Comando permite que os objetos gráficos façam requisições de objetos não especificados, tornando a requisição em si um objeto. Esse objeto pode ser armazenado e repassado como qualquer outro objeto.

Além dos *design patterns* propostos em (GAMMA et al., 1995), os quais tratam de problemas mais gerais, aplicáveis também ao projeto da IU, há padrões específicos dedicados ao projeto da visão. Por exemplo, Nudelman (2013) apresenta um catálogo de padrões (e alguns anti-padrões) de projeto de visão para a plataforma Android. Há padrões para experiência de boas-vindas, tela principal, busca, ordenação e filtragem, entradas de dados, formulários e navegação, dentre outros. Os padrões são descritos com foco em *smartfones*, contendo uma descrição geral do padrão, informações sobre como ele funciona, exemplo, quando e onde usar, por que usar, outros usos e variação para aplicativos para *tablets*.

Capítulo 6 – Projeto da Gerência de Dados

A maioria dos sistemas requer alguma forma de armazenamento de dados. Para tal, há várias alternativas, dentre elas a persistência em arquivos e bancos de dados. Em especial os sistemas de informação envolvem grandes quantidades de dados e fazem uso de sistemas gerenciadores de bancos de dados (SGBDs). Há diversos tipos de SGBDs, dentre eles os relacionais e os orientados a objetos, sendo os primeiros os mais utilizados atualmente no desenvolvimento de sistemas de informação.

Quando SGBDs Relacionais são utilizados, é necessário um mapeamento entre as estruturas de dados dos modelos orientado a objetos e relacional, de modo que objetos possam ser armazenados em tabelas. Dentre as principais diferenças entre esses modelos, destacam-se as diferentes formas como objetos e tabelas tratam ligações e na ausência do mecanismo de herança no modelo relacional. Essas diferenças levam à necessidade de transformações das estruturas de dados entre objetos e tabelas, tratadas como mapeamento objeto-relacional.

Além das diferenças estruturais, outros aspectos têm de ser tratados durante o projeto da persistência, dentre eles o modo como a camada de lógica de negócio se comunica com o banco de dados, o problema comportamental que diz respeito a como obter vários objetos do banco e como salvá-los, e o tratamento de conexões com o banco de dados e transações (FOWLER, 2003).

É importante enfatizar que muitos desses problemas são tratados por *frameworks* de persistência de objetos em bancos de dados relacionais (ou *frameworks* de mapeamento objeto-relacional), tal como o Hibernate[7]. Os desenvolvedores desses frameworks têm despendido muitos esforços trabalhando nesses problemas e tais ferramentas são bem mais sofisticadas do que a maioria das soluções específicas que podem ser construídas pelos próprios desenvolvedores em um projeto específico. Contudo, mesmo quando um framework de mapeamento objeto-relacional (O/R) é utilizado, é importante estar ciente dos padrões usados. Boas ferramentas de mapeamento O/R dão várias opções de mapeamento para um banco de dados e esses padrões ajudam a entender quando selecionar as diferentes opções (FOWLER, 2003).

Por fim, deve-se enfatizar que um bom projeto do mecanismo de persistência deve levar em conta a ideia de separação entre interface com o usuário, lógica de negócio e persistência, conforme discutido no Capítulo 3. Assim, o presente capítulo visa discutir os principais aspectos relacionados ao projeto da persistência de objetos em bancos de dados relacionais, primando pelo isolamento do banco de dados, de modo que seja possível, por exemplo, a substituição de um SGBD relacional por outro ou até mesmo por um outro tipo de SGBD.

Este capítulo está estruturado da seguinte forma: a Seção 6.1 apresenta sucintamente o modelo relacional, que constitui a base dos SGBDs relacionais; a Seção

[7] http://www.hibernate.org/

6.2 discute o mapeamento objeto-relacional; a Seção 6.3 discute padrões arquitetônicos para o projeto da camada de persistência; e finalmente a Seção 6.4 fala sobre *frameworks* de persistência.

6.1 – O Modelo Relacional

Em um modelo de dados relacional, os conjuntos de dados são representados por tabelas de valores. Cada tabela, denominada de relação, é bidimensional, sendo organizada em linhas e colunas. Esse modelo está fortemente baseado na teoria matemática sobre relações, daí o nome relacional. Os principais conceitos do modelo relacional são os seguintes:

- **Tabela ou Relação**: tabela de valores bidimensional organizada em linhas e colunas. A Figura 6.1 mostra um exemplo de uma tabela *Funcionários*.

Matrícula	Nome	CPF	Dt-Nasc
0111	Marcos	17345687691	11/04/66
0208	Rita	56935101129	21/02/64
0789	Mônica	81176628911	01/11/70
1589	Márcia	91125769120	20/10/80

Figura 6.1 – Tabela Funcionários.

- **Linha ou Tupla**: representa uma entidade de um conjunto de entidades. Ex: A funcionária Mônica do conjunto de funcionários.
- **Coluna**: representa um atributo de uma entidade. Ex.: Matrícula, Nome, CPF, Dt-Nasc.
- **Célula**: Item de dado da linha *i*, coluna *j*. Ex.: Rita (linha 2, coluna 2).
- **Chave Primária**: coluna ou combinação de colunas que possui a propriedade de identificar de forma única uma linha da tabela e que é utilizada para estabelecer associações entre entidades via transposição de chave.
- **Chave Estrangeira ou Transposta**: é a forma utilizada para associar linhas de tabelas distintas. A chave primária de uma tabela é transposta como uma coluna na outra tabela, onde é considerada uma chave estrangeira. A Figura 6.2 ilustra um relacionamento 1:N entre as tabelas *Departamentos* e *Funcionários*, indicando que um departamento pode lotar vários funcionários, enquanto um funcionário tem de estar lotado em um departamento.

Departamentos

Código	Nome
INF	Informática
LET	Letras
MAT	Matemática

Funcionários

Matrícula	Nome	Cod-Depto
0158	José	MAT
5295	Ricardo	INF
7712	Rosane	INF

Chave Estrangeira

Figura 6.2 – Exemplo de ligação entre tabelas, por meio de chave estrangeira.

- **Tabelas Associativas**: usadas para representar relacionamentos *n*-para-*n* entre tabelas. No exemplo da Figura 6.3, uma pessoa pode ter interesse em vários assuntos, enquanto um assunto pode ser de interesse de várias pessoas. A tabela *Interesses* é uma tabela associativa, sendo suas duas colunas chaves transpostas de outras tabelas.

Pessoas

CPF	Nome
96100199	José
83467187	Maria
02765140	Luiza

Interesses

CPF-Pessoa	Código-Assunto
96100199	COMP
96100199	MUS
02765140	ENG

Assuntos

Código	Nome
ENG	Engenharia
COMP	Computação
MUS	Música

Figura 6.3 – Exemplo de Tabela Associativa.

O modelo relacional tem diversas propriedades que precisam ser respeitadas, a saber:

- Cada tabela possui um nome, o qual deve ser distinto do nome de qualquer outra tabela da base de dados.
- Nenhum campo parte de uma chave primária pode ser nulo.
- Cada célula de uma relação pode ser vazia (exceto os campos de chaves primárias) ou, ao contrário, pode conter no máximo um único valor.
- Não há duas linhas iguais.
- A ordem das linhas é irrelevante.
- Cada coluna tem um nome, o qual deve ser distinto dos demais nomes das colunas de uma mesma tabela.
- Usando-se os nomes para se fazer referência às colunas, a ordem destas torna-se irrelevante.
- Os valores de uma coluna são retirados todos de um mesmo conjunto, denominado *domínio* da coluna.
- Duas ou mais colunas distintas podem ser definidas sobre um mesmo domínio.
- Um campo que seja chave estrangeira (ou parte dela) só pode assumir valor nulo ou um valor para o qual exista um registro na tabela onde ele é chave primária.

Muitas vezes, durante o projeto de bancos de dados relacionais, é útil representar graficamente as tabelas e as ligações entre elas. Para tal, um Diagrama Relacional pode ser desenvolvido, representando as ligações entre tabelas de um modelo relacional. A Figura 6.4 mostra um exemplo de um fragmento de um diagrama relacional e suas tabelas correspondentes.

Diagrama Relacional

Tabelas do Modelo Relacional

Departamentos		
Código	Nome	Matrícula-Chefe
INF	Informática	00877
MAT	Matemática	06001
QUI	Química	13888

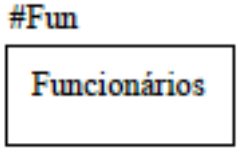
Funcionários	
Matrícula	Nome
13888	Jorge
00877	Dede
06001	Pedro

Figura 6.4 – Exemplo de Diagrama Relacional e as respectivas tabelas

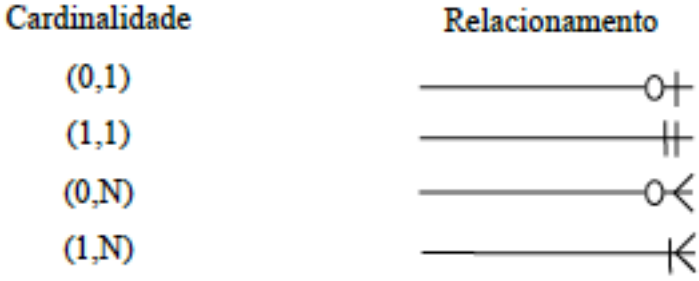
Nesse exemplo a coluna *Matrícula* de *Funcionários* foi considerada a chave primária da tabela *Funcionários* e foi transposta para a relação *Departamentos*. O contrário também poderia ser feito, isto é, transpor a chave primária de *Departamentos* para *Funcionários*. A primeira opção é mais indicada, porque há poucos funcionários que são chefes, enquanto todos os departamentos têm chefes. Assim, a coluna *Matrícula-Chefe* não terá valores vazios e, portanto, ela é mais densa do que seria a coluna resultante da transposição da chave primária de *Departamentos* para a tabela *Funcionários*.

Em um Diagrama Relacional são representados os seguintes elementos:

- Tabelas: são representadas por retângulos, com uma referência à chave primária em cima da tabela.



- Relacionamentos: são representadas por linhas contínuas, associadas aos símbolos abaixo:
-



- Chaves estrangeiras: quando uma chave transposta não fizer parte da chave primária da relação destino, a mesma é representada em cima do retângulo da relação destino com um subscrito "t", como ilustra a Figura 6.4.

Colunas que não são chaves primárias ou estrangeiras não são representadas nos diagramas, mas sim em um dicionário de tabelas do modelo relacional.

Outra opção para representar diagramas relacionais é utilizar um perfil UML. Neste caso, tabelas são representadas como classes com o estereótipo <<table>>, tabelas associativas são representadas como classes com o estereótipo <<association table>> e colunas são representadas como atributos. Quando uma coluna é chave primária (ou parte da chave primária), ela é estereotipada com <<pk>>. Quando uma coluna é chave estrangeira (ou parte da chave estrangeira), ela é estereotipada com <<fk>>. A Figura 6.5 mostra o exemplo da Figura 6.4 usando o perfil UML descrito.

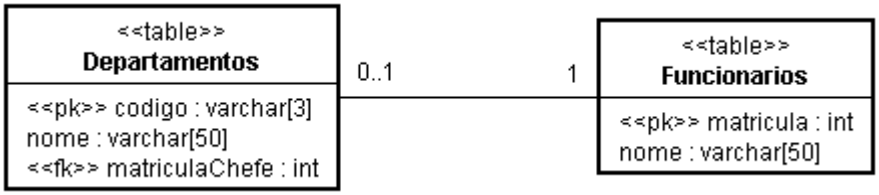


Figura 6.5 – Exemplo de Diagrama Relacional usando Perfil UML.

Ainda que no exemplo anterior as chaves primárias tenham sido escolhidas dentre atributos das entidades (*Departamentos* e *Funcionários*), essa não é necessariamente a melhor escolha. Muito pelo contrário. Para trabalhar bem a manutenibilidade dos sistemas resultantes, sobretudo quando se utiliza o paradigma orientado a objetos, é mais interessante utilizar como chave primária de uma tabela uma coluna criada exclusivamente para este fim, sem nenhum significado no domínio do problema e, portanto, uma coluna que não será manipulada pela camada de Lógica de Negócio, mas apenas pela camada de persistência. Essa abordagem, além de facilitar a manutenção, facilita grandemente o desenvolvimento de infraestruturas genéricas de persistência de objetos, uma vez que todas as chaves primárias são do mesmo tipo de dados e podem ser tratadas uniformemente.

6.2 – Mapeamento Objeto-Relacional

Conforme se pode observar pela Seção 6.1, há diferenças significativas entre o modelo de classes de um projeto orientado a objetos e o modelo relacional. Uma diferença significativa é a forma como relacionamentos são tratados nos modelos de objetos e relacional: objetos armazenam referências a outros objetos (p.ex., endereços de memória), enquanto bancos de dados relacionais ligam tabelas por meio de chaves transpostas. Ainda neste contexto, objetos usam coleções para tratar relacionamentos e atributos multivalorados, enquanto células de uma tabela só podem ter no máximo um valor. Outra diferença substancial é o mecanismo de herança, o qual não é suportado por bancos de dados relacionais. Por outro lado, tabelas têm de ter uma chave primária, enquanto objetos são únicos por essência, ficando transparente para o desenvolvedor a existência de identificadores. Assim, para que a persistência de objetos seja feita em um

banco de dados relacional, é necessário realizar um mapeamento entre esses dois tipos de modelos.

O termo mapeamento objeto-relacional (O/R) é usado tipicamente para referenciar um mapeamento estrutural entre objetos que residem em memória e tabelas em bancos de dados. Ele é fundamental para o projeto da camada de persistência quando um SGBD relacional é utilizado e só deve ser visível na camada de persistência, isolando as demais camadas da arquitetura de software do impacto da tecnologia de bancos de dados.

No mapeamento O/R, as seguintes questões devem ser abordadas: (i) mapeamento de classes e objetos; (ii) mapeamento de herança; e (iii) mapeamento de associações entre objetos.

6.3.1 - Mapeando Classes e Objetos

Quando não há herança, cada classe tipicamente é mapeada em uma tabela e cada instância da classe (objeto) em uma linha dessa tabela. O modelo de classes deve ser normalizado previamente, eliminando-se atributos multivalorados. Nesse processo de normalização das classes, surge uma importante questão. No modelo relacional, toda tabela tem de ter uma chave primária, isto é, uma ou mais colunas, cujos valores identificam univocamente uma linha da mesma. Objetos, por sua vez, têm identidade própria, independentemente dos valores de seus atributos. Assim, deve-se definir que identificador único deve ser usado para designar objetos no banco de dados relacional.

Uma solução possível consiste em observar se há um atributo na classe com a propriedade de identificação única e utilizá-lo, então, como chave primária. Caso não haja um atributo com tal característica, deverá ser criado um. Contudo, conforme discutido na seção 6.1, essa abordagem não é a mais indicada. De fato, ela só deve ser utilizada quando já houver uma base de dados legada sendo utilizada por outros sistemas.

Uma maneira mais eficaz, sobretudo para permitir a construção de componentes mais genéricos de persistência, consiste em dar a cada objeto um atributo chamado de identificador de objeto (*id*). Os *ids* são utilizados como chaves primárias nas tabelas do banco de dados relacional e não devem possuir nenhum significado de negócio. Fowler (2003) denomina essa abordagem de padrão Campo de Identidade (*Identity Field*), no qual o identificador salva a chave primária da correspondente linha da tabela no objeto, de modo a manter um rastro entre o objeto em memória e sua representação como linha de uma tabela do banco de dados.

6.3.2 - Mapeando Herança

Uma vez que os bancos de dados relacionais não suportam o mecanismo de herança, é necessário estabelecer uma forma de mapeamento desse mecanismo. A grande questão no mapeamento da herança diz respeito a como organizar os atributos herdados no banco de dados. Existem três soluções principais para mapear herança em um banco de dados relacional, a saber (AMBLER, 1998) (FOWLER, 2003):

- Utilizar uma tabela para toda a hierarquia;
- Utilizar uma tabela por classe concreta na hierarquia;
- Utilizar uma tabela por classe na hierarquia.

No primeiro caso, a tabela derivada contém os atributos de todas as classes na hierarquia. Fowler (2003) descreve esta solução como o padrão Herança em Tabela Única (*Single Table Inheritance*). A vantagem dessa solução é a simplicidade. Além disso, ela suporta bem o polimorfismo e facilita a designação de *ids*, já que todos os objetos estão em uma única tabela. O problema fundamental dessa solução é que, se as subclasses têm muitos atributos diferentes, haverá muitas colunas que não se aplicam aos objetos individualmente, provocando grande desperdício de espaço no banco de dados. Além disso, sempre que um atributo for adicionado a qualquer classe na hierarquia, um novo atributo deve ser adicionado à tabela. Isso aumenta o acoplamento na hierarquia, pois, se um erro for introduzido durante a adição desse atributo, todas as classes na hierarquia podem ser afetadas e não apenas a classe que recebeu o novo atributo.

No segundo caso, utiliza-se uma tabela para cada classe concreta na hierarquia. Cada tabela derivada para as classes concretas inclui tanto os atributos da classe quanto os de suas superclasses. Fowler (2003) descreve essa solução como o padrão Herança em Tabela Concreta (*Concrete Table Inheritance*). A grande vantagem é a facilidade de processamento sobre as subclasses concretas, já que todos os dados de uma classe concreta estão armazenados em uma única tabela. Da mesma forma que o caso anterior, a designação de *ids* é facilitada, com a vantagem de se eliminar o desperdício de espaço. Contudo, há também desvantagens. Quando uma superclasse é alterada, é necessário alterar as tabelas. Além disso, quando há muito processamento envolvendo a superclasse, há uma tendência de queda do desempenho da aplicação, já que passa a ser necessário manipular várias tabelas ao invés de uma.

A terceira solução é a mais genérica: utiliza-se uma tabela por classe, não importando se concreta ou abstrata. Deve haver uma tabela para cada classe e visões para cada uma das classes derivadas (subclasses). Fowler (2003) descreve essa solução como o padrão Herança em Tabela de Classe (*Class Table Inheritance*). Essa abordagem é a que provê o mapeamento mais simples entre classes e tabelas. É muito mais fácil modificar uma superclasse e acrescentar subclasses, já que é necessário apenas alterar ou acrescentar uma tabela. Uma desvantagem é o grande número de tabelas no banco de dados, uma para cada classe. Além disso, pode levar mais tempo para acessar dados de uma classe, uma vez que pode ser necessário acessar várias tabelas. Podem ser necessárias múltiplas uniões (*joins*) de tabelas para recuperar um único objeto, o que usualmente reduz o desempenho (FOWLER, 2003).

6.3.3 - Mapeando Associações

Conforme discutido na introdução desta seção, há diferenças substanciais na forma de lidar com relacionamentos nos modelos orientado a objetos e relacional. Assim, é fundamental mapear associações em um modelo de objetos para um modelo relacional. Associações 1:1 e 1:N são mapeadas por meio da transposição de chaves. Para tal, aplica-se o padrão Mapeamento de Chave Estrangeira (*Foreign Key Mapping*) (FOWLER, 2003). Já associações N:N requerem uma tabela associativa, com preconiza o padrão Mapeamento de Tabela Associativa (*Association Table Mapping*) (FOWLER, 2003). A seguir, algumas particularidades de cada um desses tipos de associações são discutidas.

Associações 1:1

O mapeamento de associações 1:1 é feito transpondo-se a chave primária de uma tabela para a outra. Quando a associação for obrigatória nas duas extremidades (multiplicidade mínima 1 em ambas as extremidades), pode-se escolher qualquer das chaves para transpor. Quando a associação for opcional em pelo menos uma das duas extremidades (multiplicidade mínima 0), é melhor transpor a chave que dará origem a uma coluna mais densa, isto é, que terá menos valores nulos. Este é o caso do exemplo da Figura 6.5. Outro fator a ser levado em consideração é a navegabilidade da associação. Sempre que possível, deve-se transpor a chave que facilite a navegabilidade escolhida. Por fim, é bom frisar que sempre que a associação for obrigatória (multiplicidade mínima 1), a coluna resultante da chave transposta não poderá ter valores nulos.

Associações 1:N

O mapeamento de associações 1:N é feito transpondo-se a chave primária da tabela correspondente à classe cuja extremidade da associação tem multiplicidade máxima 1 para a tabela que corresponde à classe cuja extremidade da associação tem multiplicidade máxima *n*, como ilustra a Figura 6.6. É bom lembrar que sempre que a associação for obrigatória (multiplicidade mínima 1), a coluna resultante da chave transposta não poderá ter valores nulos, como ilustra a Figura 6.6.

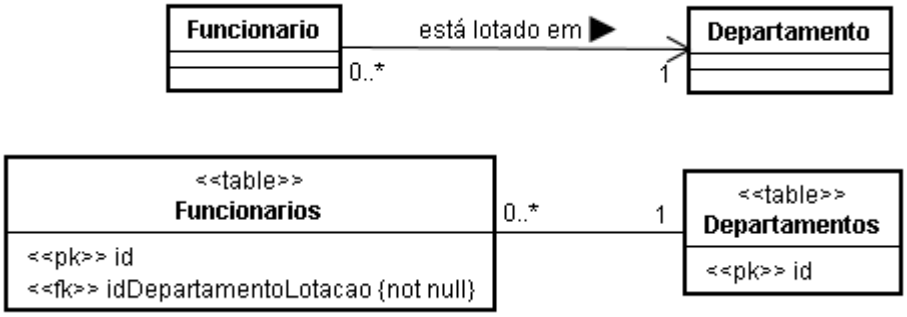


Figura 6.6 – Mapeando associações 1:N.

Associações N:N

O mapeamento de associações N:N é feito utilizando-se uma tabela associativa, uma vez que bancos de dados relacionais não são capazes de manipular diretamente relacionamentos N:N. A Figura 6.7 ilustra este caso. Vale frisar que, muitas vezes, as chaves transpostas são parte da chave primária da tabela associativa. Quando este for o caso, elas são identificadas no modelo como sendo chaves primárias, como ilustra a Figura 6.7.

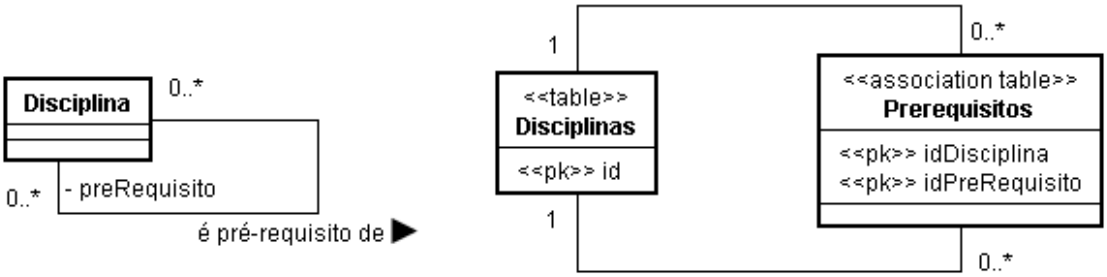


Figura 6.9 – Mapeando associações N:N.

Ainda que os mapeamentos anteriormente discutidos sejam válidos, Waslawick (2004) advoga em favor do uso de tabelas associativas para o mapeamento de quaisquer associações e não somente as associações N:N. Sua argumentação é a seguinte: Embora representar associações 1 ou 0..1 como chaves estrangeiras "possa parecer interessante em um primeiro momento, por evitar a criação de uma nova tabela para representar uma associação, pode se tornar uma dor de cabeça quando for preciso dar manutenção ao sistema. [...]. É mais cômodo deixar as associações como elementos independentes [...] a entremeá-las na estrutura de tabelas responsáveis pela representação dos conceitos. Quando for necessário fazer alterações na estrutura da informação, a vantagem das tabelas associativas é evidente".

6.3 – Padrões Arquitetônicos para a Camada de Gerência de Dados

A Camada de Gerência de Dados - CGD (ou Camada de Persistência) provê a infraestrutura básica para o armazenamento e a recuperação de objetos no sistema. Sua finalidade é isolar os impactos da tecnologia de gerenciamento de dados sobre a arquitetura do software (COAD; YOURDON, 1993).

A despeito da opção de persistência adotada (SGBD relacional, SGBD orientado a objetos, arquivos), há uma importante questão a ser considerada no projeto da CGD: Que classes devem suportar a persistência dos objetos?

Uma alternativa é tornar cada classe a ser persistida (tipicamente, classes do domínio do problema), ao longo de toda a arquitetura de software, responsável por suas próprias atividades de persistência. Essa abordagem é descrita no padrão Registro Ativo (*Active Record*) (FOWLER, 2003), no qual uma classe mantém uma linha de uma tabela ou visão do banco de dados, encapsula o acesso à base de dados e adiciona à lógica de domínio o tratamento da persistência de seus dados. Em outras palavras, esse padrão coloca a lógica de acesso a dados nas classes de domínio do problema e, portanto, não há efetivamente uma camada de persistência, já que não há separação entre lógica de negócio e gerência de dados. Obviamente, nessa abordagem, a arquitetura torna-se completamente dependente da tecnologia de persistência e, se, por exemplo, a organização migrar de um SGBD relacional para outro, essa migração provavelmente vai ter impactos em várias classes do sistema. Em geral, essa abordagem é desaconselhável, só devendo ser aplicada em aplicações muito simples.

Uma abordagem mais elegante consiste em isolar completamente a lógica de negócio e o banco de dados, criando uma camada responsável pelo mapeamento entre

objetos do domínio e tabelas do banco de dados. Os padrões Mapeador de Dados (*Data Mapper*) (FOWLER, 2003) e Objeto de Acesso a Dados (*Data Access Object* - DAO) (BAUER; KING, 2007) adotam esta filosofia, de modo que apenas uma parte da arquitetura de software fica ciente da tecnologia de persistência adotada. Essa parte, o Componente de Gerência de Dados (CGD), serve como uma camada intermediária separando objetos do domínio de objetos de gerência de dados. Via conexões de mensagem, o CGD lê e escreve dados, estabelecendo uma comunicação entre a base de dados e os objetos do sistema. Qualquer código SQL[8] está confinado nessas classes, de modo que não há código desse tipo em outras classes da arquitetura do software.

6.3.1 – O Padrão *Data Mapper*

O padrão Mapeador de Dados (FOWLER, 2003) prescreve uma camada de objetos mapeadores que transferem dados entre objetos em memória e o banco de dados, mantendo-os independentes uns dos outros e dos mapeadores em si. Os objetos de domínio não têm qualquer conhecimento acerca do esquema do banco de dados e não precisam de nenhuma interface para código SQL. De fato, eles não precisam saber sequer que há um banco de dados. O banco de dados, por sua vez, desconhece completamente os objetos que o utilizam.

Em sua versão mais simples, para cada classe a ser persistida em uma tabela, há uma correspondente classe mapeadora (ou classe sombra). Seja o exemplo da Figura 6.8. Nesse exemplo, a classe mapeadora *PersonMapper* intermedeia a classe *Person* da lógica de negócio e o acesso a seus dados no banco de dados, na correspondente tabela (FOWLER, 2003).

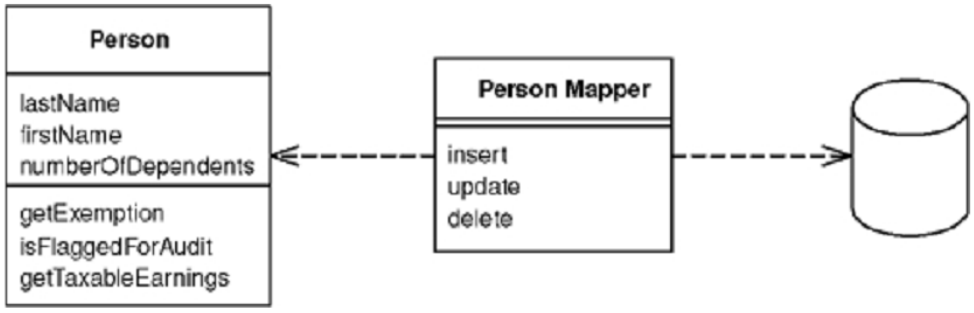


Figura 6.8 – Padrão *Data Mapper* (FOWLER, 2003).

6.3.2 – O Padrão DAO

O padrão DAO define uma interface de operações de persistência, incluindo métodos para criar, recuperar, alterar, excluir e fazer consultas de diversos tipos, relativa a uma particular entidade persistente, agrupando o código relacionado à persistência daquela entidade (BAUER; KING, 2007). A estrutura básica do padrão, como proposto em (BAUER; KING, 2007), é apresentada na Figura 6.9.

Seguindo esse padrão, a camada de persistência é implementada por duas hierarquias paralelas: interfaces à esquerda e implementações à direita. As operações básicas de armazenamento e recuperação de objetos são agrupadas em uma interface

[8] SQL é a abreviatura de *Structured Query Language* (Linguagem Estruturada de Consulta), a linguagem de consulta dos bancos de dados relacionais.

genérica (*GenericDAO*) e uma superclasse genérica (no exemplo da Figura 6.9, *GenericDAOHibernate*). Esta última implementa as operações com uma particular solução de persistência (no caso, Hibernate). A interface genérica é estendida por interfaces para entidades específicas que requerem operações adicionais de acesso a dados. O mesmo ocorre com a hierarquia de classes de implementação. Uma característica marcante desta solução é que é possível ter várias implementações de uma mesma interface DAO (BAUER; KING, 2007).

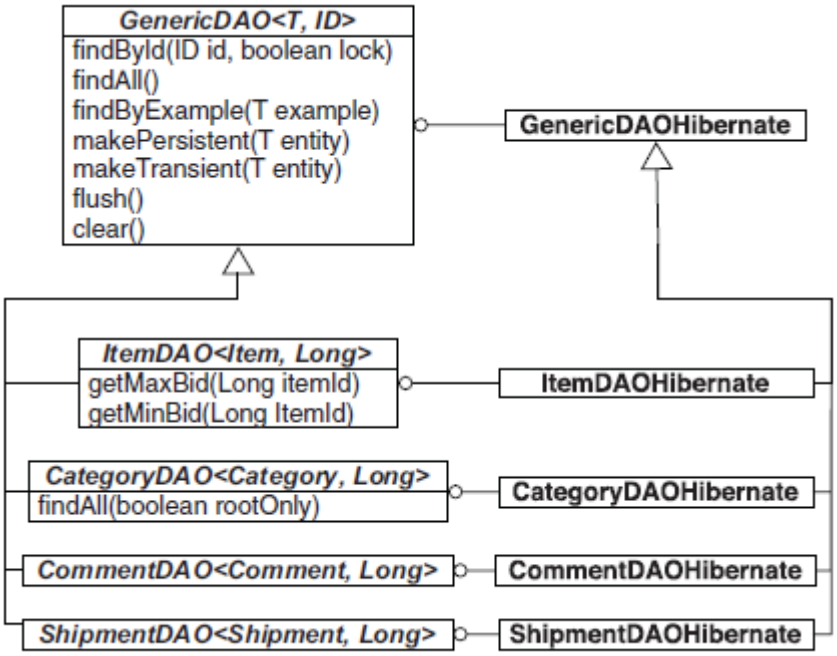


Figura 6.9 – Padrão DAO (BAUER; KING, 2007).

6.4 – *Frameworks* de Persistência

Atualmente há muitos *frameworks* de persistência disponíveis, tais como Hibernate, Java Data Objects – JDO[9] e Oracle Toplink[10], todos esses para a linguagem Java. Esses *frameworks* se encarregam do mapeamento objeto-relacional e tornam o projeto da camada de persistência mais simples.

Usando um *framework* dessa natureza, ao invés de obter os dados dos objetos e mesclá-los a uma *string* de consulta SQL a ser enviada ao SGBD relacional, o desenvolvedor deve informar ao *framework* como transformar objetos/atributos em tabelas/colunas e chamar métodos simples, como *salvar()*, *excluir()* e *recuperarPorId()*, disponíveis no *framework*. A Figura 6.10 ilustra o funcionamento de um *framework* de mapeamento O/R. Em geral, esses *frameworks* disponibilizam também uma linguagem de consulta similar à SQL, porém orientada a objetos, para que consultas possam ser

[9] http://java.sun.com/products/jdo
[10] http://www.oracle.com/technology/products/ias/toplink

realizadas com facilidade (SOUZA, 2005). O Hibernate, por exemplo, disponibiliza a linguagem de consulta HQL.

Usando um *framework* de persistência, é praticamente desnecessário projetar tabelas, formatos de campos e outros aspectos típicos do projeto de bancos de dados relacionais. Contudo, às vezes é necessário efetuar ajustes no mecanismo de persistência para acomodar objetos com características especiais ou satisfazer requisitos não funcionais específicos do sistema (WAZLAWICK, 2004). Assim, é muito importante que o projetista saiba como o mapeamento é tipicamente feito nesses *frameworks*.

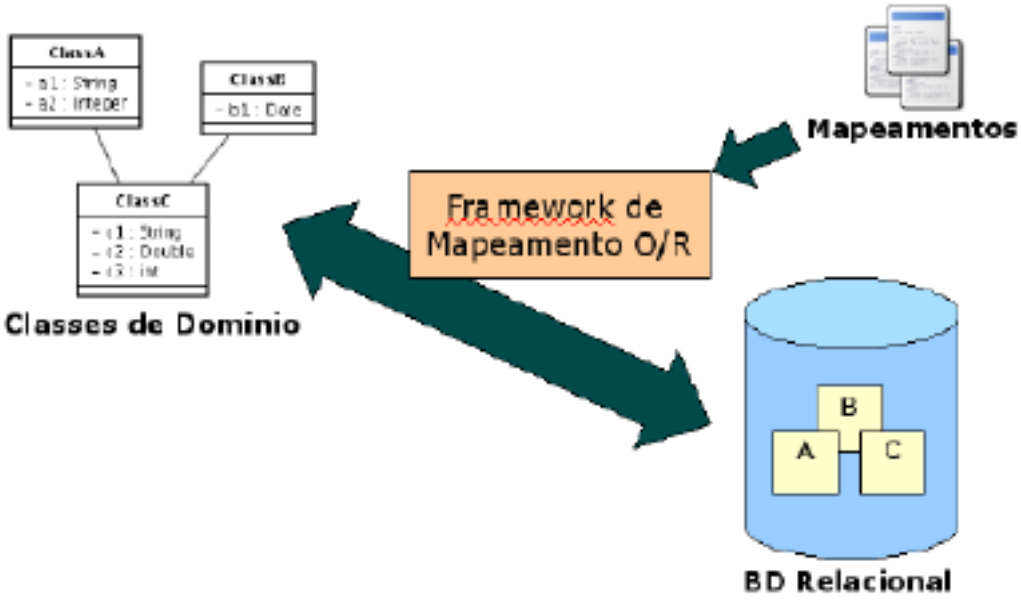


Figura 6.10 – Funcionamento de *framework* de mapeamento O/R (SOUZA, 2005).

Usando o padrão DAO em conjunto com um *framework* de persistência, o projeto da camada de persistência se restringe à definição das interfaces e classes relativas às entidades do domínio a serem persistidas, como ilustrado na Figura 6.9.

Em 2006, dada a existência de vários *frameworks* para persistência de objetos Java, foi lançada *Java Persistence API* (JPA). A JPA permite o armazenamento de dados em bancos de dados relacionais sem se ficar preso a um *framework* de persistência específico. Por meio da JPA, operações de manipulação de tabelas são delegadas para um *framework* de persistência que implemente a JPA. Realizado o mapeamento O/R, é possível utilizar um framework, que suporte JPA (como Hibernate, Oracle TopLink, Kodo, OpenJPA, entre outros), para fazer as devidas inserções, buscas, exclusões e alterações nos dados da aplicação nas tabelas do banco de dados. Dessa forma, o código da aplicação fica independente de *frameworks* de persistência, pois todo o código utilizado para manipular o banco de dados passa a ser da JPA (VILLELA; SILVA, 2010).

Capítulo 7 – Projeto de Classes e Avaliação da Qualidade do Projeto de Software

Uma vez definidos e refinados todos os componentes da arquitetura de software, deve-se passar ao projeto detalhado das classes, quando se projetam detalhadamente os atributos, as associações e os métodos de cada classe. Neste momento, deverão ser definidos, dentre outros, as interfaces dos métodos, os algoritmos usados para implementá-los e a visibilidade de atributos, associações e métodos.

Inicialmente, uma descrição do protocolo de cada classe deve ser estabelecida, indicando o conjunto de métodos da classe acessíveis a objetos de outras classes, i.e. sua interface pública. A seguir, deve-se fazer uma descrição da implementação da classe, provendo detalhes internos necessários para a implementação, mas não necessários para a comunicação entre objetos.

Concluído o projeto de classes, a fase de projeto pode ser considerada finalizada. Contudo, antes de dar como concluída essa fase, é imprescindível avaliar a qualidade do Documento de Projeto (ou Especificação de Projeto), artefato contendo os modelos produzidos e informações relevantes acerca das decisões tomadas. Na verdade, não é necessário concluir a fase de projeto para avaliar a qualidade do que está sendo produzido nessa fase. Essa avaliação pode, e deve, ser conduzida em pontos de controle demarcados previamente, visando avaliar subprodutos da fase de projeto, tais como a arquitetura de software e os modelos dos componentes da arquitetura.

Este capítulo discute brevemente o projeto detalhado das classes e a avaliação da qualidade do projeto de software como um todo. Ele está estruturado da seguinte forma: a Seção 7.1 trata de aspectos relacionados ao projeto de atributos e associações; a Seção 7.2 trata de aspectos relacionados ao projeto de métodos; finalmente, a Seção 7.3 discute brevemente a avaliação do projeto de software.

7.1 – Projetando Atributos e Associações

Tipicamente, classes de um projeto orientado a objetos são diretamente implementadas na forma de classes em uma linguagem de programação. As exceções ficam por conta de páginas Web tradicionais, caso as mesmas tenham sido projetadas como classes do Componente de Interface com o Usuário.

Atributos e associações são implementados como variáveis de instância da respectiva classe. No caso de atributos monovalorados, o tipo da variável de instância é um tipo básico da linguagem de programação adotada, tal como *int*, *float* e *string*, ou um tipo de dado de domínio previamente estabelecido, conforme definido no projeto do componente correspondente. No caso de associações navegáveis com multiplicidade máxima 1, a classe de origem da associação deve ter uma variável de instância do tipo da classe de destino.

Recomenda-se fortemente que todos os atributos e associações sejam definidos como privados. Como decorrência disso, a classe deve implementar métodos para acessar (*get*) e alterar (*set*) os valores de seus atributos. Aconselha-se que esses métodos sejam nomeados com o nome do atributo/associação precedido pelas palavras *get* e *set*, respectivamente. Esse é um padrão indicado, pois é adotado pela maioria das ferramentas CASE durante a geração de código. Além disso, quando a camada de persistência é construída usando o *framework* Hibernate, p.ex., essa nomeação é fundamental para o funcionamento do mecanismo de persistência.

Wazlawick (2004) reforça que, para viabilizar o funcionamento do mecanismo de persistência, é fundamental que os atributos sejam acessados e modificados exclusivamente pelas operações *get* e *set*. Em hipótese alguma outro método, mesmo sendo da própria classe, poderá acessar ou modificar tais variáveis diretamente.

Quando uma classe possui um atributo obrigatório ou uma associação navegável de multiplicidade mínima 1, seu método construtor deve ter um parâmetro do tipo definido a ser atribuída à variável de instância correspondente, de modo a manter a consistência. Quando isso não ocorrer no método construtor, é necessário que a criação do objeto aconteça no contexto de uma transação que envolva também a atribuição do valor (método *set*).

No caso de atributos multivalorados ou associações navegáveis com multiplicidade máxima *n*, a implementação é feita tipicamente por meio de uma variável de instância de um tipo de uma estrutura de dados que comporte uma coleção de elementos, tal como o tipo Conjunto (*Set*). Para facilitar a identificação no código de que se trata de um conjunto de valores, recomenda-se utilizar o nome da variável no plural. Além disso, é importante garantir que a classe usada para implementar o tipo Conjunto tenha métodos para adicionar e remover elementos do conjunto. Nesse caso, os métodos *get* e *set* são usados para obter e atribuir a coleção inteira e não um elemento da coleção. Por fim, caso a coleção seja ordenada, deve-se utilizar uma estrutura de dados que trabalhe a ordenação, tal como uma lista.

Quando uma associação for bidirecional, i.e., navegável nos dois sentidos, ela deve ser implementada em ambas as classes (seguindo as diretrizes apresentadas anteriormente). Contudo, atenção especial deve ser dada ao controle da redundância, de modo a se manter a consistência.

7.2 – Projetando Métodos

No que concerne aos métodos, é necessário definir os tipos e estruturas de dados para as interfaces (parâmetros e retorno), bem como uma especificação do algoritmo de cada método. No caso de operações complexas, é uma boa opção dividi-las, criando métodos de nível mais baixo, estes privadas à classe. O projeto algorítmico de uma operação pode revelar a necessidade de variáveis locais aos métodos ou de variáveis globais à classe para tratar detalhes internos. Definir as estruturas de dados a serem utilizadas para essas variáveis também é parte do projeto detalhado dos métodos.

Para o projeto dos algoritmos, pode-se utilizar pseudocódigo. Contudo, há que se avaliar a utilidade de especificações detalhadas de algoritmos e se há necessidade de fazer essas especificações para todos os métodos. Como regra geral, essa estratégia só

deve ser aplicada a métodos em que não é intuitivo saber o que se espera deles ou quando há regras de negócio específicas a serem tratadas.

Durante o projeto detalhado dos métodos, deve-se levar em conta que projetistas muitas vezes não conhecem os recursos da linguagem de programação adotada tão bem quanto os programadores. Por exemplo, as linguagens de programação oferecem uma variedade de estruturas de dados, tais como vetores, listas, filas, pilhas, conjuntos, mapas etc., e o projetista pode não fazer a melhor escolha em relação às estruturas a serem utilizadas. O mesmo ocorre em relação ao projeto dos algoritmos. De fato, o projeto de métodos está na tênue fronteira entre o projeto detalhado e a implementação. Assim, pode ser mais produtivo deixá-lo por conta dos programadores, supervisionados pelos projetistas.

7.3 – Avaliando a Qualidade do Documento de Projeto

A qualidade de um produto de software não se atinge de forma espontânea. Ela deve ser construída ao longo do processo de desenvolvimento de software. Para tal, é necessário avaliar a qualidade dos diversos produtos intermediários do processo de software, dentre eles o Documento de Projeto de Software ou Especificação de Projeto.

O Documento de Projeto é um documento valioso. Ele servirá de base para as etapas de implementação e testes. Assim, a qualidade desse documento é vital para a qualidade do produto de software resultante.

Diversos aspectos do projeto devem ser avaliados. É importante lembrar que há vários projetos possíveis que podem implementar corretamente um conjunto de requisitos. Um bom projeto equilibra custo e benefício, de modo a minimizar o custo total do sistema ao longo de seu tempo de vida total. Coad e Yourdon (1993) propõem alguns critérios para avaliação da qualidade de projetos orientados a objetos, dentre eles:

- *Acoplamento*: conforme discutido no Capítulo 2, acoplamento diz respeito ao grau de interdependência entre componentes de software. O objetivo é minimizar o acoplamento, isto é, tornar os componentes tão independentes quanto possível. No projeto orientado a objetos, deve ser avaliado tanto o acoplamento entre classes como o acoplamento entre subsistemas. A meta é minimizar o número de mensagens trocadas e a complexidade e o volume de informação nas mensagens.
- *Coesão*: define como as atividades de diferentes componentes de software estão relacionadas umas com as outras. Vale a pena ressaltar que coesão e acoplamento são interdependentes e, portanto, uma boa coesão, geralmente, conduz a um pequeno acoplamento. No projeto orientado a objetos, três níveis de coesão devem ser verificados:
    - Coesão de métodos individuais: um método deve executar uma e somente uma função;
    - Coesão de classes: atributos e operações encapsulados em uma classe devem ser altamente coesos, isto é, devem estar estreitamente relacionados; e

  - Coesão de hierarquias de classes: a coesão de uma hierarquia pode ser avaliada examinando-se até que extensão uma subclasse redefine ou cancela atributos e métodos herdados da superclasse. Além disso, é fundamental garantir que subclasses são realmente especializações das superclasses, evitando-se a herança por conveniência.

- *Clareza*: um projeto deve ser passível de entendimento por programadores, testadores e outros projetistas.
- *Reutilização*: bons projetos devem ser fáceis de serem reutilizados e devem, sempre que possível, reutilizar porções de projeto previamente elaborados e padrões de projeto (padrões arquitetônicos e *design patterns*) consagrados.
- *Efetivo Uso da Herança*: para sistemas médios, hierarquias de classe não devem ter mais do que sete níveis de generalização-especialização. Projetos com uso intensivo de herança múltipla devem ser evitados, pois são mais difíceis de serem entendidos e, consequentemente, de serem reutilizados e mantidos.
- *Protocolo de Mensagens Simples*: protocolos de mensagem complexos são uma indicação comum de acoplamento excessivo entre classes. Assim, a passagem de muitos parâmetros deve ser evitada.
- *Métodos Simples*: os métodos que implementam as operações de uma classe devem ser mantidos pequenos. Se um método envolve muito código, há uma indicação de que as operações da classe foram pobremente divididas.
- *Habilidade de "avaliar por cenário"*: é importante que um projeto possa ser avaliado a partir de um cenário particular escolhido. Revisores devem poder representar o comportamento de classes e objetos individuais e, assim, verificar o comportamento dos objetos nas circunstâncias desejadas. Essa característica é igualmente importante para testar posteriormente o produto de software.

Assim como os demais documentos produzidos ao longo do processo de software, o Documento de Projeto deve ser revisado. Durante uma revisão desse documento, os seguintes aspectos devem ser observados:

- Aderência a padrões de documento de projeto estabelecidos pela organização.
- Aderência a padrões de nomenclatura estabelecidos pela organização, incluindo nomes de classes, atributos, métodos, pacotes etc.
- Coerência com os modelos de análise e de especificação de requisitos.

Do ponto de vista de coerência entre modelos, os seguintes aspectos devem ser observados:

- As classes da Camada de Lógica de Negócio devem ser necessárias e suficientes para cumprir as responsabilidades apontadas pelos casos de uso do documento de especificação de requisitos, agora já com uma perspectiva de implementação, i.e., levando-se em conta os requisitos não funcionais.

- As classes da Camada de Interface com o Usuário devem ser necessárias e suficientes para permitir o acesso e a realização de todos os casos de uso do documento de especificação de requisitos.
- As classes da Camada de Gerência de Dados devem ser necessárias e suficientes para tratar do armazenamento e recuperação de objetos de todas as classes persistentes do sistema (tipicamente, as classes do Componente de Domínio do Problema).
- Alterações não decorrentes da tecnologia, mas da detecção de um erro na especificação de requisitos ou na análise devem ser feitas nos correspondentes modelos de especificação e análise de requisitos. Não basta alterar o documento de projeto. Lembre-se que é fundamental manter a coerência entre os modelos de análise e projeto.

Referências

ALBIN, S.T., *The Art of Software Architecture: Design Methods and Techniques*, John Wiley & Sons, 2003.

AMBLER, S., *Análise e Projeto Orientados a Objetos*. IBPI Press, 1998.

AMBLER, S.W., *Modelagem Ágil*, Artmed, 2004.

BASS, L., CLEMENTS, P., KAZMAN, R., *Software Architecture in Practice*, Second edition, Addison Wesley, 2003.

BAUER, C., KING, G., *Java Persistence with Hibernate*, Manning, 2007.

BLAHA, M., RUMBAUGH, J., *Modelagem e Projetos Baseados em Objetos com UML 2*, Elsevier, 2006.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., *UML Guia do Usuário*, 2a edição, Elsevier Editora, 2006.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., *Pattern-Oriented Software Architecture: A System of Patterns*, Volume 1, Wiley, 1996.

CARDOSO, A.C., MARQUES, A.S., *Sistemas Baseados em Regras*, 2007. Disponível em http://mestradosiad.blogspot.com/2007/11/sistemas-baseados-em-regras.html.

CASTELEYN, S., DANIEL, F., DOLOG, P., MATERA, M., *Engineering Web Applications*, Springer, 2009.

COAD, P., YOURDON, E., *Projeto Baseado em Objetos*, Editora Campus, 1993.

FOWLER, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

FOWLER, M., *Anemic Domain Model*, 2003a. Disponível em http://www.martinfowler.com/bliki/AnemicDomainModel.html. Último acesso em 06.05.2010.

FRAGMENTAL Bliki, *Evitando VOs e BOs*, 2007. Disponível em http://www.fragmental.com.br/wiki/index.php?title=Evitando_VOs_e_BOs. Último acesso em 06.05.2010.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J.M., *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GIMENES, I. M. S., HUZITA, E. H. M., *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, Ciência Moderna, 2006.

GORTON, I., *Essential Software Architecture*, Springer, 2006.

ISO/IEC 9126-1, Software Engineering - Product Quality - Part 1: Quality Model, 2001.

ISO/IEC TR 9126-2:2003, Software Engineering – Product Quality – Part 2: External Metrics, 2003.

ISO/IEC TR 9126-3:2003, Software Engineering – Product Quality – Part 3: Internal Metrics, 2003.

KOSCIANSKI, A., SOARES, M.S., *Qualidade de Software*, Novatec, 2006.

LARMAN, C., *Utilizando UML e Padrões*, 3ª edição, Bookman, 2007.

MARINHO, E.H., RESENDE, R.F., "Extensão de um Metamodelo de Aplicações Baseadas na Web considerando Ájax", Anais do VII Simpósio Brasileiro de Sistemas de Informação, Salvador, Brasil, pp. 141 – 152, 2011.

MENDES, A., *Arquitetura de Software: desenvolvimento orientado para arquitetura*, Editora Campus, 2002.

MURUGESAN, S., GINIGE, A., "Web Engineering: Introduction and Perspectives", In: *Software Engineering for Modern Web Applications: Methodologies and Technologies*, pp. 1-24, IGI Global, 2008.

NUDELMAN, G., *Padrões de Projeto para o Android: Soluções de Projetos de Interação para Desenvolvedores*. Wiley Novatec, 2013.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, Prentice Hall, 2ª edição, 2004.

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

PRESSMAN, R.S., LOWE, D., *Web Engineering: A Practitioner's Approach*, Mc Graw Hill, 2009.

RUBLE, D.A., *Practical Analysis and Design for Client/Server and GUI Systems*, Yourdon Press Computing Series, 1997.

SHAW, M., GARLAN, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

SOUZA, V.E.S., FrameWeb: um Método baseado em *Frameworks* para o Projeto de Sistemas de Informação *Web*, Dissertação de Mestrado, Programa de Pós-Graduação em Informática, UFES, Universidade Federal do Espírito Santo, 2005.

VILLELA, M., SILVA, E., JPA 2.0 - Persistência a Toda Prova, Java Magazine 81, 2010.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

XAVIER, C.M.S., PORTILHO, C., *Projetando com Qualidade a Tecnologia de Sistemas de Informação*, Livros Técnicos e Científicos Editora, 1995.

Anexo A – Padrões de Projeto

Projetistas experientes fazem bons projetos, normalmente reutilizando soluções que já funcionaram no passado. Quando encontram uma boa solução, eles a reutilizam várias vezes. Projetistas novatos, por outro lado, tendem a resolver os problemas usando apenas os princípios básicos. Demora muito tempo até que os novatos ganhem experiência e aprendam a fazer bons projetos (GAMMA et al., 1995).

Os padrões (*patterns*) visam capturar esse conhecimento, procurando torná-lo mais geral e amplamente aplicável, desvinculando-o das especificidades de um determinado projeto ou sistema. Um padrão é uma solução testada e aprovada para um problema geral e apresenta diretrizes sobre quando usá-lo, bem como vantagens e desvantagens de seu uso. Um padrão já foi cuidadosamente considerado por outras pessoas e aplicado diversas vezes na solução de problemas anteriores de mesma natureza. Assim, tende a ser uma solução de qualidade, com maiores chances de estar correto e estável do que uma solução nova, específica, ainda não testada (BLAHA; RUMBAUGH, 2006). Padrões relativos à fase de projeto ajudam projetistas a reutilizar projetos bem sucedidos, ao basear novos projetos em experiências anteriores. Um projetista familiarizado com tais padrões pode aplicá-los imediatamente em problemas de projeto sem ter de redescobri-los (GAMMA et al., 1995).

Padrões permitem reusar arquiteturas e projetos bem sucedidos. Um padrão sistematicamente nomeia, explica e avalia uma porção de projeto importante e recorrente. Ao expressar abordagens comprovadas na forma de padrões, esse conhecimento torna-se acessível a projetistas de novos sistemas (GAMMA et al., 1995).

No que se refere a padrões relativos à fase de projeto, há dois principais tipos de padrões: padrões arquitetônicos, que definem uma estrutura global do sistema, e padrões de projeto (*design patterns*), que descrevem uma estrutura comumente recorrente de componentes que se comunicam, a qual resolve um problema de projeto geral dentro de um particular contexto (GAMMA et al., 1995).

Este anexo apresenta uma visão geral do catálogo de padrões de projeto proposto por Gamma et al. (1995), apresentando alguns dos padrões propostos nesse catálogo.

A.1 – O Catálogo de Gamma et al. (1995)

Um dos principais catálogos de padrões de projeto orientado a objetos publicados até o momento é o apresentado em (GAMMA et al., 1995). A descrição dos padrões nesse catálogo consiste de:

- **Nome**: nome dado ao padrão no catálogo.
- **Classificação**: é feita segundo dois critérios: propósito e escopo. O propósito reflete o que o padrão faz, isto é, sua funcionalidade. De acordo com esse critério, um padrão pode ser: criativo (diz respeito ao processo de criação de objetos); estrutural (lida com a composição de classes ou objetos); ou comportamental (caracteriza os meios pelos quais classes ou objetos interagem e distribuem responsabilidades). O escopo, por sua vez, especifica se o padrão está centrado em classes (e neste caso, faz intenso uso de herança) ou em objetos (e, portanto, mais apoiado em associações).
- **Intenção (Propósito)**: descreve sucintamente o que faz o padrão, seu propósito e o problema endereçado.
- **Também conhecido como**: apresenta outros nomes pelos quais o padrão é conhecido (se houver).
- **Motivação**: apresenta um cenário que ilustra o problema endereçado pelo padrão e como a estrutura proposta pelo padrão resolve esse problema.
- **Aplicabilidade**: trata de situações nas quais o padrão pode ser aplicado e como reconhecer essas situações.
- **Estrutura**: apresenta o modelo de classes do padrão e, opcionalmente, diagramas de interação para ilustrar sequências de requisições e colaborações entre objetos.
- **Participantes**: fornece uma descrição das classes e/ou objetos que participam do padrão e suas responsabilidades.
- **Colaborações**: descreve como os participantes colaboram para realizar suas responsabilidades.
- **Consequências**: trata dos comprometimentos e resultados quando se aplica o padrão, tanto positivos como negativos.
- **Implementação**: discute armadilhas e sugestões na implementação do padrão, bem como técnicas e questões específicas de linguagem.
- **Código-Exemplo**: apresenta fragmentos de código em C++ ou Smalltalk que ilustram como o padrão pode ser implementado.
- **Usos conhecidos**: apresenta exemplos de uso do padrão encontrados em sistemas reais.
- **Padrões relacionados**: faz referência a outros padrões proximamente relacionados com o padrão em questão, discutindo diferenças. Relaciona, também, outros padrões que devem ser utilizados juntamente com este.

Tendo em vista a classificação proposta por Gamma et al. (1995), é possível apontar os objetivos gerais de cada grupo de padrões. Quanto ao propósito, os seguintes objetivos são válidos:

- **Padrão Criativo**: abstrai o processo de instanciação (criação) de objetos, ajudando a tornar um sistema independente de como seus objetos são criados, compostos e representados.
  - Padrão Criativo de Classe: utiliza herança para variar a classe instanciada, adiando alguma parte da criação de objetos para subclasses.
  - Padrão Criativo de Objeto: delega a instanciação de um objeto para outro objeto ou adia alguma parte da criação de um objeto para outro objeto.
- **Padrão Estrutural**: diz respeito a como classes e objetos são compostos para formar estruturas maiores.
  - Padrão Estrutural de Classe: utiliza herança para compor classes.
  - Padrão Estrutural de Objeto: descreve meios de compor objetos a partir de outros objetos, visando obter nova funcionalidade. A flexibilidade adicional da composição de objetos advém da habilidade de alterar uma composição em tempo de execução, o que é impossível com a composição estática de classes.
- **Padrão Comportamental**: diz respeito a algoritmos e a atribuição de responsabilidades entre objetos.
  - Padrão Comportamental de Classe: utiliza herança para distribuir comportamento entre classes, ou seja, para descrever algoritmos e fluxos de controle.
  - Padrão Comportamental de Objeto: utiliza composição de objetos para distribuir o comportamento. Descreve como um grupo de objetos coopera para realizar uma tarefa que nenhum objeto poderia realizar sozinho.

A Tabela A.1 apresenta o catálogo de padrões proposto por Gamma et al. (1995). Na sequência, é apresentada uma breve descrição de cada um dos padrões.

Tabela A.1 – Catálogo de Padrões proposto em (GAMMA et al., 1995).

	Propósito		
Escopo	Criativo	Estrutural	Comportamental
Classe	Método-Fábrica	Adaptador (classe)	Interpretador Método Modelo
Objeto	Construtor Fábrica Abstrata Protótipo Singular	Adaptador (objeto) Composto Decorador Fachada Peso-Mosca Ponte Procurador (*Proxy*)	Cadeia de Responsabilidade Comando Iterador Mediador Memorial Observador Estado Estratégia Visitador

Padrões de Classe:

- **Método-Fábrica (*Factory Method*)**: define uma interface para a criação de objetos, mas deixa que uma classe adie a instanciação para suas subclasses.
- **Adaptador (*Adapter*)**: converte a interface de uma classe em outra interface, permitindo que classes trabalhem em conjunto, quando isto não seria possível por causa da incompatibilidade de interfaces.
- **Método Modelo (*Template Method*)**: define o esqueleto de um algoritmo em uma operação, adiando alguns de seus passos para as subclasses, permitindo que as subclasses redefinam certos passos do algoritmo sem alterar sua estrutura.
- **Interpretador (*Interpreter*)**: Dada uma linguagem, define uma representação para sua gramática, junto com um interpretador que utiliza essa representação para interpretar sentenças na linguagem.

Padrões de Objetos:

- **Construtor (*Builder*)**: separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção pode criar diferentes representações.
- **Fábrica Abstrata (*Abstract Factory*)**: provê uma interface para a criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.
- **Protótipo (*Prototype*)**: especifica os tipos de objetos que podem ser criados a partir de uma instância prototípica e cria novos objetos copiando este protótipo.

- **Singular (*Singleton*)**: garante que uma classe possui uma única instância e provê um ponteiro global para acessá-la.
- **Composto (*Composite*)**: compõe objetos em estruturas de árvore para representar hierarquias todo-parte, permitindo que clientes tratem objetos individuais e compostos uniformemente.
- **Decorador (*Decorator*)**: anexa responsabilidades adicionais a um objeto dinamicamente, permitindo estender sua funcionalidade.
- **Fachada (*Facade*)**: provê uma interface unificada para um conjunto de interfaces em um subsistema. Define uma interface de nível mais alto para o subsistema, tornando-o mais fácil de ser usado.
- **Peso-Mosca (*Flyweight*)**: utiliza compartilhamento para suportar eficientemente um grande número de objetos de granularidade muito fina.
- **Ponte (*Bridge*)**: desacopla uma abstração de sua implementação, de modo que ambas possam variar independentemente.
- **Procurador (*Proxy*)**: provê um substituto/procurador (*proxy*) que tem autorização para controlar o acesso a um objeto.
- **Cadeia de Responsabilidades (*Chain of Responsibility*)**: evita o acoplamento entre o objeto emissor de uma mensagem e o receptor, dando chance para mais de um objeto tratar a solicitação. Encadeia os objetos receptores e passa a mensagem adiante na cadeia até que um objeto a trate.
- **Comando (*Command*)**: encapsula uma requisição como um objeto, permitindo, assim, parametrizar clientes com diferentes requisições e desfazer operações (comando *undo*).
- **Iterador (*Iterator*)**: provê um meio de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação básica.
- **Mediador (*Mediator*)**: define um objeto que encapsula como um conjunto de objetos interage.
- **Memento (*Memento*)**: sem violar o encapsulamento, captura e externaliza o estado interno de um objeto de modo que se possa posteriormente restaurar o objeto para este estado.
- **Observador (*Observer*)**: define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- **Estado (*State*)**: permite que um objeto altere o seu comportamento quando seu estado interno muda, fazendo parecer que o objeto mudou de classe.
- **Estratégia (*Strategy*)**: define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Deste modo, o algoritmo varia independentemente dos clientes que o utlizam.
- **Visitador (*Visitor*)**: representa uma operação a ser executada sobre os elementos de uma estrutura de um objeto. Permite definir uma nova operação sem alterar as classes dos elementos sobre as quais ele opera.

Gamma et al. (1995) sugerem que, para se utilizar um padrão do catálogo, os seguintes passos devem ser seguidos:

1. Leia o padrão uma vez para obter uma visão geral, concentrando a atenção nas seções de Aplicabilidade e Consequências para garantir que este é o padrão certo para o seu problema.
2. Volte e estude as seções de Estrutura, Participantes e Colaborações. Tenha a certeza de que compreendeu as classes e objetos no padrão e como se relacionam entre si.
3. Olhe a seção Código de Exemplo para ver um exemplo concreto do padrão em código. Isto vai ajudá-lo a aprender a implementar o padrão.
4. Escolha nomes para os participantes (classes e/ou objetos) do padrão que sejam significativos no contexto de sua aplicação. Os nomes em um padrão de projeto são geralmente muito abstratos para aparecerem diretamente em uma aplicação. Contudo, é útil incorporar o nome do participante do padrão de projeto ao seu nome na aplicação, de modo a tornar o padrão mais explícito na implementação.
5. Defina as classes.
6. Defina nomes específicos da aplicação para as operações no padrão.
7. Implemente as operações para realizar as responsabilidades e colaborações do padrão.

Padrões de projeto não devem ser aplicados indiscriminadamente. Frequentemente, eles alcançam flexibilidade e variabilidade através da introdução de níveis adicionais de indireção que podem complicar um projeto e/ou resultar em queda de desempenho.

Um padrão de projeto só deve ser aplicado quando a flexibilidade que ele proporciona for realmente necessária. A seção de Consequências é muito útil para a avaliação dos benefícios e obrigações de um padrão.

Padrões de projeto são bastante úteis para a criação de projetos robustos, aptos a suportar mudanças, garantindo que o sistema pode ser alterado de certas maneiras. Cada padrão permite que algum aspecto da estrutura do sistema varie de forma independente de outros aspectos, tornando o sistema mais robusto para um particular tipo de alteração.

A seguir, são parcialmente apresentados alguns dos padrões de projeto propostos em (GAMMA et al., 1995).

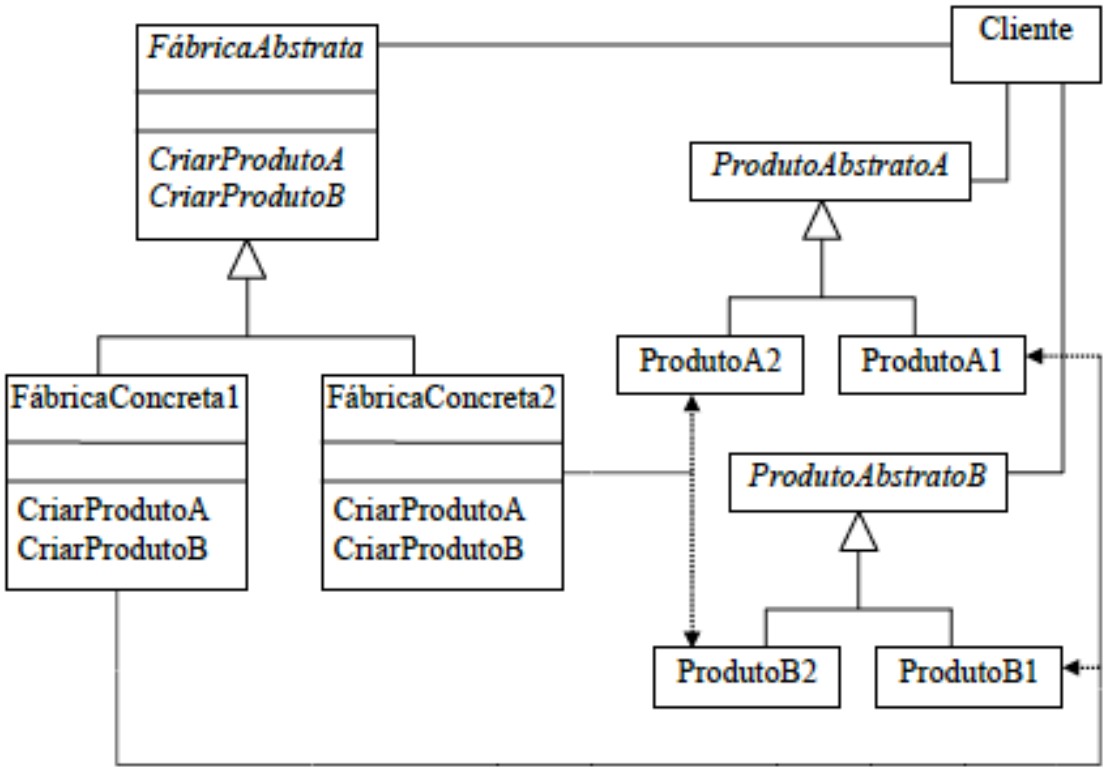
A 1.1 - Fábrica Abstrata (*Abstract Factory*)

- Classificação: Padrão Criativo de Objeto
- Propósito: Prover uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.
- Também conhecido como: Kit.
- Motivação: *Toolkit* de Interface Gráfica com o Usuário, suportando diferentes padrões de apresentação (Motif, Presentation Manager,...). Cada padrão de apresentação define diferentes comportamento e aparência para objetos de

interface, tais como janelas, botões, barras de scroll, etc. Para ser portável ao longo de diferentes padrões de apresentação, uma aplicação não pode se comprometer com um padrão específico.

- Aplicabilidade:
  - Sistema deve ser independente de como seus produtos são criados, compostos e representados.
  - Sistema deve ser configurado com uma dentre várias famílias de produtos.
  - Uma família de produtos relacionados foi projetada para ser usada em conjunto e esta restrição tem de ser garantida.

- Estrutura:

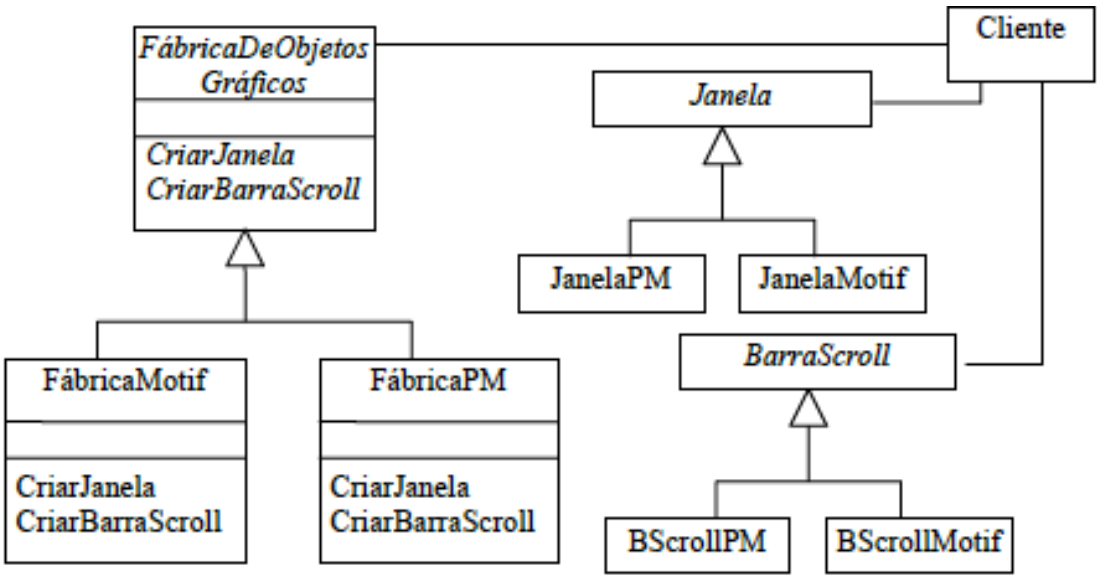


- Participantes:
  - Fábrica Abstrata: declara uma interface para operações criam objetos-produto abstratos;
  - Fábrica Concreta: implementa as operações para criar objetos-produto concretos;
  - Produto Abstrato: declara uma interface para um tipo de objeto produto.
  - Produto Concreto: implementa a interface abstrata de Produto Abstrato e define um objeto-produto a ser criado pela Fábrica Concreta correspondente.

  - Cliente: utiliza apenas as interfaces declaradas por Fábrica Abstrata e Produto Abstrato.

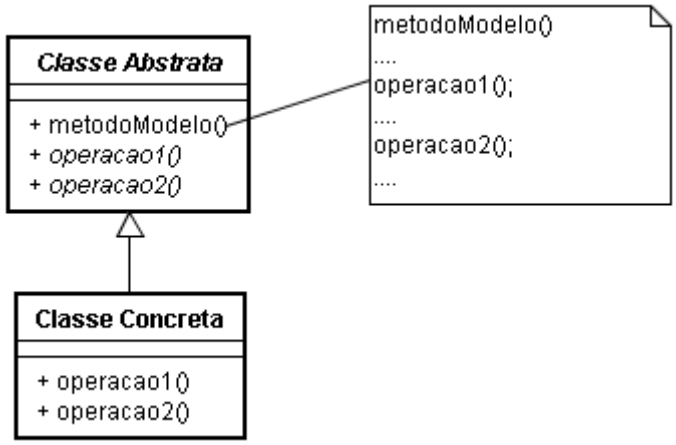
- Colaborações: Fábrica Abstrata adia a criação de objetos-produto para suas subclasses Fábricas Concretas.

- Consequências:
  - Isola classes concretas: uma vez que uma fábrica encapsula a responsabilidade e o processo de criação de objetos-produto, ela isola clientes das classes de implementação.
  - Fica mais fácil a troca de uma família de produtos, bastando trocar a fábrica concreta usada pela aplicação.
  - Promove consistência entre produtos. Quando objetos-produto em uma família são projetados para trabalhar juntos, é importante que uma aplicação utilize apenas objetos desta família.
  - O suporte a novos tipos de produtos é dificultado, já que a interface da Fábrica Abstrata fixa o conjunto de produtos que podem ser criados. Para suportar novos tipos de produtos, é necessário alterar a interface da fábrica, o que envolve alterações na Fábrica Abstrata e em todas as suas subclasses.



A 2.2 - Método Modelo (*Template Method*)

- Classificação: Padrão Comportamental de Classe.
- Propósito: Definir o esqueleto de um algoritmo em uma operação, adiando alguns passos para as subclasses.
- Aplicabilidade: Para implementar apenas uma vez as partes que não variam de um algoritmo e deixar a cargo das subclasses a implementação do comportamento variável.
- Estrutura:

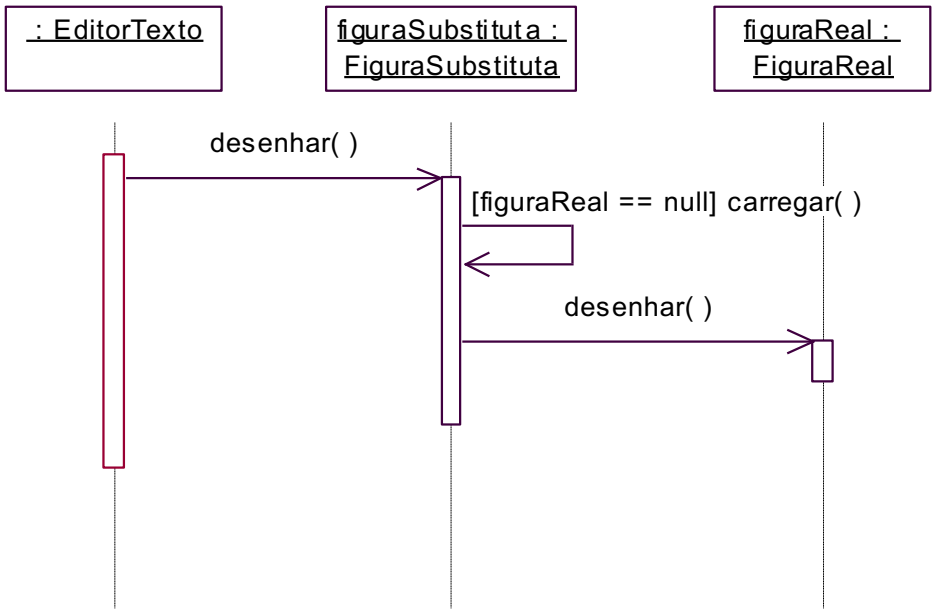


- Participantes:
  - Classe Abstrata: implementa um método modelo, definindo o esqueleto de um algoritmo e define operações primitivas abstratas que as subclasses concretas têm de definir para implementar os passos do algoritmo;
  - Classe Concreta: implementa as operações primitivas para realizar passos do algoritmo que são específicos da subclasse.
- Colaborações: A Classe Concreta conta com a Classe Abstrata que implementa os passos que não variam do algoritmo.
- Padrões Relacionados:
  - Método Fábrica: métodos-fábrica normalmente são chamados por métodos-modelo;
  - Estratégia: enquanto os métodos-modelo utilizam herança para variar partes de um algoritmo, as estratégias usam delegação para variar o algoritmo inteiro.

A 2.3 - Procurador (*Proxy*)

- Classificação: Padrão Estrutural de Objeto
- Propósito: provê um substituto/procurador que tem autorização para controlar o acesso ao objeto.

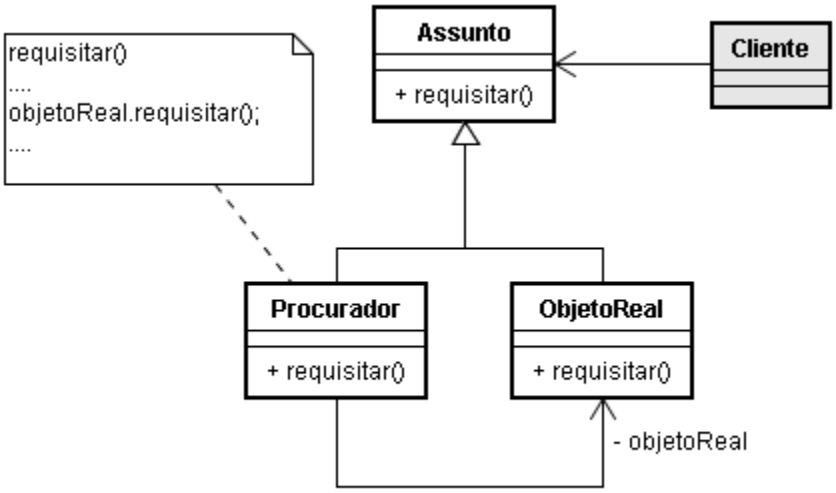
- Também conhecido como: Substituto, *Proxy*.
- Motivação: Uma razão para se controlar o acesso a um objeto é tentar adiar o alto custo de criação e inicialização deste objeto até o momento em que ele for ser realmente utilizado. Considere um editor de texto que pode embutir objetos gráficos em um documento. Alguns desses objetos, como figuras complexas, podem ter um alto custo de criação. Contudo, a abertura de um documento deve ser rápida. Assim, é desejável evitar a criação de todos esses objetos no momento em que o documento é aberto, até porque muitos deles não estarão visíveis ao mesmo tempo. Esta restrição sugere a criação dos objetos complexos sob demanda, isto é, o objeto só será criado no momento em que sua imagem se tornar visível. Mas o que colocar no documento no lugar da imagem? E como esconder essa abordagem sem complicar a implementação do editor? A solução consiste em criar um outro objeto, o procurador (*proxy*) da imagem, que atuará como substituto para a imagem real. Este procurador agirá exatamente como a imagem e cuidará de sua instanciação quando a mesma for requerida. O procurador da imagem criará a imagem real somente quando o editor de texto requisitar a ele que exiba a imagem, através da operação **desenhar( )**. A partir daí, o procurador passará adiante as requisições subsequentes diretamente para a imagem, como ilustra o diagrama de sequência abaixo.



- Aplicabilidade: O padrão Procurador é aplicável sempre que houver necessidade de uma referência mais versátil ou sofisticada do que um simples ponteiro. A seguir são listadas algumas situações nas quais este padrão é aplicável:
  - Um Procurador Remoto provê uma representação local para um objeto que se encontra em um outro espaço de endereçamento.
  - Um Procurador Virtual cria objetos complexos sob demanda, como no caso do exemplo da motivação.

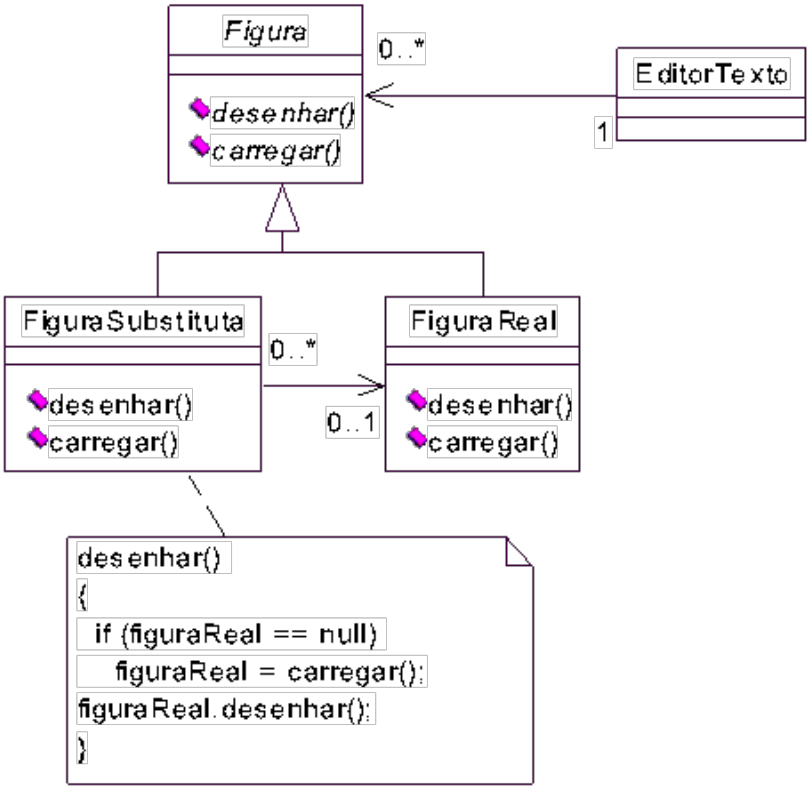
  - Um Procurador de Proteção controla o acesso ao objeto original. Este tipo de procurador é amplamente utilizado quando há diferentes direitos de acesso ao objeto original. Neste caso, o procurador serve como uma espécie de filtro.
  - Um Procurador de Referência Inteligente é uma substituição para um ponteiro simples, que realiza operações adicionais quando o objeto é acessado. Isto pode ser útil em muitas situações, tais como:
    - contar o número de referências ao objeto real, de forma tal que ele possa ser liberado automaticamente quando não houver mais referências a ele;
    - carregar um objeto persistente para a memória quando ele for referenciado pela primeira vez;
    - verificar se o objeto real não está bloqueado antes de permitir um acesso a ele, garantindo que nenhum outro objeto poderá altera-lo.

- Estrutura:



- Participantes:
  - Assunto: define uma interface comum para o ObjetoReal e para o Procurador, de modo que o procurador possa ser usado no lugar do objeto real;
  - Procurador: representa um substituto para o objeto real. Para tal, mantém uma referência ao objeto real, que o permite acessar este objeto. Sua interface deve ser idêntica à do objeto real, de modo que possa ser por ele substituído. Além disso, controla o acesso ao objeto real e pode ser responsável pela criação e exclusão do mesmo. Outras responsabilidades podem lhe ser atribuídas em função do tipo do procurador;
  - ObjetoReal: define o objeto real que o procurador representa.
- Colaborações: O procurador envia requisições para o objeto real, quando for apropriado, dependendo do tipo de Procurador.

- Consequências: O padrão Procurador introduz um nível de indireção quando se acessa o objeto. Essa indireção adicional tem muitos usos, dependendo do tipo de procurador. Um Procurador Remoto, por exemplo, pode esconder o fato de um objeto residir em um espaço de endereçamento diferente. Um Procurador Virtual pode realizar otimizações, como criar um objeto sob demanda. Procuradores de Proteção e de Referência Inteligente, por sua vez, permitem tarefas adicionais de gerenciamento quando um objeto é acessado.

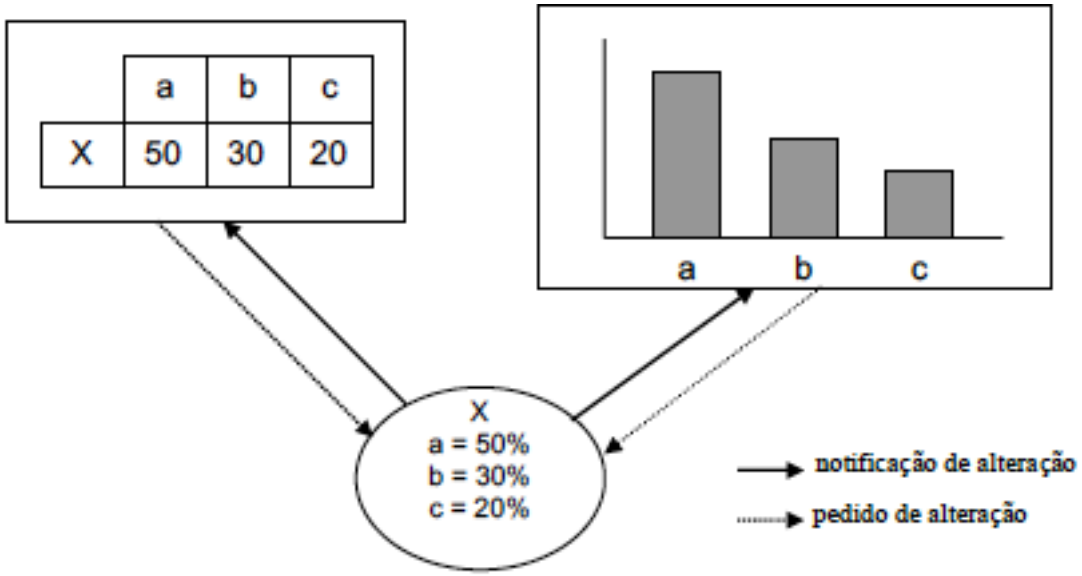


- Padrões Relacionados:
  - Adaptador: um adaptador provê uma interface diferente para o objeto que ele adapta. O procurador provê a mesma interface.
  - Decorador: Um decorador adiciona responsabilidades a um objeto, enquanto o procurador controla o acesso ao objeto.

A 1. 5 – Observador (*Observer*)

- Classificação: Padrão Comportamental de Objeto
- Propósito: define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- Também conhecido como: Dependentes.

- Motivação: É uma boa opção para ferramentas de interfaces gráficas com o usuário separar aspectos de apresentação dos respectivos dados da aplicação. As classes dos componentes de domínio do problema e de interface com o usuário podem ser reutilizadas independentemente, assim como podem trabalhar juntas. Por exemplo, os mesmos dados estatísticos podem ser apresentados em formato de gráfico de barras ou planilha, usando apresentações diferentes. O gráfico de barras e a planilha devem ser independentes, de modo a permitir reuso. Contudo, eles têm de se comportar consistentemente, isto é, quando um usuário altera a informação na planilha, o gráfico de barras reflete a troca imediatamente e vice-versa.



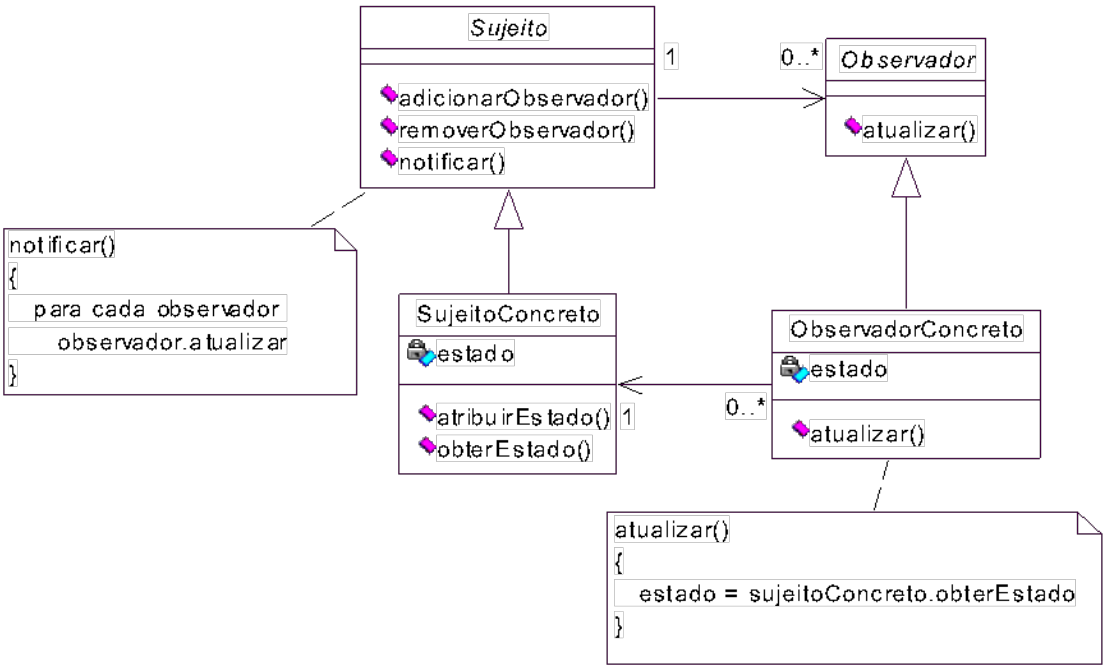
Este comportamento implica que a planilha e o gráfico de barras são dependentes do mesmo objeto de dados e, portanto, devem ser notificados quando ocorre alguma mudança no estado desse objeto.

O padrão Observador descreve como se estabelecem estes relacionamentos. Os objetos principais deste padrão são Sujeito e Observador. O sujeito, no exemplo o objeto **X**, pode ter qualquer número de observadores, no caso a planilha e o gráfico de barras. Todos os observadores são notificados sempre que ocorre uma mudança no estado do sujeito. Em resposta, cada observador irá consultar o sujeito para sincronizar seu estado com o estado do sujeito.

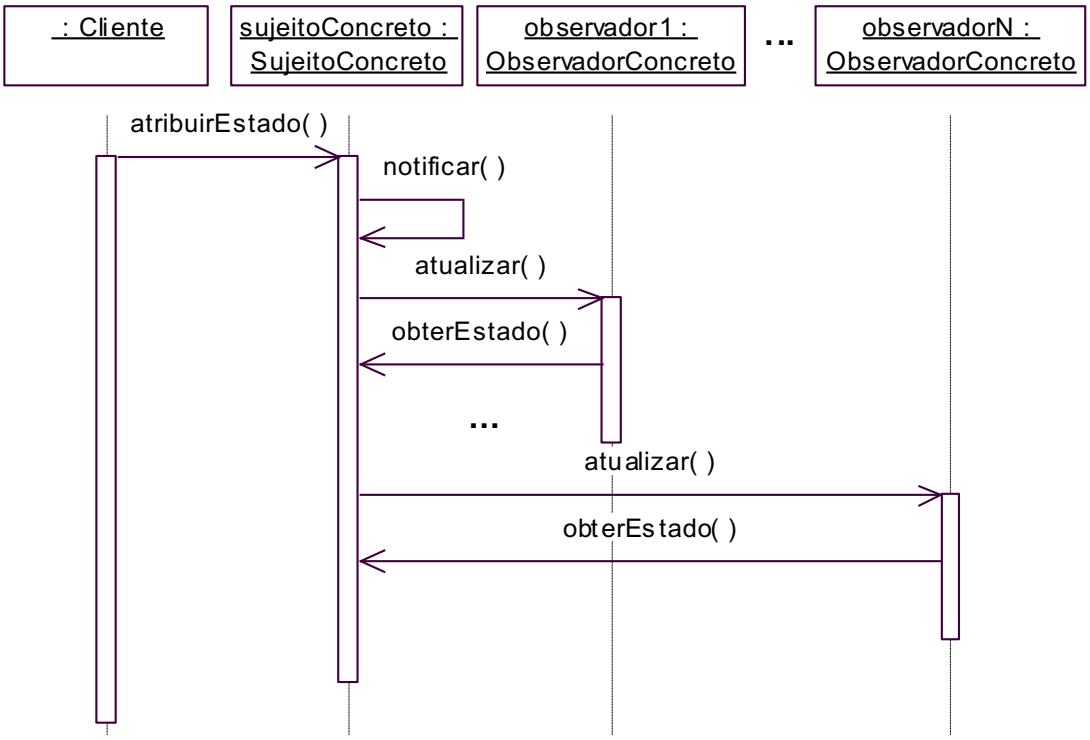
- Aplicabilidade: O padrão Procurador é aplicável em qualquer uma das seguintes situações:
  - Quando uma abstração possui dois aspectos, um dependente do outro. Encapsular esses aspectos em objetos separados permite variá-los e reutilizá-los independentemente.
  - Quando uma alteração em um objeto requer alterações em outros e não se sabe quantos objetos precisam ser alterados.

– Quando um objeto deveria ser capaz de notificar outros objetos sem fazer nenhuma suposição sobre como são esses objetos, ou seja, não se quer esses objetos fortemente acoplados.

- Estrutura:



- Participantes:
  - Sujeito: conhece seus observadores e provê uma interface para adicionar e remover objetos observadores. Qualquer número de observadores pode observar um sujeito.
  - Observador: define uma interface de atualização para os objetos que devem ser notificados das mudanças no sujeito.
  - SujeitoConcreto: armazena o estado de interesse para os observadores concretos e envia notificações para eles quando o seu estado é alterado.
  - ObservadorConcreto: mantém uma referência para um objeto SujeitoConcreto, armazena o estado que deve ficar consistente com o estado do sujeito e implementa a interface de atualização do Observador, de modo a manter seu estado consistente com o do sujeito.
- Colaborações: O sujeito concreto notifica seus observadores sempre que ocorre uma alteração que pode tornar o estado de seus observadores inconsistente com o seu estado. Após ser informado de uma mudança no sujeito concreto, um objeto ObservadorConcreto pode consultar o sujeito, usando esta informação para reconciliar seu estado com o estado do sujeito.



- Consequências: O padrão Observador permite variar sujeitos e observadores independentemente. Deste modo é possível reutilizar sujeitos sem reutilizar observadores e vice-versa. Isso permite adicionar observadores sem modificar o sujeito ou outros observadores. Outros benefícios e obrigações desse padrão incluem:
  - Acoplamento abstrato entre Sujeito e Observador: Tudo que um sujeito sabe é que ele possui uma lista de observadores, todos em conformidade com a interface simples da classe abstrata Observador. O sujeito não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre sujeitos e observadores é abstrato e mínimo.
  - Suporte para comunicação *broadcast*: Ao contrário de uma notificação individual, a notificação que um sujeito envia não precisa especificar seus receptores. A notificação é enviada automaticamente para todos os objetos interessados. Assim, a responsabilidade de um sujeito é limitada apenas à notificação de seus observadores. Isso oferece liberdade de adicionar e remover observadores a qualquer momento.
  - Atualizações inesperadas: Uma vez que os observadores não têm conhecimento da presença uns dos outros, eles podem não ser conscientes do custo de uma alteração no sujeito. Assim, uma operação aparentemente inócua sobre o sujeito pode provocar uma atualização em cascata para seus observadores e objetos dependentes. Além disso, critérios de dependência não bem definidos geralmente levam a atualizações falsas que podem ser difíceis de propagar.