CAPÍTULO VI: SISTEMAS DE INFORMAÇÃO - DESENVOLVIMENTO

6.1 Tipos de Sistemas do Ponto de Vista da Engenharia
6.2 Ciclos de Vida do Software e suas Características
6.3 Modelagem de Negócio
6.4 Requisitos
6.5 Análise e Design
6.6 Modelagem Funcional
6.7 Geração de Código e Teste
6.8 Comentários Bibliográficos
6.9 Exercícios

Neste capítulo apresentaremos algumas técnicas para o desenvolvimento de sistemas de informação. Neste ponto, é fundamental entender que sistemas de informação e software costumam ser desenvolvidos em conjunto, embora conceitualmente sejam coisas diferentes. Por exemplo, é possível ter software que não seja sistema de informação, como compiladores, jogos e sistemas operacionais; também é possível ter sistemas de informação sem software, afinal as empresas, muito antes de existirem computadores, já tinham seus sistemas de informação, mesmo que baseados em lápis, papel, gavetas e grampeadores.

Iniciamos o capítulo examinando alguns tipos de sistemas computacionais e caracterizando o software para sistemas de informação como uma das muitas possibilidades existentes. O importante é mencionar que cada diferente tipo de sistema pode exigir diferentes técnicas de desenvolvimento, não existindo assim nenhum modelo universal para desenvolvimento de software.

Na sequência explicaremos também porque não há modelos universais para desenvolvimento de sistemas de informação, analisando as características de vários ciclos de vida e indicando porque tais características foram introduzidas e para que tipo de sistemas e projetos elas são apropriadas.

Finalmente, veremos algumas técnicas para elaboração de sistemas de informação, o que usualmente inicia com a modelagem de negócio, seguida pela análise de requisitos e modelagem conceitual. Ficará para capítulos posteriores a apresentação de técnicas de implementação de tais sistemas com o uso de bancos de dados.

6.1 Tipos de Sistemas do Ponto de Vista da Engenharia

Existe, às vezes, uma tendência a se pensar que sistemas computacionais em geral são muito parecidos entre si, e tal tendência pode levar a imaginar que o processo de desenvolvimento de software possa ser algo que evolui sempre da mesma forma, havendo assim uma espécie de processo ideal, que sempre funcionaria. Isso, porém é apenas ilusão. Existem muitos tipos de sistemas e cada um deles demanda diferentes técnicas de desenvolvimento.

Durante os anos 1980, na assim chama era das "balas de prata" para a engenharia de software, acreditava-se que seria possível definir uma técnica universal de desenvolvimento e até uma linguagem universal de programação, que substituiria todas as outras. Assim, cada nova técnica era apresentada como a solução para os problemas do desenvolvimento de software, que desde a década anterior eram conhecidos como a "crise do software", ou seja, a incapacidade de os programadores acompanharem a rápida evolução do hardware.

Porém, a diversidade sempre foi uma constante na área de desenvolvimento de software. Desde os primórdios da área, nos anos 1950, já ficou bem claro que haveria pelo menos duas vertentes: os *sistemas científicos*, usualmente atendendo a profissionais como físicos, engenheiros, matemáticos e militares, e os *sistemas comerciais*, que atenderiam às necessidades das mais diferentes empresas e organizações.

Para os sistemas científicos foi criada, no final da década de 1950 a linguagem de programação FORTRAN e para os sistemas comerciais, logo no início da década seguinte, a linguagem COBOL. Assim, duas vertentes de sistemas computacionais surgiram e continuariam a evoluir de forma independente, embora com muitos pontos em comum.

Da vertente de sistemas comerciais, surge então a área que hoje é denominada "sistemas de informação", ou seja, sistemas que atendem às necessidades de registro, processamento e consulta à informação, principalmente por parte de empresas e organizações dos mais diversos tipos.

Para efeito de classificação de tipos de sistemas, considera-se hoje uma multiplicidade de classes bem maior:

- *Sistemas de informação*: são sistemas fortemente baseados em informação que deve ser recebida, armazenada, processada e consultada sempre que necessário. Usualmente quando se fala em sistemas de informação, logo vêm à mente os sistemas do tipo comercial.
- *Sistemas de controle*: estes são usualmente presentes na indústria, onde computadores mantém o controle sobre linhas de produção que muitas vezes também envolvem atividades automatizadas por robôs. Também existem informações aqui que devem ser registradas e consultadas, mas o foco do sistema não está na informação em si, mas no controle das atividades.
- *Software básico*: trata-se dos programas que permitem que os computadores sejam mais facilmente operados por seres humanos, como por exemplo, os sistemas operacionais, compiladores de linguagens de programação, *drivers* de dispositivos eletromecânicos etc. Sem estes sistemas, o computador teria que ser controlado a partir de instruções muito elementares em código de máquina e seria praticamente impossível termos toda a complexidade dos atuas aplicativos com o uso dessa técnica. Novamente, não se trata de sistemas que usam informação de forma tão intensa quanto os sistemas comerciais. O foco aqui está no acesso às funções mais elementares do computador e na abstração dessas funções para funções de nível cada vez mais alto até que um usuário possa, por exemplo, apertar um botão com o mouse e desencadear a execução de milhões de instruções de máquina a partir desse simples gesto.
- *Sistemas embarcados*: estes sistemas são concebidos para funcionar dentro de dispositivos eletromecânicos como automóveis, televisores, eletrodomésticos etc. Aqui, novamente, a ênfase não está na informação, mas na operação de tais dispositivos. Usualmente com interfaces com usuário muito simplificadas ou ainda agindo de forma completamente autônoma, estes sistemas fazem medições usando sensores e após processar essas informações de acordo com uma programação, atuam sobre mecanismos que possam produzir algum efeito no mundo real, como, por exemplo, freando um automóvel de forma a evitar que ele derrape na pista, como no caso dos freios ABS. Muitas vezes estes sistemas são também considerados como sistemas de tempo real, pois devem produzir resultados no mesmo instante em que as leituras são feitas, não havendo usualmente prazo mais estendido para realizar o processamento. Já os sistemas de informação usualmente não têm restrições de tempo de processamento tão severas, afinal se uma consulta

é feita em um segundo ou um microssegundo, isso não chega a afetar a produtividade do usuário do sistema, mas em um sistema de freios essa diferença seria crítica.

- *Sistemas pessoais*: são considerados sistemas pessoais aqueles que atendem ao indivíduo em sua vida pessoal e profissional. Exemplos desse tipo de sistema são os aplicativos como editores de texto, planilhas, organizadores de e-mail, ferramentas de chat, etc. Esses também usualmente não são entendidos como sistemas de informação, pois seu objetivo é facilitar ao indivíduo a execução de determinadas tarefas as quais até produzem resultados como textos, planilhas, imagens etc., mas o objetivo da ferramenta usualmente é apenas de produzir estes resultados e não de processar ou organizar os mesmos.
- *Jogos*: embora existam muitos tipos de jogos eletrônicos, eles usualmente são considerados sistemas de alta complexidade visto que normalmente precisam de respostas em tempo real, bom design e jogabilidade que produza prazer no usuário, visto que o objetivo deste tipo de sistema é justamente permitir o entretenimento. A indústria de jogos possui todo um conjunto próprio de técnicas e ferramentas de desenvolvimento, até porque esse tipo de sistema usualmente exige equipes interdisciplinares.
- *Inteligência artificial*: apesar de ser considerada também como uma das subáreas da Ciência da Computação, a IA, como também é chamada a inteligência artificial, é uma área emergente e cada vez mais desafiadora em termos de desenvolvimento de software. Embora tenha sido desacreditada em alguns momentos durante o século passado, a IA ressurgiu nesse início de século como uma nova aposta visto que novos algoritmos e ferramentas aliados a computadores cada vez mais poderosos em termos de memória e capacidade de processamento tem tornado possível realizar atividades que até há poucos anos apenas seres humanos conseguiam realizar. Esses sistemas usualmente também não são considerados sistemas de informação, embora eventualmente possam ser partes deles, ajudando a elaborar diagnósticos ou reconhecer padrões em imagens ou sons.

Vimos assim que existe uma grande diversidade entre tipos de sistemas que podem ser desenvolvidos com o uso de software. Observamos que os assim chamados *sistemas de informação*, que são o alvo deste livro, são um tipo especial de sistema que utiliza fortemente a informação recebida usualmente de seres humanos e que é armazenada, processada e fica disponível para consulta, permitindo a realização especialmente de atividades comerciais, mas também de atividades de outras organizações em geral.

Deve-se observar, porém, que a lista acima não é completa, pois muitos outros tipos de sistemas existem e classificá-los aqui seria inviável. Além disso, sistemas nem sempre se categorizam em um único tipo. Existem sistemas que embora pertençam primordialmente a uma das categorias podem ter elementos pertencentes às outras.

Vamos nos concentrar neste capítulo nos *sistemas de informação*, conforme mencionado, suas necessidades em termos de processo de desenvolvimento e algumas técnicas para realizar esse processo.

6.2. Ciclos de Vida do Software e suas Características

No início da história da computação, o hardware era muito limitado e os programas, consequentemente eram muito curtos. Assim, matemáticos, engenheiros e físicos, usualmente encarregados de programar essas máquinas faziam o que bem entendiam em termos de programação.

Porém, o hardware cresceu de forma exponencial em poder de processamento e armazenamento desde o início da era da computação. Rapidamente, programas tornaram-se suficientemente complexos para que novas profissões fossem criadas, como programadores e analistas de sistemas. Porém, as técnicas de programação ainda eram incipientes e caóticas, o que levou alguns cientistas a cunharem o termo "crise do software" no final dos anos 1960, já que a falta de um processo sistemático e padronizado de programação faziam com que programas fossem complexos, desorganizados e incompatíveis, muitas vezes frustrando projetos que tinham que ser cancelados ou que atrasavam muito ou ainda deixavam a desejar em qualidade.

Um marco para a organização da área de desenvolvimento de software foi a publicação do artigo de Winston Royce em 1970 sobre processo de desenvolvimento de software. Ele inicialmente indicava que se poderia conceber o processo de desenvolvimento em diferentes fases que seriam:

- *Requisitos de sistema e software*: que produziria um documento de requisitos.
- *Análise*: que produziria um detalhamento dos requisitos na forma de modelos e regras de negócio.
- *Design*: que produziria a arquitetura do software a realizar os requisitos.
- *Codificação*: que produziria os componentes do software executáveis, bem como sua integração.
- *Teste*: que revisaria o produto para garantir sua confiabilidade e que estivesse de acordo com as especificações.
- *Operação*: que colocaria o produto em uso após migração de dados, testes finais, treinamento de usuários etc.

Ironicamente, Royce apresentou esse modelo como a personificação de algo que na opinião dele *não iria funcionar*. E realmente não funcionava para a maioria dos projetos. Mas muita gente viu nisso um modelo razoável e passou a usar essa referência e o modelo passou a chamar-se "*Waterfall*" ou "Modelo Cascata" a partir de 1976.

Esse modelo, de fato, passou a trazer alguma organização para o desenvolvimento de software, e suas fases bem definidas foram sua principal contribuição, pois ele estabelecia uma série de documentos que deveriam ser produzidos em cada fase para que se pudesse passar à fase seguinte. Porém, como o próprio Royce já havia observado, ele não funcionava bem na prática, pois muitas vezes a necessidade de mudanças era percebida tardiamente e para mudar o código, por exemplo, você deveria retornar às fases anteriores e produzir mudanças também nos requisitos, nos modelos, na arquitetura, etc. Isso era bastante trabalhoso e dificilmente colocado em prática, o que causou muitos problemas, especialmente em relação à inconsistência entre requisitos, modelos, documentação em geral e código existente.

Várias propostas de variações do modelo Cascata foram feitas, produzindo os assim chamados "modelos cascata modificados". Entre elas podemos citar:

- *Cascata com subprojetos*: um modelo que seguia igualzinho ao Cascata original até a fase de design. A partir dali, com a arquitetura definida, vários subprojetos de codificação eram estabelecidos e poderiam ser desenvolvidos por várias equipes em paralelo. Sua contribuição foi permitir a aceleração do processo de desenvolvimento pelo paralelismo nas fases finais, e sua maior deficiência estava no fato de que era extremamente difícil integrar ao final vários módulos desenvolvidos por equipes independentes. Essa integração "*big-bang*", como é chamada, usualmente produz uma quantidade enorme de erros, não padronização, e a necessidade de grandes refatorações em todo o código.
- *Cascata com redução de risco*: um modelo que acrescentava uma fase de redução de risco ao início do projeto, o qual depois poderia prosseguir como qualquer

variante do modelo Cascata. O objetivo deste modelo era minimizar a probabilidade e impacto de riscos do projeto como por exemplo, problemas de arquitetura, problemas de acesso a dados, problemas de compreensão de requisitos etc. Os principais riscos do projeto seriam analisados previamente e ações seriam realizadas para minimizar sua exposição. Assim, quando as fases tradicionais de desenvolvimento iniciassem, os aspectos mais obscuros e arriscados do projeto já teriam sido mitigados. Uma das principais evoluções desse modelo foi o Modelo Espiral de Boehm, que incorpora a análise de risco e prototipação na forma de grandes ciclos de desenvolvimento, e que termina com as fases tradicionais do Modelo Cascata.

- *Sashimi*: um modelo criado para reduzir a rigidez das fases do Modelo Cascata. Com Sashimi a ideia é que questões relacionadas à fase seguinte já podem ser tratadas na fase atual e questões relacionadas à fase anterior ainda podem ser abordadas na fase atual. Assim, a cada instante o desenvolvimento está formalmente em uma fase, mas sem deixar de abordar questões das fases anteriores e posteriores. Infelizmente, essa abordagem não reduzia substancialmente o problema do retrabalho ou mesmo da inconsistência entre artefatos de documentação e código.
- *Modelos V e W*: são variações do Modelo Cascata, mais populares na Alemanha, e que propõem uma forte disciplina de testes. Para cada fase construtiva do Modelo Cascata ("requisitos", "análise" e "codificação"), haveria uma fase correspondente para testar os elementos e modelos da fase construtiva ao final. Haveria assim, inicialmente testes para a codificação, depois para a arquitetura e regras de negócio e finalmente para requisitos.
- *Entrega em Estágios*: uma variação do Modelo Cascata com subprojetos na qual os subprojetos após a fase de design não são feitos em paralelo, mas sequencialmente no tempo, na forma de ciclos de desenvolvimento, com integração incremental. Isso evita o problema da integração "big-bang". A ideia de ciclos de desenvolvimento ou iterações está presente em praticamente todos os modelos de desenvolvimento mais modernos, inclusive nos ágeis.
- *Modelo Orientado a Cronograma*: uma variação da Entrega em Estágios na qual o ponto de parada não é o momento em que todos os requisitos foram implementados, mas um instante de tempo predefinido. Assim, os requisitos, com este modelo, precisam ser priorizados dos mais importantes para os menos importantes. O modelo garante que no prazo fatal alguma coisa será entregue e que preferencialmente serão os requisitos mais importantes. Isso evita que projetos atrasem ou percam prazos fatais porque os desenvolvedores ficaram desenvolvendo coisas não tão essenciais, mas que atrasaram o projeto. O modelo segue o assim chamado *Princípio de Pareto* que propõe que os 20% mais importantes dentre os requisitos devem atender a 80% das necessidades de um projeto.

Paralelamente à evolução do Modelo Cascata e suas variações, uma outra vertente crescia em importância ao longo das últimas décadas do século passado: os modelos baseados em *prototipação*. A ideia de prototipação já aparecia no artigo de Royce em 1970 como uma das formas de garantir a produção de software com mais qualidade. Royce propunha que o sistema fosse desenvolvido duas vezes: a primeira para aprender sobre ele e a segunda para produzir de forma organizada e coerente um código de qualidade que atendesse a todos os requisitos.

Algumas variações da técnica de prototipação acabaram sendo conhecidas como modelos de desenvolvimento. Entre elas:

- *Prototipação Evolucionária*: uma forma organizada de planejar e executar protótipos para avaliar diferentes aspectos de um sistema antes de entregar uma versão final.
- *DSDM, ou Método Dinâmico de Desenvolvimento de Sistemas*: um processo baseado em ciclos de prototipação funcional, prototipação de design e construção e implantação.

Muitos dos autores que estavam convencidos de que prototipação e interação com o usuário/cliente eram mais importantes do que ter simplesmente um processo industrial para seguir na produção de software acabaram evoluindo outras ideias que coletivamente eram conhecidas inicialmente como "processos leves", mas que após um encontro de vários metodologistas em 2001 passaram a ser conhecidas como "modelos ágeis". Houve uma mudança de foco aqui: primeiramente, não seriam mais processos, ou seja, descrições passo-a-passo sobre como fazer software, mas "modelos", ou seja, um conjunto de regras, sugestões e práticas que estabelecem como a equipe deve se organizar e como se comunicar, incluindo questões relacionadas ao ambiente de trabalho, aos valores e princípios etc.

Os modelos ágeis usualmente indicam que não são centrados em processos, mas centrados em pessoas. Assim, metaforicamente, em vez de investir na receita de bolo, eles propõem que se contratem bons confeiteiros, porque eles saberão como fazer o bolo, e que se dê a eles uma boa estrutura de trabalho.

Outra importante contribuição que surgiu no final do século passado e ainda hoje é bastante influente foi o **Processo Unificado**, um *framework* de processo com várias implementações, algumas delas ágeis, inclusive.

O Processo Unificado é importante porque surgiu a partir da unificação de diversos modelos e processos e da integração de várias ferramentas de desenvolvimento, além da produção de uma linguagem de modelagem conhecida como UML, ou *Linguagem de Modelagem Unificada*.

O Processo Unificado apresenta quatro fases, dentro das quais as disciplinas de processo como requisitos, análise e design, modelagem de negócio etc. são praticadas com maior ou menor intensidade. As quatro fases são:

- *Concepção*: na qual os requisitos devem ser compreendidos em sua essência e extensão, mas não necessariamente em profundidade. Essa fase deve resolver os principais riscos relacionados aos requisitos e ao cliente e produzir uma estimação de esforço global para o projeto além de um plano de desenvolvimento.
- *Elaboração*: na qual os requisitos serão detalhados e uma arquitetura de software definida iterativamente, ou seja, ao longo de vários ciclos de desenvolvimento. O objetivo principal dessa fase é resolver os principais riscos de arquitetura para que a fase seguinte ocorra com o mínimo de retrabalho.
- *Construção*: na qual o código final é produzido e testado. Seu objetivo é gerar o produto final que será disponibilizado ao cliente.
- *Transição*: na qual o produto é colocado em uso pelo cliente. Essa fase pode ser quase instantânea no caso, por exemplo, de aplicativos para *download* pela Web, ou bastante demorada caso migração de dados, múltiplas instalações e treinamento de usuários, por exemplo, sejam necessários.

Usualmente, as fases centrais de Elaboração e Construção são executadas em diversas iterações, que são ciclos de desenvolvimento completos para um subconjunto de casos de uso (os mais prioritários primeiro), riscos e modificações solicitadas. As iterações da fase de Elaboração usualmente têm como objetivos mitigar riscos, especialmente os relacionados à arquitetura, enquanto que as iterações de construção tem como objetivos produzir o produto final.

O Processo Unificado, assim, propõe que as disciplinas (antigas fases do Modelo Cascata) ocorrem com maior ou menor intensidade em cada uma das quatro fases. Por exemplo, haverá bastante análise de requisitos na fase de concepção e também na elaboração, mas ela diminui durante a construção e transição, embora não desapareça totalmente.

A quantidade e o tipo de disciplinas variam conforme a implementação do processo. As disciplinas que nos interessam neste capítulo estão associadas às atividades de modelagem e construção de sistemas e aparecem em várias versões do processo. Assim, das nove disciplinas originais, vamos trabalhar com as cinco seguintes:

- *Modelagem de negócio*: visa estabelecer uma compreensão sobre o contexto de negócio no qual o sistema a ser desenvolvido se situa, para que eventuais interrelações sejam contempladas.
- *Requisitos*: visa levantar os requisitos de sistema a partir do escopo de automatização estabelecido no final da modelagem de negócio.
- *Análise e design*: visa a construção de um modelo orientado a objetos, que servirá de base para a implementação do sistema a partir das informações obtidas na disciplina de requisitos.
- *Implementação*: visa a construção do código que vai realizar os requisitos de acordo com o modelo de design.
- *Teste*: verifica se o código atende aos requisitos de qualidade estabelecidos, sendo que o principal destes, usualmente, é estar livre de erros.

Outras disciplinas incluem implantação, ambiente e diversas formas de gerenciamento. Mas não serão vistas aqui, pois o objetivo deste capítulo é concentrar nas atividades de modelagem e construção de sistemas de informação.

O Processo Unificado, assim como a UML são hoje padrões de fato e de direito para produção de software. O Processo Unificado procurou absorver todas as contribuições positivas de modelos anteriores como análise de risco, teste, ciclos iterativos, desenvolvimento orientado a cronograma, centrado em arquitetura e dirigido por casos de uso. Ele é, por suas características, um modelo direcionado para a produção de software para sistemas de informação. Ele apresenta variações prescritivas como RUP (*Rational Unified Process*), OUM (*Oracle Unified Method*) e EUP (*Enterprise Unified Process*), mas também versões ágeis como OpenUP (*Open Unified Process*) e DAD (*Disciplined Agile Delivery*), que sucedeu a abordagem AUP (*Agile Unified Process*), mais antiga.

Outra contribuição para os métodos de desenvolvimento de software que não é necessariamente atrelada a nenhum dos modelos anteriores, mas compatível com a maioria deles, é o **Desenvolvimento Dirigido por Modelos** (*Model-Driven Development/Engineering*). Esta filosofia considera que os primeiros modelos de um sistema devem ser puramente conceituais, ou seja, usando elementos pertencentes ao domínio da aplicação ou negócio e não ao domínio da computação. Assim, nos modelos conceituais sobre os quais falaremos mais adiante, não aparecem coisas como "tela", "teclado", "banco de dados", "processamento", etc.

Na seção seguinte apresentamos algumas técnicas de concepção e modelagem de sistemas de informação que são compatíveis com modelos baseados no Processo Unificado, considerando as cinco disciplinas de interesse que foram mencionadas anteriormente.

6.3. Modelagem de Negócio

A *modelagem de negócio* é usualmente a primeira atividade de análise que se realiza para a concepção de um sistema informatizado. Um de seus objetivos é dar à equipe de desenvolvimento uma ideia geral do contexto de negócio no qual o sistema se situa.

Segundo Philippe Kruchten, um dos criadores do RUP, diferentes projetos poderão ter diferentes necessidades de modelagem de negócio. A seguinte lista apresenta uma caracterização desses diferentes níveis de necessidade do mais simples para o mais complexo:

- *Organograma*: trata-se de projetos relacionados com sistemas autossuficientes usualmente não associados a um contexto de negócio mais amplo. Um exemplo poderia ser o desenvolvimento de uma simples agenda de contatos, que tem objetivos fechados em si mesma e não interage com outros sistemas nem áreas de negócio. Este tipo de projeto usualmente não afeta as rotinas de trabalho da empresa e, portanto, não necessita de nenhuma modelagem de negócio, podendo-se ir diretamente para a análise de requisitos. Porém, é interessante que seja estabelecido previamente um organograma para que se saiba quem são as pessoas chave na organização cliente, se houver, e quais os poderes de decisão, por exemplo, em relação à definição de requisitos.
- *Modelagem de domínio*: trata-se de projetos onde é desenvolvido um único sistema relativamente autossuficiente, com pouca ou nenhuma interação com outros sistemas, mas que afeta de alguma forma as rotinas de trabalho da empresa. Trata-se de contextos de negócio simples, como o encontrado em microempresas ou empresas individuais, nos quais um único sistema com complexidade relativamente baixa dá conta de todos os processos e atividades necessárias, bem como das necessidades de registro de informação. Neste caso, a modelagem de negócio é feita concomitantemente à análise de requisitos. Assim, ao se estudar os requisitos do sistema, está-se ao mesmo tempo estudando o contexto de negócio do sistema. Neste caso, haverá um único modelo que servirá tanto para entender a empresa quanto o sistema.
- *Uma empresa, vários sistemas*: trata-se de projetos para empresas de médio e grande porte, as quais usualmente dividem suas atividades em diferentes áreas de negócio. Essas empresas usualmente terão toda uma coleção de sistemas e estes sistemas devem idealmente ser capazes de interagir. Assim, a modelagem de negócio ajuda a entender a empresa como um todo, com suas diferentes áreas, e, para as áreas que serão automatizadas, será desenvolvido um novo modelo, mais detalhado. Assim, a modelagem de negócio, neste caso e nos seguintes, será uma atividade prévia a ser desenvolvida para a compreensão do contexto geral da organização e, posteriormente, a modelagem de domínio vai detalhar as informações e processos necessários apenas às áreas que serão automatizadas.
- *Modelo de negócio genérico*: trata-se de projetos nos quais vai-se desenvolver um ou mais sistemas não para uma empresa, mas para um conjunto de potenciais clientes. Em muitas situações, as necessidades de negócio das diferentes organizações, embora tenham muitos pontos em comum, poderão ter também pontos divergentes. Assim, é necessário desenvolver um modelo genérico de negócio que contemple todas as potenciais variações, e precisa-se também estabelecer os pontos de variação nos quais uma empresa poderá ter necessidades diferentes de outra empresa. Usualmente a técnica de Linha de Produto de

Software (SPL[1]) é indicada nestes casos; essa técnica permite o planejamento e desenvolvimento de famílias de produtos de software com reutilização sistemática de partes ou componentes comuns.

- *Novo negócio*: trata-se de um tipo de projeto no qual um novo modelo de negócio será desenvolvido concomitantemente aos sistemas que lhe darão suporte. Este tipo de projeto é mais complexo porque não existe uma referência de modelo de negócio para ser analisada: ela terá que ser criada. Posteriormente os sistemas de informação que darão suporte a este novo sistema serão concebidos.
- *Renovação*: trata-se de projetos nos quais toma-se uma empresa que opera segundo determinados padrões e se faz toda uma reformulação na forma dessa empresa trabalhar. Por exemplo, uma livraria que vende livros em papel em um ponto comercial pode estar querendo estender suas atividades para a venda de livros digitais pela Web. Assim, é necessário que se proceda a uma extensa modelagem de negócio que irá produzir uma compreensão do negócio atual, sua reformulação em um novo negócio e a partir dali a concepção dos sistemas de informação para dar suporte a esse negócio renovado.

Nesta seção ainda apresentaremos algumas técnicas para realizar modelagem de negócio usando a linguagem UML. Conforme explicado, esse tipo de modelagem seria necessário no caso de projetos que necessitassem mais do que uma simples modelagem de domínio.

Primeiramente, convém mencionar que na modelagem de negócio o objetivo é entender o funcionamento da organização como um todo e não de um de seus sistemas especificamente.

Para realizar modelagem de negócio com UML temos três diagramas a disposição que podem ser muito úteis:

- *Diagrama de casos de uso*, que pode ser usado para descrever *quais* são os principais processos de negócio.
- *Diagrama de atividades*, que pode ser usado para *detalhar* cada um dos casos de uso de negócio descritos no diagrama de casos de uso.
- *Diagrama de máquina de estados*, que pode ser usado para especificar o comportamento dos *objetos-chave de negócio*.

No diagrama de casos de uso de negócio devemos descrever apenas os principais *processos de negócio*. Esses processos usualmente duram mais de um dia e envolvem vários atores. Isso os diferencia dos *processos de sistema*, representados por casos de uso de sistema que usualmente são mais pontuais.

Assim, supondo que nossa organização alvo seja uma biblioteca, quais grandes processos de negócio podemos identificar? Inicialmente vamos identificar os processos através dos quais a organização realiza seus objetivos. O principal caso de uso de negócio, assim, seria "Emprestar Livros". Note que esse caso de uso é um processo de negócio. Assim, ele possivelmente ocorre ao longo de vários dias e não se trata de ações ou processos instantâneos como os de sistema.

O ator envolvido em um caso de uso de negócio é também um ator de negócio. Neste caso, o ator seria o usuário da biblioteca. Porém, o nome "Usuário" é demasiadamente genérico para ser informativo; assim, talvez seja interessante usar um nome mais especificamente associado com bibliotecas. No caso, o nome "Leitor" deixa mais claro qual o papel ou função deste ator no modelo de negócio da biblioteca.

A notação usada para indicar atores e casos de uso de negócio consiste em incluir uma barra inclinada no desenho do ator e do caso de uso. Isso não é um padrão da UML,

---

[1] https://www.sei.cmu.edu/productlines/

mas é tão universalmente usado que é como se fosse. Assim, nosso ator e seu principal caso de uso podem ser representados como na Figura 1.

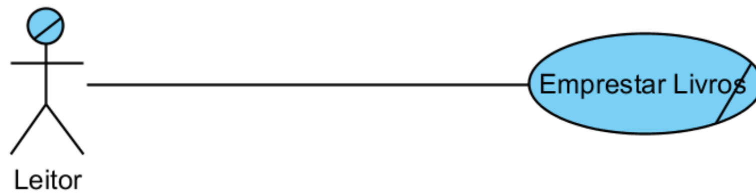

Figura 6.1: Um ator de negócio e seu principal caso de uso de negócio.

Neste contexto de negócio, emprestar livros é algo que inicia quando um leitor resolve que vai precisar de um livro, por exemplo: ele vai à biblioteca, examina, possivelmente seus índices ou estantes, escolhe os livros que precisa, usualmente leva estes livros a um guichê onde são marcados como emprestados a ele, leva os livros para casa e, após um determinado prazo, retorna à biblioteca para devolver os livros, podendo ou não pagar uma multa caso atrase a devolução.

Essa sequência de atividades relacionadas ao caso de uso "Emprestar Livros" pode ser descrita com o *diagrama de atividades* da UML, embora atualmente muitos analistas estejam preferindo usar o diagrama BPMN, ou *Business Process Modeling and Notation*, que tem mais recursos para modelagem de negócio, apresentando já de saída vários nodos especializados que não existem no diagrama de atividades, este bem mais simples e genérico.

Seja um ou outro, o diagrama vai descrever as diferentes atividades realizadas tanto pelo cliente quanto pela bibliotecária e isso se configura em um detalhamento do caso de uso de negócio.

Ao fazermos a descrição ou detalhamento do caso de uso de negócio com um diagrama de atividades, percebemos também que o ator "Leitor" não está sozinho neste processo. Em muitos casos, ele interage com um funcionário de guichê, o qual faz o registro do empréstimo. A rigor, este ator, como é funcionário e parte do sistema, pode ser identificado com o estereótipo "worker", que indica que ele é um *trabalhador de negócio*. A diferença entre ele e os atores que não são trabalhadores é que os papeis dos trabalhadores eventualmente poderão ser automatizados, ou seja, realizados pelo próprio sistema de informação. Já os atores de negócio que não são trabalhadores, não podem ser automatizados, via de regra. Assim, o novo modelo de casos de uso de negócio, incluindo esse trabalhador, é mostrado na Figura 2.

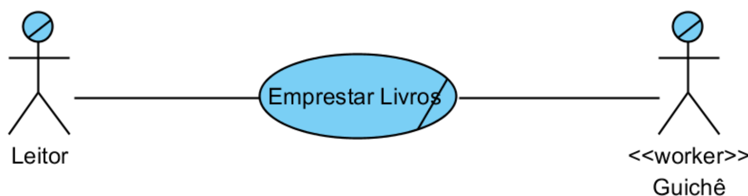

Figura 6.2: Evolução do modelo de casos de uso de negócio.

O diagrama de atividades é formado por uma "piscina" na qual as diferentes "raias" incluem as atividades realizadas por cada um dos atores participantes. As atividades são representadas por retângulos arredondados e são ligadas por fluxos que indicam dependência, ou seja, qual atividade deve obrigatoriamente ser realizada antes de qual. A Figura 3 mostra um possível diagrama de atividades descrevendo o processo de negócio "Emprestar Livros", conforme descrito mais acima.

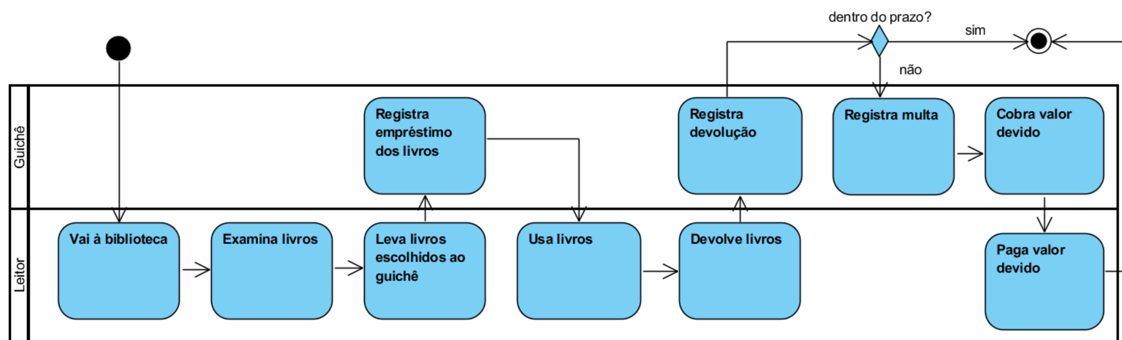

Figura 6.3: Diagrama de atividades detalhando o caso de uso de negócio "Emprestar Livros".

Observe que há duas raias: uma para o trabalhador de negócio "Guichê" e outra para o ator de negócio "Leitor". O processo inicia com a bolinha preta, mais à esquerda, que é um pseudonodo que indica início de processo. Já a bolinha preta circulada mais à direita indica o final do processo. Estes pseudonodos não precisam ficar necessariamente dentro das raias, podendo ficar fora delas ou em qualquer raia, pois isso não afeta seu significado.

No diagrama vemos uma sequência de atividades que iniciam com o “Leitor” e que passam ao “Guichê” também. Após a atividade "Registra devolução" por parte do Guichê, vemos um símbolo no formato de losango indicando um ponto de decisão. Se os livros forem devolvidos dentro do prazo, então o processo termina em seguida, com o fluxo "sim" indo para o pseudonodo final. Mas se pelo menos um dos livros estiver fora do prazo, mais algumas atividades são realizadas, seguindo-se o fluxo "não", conforme indicado no diagrama.

Observe que neste diagrama de atividades, há atividades de todos os tipos e durações. Por exemplo, registrar o empréstimo dos livros é algo que o “Guichê” faz rapidamente e que implica em alimentar um sistema de informação com este fato de que determinados livros foram emprestados para um determinado usuário. Já a atividade "Usa livros" é algo que o “Leitor” vai fazer provavelmente em casa, coisa que vai durar muitos dias, e para a qual nenhum registro no sistema de informação é feito. Embora o diagrama de atividades possa ter quaisquer atividades registradas, normalmente as mais úteis são aquelas que implicam em alimentar ou obter dados de um sistema de informações, como no primeiro caso, porque estas, posteriormente darão origem aos requisitos funcionais do sistema. Já as demais atividades servem apenas para ajudar a entender o contexto dos processos de negócio.

Além do processo de emprestar livros, podemos imaginar outros que sejam importantes para o ator "Leitor", como credenciar-se, descredenciar-se, pesquisar por livros, reservar um livro, pagar multa, etc. Porém, devemos evitar de colocar no diagrama de casos de uso de negócio processos que possam ser parte de casos de uso que já existam. Assim, por exemplo, verificamos que os processos de credenciar e descredenciar não fazem parte das atividades previstas no caso de uso de negócio "Emprestar Livros"; assim, eles são bons candidatos a serem considerados também como casos de uso de negócio. Por outro lado, os processos relacionados à pesquisa de livros e pagamento de multa, já pertencem ao escopo de atividades do caso de uso de negócio "Emprestar Livros" e assim, deve-se evitar considerá-los como casos de uso de negócio, pois são apenas fragmentos deste.

Alguns analistas colocariam "pesquisar livros" e "pagar multa" como casos de uso subordinados a "Emprestar Livros" com o uso de associações como "extends" ou "uses". Porém, isso não é recomendado porque:

- O diagrama deve conter casos de uso de negócio, e estes processos não o são.
- Estes processos são melhor descritos no diagrama de atividades, que além de mencioná-los explicitamente, ainda os relaciona de acordo com a precedência temporal, coisa que o diagrama de casos de uso não faz.
- Porque seria útil colocar estes dois processos no diagrama de casos de uso, mas não colocar todos os outros processos mencionados no diagrama de atividades da Figura 3? Ou se coloca todos ou nenhum. Mas como eles já aparecem no diagrama de atividade, não há motivo algum para repeti-los no diagrama de casos de uso, que desta forma ficaria redundante e complexo.

Para continuar completando o diagrama de casos de uso de negócio, podemos verificar se existem outros atores e quais seriam seus principais processos de negócio. Por exemplo, os livros não aparecem do nada na biblioteca, então deve haver um setor de aquisições e "Adquirir Livros" seria um processo de negócio que inicia quando alguém solicita a compra de um livro (possivelmente o Leitor) e termina quando os livros são registrados e disponibilizados na biblioteca. Assim, teremos um novo ator "Comprador", já que não é necessariamente o “Guichê” que faz a aquisição de livros. E como o Leitor não participa do processo, ele não é ligado a ele. O diagrama, agora mais completo, é mostrado na Figura 6.4.

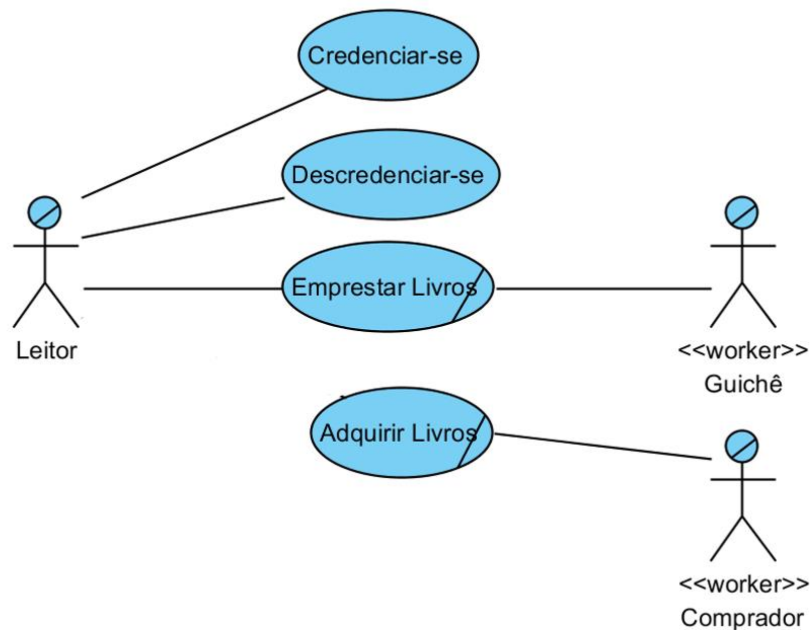

Figura 6.4: Um diagrama de casos de uso de negócio um pouco mais completo.

Se um estudo posterior (modelagem com diagrama de atividades) demonstrar que o funcionário do guichê deve participar dos casos de uso de credenciamento e descredenciamento de leitor, então uma ligação entre ele e esses casos de uso deverá ser estabelecida.

Outro diagrama que pode ser muito útil nesta fase é o *diagrama de máquina de estados*. Ele não é usado para descrever um processo, como o diagrama de atividades que acabamos de apresentar, mas para descrever os diferentes estados pelos quais um objeto de negócio pode passar. A confecção deste diagrama para os principais objetos de negócio pode ajudar a cobrir algumas lacunas que a simples identificação de casos de uso de negócio não faz.

No caso da nossa biblioteca, podemos considerar que os principais objetos-chave de negócio são o livro, o leitor e o empréstimo. Os dois primeiros correspondem a *objetos concretos*, sendo ainda que o segundo deles corresponde também a um ator de negócio.

Já o empréstimo é um *objeto abstrato*, pois trata-se de uma transação e não de uma coisa física. Vamos estudar aqui o livro para exemplificar a elaboração do diagrama de máquina de estados do objeto de negócio.

Primeiramente, precisamos definir que em relação ao livro existem pelo menos dois conceitos distintos: a *obra bibliográfica* e o *exemplar*. Quando dizemos que não existem dois livros com o mesmo ISBN, por exemplo, estamos nos referindo à obra bibliográfica. Mas se nos referimos aos objetos físicos, então dois exemplares do mesmo livro terão o mesmo ISBN. Assim, de um lado temos um conceito abstrato de obra bibliográfica que serve como uma *especificação* para um conjunto de exemplares de livro. O objeto concreto a que nos referimos, e que vamos estudar neste próximo diagrama é, portanto, o exemplar, ou seja, o livro físico em papel.

Podemos dizer que um primeiro estado para este objeto é alcançado quando ele é registrado pela primeira vez na biblioteca. Isso acontece quando ele é recebido pelo setor de aquisições. A obra possivelmente já estará catalogada e o exemplar será associado a ela e receberá um número de exemplar ou cópia. Posteriormente ele é disponibilizado na estante para consulta ou empréstimo. Durante a existência desse exemplar, ele passará por vários ciclos de empréstimo e devolução. Eventualmente, o livro poderá ser encaminhado para a restauração quando se detectar essa necessidade. Isso, claro, só poderá acontecer se o livro não estiver emprestado. Enfim, há várias formas de um livro sair do sistema: ele pode ser dado como imprestável pelo setor de restauração, se não houver mais como consertar os danos, também poderá desaparecer enquanto disponível na biblioteca, em função de extravio ou furto, ou ainda poderá ser emprestado e nunca devolvido. Porém, no caso de atraso, furto ou extravio, não se considera que ele esteja em um estado final, pois por mais tempo que passe, um dia, mesmo que muitos anos depois, ele poderá ser localizado ou devolvido. Mas enquanto estiver em um destes estados, o livro não poderá ser disponibilizado para novos empréstimos.

A Figura 6.5 mostra os estados possíveis de um exemplar de livro, de acordo com essa descrição.

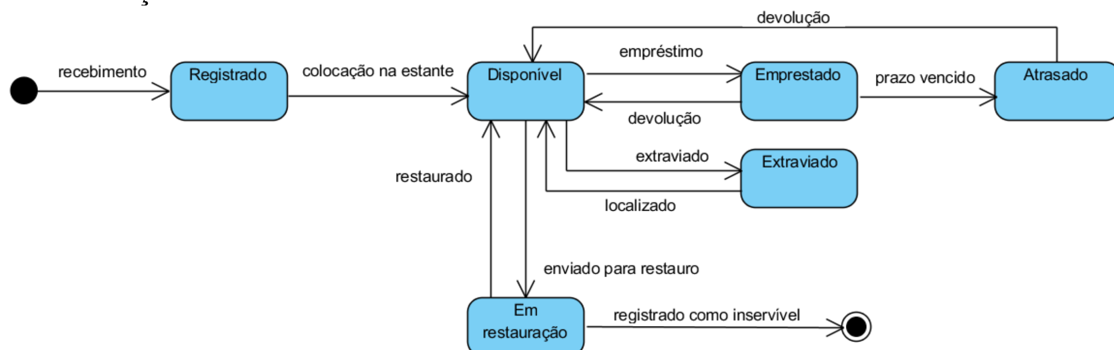

Figura 6.5: Diagrama de máquina de estados para o objeto de negócio "livro".

Esse diagrama tem similaridades com o diagrama de atividades. Entre outras coisas, os pseudonodos que representam os estados inicial e finais, são representados da mesma forma. Porém, os nodos e fluxos têm significado distinto nos dois diagramas. No diagrama de atividades, os fluxos simplesmente representam dependências, ou seja, qual atividade deve ser executada para que outra possa ser executada. Aqui os fluxos representam eventos, ou seja, situações que devem ocorrer no mundo real como um registro, uma solicitação, um prazo cumprido etc. Assim, cada fluxo desse diagrama deve ser rotulado com o nome de um evento que possa acontecer no mundo real. Já os nodos diferem também porque no diagrama de atividades eles representavam atividades ou sequencias de ações que os atores executavam. Já aqui os estados correspondem a uma situação e não a uma atividade. Por exemplo, um livro que está disponível não está

executando nenhuma atividade, mas está em um estado. Outra diferença é que neste diagrama pode não haver nenhum estado final, enquanto que um diagrama de atividades, quando usado para descrever processos de negócio, quase sempre deverá ter um nodo final.

6.4. Requisitos

Uma vez concluída a modelagem de negócio, deve ser estabelecida a fronteira de automatização de sistema, ou seja, a definição do escopo de informatização que será feita no modelo de negócio. Isso pode ser feito através de um documento descritivo, mas também é possível desenhar a fronteira de sistema no próprio diagrama de casos de uso de negócio. Usa-se, neste caso, um retângulo para indicar que os casos de uso dentro dele serão informatizados. Possivelmente alguns trabalhadores de negócio também poderão ter seu papel total ou parcialmente informatizado. Neste caso, o ator deve ser incluído na mesma fronteira de sistema. Caso o papel seja informatizado parcialmente, deve-se desdobrar esse trabalhador de negócio em dois atores: um que fica dentro da fronteira e um que fica fora dela.

Uma vez estabelecidos claramente quais os casos de uso de negócio a serem informatizados, pode-se passar a identificar os requisitos de sistema. Para isso, teremos duas fontes de informação: os diagramas de atividades que foram feitos para cada caso de uso de negócio que vai ser informatizado e os diagramas de máquina de estado desenvolvidos para os principais objetos de negócio.

Suponha, assim, que decidiu-se informatizar os casos de uso de negócio relacionados ao Leitor, mas não o caso de uso de aquisição de livros. Suponha também que é requerido que o futuro sistema dispense a existência de um funcionário de guichê, podendo o próprio leitor fazer o empréstimo usando alguma tecnologia como leitor de código de barras. Assim, a fronteira de informatização do sistema será anotada conforme mostrado na Figura 6.6.

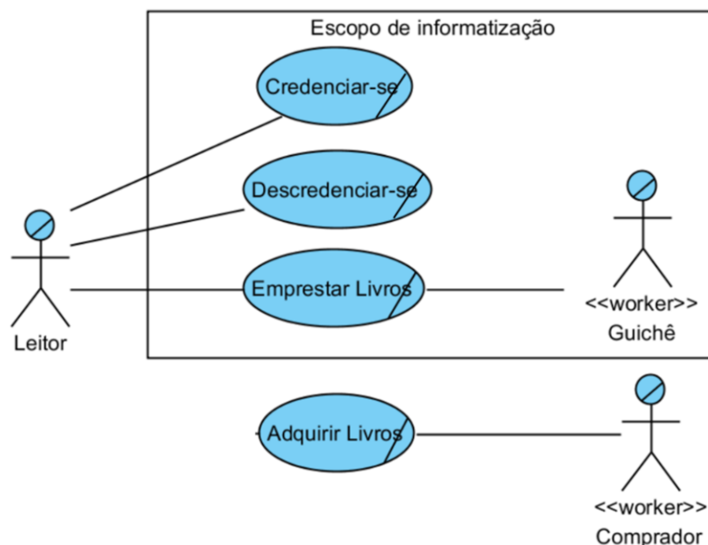

Figura 6.6: Escopo de informatização representado no diagrama de casos de uso de negócio.

Apenas atores que estejam do lado de fora do escopo e que tenham relação com algum caso de uso que esteja dentro do escopo serão considerados atores de sistema. Assim, neste exemplo, haverá um único ator de sistema para a disciplina de requisitos, o “Leitor”. O “Guichê” não será considerado ator de sistema porque ele está dentro do escopo de informatização e assim é um trabalhador de negócio cujas funções serão

automatizadas. Já o Comprador também não será promovido a ator de sistema porque embora ele seja um trabalhador que está fora do escopo de automatização, ele não se relaciona com nenhum caso de uso que esteja dentro do escopo de automatização.

O próximo passo para efetuar a análise de requisitos é identificar os casos de uso de sistema. Ao contrário dos casos de uso de negócio que são de longo prazo e multi-atores, os casos de uso de sistema correspondem a sessões de uso do sistema por um ator que realiza uma tarefa completa do início ao fim, sem interrupções. Há pelo menos duas maneiras para se encontrar esses casos de uso: (1) examinar os diagramas de atividade de negócio e agrupando atividades em casos de uso de sistema e (2) examinar os diagramas de máquina de estado, associando casos de uso de sistema às atividades que fazem um objeto de negócio mudar de um estado para outro.

No caso dos diagramas de atividade, é necessário identificar quais trabalhadores de negócio terão suas atividades informatizadas, pois eles não serão transformados em atores de sistema. No nosso exemplo, o ator "Guichê", que será automatizado, não será promovido a ator de sistema, e assim, todas as atividades feitas por ele serão agora feitas pelo sistema e incorporadas nos casos de uso dos quais o Leitor é ator.

Em relação às atividades descritas neste diagrama, precisamos determinar que aquelas que obrigatoriamente ocorrem no mesmo momento, seja em sequência, seja em paralelo, devem pertencer ao mesmo caso de uso de sistema. Já as atividades que podem ocorrer em momentos diferentes (por exemplo, no dia seguinte), devem pertencer a casos de uso de sistema distintos.

Assim, observamos no diagrama de atividades que as atividades "Vai à biblioteca", "Examina livros" e "Leva livros escolhidos ao guichê" são atividades que o ator Leitor necessariamente realiza em sequência e elas, portanto, vão pertencer a um mesmo caso de uso de sistema. A atividade seguinte "Registra empréstimo dos livros" seria realizada pelo guichê imediatamente após a atividade anterior. Assim, ela também pertence a esse primeiro caso de uso de sistema.

Até poderíamos chamar este caso de uso de sistema como "Emprestar Livros", mas para evitar confusão com o caso de uso de negócio de mesmo nome, usaremos "Obter Livros", o que até faz sentido já que o processo de empréstimo implica na obtenção, uso e devolução dos livros. E essas quatro atividades incluídas aqui correspondem apenas ao processo de obtenção dos livros.

Já a atividade seguinte "Usa livros" não ocorre necessariamente depois do registro do empréstimo. O leitor vai usar os livros no momento em que julgar conveniente e não obrigatoriamente logo depois do empréstimo. Por outro lado, essa atividade ocorre independentemente do sistema de informação que vamos informatizar e assim, ela também não é candidata a virar caso de uso de sistema.

Já a atividade seguinte "Registra devolução", é um novo caso de uso do qual o ator de sistema é o Leitor. Essa atividade é necessariamente seguida pelo registro da multa e cobrança, caso algum livro tenha sido devolvido depois do prazo. Assim, essas três atividades vão pertencer ao caso de uso de sistema "Devolver Livros".

A atividade de pagamento do valor devido não ocorre necessariamente no momento em que a multa é cobrada. O pagamento pode ser feito posteriormente. Assim, trata-se de um terceiro caso de uso de sistema que chamaremos de "Pagar Multa". A Figura 6.7 mostra esquematicamente como os três casos de uso de sistema são obtidos a partir do diagrama de atividades.

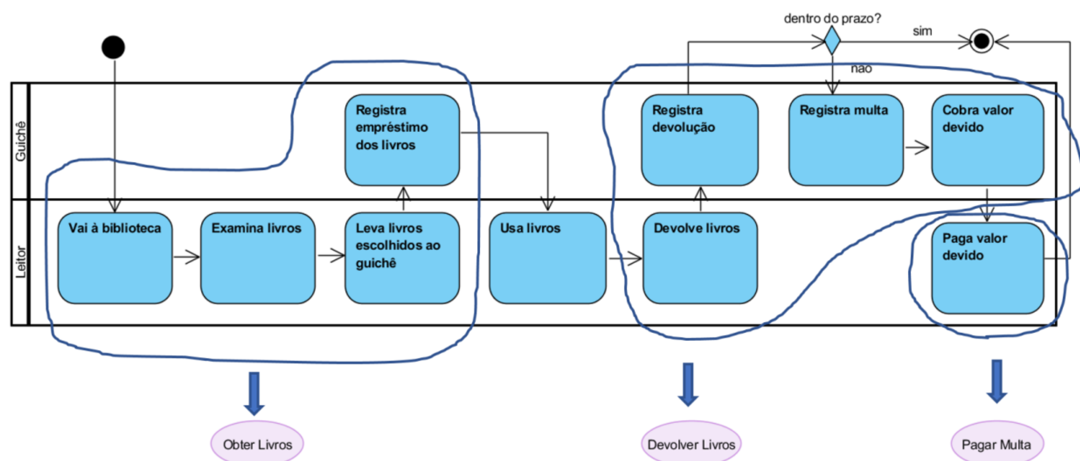

Figura 6.7: Representação esquemática da obtenção de casos de uso de sistema a partir do diagrama de atividades de negócio.

Já o diagrama de máquina de estados vai produzir possivelmente um caso de uso de sistema para cada transição de estado representada nele, desde que a transição tenha que ser realizada necessariamente a partir de um registro no sistema. Analisando cada uma das transições representadas no diagrama de máquina de estados da Figura 6.5, temos o seguinte resultado:

- A transição "recebimento" produz um novo caso de uso de sistema "Registrar Recebimento de Livro", cujo ator é o “Comprador”.
- A transição "colocação na estante" é meramente um ato físico que não precisa ser associado a um caso de uso de sistema, pois podemos considerar que o livro já está disponível no momento em que sua chagada é registrada na biblioteca. Ser capaz de emprestá-lo depende apenas de sua localização física.
- A transição "empréstimo" já é realizada pelo caso de uso de sistema "Obter Livro", que identificamos no diagrama de atividades.
- A transição "enviado para restauro" produz um novo caso de uso de sistema "Enviar Livro para Restauro", que pode ser atribuída a um novo ator, o “Restaurador”.
- A transição "restaurado" produz um novo caso de uso de sistema "Disponibilizar Livro Restaurado", que também pode ser atribuído ao Restaurador.
- A transição "registrado como inservível" produz um novo caso de uso de sistema "Dar Baixa em Livro", que também pode ser atribuído ao Restaurador já que, a princípio, é ele que deve chegar a essa conclusão (embora, dependendo de como a biblioteca se organiza, possa ser algum outro cargo ou funcionário.
- A transição "extraviado" produz um novo caso de uso de sistema "Registrar Extravio de Livro". Nenhum dos atores identificados até aqui parece ideal para ser associado a esse caso de uso. Assim, vamos considerar que descobrimos com o cliente que ele é realizado pela direção da biblioteca. Assim, criaremos o ator "Direção" e o associaremos a este caso de uso.
- A transição "localizado" produz um novo caso de uso de sistema "Registrar Recuperação de Livros Extraviados", que associamos ao ator Direção.
- A transição "prazo vencido" não necessita ser transformada em caso de uso de sistema, pois não há necessidade, pelo menos no momento, de avisar ao sistema que um livro está atrasado no momento em que isso passa a ser verdade. Basta que no momento da devolução do livro essa informação possa ser verificada.

- A transição "devolução" que sai do estado "Emprestado", é realizada pelo mesmo caso de uso "Devolver Livros" que já identificamos no diagrama de atividades.
- A transição "devolução" que sai do estado "Atrasado", é realizada pelo mesmo caso de uso "Devolver Livros" que já identificamos no diagrama de atividades.

Ficamos assim com um conjunto de casos de uso de sistema bem completo para as atividades de empréstimo e devolução de livros que estamos examinando. Esses casos de uso de sistema são considerados como os requisitos de alto nível para o sistema informatizado que será desenvolvido. A Figura 6.8 apresenta estes casos de uso na forma de um diagrama.

Figura 6.8: Diagrama de casos de uso de sistema (requisitos de alto nível).

Depois de obter a lista completa dos casos de uso de sistema, podemos examinar seus nomes para identificar os conceitos que vão compor o modelo conceitual preliminar (Figura 6.9). Normalmente os verbos e substantivos presentes nos nomes dos casos de uso de sistema podem dar boas pistas sobre quais informações o sistema vai trabalhar. Mas sempre é bom pensar um pouco sobre o modelo de informação antes de começar a criar classes cegamente.

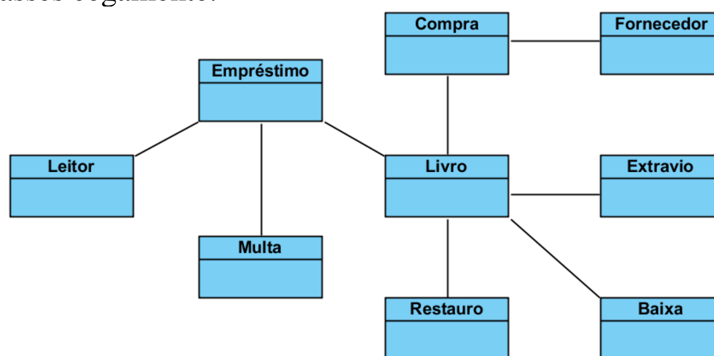

Figura 6.9: Diagrama de classes representando um modelo conceitual preliminar.

Os casos de uso "Obter Livros" e "Devolver Livros" levam à criação das classes "Livro", "Empréstimo", que incorpora as ações de obtenção e devolução de livros, e "Leitor", já que um empréstimo precisa estar obrigatoriamente ligado a um usuário da biblioteca. O caso de uso "Pagar multa" leva à criação da classe "Multa" que pode ter ou não instâncias associadas a um empréstimo. E assim por diante.

Usualmente esta atividade encerra a fase de concepção do processo unificado, pois os casos de uso de sistema agora identificados podem ser priorizados e seu esforço para

desenvolvimento estimado de forma que se possa planejar os ciclos de desenvolvimento abordando inicialmente os de maior prioridade e que cabem dentro do esforço possível em cada iteração.

Assim, durante cada iteração de desenvolvimento, vamos fazer o detalhamento dos casos de uso respectivos, bem como refinar o modelo conceitual a partir das informações descobertas neles. Possivelmente também poderemos fazer a implementação do caso de uso, se isso for julgado necessário na mesma iteração.

Uma vez identificados os casos de uso de sistema a serem abordados na iteração corrente, podemos passar ao detalhamento destes, o que vai produzir os *requisitos do sistema* para estes casos de uso.

O detalhamento de um caso de uso consiste em se definir o *fluxo principal* e os *fluxos alternativos*. O fluxo principal é uma sequência de passos de interação do ator com o sistema. Ele é também chamado de "caminho feliz" porque se considera que o fluxo principal deve conter uma descrição do processo quando tudo dá certo.

Há dois tipos de fluxos alternativos: as *exceções* e as *variantes*. Uma exceção é uma situação anômala que impede o prosseguimento dos passos do caso de uso e o fluxo alternativo deve indicar os passos que possivelmente vão corrigir essa situação. Por exemplo, se um usuário tenta emprestar um livro mas ele já tem um número de livros emprestados que corresponde ao limite máximo permitido, então ocorre uma exceção que impede ele de levar esse novo livro; as ações corretivas para que o caso de uso possa prosseguir incluiriam ele devolver um ou mais livros que tivesse já emprestado, dessa forma liberando a possibilidade de emprestar outro livro.

Não se deve considerar na descrição do caso de uso situações que não envolvam o sistema de informação; exceção ocorre quando o usuário passa ao sistema uma informação que o sistema não pode aceitar para dar prosseguimento ao caso de uso, como no exemplo mencionado acima. Devemos evitar colocar aqui situações não relacionadas à informação como, por exemplo, queda de energia elétrica, defeito no computador, acidentes com o usuário, etc.

Já a variante é simplesmente uma forma diferente de fazer as coisas no fluxo principal. Por exemplo, o usuário pode ou não querer cadastrar um novo endereço quando estiver fazendo uma compra. Isso não é uma exceção pois não há aqui nenhum impedimento para o caso de uso prosseguir. É apenas algo que ele pode ou não fazer de acordo com sua vontade, e, portanto, uma variante. Já uma senha errada não é opção do usuário, é uma exceção.

Deve-se procurar colocar nos passos de um caso de uso apenas transações que indiquem entrada e saída de informação do sistema. Assim, quando o usuário passa informações ao sistema, seja digitando ou fazendo uma escolha em um menu, temos um *passo de entrada de informação*. Já quando o sistema apresenta informações sejam armazenadas internamente ou obtidas a partir de um processamento dessa informação, temos um *passo de saída de informação*. Portanto, uma simples consulta ao acervo da biblioteca não deve ser considerado um caso-de-uso.

Usualmente bons casos de uso são descritos como uma sequência alternada de passos de entrada e saída. Considera-se também que na fase de análise de requisitos é mais vantajoso trabalhar com *casos de uso essenciais*, ou seja, aqueles que não mencionam a tecnologia de interface. Assim, um bom passo dirá "o usuário *informa* seu CPF" e não "o usuário *digita* o CPF", pois digitação implica usar o teclado como tecnologia de interface. Com casos de uso essenciais, podemos definir a tecnologia de implementação mais tarde, depois de conhecer melhor as necessidades de informação do sistema. Assim, por exemplo, mais adiante poderíamos decidir que o usuário poderá usar uma interface de voz ou uma leitora de código de barra para informar seu CPF, e não a

simples digitação. Casos de uso essenciais nos deixam abertos para novas escolhas tecnológicas.

Assim, o caso de uso "Obter livros", por exemplo, poderia ser definido inicialmente pelo seguinte fluxo principal (sequência de ações na qual os objetivos do caso de uso são obtidos sem desvios nem percalços):

1. O leitor se identifica.
2. O sistema confirma a identificação.
3. O leitor informa o livro que deseja emprestar.
4. O sistema confirma o empréstimo e informa o prazo para devolução.
5. O leitor confirma e encerra o caso de uso.

Note que os passos impares são realizados pelo ator e os passos pares pelo sistema. Na maioria dos casos de uso essa será a regra. Aqui está representada a situação de obtenção dos objetivos sem percalços ou desvios. Mas poderíamos pensar inicialmente no que aconteceria se alguma das informações passadas pelo usuário não fosse aceita pelo sistema, ou seja, se alguma regra de negócio pode implicar em exceção neste caso de uso. Pensando um pouco sobre o problema e possivelmente conversando com um especialista de domínio para obter as regras de negócio, poderíamos chegar às seguintes exceções:

- 1a. Usuário inválido (não reconhecido)
  - Retorna ao passo 1 (leitor se identifica novamente)
- 1b. Usuário suspenso (possivelmente por não pagar multas)
  - Executa-se o caso de uso "Pagar multa"
  - Retorna ao passo 1
- 3a. Livro desconhecido
  - O caso de uso é encerrado
- 3b. Livro reservado a outro usuário
  - O caso de uso é encerrado
- 3c. Livro pertencente a reserva técnica (apenas consulta local)
  - O caso de uso é encerrado
- 3d. Livro danificado
  - Executa-se o caso de uso "Enviar livros para restauro"
  - O caso de uso é encerrado

Observe que ao fazer essa analise sistemática do caso de uso estamos detalhando os requisitos do sistema. Vários novos conceitos podem surgir desta análise e também novos atributos de conceitos já existentes no modelo conceitual.

Outra situação que poderia acontecer e que não é uma exceção nem regra de negócio, mas uma opção do usuário seria ele levar de uma só vez vários livros. Assim, no passo 5, em vez de confirmar e encerrar o caso de uso ele pode escolher registrar o empréstimo de pelo menos mais um livro. Neste caso, ele não precisa se identificar novamente e então o caso de uso pode reiniciar diretamente do passo 3. Isso pode ser representado por uma variante para o passo 5:

- 5.1. O leitor deseja levar mais um livro:
  - Retorna ao passo 3

Wazlawick [Waz14] apresenta em maior detalhe técnicas de modelagem de casos de uso para obtenção de requisitos.

6.5. Análise e Design

Há dois momentos distintos nos quais fazemos uma modelagem de conceitos ou classes quando usamos o Processo Unificado. O primeiro momento é na fase de concepção, quando dispondo apenas dos nomes dos casos de uso de sistema podemos

criar o assim chamado *modelo conceitual preliminar* (Figura 6.9). Ele vai conter apenas as classes (representando conceitos) e suas associações, sem maiores detalhes. Essas classes podem ser inferidas a partir dos nomes dos casos de uso: substantivos e verbos que indicam transações são ótimas classes candidatas ao modelo conceitual.

De posse deste modelo conceitual preliminar podemos fazer uma revisão nos nossos casos de uso de sistema. Cada conceito ou classe existente no modelo deve poder ser criado e consumido em pelo menos um caso de uso. A rigor, são quatro operações possíveis:

- *Criação*: um caso de uso que produz instancias novas daquela classe, como por exemplo, um registro de um novo cliente.
- *Consulta*: um caso de uso que apresenta ao usuário as informações relativas àquele conceito, como por exemplo, uma informação sobre a disponibilidade de um livro.
- *Atualização*: um caso de uso que altera as informações armazenadas sobre uma instância preexistente, como por exemplo, um caso de uso que muda o status de um livro para "emprestado".
- *Destruição*: um caso de uso que deleta uma instância de um conceito, como por exemplo, um caso de uso que deleta um item de um pedido.

Essas quatro operações normalmente são conhecidas pela sigla *CRUD*, que vem do inglês *Create*, *Retrieve*, *Update* e *Delete*. Porém, nem todos os conceitos precisam passar pelas quatro operações. Por exemplo, conceitos cujas instâncias são criadas em outro sistema, poderiam ser em nosso sistema apenas consultadas, mas não criadas nem deletadas ou atualizadas; instâncias de conceitos considerados *imutáveis* podem ser criadas, destruídas e consultadas, mas não alteradas, e muitas vezes nem destruídas elas podem ser. É usual em sistemas de informação que instancias não sejam destruídas realmente, mas apenas marcadas como inativas, assim, se um cliente, por exemplo, faleceu, você não deve apagá-lo da base de dados, mas marcá-lo como inativo; caso contrário, pedidos feitos no passado não serão mais corretamente indicados no sistema.

Sabendo então quais das quatro operações cada conceito deve implementar, verifique no seu modelo de casos de uso de sistema se pelo um caso de uso implementa aquelas operações que o conceito deve ter. Em não havendo nenhum caso de uso com essa operação, deve-se considerar seriamente criar um e adicioná-lo ao modelo de casos de uso de sistema.

Analisando as classes do nosso modelo conceitual preliminar da Figura 6.9, podemos identificar quais casos de uso implementam as operações CRUD usando, por exemplo, uma tabela, como mostrado na tabela 6.1. No caso, como usualmente registros criados em sistemas de informação não são destruídos, mas apenas invalidados, interpretamos a operação "delete" não como o apagamento da informação, mas como a marcação dela como não mais ativa. Para algumas operações também é discutível se elas representam um *update* ou um *delete*. Por exemplo, se o conceito de "Empréstimo" for algo temporário que deixa de existir quando o livro emprestado for devolvido, "Devolver livro" deveria ser um "*Delete*". Na tabela a devolução do livro significa uma mudança de estado do empréstimo de "atual" para "encerrado". Raciocínio análogo pode ser aplicado às outras operações de "Update".

Tabela 6.1: Identificação dos casos de uso que implementam as operações CRUD para as classes do modelo conceitual preliminar.

Conceito	Create	Retrieve	Update	Delete
Leitor				
Empréstimo	Obter livro		Devolver livros	
Multa	Devolver livros		Pagar multa	

Compra			Registrar recebimento de livros	
Livro	Registrar recebimento de livros			Dar baixa em livros
Restauro	Enviar livros para restauro		Disponibilizar livros restaurados	
Fornecedor				
Extravio	Registrar extravio de livros		Registrar recuperação de livros extraviados	
Baixa	Dar baixa em livros			

Fazendo esta análise podemos perceber que há vários casos de uso faltando, pois para alguns conceitos não há definição sobre como suas instâncias são criadas, consultadas, alteradas ou desativadas. Assim, poderíamos completar a tabela adicionando novos casos de uso. Quando uma das quatro operações não se aplica ao conceito, marcamos a posição referente com "---".

Tabela 2: Lista de casos de uso completada.

Conceito	Create	Retrieve	Update	Delete
Leitor	Cadastrar livro	Visualizar informações de leitor	Atualizar informações de leitor	Descredenciar leitor
Empréstimo	Obter livros	Listar empréstimos em aberto	Devolver livros	Cancelar empréstimo
Multa	Devolver livros	Listar multas em aberto	Pagar multa	Cancelar multa
Compra	Comprar livros	Listar compras em aberto	Registrar recebimento de livros	Cancelar compra
Livro	Registrar recebimento de livros	Consultar livro	---	Dar baixa em livros
Restauro	Enviar livros para restauro	Listar restauros em andamento	Disponibilizar livros restaurados	---
Fornecedor	Cadastrar fornecedor	Consultar informações de fornecedor	Atualizar informações de fornecedor	Descredenciar fornecedor
Extravio	Registrar extravio de livros	Listar extravios	Registrar recuperação de livros extraviados	---
Baixa	Dar baixa em livros	Listar baixas	---	---

Note que foi definido que o livro e a baixa são conceitos imutáveis, já que seus atributos não podem ser modificados. Outros conceitos foram definidos como não deletáveis; restauros, extravios e baixas de livro não são mais desativados ou deletados. Já outros conceitos como empréstimo, multa e compra podem ser cancelados, o que

significa que embora continuem constando no sistema foram considerados anulados. Por outro lado, leitores e fornecedores podem ser descredenciados.

O segundo momento em que se pode fazer alterações no modelo conceitual é na fase de Elaboração, quando, após fazer o detalhamento de cada caso de uso, deve-se observar os detalhes das informações transmitidas nos passos de entrada e saída e verificar se o modelo já faz constar os conceitos ou atributos que representam essa informação. Dessa forma, a cada novo caso de uso que for detalhado, o modelo será também refinado. Isso pode acontecer ao longo de vários ciclos de elaboração. Ao final do processo espera-se que a arquitetura de classes construída a partir desse refinamento cada vez maior do modelo conceitual esteja suficientemente estável para permitir a construção do sistema propriamente dito na fase de construção.

Analisando, por exemplo, o modelo conceitual da Figura 6.9 e o caso de uso expandido "Obter Livros", teremos que incluir no modelo conceitual todas as informações referenciadas no caso de uso. Por exemplo, o caso de uso menciona que o usuário pode estar suspenso, mas não há qualquer atributo ou conceito que permita registrar que um usuário está suspenso. Assim, uma solução para poder representar essa informação seria adicionar um atributo "suspenso" à classe Leitor que pode ser verdadeiro ou falso. A Figura 6.10 mostra um modelo conceitual evoluído ao qual foram adicionadas algumas informações obtidas do caso de uso expandido "Obter livros".

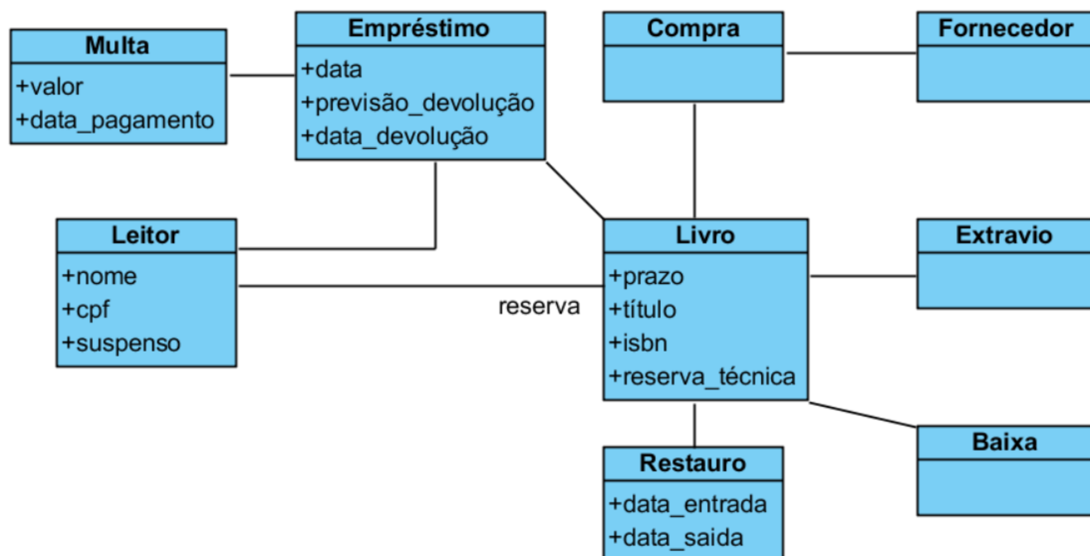

Figura 6.10: Modelo conceitual refinado a partir do caso de uso "Obter livros".

O processo de completar o modelo conceitual com novos atributos e classes continua a partir da análise dos outros casos de uso expandidos. Usualmente recomenda-se iniciar com os casos de uso mais complexos, pois com eles descobre-se mais rapidamente novos conceitos e atributos que serão necessários. Dessa forma, à medida que o progresso com a expansão dos casos de uso avança, cada vez menos novas informações são adicionadas ao modelo, de forma que ele vai se estabilizando.

Vale destacar que o modelo conceitual mostrado na figura 6.10 ainda é um modelo preliminar. Na modelagem detalhada serão explicitadas melhor as associações, determinado as abstrações e definido métodos.

6.6. Modelagem Funcional

Até aqui temos dois artefatos importantes para a análise de requisitos e de domínio de um sistema: os casos de uso expandidos (ou detalhados) e o modelo conceitual agora já refinado com as informações contidas nestes casos de uso detalhados. Idealmente deve

ser possível realizar todas as transações indicadas nos passos dos casos de uso sobre os elementos (conceitos, atributos e associações) do modelo conceitual. Assim, nenhuma transação dos casos de uso pode referenciar elementos que não constam no modelo e todos os elementos do modelo são referenciados por pelo menos uma transação de pelo menos um caso de uso.

Para a próxima fase do processo de modelagem do sistema podemos passar a definir exatamente como cada transação do caso de uso expandido modifica os elementos representados no modelo conceitual. Esse processo é conhecido como "modelagem funcional" e pode ser feito com o uso de *contratos de operação de sistema*. Esses contratos tanto podem ser escritos em linguagem natural quanto em alguma linguagem de especificação formal, como por exemplo a OCL (*Object Constraint Language*). Para manter a leitura do capítulo menos trabalhosa, usaremos linguagem natural nos nossos exemplos.

Mas independentemente da linguagem usada, os contratos devem obedecer a uma série de regras. Inicialmente identificamos que existem dois tipos de operações de sistema e, portanto, dois tipos de contratos:

- *Comandos de sistema*: são operações que têm como objetivo alterar alguma informação armazenada no sistema, seja criando, modificando ou deletando elementos de informação. Contratos de comandos de sistema obrigatoriamente têm *pós-condições* e opcionalmente podem ter *precondições* e *exceções*. Comandos de sistema não devem apresentar *retornos de informação* ao ator.
- *Consulta de sistema*: são operações que têm como objetivo apenas retornar a um ator alguma informação armazenada no sistema, seja da forma como se encontra armazenada, seja aplicando alguma modificação, como somas, cálculo de médias, , etc. Contratos de consultas de sistema não podem alterar dados armazenados e, portanto, não têm pós-condições. Obrigatoriamente esses contratos devem ter retornos de informação e opcionalmente podem ter precondições ou exceções.

Assume-se pelo *princípio de separação comando-consulta*, que comandos não retornam informação e consultas não alteram dados. Quando uma operação não respeita esse princípio, diz-se que ela sofre de *efeitos colaterais*, e isso não é desejado em um bom projeto de software.

As pós-condições mencionadas acima são então, alterações básicas que um comando pode fazer na informação representada no modelo conceitual. Considera-se que apenas 5 tipos de operações são básicos ou elementares em sistemas orientados a objetos:

- *Criação de instância*: quando a operação deve criar um objeto como instância de uma determinada classe.
- *Destruição de instância*: quando a operação vai deletar ou invalidar um objeto preexistente.
- *Alteração de valor de atributo*: quando a operação vai modificar o valor de um atributo de algum objeto preexistente.
- *Adição de ligação* (*link*): quando a operação adiciona uma ligação entre dois objetos. Essa ligação deve corresponder a uma associação entre as classes dos objetos que serão ligados.
- *Remoção de ligação*: quando a operação vai remover uma ligação preexistente entre dois objetos.

Assim, o contrato de um comando de sistema terá entre suas pós-condições operações básicas dentre os cinco tipos acima. Vamos considerar como exemplo o caso de uso "obter livros" da seção 6.4 e o modelo conceitual refinado da Figura 6.10. Neste momento, como já comentamos, o modelo deve conter todas as informações

referenciadas no caso de uso. Caso alguma operação básica não seja possível de realizar no modelo pode ser o caso de refinar o modelo ainda mais.

Para o exemplo, vamos trabalhar com o passo 3 do caso de uso, no qual o leitor informa os livros que deseja emprestar. Observando o modelo conceitual da Figura 6.10, podemos imaginar que as pós-condições deste comando serão pelo menos:

- Criar uma nova instância de Empréstimo, para cada um dos livros informados pelo leitor.
- Adicionar uma ligação entre cada novo empréstimo e a instância que corresponde ao leitor identificado.
- Adicionar uma ligação entre cada novo empréstimo e a instância correspondente de Livro, conforme indicado pelo leitor.
- Se existir uma ligação de reserva entre o leitor e algum destes livros ela deve ser removida.
- Deve-se atualizar o atributo "data" de cada novo empréstimo para corresponde à data corrente.
- O atributo "previsão_devolução" de cada novo empréstimo deve ser atualizado para corresponder à data do empréstimo somada ao atributo "prazo" do livro correspondente.

Precondições são situações referentes a informações que são assumidas como verdadeiras antes de uma operação de sistema ser executada. Assim, como a operação assume que a condição é verdadeira, ela usualmente não precisa verificar isso.

Um exemplo de pré-condição para este comando é o fato de que tanto o leitor quanto os livros indicados efetivamente estão cadastrados no sistema. Ocorre que usualmente o sistema recebe da interface com o usuário não um objeto, mas um código que possivelmente representa um objeto. Se certas precauções forem tomadas, pode-se garantir que apenas códigos válidos são enviados. Assim, para garantir que o código do identificador do leitor seja válido, pode-se habilitar no sistema a opção de informar livro apenas depois que o leitor efetuar *login* com sucesso. Da mesma forma, para garantir que apenas códigos de livros cadastrados são enviados, pode-se fazer com que os livros a serem emprestados sejam selecionados de uma lista, ou, como é mais comum, identificados a partir de chips, códigos de barra ou *QR codes* gerados pelo próprio sistema.

Já as **exceções** são situações de erro referente às informações prestadas que não podem necessariamente ser garantidas antes de se tentar executar a operação. Assim, exceções são situações de erro que devem ser testadas pela operação de sistema.

Por exemplo, se o leitor estiver emprestando livros que ele pegou em uma estante e um desses livros estiver reservado para outro leitor, o sistema não estará garantindo que ele não possa tentar emprestar o livro. Mas se ele tentar, o sistema deve identificar a situação e provocar a exceção, já que ele não pode emprestar um livro que está reservado para outra pessoa. Isso tem que ser verificado pela operação.

Pode-se, também, se for o caso, converter alguma precondição em exceção se for verificado que não é possível garantir que ela seja sempre verdadeira antes de executar a operação. Assim, se você acredita que a operação de sistema deve verificar uma determinada condição quando ela estiver sendo executada, você deve considerar essa condição como exceção e não como precondição, já que essas últimas são garantidas antes e não testadas durante a operação.

Os contratos de consulta de sistema costumam ser bem mais simples pois usualmente apresentam apenas uma descrição da informação retornada pela consulta.

Précondições são raras e não são obrigatórias nesses contratos, e exceções também ocorrem com pouca frequência neste tipo de operação.

No nosso caso de uso "obter livros" observamos que a linha 4 corresponde a uma consulta de sistema, na qual o sistema vai confirmar o empréstimo e informar a data prevista de devolução para cada livro. Assim, o retorno dessa consulta poderia ser descrito como "*Para cada empréstimo associado ao leitor identificado com atributo 'data_devolução' indefinido, retorne o título do livro correspondente e o valor de 'previsão_devolução' do empréstimo*".

Para esta descrição de retorno de informação adicionamos a condição de que a data de devolução esteja indefinida porque senão todos os empréstimos realizados pelo leitor seriam informados, não apenas os que ainda estão ativos. Por outro lado, desta forma, serão listados todos os empréstimos ativos e não apenas os realizados naquele momento. Se o leitor tem algum livro emprestado anteriormente e que ainda não tenha sido devolvido, ele será listado neste momento, o que poderá servir como lembrete de que ele deve devolve-lo.

6.7. Geração de Código e Teste

Uma vez que o modelo conceitual tenha sido plenamente refinado e os contratos das operações de sistema tenham sido estabelecidos, podemos passar ao processo de geração de código. Em função dos artefatos, podemos identificar também dois momentos no processo:

- Geração da estrutura de classes (arquitetura).
- Geração dos métodos.

Para gerar a estrutura de classes em uma linguagem de programação deve-se seguir algumas regras:

- Cada classe no modelo conceitual corresponde a uma classe na linguagem de programação.
- Cada atributo de classe corresponde a um atributo privado na linguagem de programação.
- Se o atributo é imutável, ele deve implementar apenas um método de consulta (*get*).
- Se o atributo não for imutável ele deve implementar métodos de consulta (*get*) e alteração (*set*).
- Todos atributos que não sejam opcionais ou derivados devem ser inicializados no construtor da classe (*constructor*). Assume-se que os atributos opcionais (aqueles que não são obrigatórios e que, portanto, podem ter valor nulo) poderão ser definidos mais tarde (ou nunca) e que os derivados são sempre calculados e assim não têm representação como atributo da classe.
- Cada associação entre classes deve ser implementada de acordo com o padrão de implementação de associação escolhido.
- Toda associação com papel obrigatório deve ser adicionada no construtor da classe.
- Associações imutáveis devem implementar um método de consulta (*get*) e as demais implementam métodos de consulta (*get*) e alteração (*add* e *remove*).

Em relação aos padrões para implementação de associações, pode-se citar três como os mais frequentes, cada um com suas vantagens e desvantagens:

- *Implementação unidirecional*: neste caso, uma das classes possui um atributo que referencia uma ou mais instâncias da outra classe e a outra classe não possui este atributo. Se o envio de mensagens entre os objetos envolvidos for sempre em um

sentido, essa é a estratégia ideal. Mas se houver necessidade de envio de mensagens no sentido oposto, é necessário implementar uma consulta na segunda classe que pesquise e descubra qual ou quais instâncias da primeira classe estão associadas a ela. Assim, essa navegação no sentido oposto será bastante lenta.

- *Amigos mútuos*: neste caso, as duas classes têm referência direta uma para a outra através de seus atributos. A navegação de mensagens é rápida nos dois sentidos, mas deve-se tomar muito cuidado na atualização destes atributos, pois devem sempre ser atualizados nas duas classes para evitar situações inconsistentes para a associação.
- *Objeto intermediário*: neste caso, a associação não é implementada em nenhuma das classes, mas em uma tabela externa que referencia instâncias das duas classes. Assim, quando um objeto precisa enviar mensagem a outro objeto através e uma ligação, ele deve consultar essa tabela para localizar o tal objeto. A navegação é um pouco mais lenta do que no caso de amigos mútuos, mas essa técnica cria classes bem mais desacopladas e fáceis de gerenciar do que no caso da estratégia anterior.

Para cada associação também devem ser implementados métodos de consulta (*get*) nos dois lados ou em apenas um lado, se ela for unidirecional. Já os métodos de modificação (*add* e *remove*) só devem ser implementados nos lados em que a associação não for imutável. Por exemplo, um empréstimo, uma vez associado a um leitor, não pode ser removido deste leitor ou associado a outro leitor. Neste caso, a associação do empréstimo para o leitor é imutável. Além disso, mesmo que a associação seja mutável dos dois lados, os métodos públicos de adição e remoção só precisam ser implementados em uma das classes, pois implementá-los em ambas as classes seria redundante já que fariam exatamente a mesma coisa. Isso não se aplica aos métodos de consulta que, caso a associação seja bidirecional, devem mesmo ser implementados em ambas as classes.

Em relação aos demais métodos a serem implementados, a primeira observação que deve ser feita é em relação aos contratos de operação de sistema obtidos a partir dos casos de uso detalhados. Cada contrato será implementado na forma de um método em uma classe que encapsula o acesso aos demais objetos. Essa classe segue o padrão *controlador-fachada* e apenas ela pode se comunicar com os objetos de domínio, não sendo permitido à interface acessar estes objetos diretamente.

Assim, a implementação de cada método, conforme a linguagem de programação escolhida, deve seguir as especificações dos contratos:

- As pós-condições dos comandos de sistema devem ser implementadas como ações que vão criar e destruir ou desativar objetos, adicionar ou remover ligações entre objetos ou ainda alterar o valor de atributos dos objetos.
- As precondições a rigor só precisam ser testadas durante a fase de testes do sistema. Depois que o sistema passar nos testes assume-se que as precondições serão sempre verdadeiras e assim, não precisam mais ser testadas.
- As exceções precisam ser testadas no código e ações devem ser tomadas caso alguma delas ocorra. Usualmente a ação a ser tomada é informar à interface que uma exceção ocorreu e fica a cargo da interface determinar o que fazer e o que comunicar ao ator neste caso.
- Os retornos de informação devem repassar à interface uma estrutura de informação que seja puramente sintática, ou seja, a interface não deve receber os objetos de domínio propriamente ditos, mas uma representação *read only* deles. Há vários padrões que podem ser usados para isso, sendo um deles o uso de *proxys* ou DTOs

(*Data Transfer Object*), ou ainda a linearização do objeto usando notações como, por exemplo, XML.

Em relação aos testes que devem ser feitos deve-se observar o seguinte:

- Elabora-se um caso de teste pelo menos para cada comportamento possível da operação. No caso do contrato para "obter livros" citado anteriormente, deverá ser feito um teste quando nenhum dos livros tiver reserva e outro teste quando pelo menos um dos livros tiver uma reserva do mesmo leitor. O fato de uma pós condição ser condicionada (com o uso de "se", ou "caso"), já dá a entender que o comando pode ter mais de um comportamento possível.
- Elabora-se um caso de teste pelo menos para cada exceção mencionada no contrato para verificar se a operação realmente detecta e acusa a exceção. Nestes testes, se a operação funcionar normalmente sem dar exceção é porque ela não está corretamente implementada. No caso do exemplo, teríamos que testar também a situação em que um leitor tenta emprestar um livro reservado para outro leitor.

Recomenda-se também que, quando um conjunto de testes for elaborado, sejam verificadas situações limite, para ver como a operação reage nestes casos. Por exemplo, no caso do empréstimo de livro, podemos verificar as seguintes situações limites:

- Empréstimo de zero livros: isso deveria causar uma exceção que ainda não tinha sido identificada no processo de análise.
- Empréstimo de um livro para um leitor que não tem empréstimos ativos: deve funcionar normalmente.
- Empréstimo de uma quantidade de livros que chegará exatamente ao limite máximo que o usuário pode levar: deve funcionar normalmente.
- Empréstimo de uma quantidade de livros que passará exatamente 1 do limite máximo que o usuário pode levar: deve acusar uma exceção.

Evidentemente que nem sempre é possível garantir que um sistema esteja livre de erros, mas a realização dos testes dentro das classes de equivalência definidas e considerando as situações limite já é muito melhor do que fazer testes simplesmente de forma aleatória ou ad-hoc.

6.8. Comentários Bibliográficos

O artigo de Royce [Roy70] tem interesse histórico, pois apresenta uma discussão inicial sobre processo de desenvolvimento de software e sugestões para melhorar este processo.

Wazlawick [Waz12] apresenta uma discussão mais detalhada sobre os modelos de desenvolvimento de software prescritivos, ágeis e também sobre o Processo Unificado. O livro também apresenta informações sobre como planejar e gerenciar um projeto de software, sobre a qualidade do processo e do produto e também sobre como testar, manter e gerenciar versões de um produto de software, especialmente aquele voltado para sistemas de informação.

Outro livro de Wazlawick [Waz15] complementa o anterior apresentando em detalhes técnicas de modelagem de negócio, requisitos, planejamento dirigido por casos de uso, modelagem conceitual, funcional e dinâmica de software. Este livro também detalha aspectos de estimação de esforço, teste de software orientado a objetos, geração de código e banco de dados.

Cockburn [Coc01] é um clássico da área de casos de uso. Em seu livro ele apresenta muitas dicas extremamente valiosas sobre como escrever casos de uso realmente úteis.

Gamma et al. [GHJV95] apresentam um conjunto de padrões de *design*, muitos dos quais úteis para solucionar problemas usuais e recorrentes em modelagem conceitual. Outros livros focados em padrões para modelagem conceitual são os de Fowler [Fow03] e de Larman [Lar04], este último usando a notação UML. O manual completo da UML pode ser encontrado em [OMG11].

Jacobson [Jac94] é um clássico que introduz a notação de casos de uso para modelagem de processos de negócio. Este livro pode ser complementado por outro, mais recente, do inventor dos casos de uso [Jac11]. Probasco [Pro01] também apresenta uma excelente discussão sobre a escolha de bons nomes para casos de uso.

Krol e Kuchten [KK03] apresentam um guia bastante elucidativo sobre todo o Processo Unificado. Este livro também pode ser complementado por outro [Kru03].

[Coc01] Cockburn, A. **Writing Effective Use Cases**. Addison-Wesley, 2001

[FOW03] Fowler, M. **Patterns of Enterprise Application Architecture**. Addison-Wesley, 2003

[GHJV95] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. **Design Patterns: Elements of reusable object-oriented software**. Addison-Wesley, 1995

[Jac94] Jacobson, I. **The Object Advantage: Business process reengineering with object technology**. Addison-Wesley, 1994

[JSB11] Jacobson, I., Spence, I., & Bittner, K. **Use Case 2.0: The guide to succeding with use cases**. Ivar Jacobson International, 2011

[KK03] Kroll, P., & Kruchten, P. The Rational Unified Process Made Easy: A practicioner's guide to RUP. Addison Wesley.

[Kru03] Kruchten, P. (2003). **The Rational Unified Process: An introduction. 3rd Ed.**. Addison-Wesley, 2003

[Lar04] Larman, C. **Applying UML and Patterns: An introduction to object-oriented analysis and design and the unified process 3rd ed**. Prentice-Hall, 2004

[OMG11] Object Management Group. OMG *Unified Modeling Language™* (*OMG UML*), *Superstructure*, *Version* 2.4.1. OMG, 2011

[Prob01] Probasco, L. *What makes a good Use Case Name?* The Rational Edge. March, 2001

[Roy70] Royce, W. Managing the Development of Large Software Systems, *Proceedings of IEEE WESCON*, **26** : pp. 1–9, 1970

[Waz12] Wazlawick, R. S. **Engenharia de Software: Conceitos e práticas**. Rio de Janeiro, Elsevier, 2012

[Waz15] Wazlawick, R. S. Análise e Design Orientados a Objetos para Sistemas de Informação: Modelagem com UML, OCL e IFML. Rio de Janeiro, Elsevier, 2015

6.9. Exercícios

1. Que tipo de projeto de sistema de informação exige maior esforço em termos de modelagem de negócio?

2. Quais as diferenças entre casos de uso e negócio e casos de uso de sistema?

3. Em modelagem de negócio, para que serve o diagrama de atividades e para que serve o diagrama de máquina de estado?

4. Qual a diferença entre exceções e variantes em casos de uso expandidos?

5. Escolha um projeto simples para uma empresa ou organização com a qual você esteja familiarizado ou tenha acesso a pessoas que conheçam a forma como ela funciona. Procure desenvolver os seguintes tópicos:

- Desenvolva o diagrama de casos de uso de negócio com seus respectivos atores.

- Escolha o caso de uso de negócio mais importante para a organização e faça o diagrama de atividades para ele.
- Escolha o objeto de negócio mais relevante para a organização e faça seu diagrama de máquina de estados.
- Identifique a partir dos três diagramas anteriores os casos de uso de sistema e seus atores.
- Construa o modelo conceitual preliminar a partir do diagrama de casos de uso de sistema.
- Identifique casos de uso que estejam faltando no seu modelo a partir das operações CRUD para os conceitos do modelo preliminar.
- Escolha um caso de uso de sistema considerado importante para a organização e apresente sua versão expandida, com fluxo principal, exceções e variantes, se existirem.
- Refine o modelo conceitual a partir de novas informações mencionadas na versão expandida do caso de uso de sistema.
- Escolha um comando e uma consulta deste caso de uso expandido e elabore seus contratos de operação de sistema.
- Usando sua linguagem de programação preferida implemente o modelo de classes do modelo conceitual refinado e as operações de sistema dos contratos elaborados.
- Elabore um conjunto de testes para as operações de sistema implementadas e execute estes testes com dados fictícios.