ANÁLISE E PROJETO DE SISTEMAS

ÍNDICE

Capítulo 1 – Conceitos Básicos de Análise de Sistemas e Projetos 4
1.1. A Crise do Software 4
1.2. Surgimento da Engenharia de Software 5
1.3. Problemas no desenvolvimento de sistemas 5
1.4. Manifesto Ágil 7
1.5. Modelagem de Sistemas 9
Capítulo 2 – Ciclo de Vida de um sistema 12
2.1. Concepção do Sistema 12
2.2. Técnicas de Levantamento 13
2.3. Análise dos requisitos 15
2.4. Projeto 16
2.5. Implementação 17
2.6. Testes 18
2.7. Implantação 19
2.8. XP 20
2.9. Scrum 22
2.9.1 Planejamento no Scrum 23
Capítulo 3 – Estudo de Viabilidade 25
3.1. Viabilidade Organizacional 25
3.2. Viabilidade Técnica 26
3.3. Viabilidade Econômica 27
3.4. Viabilidade Operacional 29
3.5. Viabilidade de Cronograma 31
3.6. Benefícios tangíveis e intangíveis 32
Capítulo 4 – Especificação de Requisitos 34
4.1. Requisitos Funcionais 34
4.2. Requisitos de Qualidade 38
4.3. Requisitos de Negócio 40

4.4. Requisitos de Hardware e Software 42
Capítulo 5 – Orientação a Objetos 44
5.1. Classes e Objetos 45
5.1.1. Relacionamentos 48
5.2. Encapsulamento 52
5.3. Comunicação entre objetos 55
5.4. Herança 57
5.5. Abstração 60
5.6. Polimorfismo 60
5.7. Interfaces 63
Apêndice A – ArgoUML 67
A.1. Download e Instalação do ArgoUML 67
A.2. Conhecendo o ArgoUML 68
A.3. Elaborando os Diagramas da UML 71
A.3.1. Diagramas de caso de uso 72
A.3.2. Diagrama de Classes 75

APRESENTAÇÃO

Essa apostila é indicada para a disciplina de Análise e Projeto de Sistemas dos cursos técnicos da área de Informática relacionados ao Novotec.

A apostila serve de base para iniciar os estudos em desenvolvimento de sistemas, em especial a tópicos relacionados a Engenharia de Software. Apresenta uma visão geral sobre o ciclo de vida do projeto de sistemas, especificação de requisitos, estudos de viabilidade e uma introdução à orientação a objetos. Além disso, você terá um primeiro contato com UML, uma linguagem de modelagem unificada usada na representação de “partes” de um sistema.

O capítulo 1 descreve os conceitos básicos de análise e desenvolvimento de sistemas, apresentando a evolução dos sistemas, os problemas gerados pela crise do software e o surgimento da Engenharia de Software.

O capítulo 2 descreve o ciclo de vida dos sistemas, descrevendo as principais etapas de um processo de desenvolvimento, desde a concepção até a implantação de um sistema.

O capítulo 3 apresenta o estudo de viabilidade, descrevendo os principais tipos de viabilidade, seus objetivos e benefícios. Isso é importante para mostrar que nem sempre um sistema desejado é viável, é melhor planejar muito bem antes de sair desenvolvendo um sistema.

O capítulo 4 descreve os principais tipos de requisitos que devem ser especificados na elaboração de um sistema. Isso é importante, para que você compreenda o impacto que os diferentes requisitos podem gerar na produção de um sistema.

O capítulo 5 descreve a orientação a objetos, o principal paradigma de desenvolvimento de software no mercado atual. Conhecer os conceitos elementares é importante para qualquer profissional na área de desenvolvimento.

O Apêndice A descreve o uso do ArgoUML, uma ferramenta freeware que você poderá usar para praticar a elaboração de desenvolver diagramas em UML, uma linguagem unificada para a modelagem de sistemas.

Capítulo 1 – Conceitos Básicos de Análise de Sistemas e Projetos

Este capítulo apresenta uma visão geral sobre a evolução dos sistemas e da Engenharia de Software, iniciando com a narrativa da crise do software e como essa crise foi resolvida, ou pelo menos minimizada com a adoção de metodologias. Aborda também questões relacionadas à modelagem e sua importância para o desenvolvimento de sistemas.

1.1. A Crise do Software

No Início do desenvolvimento de software, mais especificamente a partir dos anos 70, existia uma forte demanda por sistemas, no entanto, as técnicas existentes para sua produção, aliadas a baixa produtividade e complexidade pertinentes ao processo, gerou muitos problemas de qualidade e manutenção dos sistemas.

O fato de desenvolver sistemas de baixa qualidade que, muitas vezes, não atendia as necessidades dos usuários, gerou o que passou a ser chamado de Crise do Software. Nessa época os sistemas eram produzidos sem muito controle, dependentes totalmente da competência dos profissionais. Obviamente isso fez aumentar a demanda por esses profissionais. Mas, isso não era suficiente.

Como os sistemas foram crescendo e se tornando cada vez mais complexos, houve a necessidade de aumentar o controle sobre as tarefas pertinentes à produção dos sistemas. Se fizermos uma analogia de um sistema produzido, no final da década de 80 com os dias atuais, veremos que a complexidade vem aumentando ano a ano. Esse foi o motivo para o surgimento da Engenharia de Software, uma tentativa de controlar a complexidade envolvida no desenvolvimento de um sistema.

Podemos dizer que os elementos responsáveis pela redução dos problemas trazidos pela crise do software foram os seguintes: a implementação de processos sistematizados, uso de técnicas e ferramentas apropriadas, treinamento contínuo da equipe e mudança na postura dos profissionais, reconhecendo que a presença do cliente durante o processo de desenvolvimento é essencial para o sucesso da grande maioria dos projetos.

Ainda assim, é importante afirmar que os problemas na produção de software continuam os mesmos para os dias atuais, boa parte dos projetos continuam gerando problemas semelhantes. Mais do que nunca, a utilização de ferramentas e processos

padronizados deve ser implementados continuamente, em especial nas empresas produtoras de sistemas.

1.2. Surgimento da Engenharia de Software

Como citamos na seção anterior, a Engenharia de Software teve como objetivo principal reduzir os impactos negativos provenientes da crise do software. Inspirada nas engenharias existentes, a Engenharia de Software define diversas metodologias, focadas no planejamento, implementação e gerenciamento do processo pertinentes ao desenvolvimento de sistemas.

A Engenharia de Software alterou a forma de trabalho, até então artesanal, usada pelos profissionais no desenvolvimento de sistemas, buscando tornar o trabalho mais contínuo, sequencial, que segue um conjunto de passos bem definidos. Isso reduz o lado pessoal e alinha a equipe para torná-la mais eficiente, eficaz e, principalmente, menos dependente de um único membro da equipe.

Atualmente, um profissional que trabalha com Engenharia de Software é alguém capaz de planejar, projetar, modelar, fazer testes e manutenções em diversos tipos de sistemas, sejam do tipo desktop, aplicativos ou para a Internet, não importa o tipo do sistema, os procedimentos e técnicas a serem utilizados são praticamente os mesmos.

Dessa forma, é de se esperar que um Engenheiro de Software esteja apto para atuar nas mais diferentes etapas de um processo de desenvolvimento, como veremos daqui por diante. As atividades mais comuns associadas a esse profissional se concentram no levantamento e especificação de requisitos, no gerenciamento do projeto, definição da arquitetura do sistema, elaboração da documentação entre outras atividades.

1.3. Problemas no desenvolvimento de sistemas

A produção de um sistema, apesar de conter passos e processos semelhantes a criação de produtos físicos, é bastante diferente e desafiadora. Mesmo softwares considerados de sucesso apresentam problemas e bugs de maneira frequente, mostrando que a complexidade, inerente ao processo de desenvolvimento e manutenção, se mostra cada vez mais presente.

Se você pesquisar na Internet pelos termos “software is hard” encontrará diversos artigos e blogs refletindo a respeito das particularidades e dos desafios existentes na produção do software. Uma pesquisa do Chaos Report (1994), alertava para diversos problemas relacionados ao desenvolvimento de aplicações, dos quais destacamos apenas cinco itens.

1. Apenas 10% dos projetos terminam dentro do prazo estimado.
2. Aproximadamente 25% dos projetos são descontinuados.
3. Mais de 50% dos projetos possuem um custo acima do esperado.
4. Apenas 2% das aplicações podem ser usadas quando entregue.
5. Quase 50% do software entregue nunca foi usado.

Apesar de o relatório ser antigo, quase 30 anos atrás, o desenvolvimento atual de sistemas ainda pode passar por problemas semelhantes.

A questão do prazo citada no item 1 continua sendo verdade para uma boa parte dos projetos, uma vez que mudanças na equipe, no orçamento ou mesmo nos requisitos, fazem com que o sistema não seja entregue no prazo estimado.

Em relação ao item 2, ainda existem muitos projetos com este problema por diversos fatores: saída de membros chave da equipe, falta de experiência com a tecnologia envolvida, mudanças políticas, etc.. Parar um projeto antes de terminar pode ser bastante frustrante para a equipe, pois quem trabalha com desenvolvimento sabe o quanto é importante ver seu software rodando. Dedicar um ano na produção de algo que sequer será utilizado, pode causar muita decepção entre os profissionais.

Já o item 3 destaca a questão do custo, fator surpresa que pode mudar com o passar do tempo, em especial se o projeto for grande (e a equipe pequena). Isso mostrava na época e, pode ser aplicado ainda hoje, que a falta de compreensão do todo em relação ao que deve ser produzido e entregue pelo sistema, pode gerar trabalho adicional e, consequentemente, alterar o custo. Normalmente, o aumento do custo está associado a mudanças de requisitos. Conforme o sistema vai sendo desenvolvido, novas necessidades aparecem.

O conteúdo do item 4 era uma verdade absoluta para a época, principalmente em função da metodologia de desenvolvimento usada no início da engenharia do software: a metodologia em cascata. Vamos tentar resumir essa forma de produção

de software. Acreditava-se que para se produzir um sistema era necessário apenas seguir um conjunto de passos sequenciais, semelhante a fazer um produto físico qualquer. Como veremos, as fases para desenvolvimento de um sistema existem até hoje, o que mudou foi a maneira de conduzir esse processo. Por diversas razões, antigamente, um analista de sistemas visitava (fisicamente) a empresa cliente interessada no sistema (que poderia estar localizada em outro estado), coletava todas as informações relacionadas ao sistema, retornava a seu escritório e iniciava a produção do sistema. Depois de um tempo considerável (meses ou mesmo anos), retornava “com o sistema embaixo do braço” e implantava para ser usado. Atualmente é fácil de entender que essa forma de trabalho não tem como dar certo! Vamos descrever melhor esse processo na seção 4 do capítulo 1, relacionado ao manifesto ágil.

No item 5 é abordada uma questão que, provavelmente, é muito semelhante ao cenário atual - praticamente metade dos sistemas produzidos não era utilizada. Imagine se isso ocorresse com automóveis, aviões, pontes, edifícios etc. Quanto prejuízo! Muitas vezes um sistema não é utilizado por mínimos detalhes. Talvez a falta de precisão em cálculos, a falta de comunicação com algum outro sistema, ou ainda, simplesmente porque o usuário não gostou “da cara” da interface gráfica.

Se considerarmos que a maioria dos aplicativos desenvolvidos para smartphones não obtém sucesso atualmente, então talvez nossa estatística tenha piorado ainda mais! Milhares de projetos e sistemas são publicados diariamente nas “stores”, apenas alguns fazem sucesso. Como dissemos, produzir um software é uma tarefa difícil, fazer com que um software se torne um sucesso é mais difícil ainda. Mas, tudo bem, para nós da área da computação, o que nos move são os desafios!

1.4. Manifesto Ágil

Elaborado a partir do ano 2000, o manifesto ágil se refere à uma “declaração de valores e princípios essenciais para o desenvolvimento de software”. Foi um tipo de reunião ocorrida em 2001, em que participantes experientes da área de desenvolvimento de sistemas, apresentaram os aspectos positivos de algumas metodologias que vinham utilizando. O evento gerou um momento de mudança de paradigma, isto é, uma mudança na maneira de conduzir o processo de desenvolvimento.

O manifesto ágil definiu quatro postulados descritos nos parágrafos seguintes.

Postulado 1 - Indivíduos e interações mais do que processos e ferramentas:

Os processos de desenvolvimento de software existentes definiam uma disciplina rígida que deveria ser seguida à risca em todas as fases. Segundo os participantes do manifesto ágil, essas regras nem sempre contribuem para aumentar a qualidade e velocidade em relação a produção do software. Desta forma, foi considerado na época que as interações entre os participantes da equipe eram mais eficientes do que seguir um plano rígido definido pelo processo de desenvolvimento. Por causa disso, uma das características presentes nas metodologias de desenvolvimento ágeis é a presença da reunião diária - uma maneira de manter a equipe focada e atualizada quanto às alterações exigidas dentro do processo.

Postulado 2 - Trabalhar no software mais do que documentação abrangente:

A maioria dos profissionais de desenvolvimento de software (os programadores) gostam mais de trabalhar na codificação de um sistema ao invés de realizar sua documentação. Esse segundo postulado descreve que é mais importante trabalhar no software do que manter constantemente atualizada a documentação do sistema. Nas metodologias da época era comum serem exigidas muitas documentações, pois se acreditava que ao documentar existiam maiores possibilidades de organizar o desenvolvimento e, consequentemente, aumentar a qualidade do sistema e também as chances de sucesso. No entanto, com o desenvolvimento de novas ferramentas que facilitam o compartilhamento e a integração, tanto do desenvolvimento de um sistema, quanto de sua documentação, manter a equipe atualizada tornou-se muito mais simples. É importante dizer que o manifesto ágil não descarta a documentação, é uma questão de prioridade, melhor trabalhar no software do que investir um tempo exagerado em sua documentação.

Postulado 3 - Colaboração do cliente mais do que negociação contratual:

Esse postulado considera essencial a participação do cliente durante todas as etapas da produção do sistema. Manter o cliente junto à equipe é um procedimento que pode representar uma economia considerável, de tempo e recursos, no momento de produzir um novo software. Isso é verdade na maioria dos casos porque

os desenvolvedores nem sempre conhecem o processo no qual o sistema será inserido. Por outro lado, espera-se que o cliente tenha toda a experiência necessária (no processo) para orientar a equipe de desenvolvimento. Produzir um sistema sem a presença constante do cliente é “viver na corda bamba”, é correr o risco de ter retrabalho e, consequentemente, aumento nos custos de produção. No século passado, muitas vezes o cliente era visto como um vilão, pois vivia exigindo alterações depois que o sistema estava pronto! Felizmente isso mudou, hoje o cliente é visto como um parceiro.

Postulado 4 - Responder às mudanças mais do que seguir um plano:

Essa premissa talvez resuma a proposta do manifesto ágil. Quando falamos a respeito de metodologia ágil, o termo ágil não representa, necessariamente, o fato de produzir um software rapidamente. Esse termo está mais ligado com a velocidade com o que a equipe e, consequentemente, o sistema, deve responder às mudanças durante as etapas do desenvolvimento. Nas metodologias anteriores, o plano a ser seguido era definido no início, desde a concepção do sistema, e deveria ser seguido rigidamente. Atualmente, num momento em que as mudanças são contínuas, seguir um plano desde o início, sem poder fazer ajustes, parece um contrassenso que normalmente não funciona. O mercado exige que as empresas sejam flexíveis, que seus colaboradores sejam flexíveis e também que seus sistemas sejam flexíveis. Enfim, é importante que a equipe envolvida na produção do sistema tenha uma mente aberta, flexível, sempre pronta para mudanças.

Resumidamente, após o manifesto ágil surgiram diversas metodologias de desenvolvimento de software seguindo esses postulados. Podemos considerar que houve uma mudança de paradigma, uma maneira diferente de pensar ou de conduzir o desenvolvimento de um sistema. Essas novas metodologias passaram a produzir um sistema em etapas, em ciclos, ou ainda em iterações. Isso ficará mais claro quando estudarmos as metodologias XP e Scrum, no próximo capítulo.

1.5. Modelagem de Sistemas

A modelagem é utilizada por diversas áreas do conhecimento, em especial no desenvolvimento de sistemas. A modelagem se refere a elaborar algum tipo de representação, um esboço que possa representar um objeto real ou de software. A

modelagem cria uma representação visual de alguma coisa, normalmente para facilitar a compreensão do objeto a ser elaborado. Quanto mais complexo for o objeto, maior será a indicação da modelagem como ferramenta para facilitar o entendimento do que deverá ser produzido.

Ao construir uma residência, por exemplo, podemos citar diversos tipos de modelo: a planta da casa, um esboço da fachada, o esquema elétrico, o esquema hidráulico, entre outros. Cada modelo representa um aspecto diferente do mesmo objeto sendo modelado. Além disso, quanto maior for a casa, ou o prédio, mais importante será o papel da modelagem.

Usando a modelagem, é possível ter uma boa ideia de como o objeto ficará depois de pronto, no caso a casa. O mesmo acontece com um sistema de software: antes de realizar sua construção, podemos elaborar diversos modelos, cada um representando um aspecto diferente do sistema. Um profissional da área de desenvolvimento, tipicamente um analista de sistemas pode elaborar diagramas (modelos) que representam as diferentes partes de um sistema, contendo seus relacionamentos e integrações. Esses diagramas podem ser encaminhados para a equipe de desenvolvimento, mais especificamente para os programadores. Os programadores podem interpretar os diagramas e realizar sua implementação.

A modelagem é mais utilizada nas fases iniciais do desenvolvimento de um sistema, mas pode ser útil a qualquer momento do ciclo de vida de um software. Um modelo pode ser usado para ajudar a entender o que deve ser feito pelo sistema, como para validar se a funcionalidade planejada está funcionando corretamente. A modelagem não é uma “ciência” exata, isso significa que pessoas diferentes podem fazer modelos diferentes para representar o mesmo sistema. Como as necessidades do sistema mudam no decorrer do tempo (às vezes, de um dia para o outro), é muito comum ajustar o modelo durante o ciclo de vida do projeto. Além dessa volatilidade das necessidades, nem sempre o analista possui a visão completa do que deve ser feito no momento em que ele está modelando o sistema.

Como já citado, o processo de modelagem ajuda na compreensão do que deve ser feito, além disso, ao se elaborar diagramas e outros artefatos, ocorre um processo de documentação do sistema. Apesar de ser uma tarefa difícil de ser

realizada, manter a documentação atualizada é muito importante para a equipe de desenvolvimento. Caso um membro da equipe venha a sair e, consequentemente, outro membro comece a fazer parte da equipe, uma boa documentação pode facilitar o trabalho de adaptação ao sistema.

O processo de modelagem, portanto, pode trazer diversas vantagens, ou diversas razões para a sua utilização. Em primeiro lugar, um modelo ajuda no gerenciamento da complexidade inerente à elaboração de um sistema. Em segundo lugar, o modelo permite realizar a comunicação com equipe envolvida: da mesma forma que um engenheiro elabora uma planta e a entrega ao pedreiro para ser executada, um analista de sistemas pode elaborar um diagrama e entregá-lo ao programador para que este realize sua codificação. Como vimos, um modelo também pode facilitar a captura dos requisitos e, dependendo da visão que a equipe possui no momento de planejar o sistema, a modelagem pode ajudar a compreender o futuro do sistema, identificando possíveis funcionalidades que deverão ser implementadas futuramente.

Resumidamente, um modelo pode ser compreendido como uma simplificação da realidade que pode ajudar um profissional, seja ele da área de sistemas ou de qualquer outra área, a entender o problema a ser realizado. Um problema grande e complexo pode ser dividido em pequenos problemas, facilitando o entendimento.

No final da apostila, mais especificamente no apêndice A, você pode encontrar a exemplificação da modelagem usando dois diagramas da UML (linguagem de modelagem unificada) por meio da ferramenta ArgoUML: o diagrama de casos de uso e o diagrama de classes, dois dos mais importantes diagramas.

REFERÊNCIAS

R.S. Pressman, B.R. Maxim, B.R., **Engenharia de Software: Uma Abordagem Profissional**, 8ª edição, Ed. McGraw-Hill, ISBN 9788563308337, 2016.

J.F. Peters, **Engenharia de Software - Teoria e Prática**, ISBN 8535207465, Campus, 2001.

Capítulo 2 – Ciclo de Vida de um sistema

O ciclo de vida de um sistema, ou ciclo de desenvolvimento de software (CLCD), compreende o conjunto de fases envolvidas no desenvolvimento de um sistema, desde sua concepção até a instalação. O nome e a quantidade das fases do ciclo podem variar dependendo da literatura consultada ou da metodologia de desenvolvimento utilizada (Scrum, XP, iRUP etc.). Os nomes mais comuns para essas etapas do ciclo são: Concepção, Levantamento de Dados, Análise de Requisitos, Estudo de Viabilidade, Design, Projeto, Codificação, Teste, Instalação, Implantação, Deploy, Manutenção, entre outras.

Cada fase tem sua função e pode exigir perfis profissionais diferentes. Projetos atuais possuem equipes multidisciplinares, ou seja, pessoas com perfis e competência diferentes e complementares. Em função disso, nem sempre as fases de um ciclo de vida são sequenciais, podendo ser executada mais de uma fase ao mesmo tempo. Seja qual for a quantidade de fases, ou mesmo a metodologia escolhida, o desenvolvimento de um sistema exige a execução de diversas tarefas para que o produto final possa ser implantado, com uma boa qualidade e dentro do prazo definido no cronograma do projeto. Dizemos que uma versão de produção do sistema estará pronta quando todas as fases forem cumpridas devidamente.

No cenário atual, a produção de um sistema é dividida em iterações, em ciclos. Isso faz com que seja necessário repetir as fases do processo (análise, design, codificação etc.) para cada ciclo. A cada ciclo encerrado, o sistema é incrementado, por isso dizemos que a metodologia de desenvolvimento é iterativa (se repete várias vezes) e incremental (aumenta várias vezes).

Dito isso, as seções seguintes apresentam as diversas fases do ciclo de vida, mas, como dissemos, não devem esgotar o assunto relacionado ao desenvolvimento de sistemas.

2.1. Concepção do Sistema

Trata-se da primeira fase a ser realizada na grande maioria dos projetos de software, afinal, antes de iniciar a produção de um sistema é preciso entender o que deve ser feito. Para isso, a melhor maneira é ter uma conversa com os principais interessados no software. Quanto melhor o analista conseguir compreender as necessidades dos usuários, maiores chances de sucesso o sistema terá. Mesmo que

essa conversa seja informal, pode-se chegar a conclusão de que o sistema tem plenas condições de ser realizado no tempo esperado, ou então, pelo contrário, concluir que a equipe disponível não poderá atender as necessidades do projeto, cancelando o início do projeto desde sua concepção.

Portanto, essa etapa é extremamente importante para se identificar um escopo inicial (abrangência do projeto). Uma coisa é fazer um sistema que funcione numa máquina local, outra coisa é fazer esse mesmo sistema funcionar num ambiente Web ou num aplicativo de smartphone. Desde a concepção pode-se definir, por exemplo, que na primeira fase do projeto o sistema funcionará apenas no ambiente de rede local (em versões futuras o sistema poderá ser usado em aplicativos, por exemplo).

2.2. Técnicas de Levantamento

Uma vez que o sistema foi concebido, pelo menos numa ideia inicial, parte-se para o levantamento de dados, uma etapa que tem o objetivo de identificar as reais necessidades dos usuários e demais interessados (clientes, fornecedores, gerência etc.). Cada usuário poderá possuir necessidades e interesses distintos e é importante que todos eles sejam atendidos, afinal, a utilização diária do sistema pelos usuários depois da implantação, estará diretamente relacionada com o sucesso do sistema.

Essa etapa de ouvir o usuário é bastante difícil, pois nem sempre os usuários sabem exatamente o que precisam, ou não conseguem expressar isso de maneira imediata. Muitas vezes os usuários “entendem” o que realmente precisam apenas no meio do processo de desenvolvimento. A literatura chama isso de volatilidade de requisitos, isto é, os requisitos mudam muito rapidamente. Você já deve ter notado a grande importância dessa etapa do ciclo de vida do projeto, em especial para o analista de requisitos - se o analista não conseguir identificar corretamente os requisitos no início do projeto, isso poderá trazer grandes problemas e custos desnecessários. Para minimizar essa falta de informação inicial, e que pode comprometer seriamente o andamento e futuro do sistema, surgiram diversas técnicas de levantamento. Furgeri (2013), descreve as principais técnicas utilizadas:

- **Entrevista**: uma conversa com o usuário (podem existir diversos usuários) com o objetivo de levantar o máximo de informações possíveis a respeito das

funcionalidades que devem estar presentes no sistema a ser construído. Até pouco tempo atrás essas entrevistas eram presenciais e exigiam deslocamento dos envolvidos, hoje com o avanço da tecnologia essa técnica se tornou muito mais fácil e economicamente viável. Imagine os custos envolvidos numa viagem para outro estado apenas para entrevistar os usuários.

- **Seleção de documentos**: técnica importante quando um processo já existente for implementado pelo sistema. Os documentos usados no processo atual podem ser físicos ou digitais e, normalmente, contém informações essenciais a serem analisadas e tratadas no sistema a ser desenvolvido. O contato com os documentos usados no processo atual pode ajudar um analista a melhorar sua visão sobre as necessidades dos usuários, tanto em relação às funcionalidades quanto aos dados envolvidos e que deverão ser mantidos em um banco de dados.
- **Questionário**: nessa técnica são elaboradas questões para que os usuários envolvidos respondam. As perguntas não precisam ser, necessariamente, voltadas apenas à identificação das funcionalidades requeridas pelo sistema que será desenvolvido, mas podem abordar também a experiência do usuário em sistemas semelhantes, os equipamentos que eles utilizam, entre outras. Por outro lado, o foco das questões deve estar concentrado no levantamento de requisitos, ou seja, nas necessidades do sistema. A técnica do questionário é mais indicada quando existe uma quantidade elevada de usuários, imagine ter que fazer uma entrevista com dezenas de usuários.
- **Brainstorm**: conhecido popularmente como “toró de parpite”, tem como característica principal proporcionar um ambiente onde todos os interessados pelo sistema podem apresentar suas sugestões. Pode ser feito fisicamente ou remotamente, o mais importante é criar um clima amigável de colaboração em que ocorra uma verdadeira “chuva de ideias”. Nessa técnica, normalmente uma pessoa da equipe lidera discussões e anota as sugestões para análise futura. Com certeza o grande ponto forte dessa técnica é a participação dos envolvidos no sistema, sejam gerentes, diretores, usuários comuns, clientes, fornecedores etc.
- **Prototipação**: trata-se de uma técnica bastante utilizada no mercado atual em que são feitos esboços das telas do sistema. Ao desenhar uma tela fica

mais fácil tirar dúvidas com os usuários, pois o usuário estará olhando um desenho como se estivesse “pilotando” o próprio sistema. Um protótipo pode funcionar como um mecanismo de validação dos requisitos, pois é possível verificar se o analista entendeu o que deve ser feito e se o usuário compreende como isso irá funcionar. Nossa experiência mostra que os usuários ficam “empolgados” quando veem uma parte do sistema desenhado (pelo menos na maioria das vezes..rsrs). Com as ferramentas disponíveis atualmente é muito fácil produzir protótipos navegáveis, uma tela pode chamar a outra simulando exatamente o que acontecerá quando o sistema estiver em funcionamento.

- **Casos de Uso**: representa uma técnica bastante aplicada, normalmente, após as citadas anteriormente. Os requisitos encontrados até o momento da fase de levantamento, podem ser representados graficamente, ou mesmo em modo textual. Independentemente do tipo utilizado, a intenção é definir funções que cada usuário poderá executar por meio do sistema, daí o nome casos de uso. Os casos de uso podem ser elaborados como histórias do usuário como, por exemplo: “como professor da universidade eu gostaria de lançar as notas dos alunos de uma turma”. Se você tiver diversas histórias semelhantes, terá descrito, mesmo que de maneira breve, diversas funcionalidades que deverão estar presentes no sistema a ser desenvolvido.

É importante salientar que não existe uma técnica de levantamento que seja a melhor, dependendo das circunstâncias pode ser que uma técnica seja mais indicada que as outras, ou então pode existir um cenário em que as diversas técnicas podem ser combinadas. Em resumo, não existe uma regra apontando qual ou quais técnicas devem ser usadas, nós usamos as que são mais apropriadas para o momento, dependendo das necessidades do sistema.

Uma vez que o sistema foi concebido e as principais funcionalidades foram capturadas por meio das técnicas de levantamento, podemos passar para a fase de análise dos requisitos.

2.3. Análise dos requisitos

Nessa etapa do projeto ainda estamos trabalhando com os requisitos, uma fase em que as informações obtidas do cliente são refinadas, analisadas e, normalmente, ampliadas. Um analista de sistemas deverá “fazer o dever de casa”,

analisando e meditando os requisitos, quase que sonhando com eles, na tentativa de identificar esses deverão ser inseridos e manipulados pelo sistema. Por meio desse estudo é possível começar a vislumbrar etapas posteriores do ciclo de vida do projeto como, por exemplo, a definição de classes (caso o software seja orientado a objetos), definição das tabelas do banco de dados (caso o banco seja do paradigma relacional), além de outras tarefas relacionadas a arquitetura utilizada para produzir o sistema.

O resultado da análise de requisitos permite gerar um documento chamado de especificação de requisitos, cujos detalhes são apresentados no capítulo 4. O conteúdo desse documento pode variar bastante dependendo do tamanho do projeto. Um projeto pequeno poderá conter apenas uma rápida descrição dos casos de uso, mas, um projeto maior poderá agregar um conjunto de modelos, diagramas da UML, diagramas de banco de dados, protótipos entre outros. Poderá conter também requisitos de hardware e software necessários para manter o sistema rodando após sua implantação.

O documento de especificação de requisitos poderá ser usado na apresentação ao cliente do que será elaborado, poderá gerar um contrato de prestação de serviços ou algum documento que sirva para validação do que será contratado. É possível que neste momento existam conflitos e uma negociação poderá ser necessária. É possível que requisitos sejam inseridos, ajustados ou ainda retirados do projeto após essa etapa de validação, o mais importante é que os usuários saiam satisfeitos e os desenvolvedores com a certeza do que deverá ser produzido.

2.4. Projeto

Como o nome sugere, na fase de projeto um analista irá reunir todas as informações levantadas nas etapas anteriores para realizar o projeto do sistema. O projeto pode ter duas direções: o lado da gestão do projeto, em que se define o cronograma e todas etapas que devem ser realizadas até o desenvolvimento final do sistema e o lado do projeto do sistema em si, fase chamada de design normalmente realizada por um arquiteto de software.

A elaboração do cronograma de execução do projeto pode ser realizada numa folha de papel, numa planilha eletrônica ou ainda por meio de uma ferramenta de

software específica para essa tarefa, a forma mais comum atualmente. Existem inúmeras ferramentas disponíveis no mercado, dentre as quais: Trello, Jira, Microsoft Project etc.

Independentemente da ferramenta utilizada, um gestor de projetos deverá listar todas as tarefas a serem realizadas e o tempo estimado para cada uma delas, juntamente com os membros da equipe responsáveis por sua execução. Baseado na experiência anterior do gestor, para cada tarefa é estimado um tempo para sua finalização, tempo que deverá ser monitorado. O objetivo do monitoramento desse tempo não é punir um membro (até pode ser), mas, verificar se as estimativas estão sendo feitas corretamente, dentro do prazo previsto. Isso pode ser um indicador para revisão do tempo em tarefas futuras, aumentando ou reduzindo o prazo para execução de tarefas semelhantes.

O projeto pode ser dividido em sprints, como será visto no Scrum (seção 2.9), ou pode assumir um outro formato, dependendo da experiência da equipe e do tamanho do projeto. O importante é controlar as tarefas, atribuindo algum tipo de sinalizador, para identificar rapidamente o status de cada tarefa (em execução, pendente, finalizada etc.).

Uma vez elaborado o cronograma do projeto, é importante que o gestor realize o acompanhamento e monitore tudo o que está acontecendo durante a elaboração do projeto. Talvez seja necessário mover, remover ou inserir membros na equipe em função do andamento do cronograma.

2.5. Implementação

A implementação se refere a fase de codificação dos programas que farão parte do projeto. É a etapa realizada principalmente pelos programadores de computador usando uma (ou várias) linguagem de computador. Os requisitos do sistema, coletados em fases anteriores, serão codificados e testados para garantir que atendam às necessidades dos usuários.

Essa fase pode consumir diversos artefatos produzidos em fases anteriores. Documentos, diagramas e protótipos podem ser usados como guias para o desenvolvimento de uma funcionalidade qualquer do sistema. O ideal é que um analista, que já conhece as necessidades do sistema, oriente a equipe de programação para desenvolver as funcionalidades, apresentando a especificação de

requisitos, narrativas de casos de uso, diagramas entre outros documentos. Eles serão de extrema importância para manter a equipe na direção correta.

Quanto maior for o sistema a ser construído, maiores serão as necessidades de um líder para guiar o desenvolvimento das funcionalidades. Elas podem ser criadas em partes (ou módulos) que serão integradas pouco a pouco no produto final. Atualmente, é bastante comum um sistema Web ser dividido em Front-End e Back-End, podemos ter programadores com perfis diferentes trabalhando no mesmo sistema. Cabe ao analista ter a flexibilidade para saber orientar os dois lados, sempre guiado pela documentação elaborada.

Em termos de codificação, existem inúmeras ferramentas disponíveis, dependendo da linguagem utilizada. Fica difícil citar uma ferramenta em especial, visto que cada linguagem pode ter uma mais indicada, aliada a preferência de cada programador. Apenas para citar, algumas das principais ferramentas atuais para a codificação são Eclipse, NetBeans, Visual Studio, Visual Studio Code, PyCharm, Atom, Jupyter, entre centenas de outras.

2.6. Testes

Etapa essencial no desenvolvimento de sistemas que ganhou destaque nos últimos anos para aumentar a qualidade do software. Não é de hoje que erros no software causam prejuízos enormes e muita dor de cabeça. Apenas para citar, em 1962, um erro de cálculo no software de voo da sonda Mariner fez com que o foguete tivesse que ser destruído. Em 1982, uma falha de programação causou a explosão num gasoduto soviético. Em 1993, a Intel teve um prejuízo milionário por um erro na divisão de números com ponto flutuante no processador Pentium.

A lista de problemas em software é muito grande, envolvendo jogos, sistemas operacionais, redes, aplicativos, enfim, em qualquer dispositivo em que esteja rodando o software. Às vezes, um bug pode ser inofensivo, ou em outros casos pode causar mortes de vidas humanas. Seja como for, os testes são essenciais para melhorar a qualidade do software, apesar de não garantir que está livre de erros.

A fase de teste também exige planejamento para definição do tipo de teste que será realizado, do que será testado, do momento em que será testado e qual ferramenta será usada, ou ainda se será feito manualmente. Teste da caixa branca, caixa preta, de estresse, de integração, são alguns tipos de teste que podem ser

considerados no planejamento. Além do planejamento, existem muitos modelos (templates) usados para teste de software, cada empresa define um padrão que melhor se adapta as suas necessidades. Para cada teste a ser realizado, é possível se definir um plano que define a sequência e as condições para a realização do teste.

Visando aumentar a qualidade do software por meio de testes, existem normas a serem seguidas no mercado. Por exemplo, a norma IEEE 829, um padrão internacional, define todos os itens que um plano de teste deve incluir. Os tópicos seguintes descrevem alguns itens exigidos pela norma:

- **Identificador**: um nome ou código que identifica unicamente o plano.
- **Descrição**: contém o objetivo do teste e os recursos necessários para sua realização.
- **Itens de teste**: descreve os elementos a serem testados e sua ordem de prioridade, podendo abranger funcionalidades do usuário ou do sistema.
- **Estratégia do teste**: descreve qual será a metodologia utilizada para cada tipo de teste e as ferramentas a serem usadas (ou não). Importante definir os critérios de aceite ou recusa da funcionalidade a partir do teste.

Atualmente existem ferramentas para testagem automática de software que facilitam principalmente a manutenção, pois qualquer alteração pode gerar impacto significativo no funcionamento de um sistema. Por isso, sempre que uma mudança for realizada no sistema, é importante rodar rotinas de teste, se estas forem automáticas o tempo de execução será reduzido.

Apenas para exemplificar, suponha que você está trabalhando no desenvolvimento de um sistema Back-End usando a linguagem Node. Durante a codificação, poderá usar a ferramenta Insomnia para criar funções de teste, ou seja, a cada funcionalidade produzida será possível testar (sem a existência da aplicação final do Front-End) sua execução e verificar se está recebendo os parâmetros corretos e, principalmente, se está retornando os resultados esperados. Poderá também usar a ferramenta Selenium para criar rotinas automáticas de teste para cada alteração realizada.

2.7. Implantação

Apesar de ligeiramente diferentes, a implantação pode ser confundida com a instalação de um sistema. A diferença é que a instalação normalmente é um

processo rápido, pontual, dizer que um sistema está instalado não quer dizer que esteja implantado, pois a implantação é algo mais amplo que exige a mobilização de setores, usuários, treinamentos etc.

No início do surgimento dos sistemas, e dos problemas relacionados à crise do software, o sistema todo (completo) era implantado de uma única vez, gerando uma série de problemas. Atualmente, a implantação de um sistema pode ocorrer em momentos diferentes do ciclo de vida. Se considerarmos as metodologias SCRUM e XP, descritas nas seções seguintes, é possível implementar o sistema em partes, incrementando as funcionalidades a cada implantação.

Implantar um sistema significa que ele passará a ser utilizado pelos usuários, ele estará em produção. Não confunda os termos usados em sistemas: uma versão de produção é uma versão que está pronta para ser utilizada (ela não está sendo produzida, já está pronta). A implantação de um sistema pode ser rápida ou demorada, dependendo do tamanho do sistema. Um software de ERP (Enterprise Resource Planning), por exemplo, pode durar anos, pois pode exigir a reestruturação de processos de uma empresa. Em função disso, a implantação pode conter diversas etapas como treinamentos, validações, adaptações, customizações entre outras.

Apesar de parecer que a implantação é a última fase do ciclo de vida, na verdade isso não é verdade. O ciclo de vida de um sistema irá durar enquanto o software estiver sendo usado, enquanto o sistema estiver em produção haverá mudanças e ajustes a serem realizados. O sistema é quase um ser vivo, precisa ser cuidado o tempo todo.

2.8. XP

A sigla XP se refere a Extreme Programing, e pode ser considerada uma metodologia ágil de desenvolvimento de software, visto que define um conjunto de "regras" que podem ser utilizadas na elaboração de sistemas. Além disso, XP tem o objetivo de promover o correto andamento de um projeto de maneira que todo o planejamento possa ser cumprido, seja ele em relação ao ciclo de vida do sistema, aos prazos e orçamentos. É considerado como uma metodologia ágil, indicado para projetos de pequeno e médio porte.

Como vimos anteriormente, definir todas as funcionalidades que um sistema deverá possuir logo no início não é uma tarefa simples. Mesmo fazendo todo o

planejamento e seguindo as melhores práticas, nem sempre será possível compreender o todo de uma única vez. Isso sempre foi um problema na metodologia em cascata, pois considerava possível levantar todos os requisitos desde a concepção do sistema. Isso não é verdade para a maioria dos sistemas e, principalmente por causa desse fato, tivemos a crise do software. Portanto, é necessário realizar um estudo minucioso na fase de levantamento de dados, para evitar que os erros se propaguem pelas diversas etapas da produção.

Utilizar a metodologia definida pelo XP significa passar pelas etapas do desenvolvimento diversas vezes, pois este propõe um desenvolvimento iterativo, isto é, a execução do projeto deve ser dividida em partes, chamadas de releases.

O XP confronta com as metodologias não ágeis (ou tradicionais), no sentido em que gera diversas entregas intermediárias ao invés de entregar o sistema todo de uma só vez. Isso pode representar uma vantagem adicional em relação ao custo do projeto, pois um erro identificado numa fase inicial, impede sua propagação pelas outras iterações. A tendência é reduzir o custo de produção, uma vez que o custo de uma modificação no software cresce em função do tempo.

Em relação aos requisitos, o XP promove a participação do cliente desde o início do ciclo de vida. Baseado nas histórias identificadas pelos clientes, organiza as histórias com base na prioridade e direciona o desenvolvimento do sistema. Como o nome do XP sugere (programação extrema), o foco é mais direcionado ao processo de codificação, deixando a documentação em segundo plano. A Figura 2.1 ilustra um quadro com pequenos cartões contendo as histórias dos usuários, procedimento bastante comum ao XP. No caso, as histórias estão divididas em To Do (a fazer), Doing (fazendo) e Done (prontas).

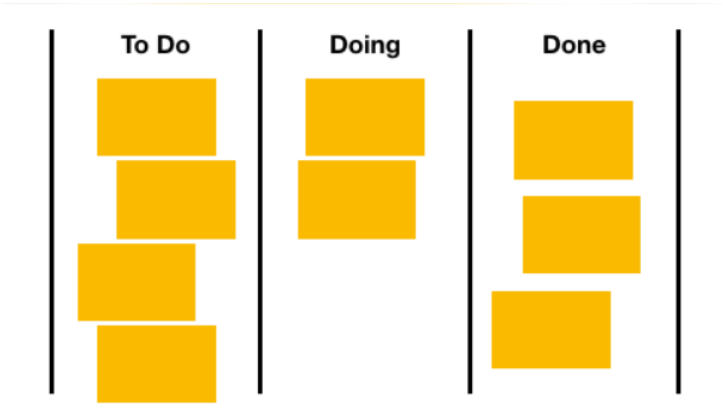

Figura 2.1 – Histórias dos clientes (https://cucumber.io/blog/bdd/user-stories-are-not-the-same-as-features/)

A gestão e controle das histórias podem ser feitas em um quadro físico, ou por meio de ferramentas de software como Trello (https://trello.com/) e Jira (https://www.atlassian.com/br/software/jira). Para manter a equipe atualizada a respeito do andamento da implementação e teste das histórias, o XP define reuniões rápidas diárias. Na maioria das vezes, alguns minutos por dia são suficientes para manter a equipe informada e não perder o foco, uma tentativa de manter o cronograma do projeto.

Outra característica proposta pelo XP é a programação em pares. Os programadores trabalham em duplas, enquanto um codifica o outro confere e sugere melhorias. Muitas empresas adaptam a disposição dos computadores para permitir esse trabalho em duplas. A proposta é interessante, desde que existam duas pessoas dispostas e com perfis para esse tipo de atividade. Nem todos os programadores se adaptam a esse modelo.

2.9. Scrum

Assim como o XP, o Scrum é uma metodologia ágil usada na gestão de projetos de software. Apesar de não ter sua origem na computação (o Scrum é proveniente da manufatura), é uma das metodologias mais usadas e exigidas no mercado de trabalho, presente em muitas vagas relacionadas a projetos de software.

Da mesma maneira que o XP, o Scrum pressupõe entregar um sistema em partes, de maneira incremental. Assim como em qualquer metodologia, o Scrum define a responsabilidade de cada integrante da equipe por meio da atribuição de papéis. A equipe possui três responsabilidades principais: a) **Product Owner**: dono ou proprietário do produto, normalmente um membro que entende dos processos do negócio para o qual o sistema será desenvolvido. Em função disso, espera-se que seja o principal responsável por ajudar a definir as funcionalidades esperadas; b) **Scrum Master**: um membro que seja facilitador, mediador, um motivador que ajude a equipe e a mantenha no rumo certo, tanto em relação ao desenvolvimento do sistema quanto a utilização das práticas do Scrum. Normalmente, alguém que atua junto ao Scrum Owner e possui a visão de todo o sistema que deverá ser desenvolvido; c) **Scrum Team**: a equipe responsável por “colocar a mão na massa”

e desenvolver o sistema, seguindo todas as etapas definidas e combinadas com o Scrum Master.

2.9.1 Planejamento no Scrum

Antes de iniciar a elaboração do sistema, o Scrum define um conjunto de funcionalidades a serem desenvolvidas. A esse conjunto de funcionalidades é dado o nome de Product Backlog. Obviamente, quanto maior for o sistema, maior será o número de funcionalidades a serem definidas. Outro aspecto importante é priorizar as tarefas e criar uma estimativa de tempo para a conclusão de cada uma delas. Dessa forma, temos um planejamento inicial das principais tarefas a serem realizadas para a produção do sistema.

A Tabela 2.1 ilustra um Product Backlog de exemplo contendo as principais funcionalidades para um sistema de controle de processos.

Product Backlog – Sistema de Controle de Processos		
Funcionalidade	**Prioridade**	**Tempo Estimado**
Definir o modelo do banco de dados	1	5
Gerenciar o cadastro de usuários	1	8
Controlar de Login	1	5
Gerenciar o cadastro de processos	2	10
Controlar as ocorrências dos processos	2	20
Controlar os anexos das ocorrências	2	20
Gerenciar alterações do Status	2	20
Elaborar o Dashboard	3	10
Gerenciar notificações no Dashboard	3	10
Gerenciar o acesso dos usuários	3	10
Total estimado		108

Tabela 2.1 – Exemplo de Product Backlog - baseada em Furgeri, 2013.

A partir da definição do Product Backlog, as tarefas podem ser divididas em partes, chamadas de Sprints (no XP são chamadas de releases). Uma estratégia é dividir as funcionalidades em Sprints a partir das prioridades. Dessa forma, podemos dividir nosso exemplo de Backlog em três Sprints. No Sprint, podemos realizar o detalhamento de cada funcionalidade em tarefas (e/ou sub tarefas) e alocar cada uma delas aos diferentes membros da equipe. Com isso podemos ter diferentes membros trabalhando em um mesmo Sprint, cada um dentro do seu perfil profissional.

Dependendo do tamanho do projeto e da equipe envolvida, é possível que vários sprints estejam acontecendo ao mesmo tempo, aumentando a complexidade do gerenciamento do projeto. Da mesma maneira que afirmamos no XP, para gerenciar um projeto no Scrum, é possível utilizar ferramentas de software, tais como Trello e Jira.

Existem outros detalhes relacionados ao Scrum que não trataremos aqui, o assunto é bastante amplo, visto a quantidade de livros existentes sobre essa importante metodologia.

REFERÊNCIAS

FURGERI, S. **Modelagem de Sistemas Orientados a Objetos** – Ensino Didático. 1. Ed. São Paulo: Editora Érica, ISBN 978-8536504612, 2013.

R.S. Pressman, B.R. Maxim, B.R., **Engenharia de Software: Uma Abordagem Profissional**, 8ª edição, Ed. McGraw-Hill, ISBN 9788563308337, 2016.

I. Sommerville, **Engenharia de Software**, 9a edição, ISBN 9788588639287, Pearson Education, 2011.

Capítulo 3 – Estudo de Viabilidade

Este capítulo traz conceitos do Estudo de Viabilidade, um dos principais processos para análise sob diversos pontos de vista, para, como o próprio nome já diz, avaliar se o projeto a ser desenvolvido é viável. Esse estudo se faz necessário quando a organização precisa decidir se o novo sistema a ser desenvolvido (ou modificado) é viável em diversos sentidos. Dependendo da literatura consultada referente ao estudo de viabilidade, podemos encontrar nomes e conceitos ligeiramente diferentes. Em função disso, determinados assuntos podem se sobrepor entre os diferentes tipos de viabilidade.

Depois de elaborado o estudo de viabilidade, a organização terá condições de decidir com mais assertividade se vale a pena investir no projeto. Além disso, o estudo de viabilidade pode ser importante para ajudar uma empresa a decidir se utilizará seus recursos internos (equipamento, pessoal etc.) para desenvolver o sistema internamente, ou ainda se contratará uma empresa terceirizada, atualmente chamada de fábrica de software, para elaborar o sistema.

Como ocorre em diversas áreas, em especial as que envolvem tecnologia, o estudo de viabilidade pode envolver questões éticas e gerar muita polêmica. Afinal, um sistema pode trazer impactos para os colaboradores de uma empresa das mais diversas áreas. Dessa forma, é possível que uma empresa opte por não implementar um sistema se ela achar que os benefícios serão pequenos em comparação com os problemas que podem ser gerados com sua implantação.

3.1. Viabilidade Organizacional

A maioria dos profissionais de desenvolvimento tem o ímpeto de sair codificando um programa, mesmo sem ter certeza de que entendem exatamente o que deve ser feito. Além disso, mesmo que se entenda exatamente qual é o problema, e qual é a solução por meio do sistema a ser desenvolvido, existe ainda a necessidade de se responder a seguinte questão: será que vale a pena investir em um novo sistema? Essa pergunta precisa ser respondida com precisão porque fazer um sistema simplesmente para informatizar uma tarefa manual nem sempre é traduzido em redução de custos, ou outro benefício qualquer.

A viabilidade organizacional está diretamente ligada a “como” e “quanto” o sistema (solução) trará de benefícios para a organização. Isso deve considerar a

cultura organizacional e os objetivos estratégicos - um mesmo sistema pode funcionar perfeitamente em uma empresa (dentro de uma cultura) e não ter nenhuma aceitação em outra empresa porque a cultura é totalmente diferente.

Dessa forma, um dos pontos da viabilidade organizacional é verificar se a solução terá aderência ao uso pelos usuários, alinhado com a cultura organizacional e a percepção dos envolvidos. Outro ponto de extrema importância se refere ao suporte adequado da direção da organização em relação ao projeto. Caso não exista apoio da direção, provavelmente o projeto não terá êxito.

3.2. Viabilidade Técnica

Como o próprio nome sugere, a viabilidade técnica está diretamente ligada ao suporte técnico que a organização responsável pelo projeto irá fornecer para seu desenvolvimento e manutenção futura. Decidir pelo desenvolvimento de um novo sistema é parecido a decidir ter um filho! É preciso reconhecer que isso trará impactos duradouros para a empresa, uma vez que todo sistema precisa ser cuidado, normalmente durante muitos anos.

A partir disso, a organização pode avaliar seus recursos atuais com o objetivo de verificar se já possui a tecnologia necessária em termos de software, hardware, infraestrutura de redes etc., para que o sistema possa ser desenvolvido e implementado. Em outras palavras, para oferecer esse suporte técnico relacionado às tecnologias necessárias, podem ser considerados diversos equipamentos e serviços, tais como: sistema de refrigeração para instalação do servidor, computadores compatíveis, boa infraestrutura de rede, etc.

Caso não possua essa capacidade de suporte para a solução proposta, ou caso não possua as tecnologias necessárias, a organização pode avaliar os investimentos necessários para a obtenção dos elementos necessários. Um item bastante comum atualmente vinculado a esse suporte técnico se refere à locação de toda a infraestrutura na nuvem (AWS, Google, Azure etc.). A migração de uma infraestrutura local para a nuvem tem se tornado cada vez mais comum nas empresas, visto as diversas vantagens envolvidas e a possibilidade de ampliação dos recursos com extrema facilidade (escalabilidade).

Outro aspecto cada vez mais importante se refere à capacidade técnica da organização em termos de recursos humanos. Mesmo que a organização tenha toda

a tecnologia necessária para suportar o projeto, ou ainda ter todos os recursos financeiros necessários para investimento, é imprescindível que existam pessoas capazes para manusear os recursos tecnológicos. Dessa forma, é essencial que a organização possua uma equipe técnica competente para manipular todos os elementos tecnológicos envolvidos na solução a ser implementada.

3.3. Viabilidade Econômica

Já citamos a questão do investimento nas seções anteriores e agora vamos abordar o assunto com mais detalhes. Dentre as diferentes viabilidades, provavelmente a econômica seja a mais crítica, chamada também de viabilidade financeira ou ainda estimativas de custo. Durante a fase inicial de um novo projeto, a viabilidade econômica se concentra em analisar os custos do projeto e os possíveis benefícios após sua implantação. Em outras palavras, é necessário que a organização realize uma análise de custo-benefício. Trata-se de uma etapa que deve ocorrer antes do planejamento do sistema. Imagine a frustração de investir um ano inteiro no desenvolvimento de um sistema e, ao final, descobrir que ele não é economicamente viável, isto é, o sistema “não se paga” mesmo durante os próximos 5 anos de utilização.

Sendo assim, nessa fase a organização avalia o potencial de retorno do projeto proposto. Isso minimiza riscos, e traz uma visão inicial sobre as possíveis vantagens da produção do sistema, sejam ganhos financeiros ou qualquer outra vantagem. Já comentamos isso anteriormente, nem todo sistema precisa gerar dinheiro, mas espera-se que ele traga algum tipo de vantagem.

Existem diversos custos a serem considerados e analisados na realização de um projeto, mas, podemos de maneira macro, classificá-los em dois tipos: os custos de concepção e desenvolvimento do sistema e os operacionais, após a implantação do sistema.

Os custos envolvidos para o desenvolvimento de sistema são aqueles que ocorrem uma única vez, ou seja, é o custo que envolve todas as fases a serem cumpridas para que o sistema seja criado. Considere todas as etapas do processo de desenvolvimento abordadas no capítulo 2, referente ao ciclo de vida de um projeto de software. Desde a concepção e análise de requisitos, passando pelo projeto, implementação e implantação do sistema, existem diversos custos com a equipe,

talvez alguns treinamentos sejam necessários, custos com aquisição de equipamentos de hardware e softwares.

Os custos operacionais são contínuos e duram enquanto o sistema “estiver no ar”, ou seja, enquanto o sistema estiver sendo utilizado pelos usuários. Em outras palavras, os custos operacionais de um sistema permanecem durante seu ciclo de vida. Muito provavelmente, enquanto o sistema estiver sendo utilizado, alguém deverá tomar conta dele, normalmente um programador ou analista de sistemas. Obviamente, existirá o custo desse profissional e de toda a infraestrutura necessária para que o sistema se mantenha rodando.

De maneira geral, os custos podem ser classificados também em duas categorias: custos **fixos** e **variáveis**. Os custos fixos ocorrem em intervalos de tempo regulares como, por exemplo, tarifas de energia, Internet, pagamentos de aluguel e de licença de softwares, salários dos colaboradores internos e externos, suporte técnico, entre outros custos envolvidos com a operação do sistema. Já os custos variáveis, como o próprio nome sugere, ocorrem esporadicamente por ocasião de aquisição ou manutenção de equipamentos, por exemplo. Num determinado mês é possível ter um custo variável, por exemplo, de R$10.000,00 com a compra do ar condicionado para um servidor, no entanto, nos meses seguintes haverá o custo fixo da energia elétrica consumida por esse equipamento.

Como na grande maioria das vezes, o lado financeiro é o mais importante, existem alguns indicadores específicos a serem considerados. Obviamente não iremos explicar como calcular cada um desses indicadores, mas apenas apresentar alguns conceitos relacionados. Os principais indicadores são:

- **Taxa de Mínima de Atratividade (TMA):** é um indicador utilizado para medir o retorno mínimo que é esperado de acordo com o que foi investido, isto é, pode ser usado para identificar se vale a pena investir em alguma coisa, por exemplo no desenvolvimento de um sistema. No nosso caso, a TMA pode ser calculada por meio da fonte de capital do projeto e da margem de lucro esperada. Quando um sistema recebe apoio de um investimento, esse fator é crucial para decidir pela elaboração do projeto, visto que um investidor não irá aplicar seus recursos em um projeto que tenha poucas e arriscadas perspectivas de retorno.

- **Valor Presente Líquido (VPL):** é um indicador usado no gerenciamento de projetos que permite realizar a análise dos fluxos de caixa esperados com o novo investimento. O objetivo é trazer o fluxo de caixa (entradas e saídas) para o dia presente, de maneira a ser possível identificar a diferença entre o valor atual e o valor aplicado no investimento inicial do sistema.
- **Tempo de Retorno do Investimento (payback):** se refere ao prazo estimado para que o projeto possa dar retorno do investimento realizado. Em outras palavras, o payback é uma métrica cujo cálculo indica um determinado tempo (meses ou anos) necessários para recuperar o investimento realizado no projeto. Esse tempo de retorno pode ser calculado de duas maneiras: o payback simples e o descontado. O payback simples não considera o dinheiro ao longo do tempo, não inclui o custo de capital, levando-se em consideração apenas o valor investido inicialmente e em quanto tempo esse investimento será recuperado. Já o payback descontado considera o valor do dinheiro no tempo e utiliza a TMA para descontar os fluxos de caixa e trazer o retorno na mesma data do investimento inicial.

3.4. Viabilidade Operacional

A viabilidade operacional pode ser direcionada de duas maneiras: a visão do desenvolvedor, quanto a eficiência e performance das operações realizadas pelo sistema, e a visão do usuário, relacionada à facilidade, utilidade e aceitação da utilização do sistema. Essa é uma questão bastante preocupante, uma vez que mesmo quando um sistema é elaborado da forma mais eficiente, usando a melhor tecnologia disponível, existe o risco da não aceitação do usuário e, consequentemente, a solução pode não obter sucesso.

Existem diversas maneiras de realizar um estudo de viabilidade operacional, uma delas se refere à utilização de frameworks para facilitar e direcionar o andamento do estudo. Dentro desse contexto, podemos citar o framework PIECES, muito utilizado para categorizar os problemas e levantar requisitos. No framework PIECES, cada letra define um problema a ser analisado: P - Performance, I - Informação, E - Economia, C - Controle, E - Eficiência e S - Serviços.

O framework PIECES, normalmente conduzido por um analista de sistemas ou um líder de projeto, pode contribuir na resolução de problemas, principalmente durante a fase de levantamento de requisitos, estudar as oportunidades e atingir os

objetivos estabelecidos no estudo de viabilidade. Comparativamente, a análise PIECES tem seu foco nos requisitos não funcionais estudados anteriormente. Lembre-se de que o estudo de viabilidade é realizado antes de se desenvolver um sistema, então, os tópicos seguintes contêm a descrição de análises esperadas que devem fazer parte do sistema a ser desenvolvido.

- **Performance (P):** procura identificar o tempo de resposta apropriado para as funcionalidades mais importantes do sistema a ser desenvolvido. O tempo de resposta a uma consulta pode ser um elemento crucial para o sucesso do sistema. Você usaria o Google se o tempo de resposta de cada pesquisa fosse superior a 30 segundos? Provavelmente não! Um outro exemplo seria considerar a quantidade máxima de processamento que o sistema deverá suportar. Quantos usuários simultâneos serão atendidos pelo sistema, para um site ou aplicativo essa pode ser uma questão bastante crítica.
- **Informação (I):** a informação tornou-se um elemento crucial para qualquer tipo de negócio. É importante analisar se o sistema a ser desenvolvido fornecerá informações corretas, precisas e dentro do tempo esperado. Num mundo cada vez mais integrado e imediatista, é de se esperar que as informações estejam disponíveis 24 horas por dia a todos os envolvidos no processo. Se um sistema precisa apresentar um dashboard, por exemplo, as informações precisam estar atualizadas e corretas para ajudar os gestores na tomada de decisão.
- **Economia (E):** procura identificar se o sistema irá oferecer custo/benefício adequado para a organização. Como já dissemos anteriormente, a produção de um novo sistema deve “valer a pena”. O sistema precisa trazer algum tipo de benefício para a empresa, seja a redução de custos, o aumento das vendas (do lucro), melhor controle, melhor visibilidade, maior alcance organizacional e assim por diante.
- **Controle (C):** analisa as partes relacionadas com a segurança de acesso e, consequentemente, a segurança dos dados a serem manipulados pelo sistema. Nessa etapa podem ser analisados e definidos os perfis de usuário e as partes do sistema que estes terão acesso. Mais do que nunca, gerenciar a segurança das informações com o objetivo de evitar fraudes e violações de

privacidade, são elementos extremamente importantes para qualquer organização.

- **Eficiência (E):** procura analisar como estão sendo utilizados os recursos e atividades que possam desperdiçar tempo, causadas principalmente por redundância. Essas redundâncias podem ser de processo, de funções do sistema, de dados etc. Fazer mais de uma vez “o que já está feito”, ou armazenar o mesmo dado mais de uma vez, são pontos que devem ser localizados e evitados.
- **Serviços (S):** procura identificar quais são os serviços necessários aos usuários e como eles devem ser entregues. Verifica se esses serviços demonstram ser confiáveis aos usuários. Caso seja necessário se comunicar com outros sistemas, é importante identificar as interfaces necessárias e se podem representar algum risco à organização.

De forma geral, os itens listados anteriormente são executados por meio de uma série de questionamentos que devem ser ponderados pela gerência e pelos usuários envolvidos no sistema. O objetivo é verificar qual é o sentimento dos usuários sobre o novo sistema e reduzir possíveis resistências na implantação. Isso pode facilitar a adaptação dos usuários frente às mudanças que o novo sistema trará. Desta forma, o estudo de viabilidade operacional é também um estudo sobre o comportamento das pessoas que utilizarão o sistema, um sistema que seja adequado e que implemente facilidades, possui muito mais chances de conquistar o engajamento dos envolvidos em sua utilização.

3.5. Viabilidade de Cronograma

Por mais organizada que uma equipe possa ser, em especial na área de desenvolvimento de sistemas, o cronograma sempre foi um desafio! Por isso, a viabilidade de cronograma deve ser estudada logo no início do projeto. Todas as funcionalidades e prazos devem ser cuidadosamente analisados pela equipe, na tentativa de verificar se o cronograma é factível, ou seja, se poderá ser cumprido no prazo esperado. Nem sempre a equipe, por mais experiente que seja, tem plena convicção da quantidade de trabalho que deverá ser realizada. Durante o desenvolvimento do sistema podem surgir imprevistos dos mais diversos que podem comprometer o andamento do projeto.

Uma maneira de facilitar a mensuração do tamanho de um projeto é dividi-lo em partes menores, como vimos na metodologia SCRUM, por exemplo, criando diversos sprints. Ao dividir um projeto em partes menores, a tendência é errar menos, pois possíveis atrasos entre os sprints podem ser melhor gerenciados. Por exemplo, se a equipe se atrasou no sprint 1, então talvez seja o caso de “apertar o passo” na execução do próximo sprint.

Uma outra vantagem de se dividir um projeto em pedaços menores é, considerando a experiência dos membros da equipe em projetos anteriores e semelhantes, é possível chegar a conclusão de que será muito difícil cumprir o prazo esperado (ou exigido). Se você tem que entregar um projeto em seis meses e tem certeza de que não terá condições de entregá-lo antes de doze meses, talvez seja melhor “não embarcar nessa viagem”.

3.6. Benefícios tangíveis e intangíveis

Como já citado anteriormente, é muito importante analisar quais serão os benefícios que o sistema oferecerá para a organização. Nesse ponto vamos classificar esses benefícios em dois tipos: **tangíveis** e **intangíveis**.

- Os benefícios **tangíveis** são aqueles que podem ser quantificados em moeda corrente, podendo ser inseridos no patrimônio da organização. Dessa forma, eles podem ser identificados e medidos facilmente. Dentre esses tipos de benefício, podemos citar: aumento no lucro, redução de custos, crescimento das vendas, diminuição de erros de processamento etc..
- Os benefícios **intangíveis** por outro lado são muito difíceis (ou impossíveis) de serem mensurados. Por exemplo, a melhoria da satisfação do cliente pode ser medida por meio de questionários de satisfação, mas é muito mais difícil de ser medida do que um valor monetário. A melhoria da imagem da empresa perante o mercado, a melhoria da relação com os fornecedores são outros exemplos dos benefícios intangíveis. Em função disso, um benefício intangível é mais difícil de ser aceito como verdade, uma vez que é muito mais difícil de ser quantificado.

REFERÊNCIAS

Delamaro et al., **Introdução ao teste de software**, 2a edição, ISBN 9788535226348, Campus, 2016.

Análise de viabilidade de projetos de aplicativos, 20 mar. 2019. Disponível em: https://blog.cronapp.io/analise-de-viabilidade-de-projetos/. Acesso em: 30 jun. 2022.

O que é Estudo de Viabilidade? 2021. Disponível em: https://www.luis.blog.br/estudo-viabilidade-tenica-economica-operacional.html. Acesso em: 1 jul. 2022.

Engenharia de Software: O Estudo de Viabilidade. [*S. l.*], 2022. Disponível em: https://bityli.com/ovlnOJ. Acesso em: 1 jul. 2022.

Capítulo 4 – Especificação de Requisitos

Como você deve ter notado nos capítulos anteriores, a produção de um sistema envolve diversos aspectos materiais, não materiais e humanos. Apenas para citar, podemos encontrar requisitos financeiros, legais, requisitos do negócio, requisitos de dados, de hardware e software, de qualidade, de teste, de usabilidade, de segurança, de ergonomia e portabilidade, enfim, a lista está se tornando cada vez maior com o passar dos anos.

Neste capítulo estamos interessados em fornecer maiores detalhes a respeito dos requisitos mais diretamente envolvidos no ciclo de vida de um software, isto é, mais ligados ao funcionamento de um sistema.

4.1. Requisitos Funcionais

Os requisitos funcionais, como o próprio nome sugere, se referem às funcionalidades esperadas para o sistema. Uma boa maneira de iniciar a definição dos requisitos funcionais é por meio do diagrama de casos de uso (DCU). Esse importante diagrama ajuda a identificar os atores (usuários) do sistema e as funcionalidades que eles precisam executar, ou seja, está fortemente relacionado com as necessidades dos usuários. Para elaborar o diagrama de casos de uso, você pode utilizar a ferramenta ArgoUML, descrita no Apêndice A.

O DCU é um tipo de diagrama de contexto e permite identificar a essência do sistema. A Figura 4.1 contém um exemplo de DCU. Como podemos observar existem dois atores diferentes, representados pelo homem palito (stickman), o ator gestor e o ator operador, um em cada lado do diagrama. Fica fácil identificar também as funcionalidades realizadas por cada um dos atores, pois as elipses representam os casos de uso, ou seja, as funcionalidades disponíveis para cada ator. Observe que existe uma funcionalidade em comum chamada Logar, pois os dois atores podem usá-la.

Uma vez logado no sistema, o ator gestor pode realizar as seguintes tarefas: monitorar acessos ao sistema, gerenciar operadores e gerenciar processos. Para este último caso de uso, observe que existem algumas opções que o ator pode realizar: estando no caso de uso gerenciar processos é possível alterar o status do

processo manualmente, acompanhar as ocorrências do processo e gerar histórico em PDF. No diagrama da UML, quando aparece o esteriótipo <<extend>> ele se refere à casos de uso opcionais, ou seja, podem ou não ser executados pelo usuário. Estando em gerenciar processos, o usuário poderá acessar uma das opções disponíveis, como se fossem opções de um menu.

Apesar de não estar representado na Figura 4.1, é possível inserir o esteriótipo <<include>> que se refere à casos de uso obrigatórios. Suponha que para utilizar o caso de uso Monitorar Acessos ao Sistema fosse necessário fornecer uma senha adicional. Nesse caso, poderia existir um caso de uso chamado Validar Acesso, que seria um <<include>> de Monitorar Acessos ao Sistema, ou seja, no diagrama ficaria “Monitorar Acessos ao Sistema” <<include>> “Validar Acesso”.

De forma semelhante, o ator operador pode acompanhar processos e pode gerenciar as ocorrências de um processo. Estando neste último caso de uso, o usuário poderá também anexar arquivos se assim for necessário.

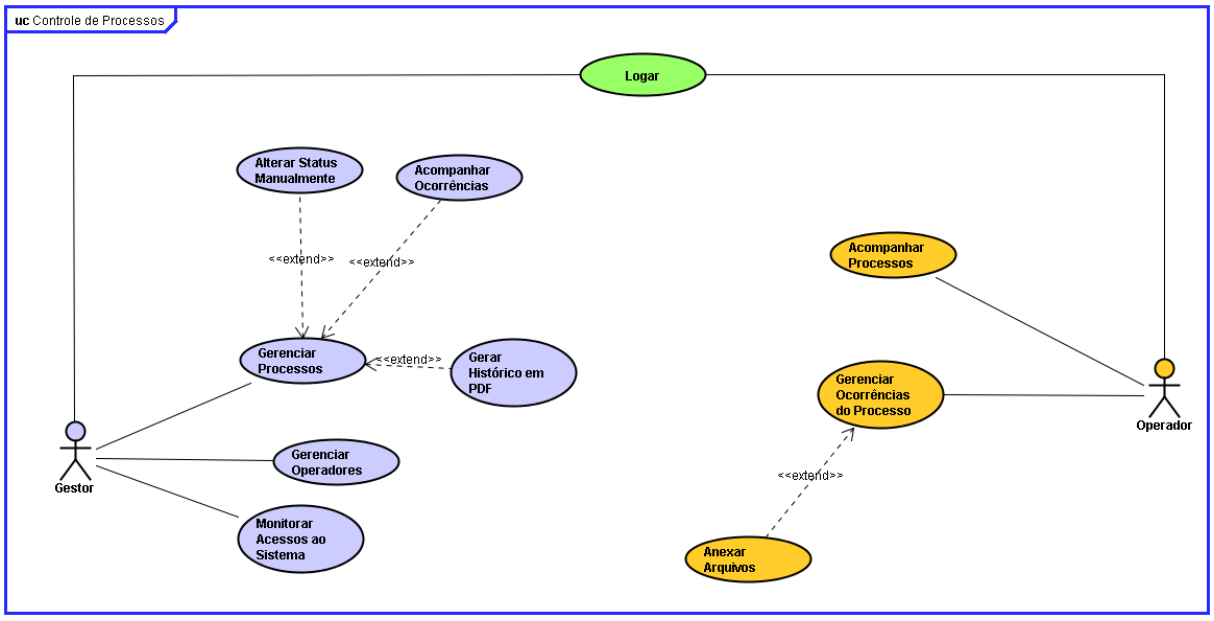

Figura 4.1 – Exemplo de Diagrama de Casos de Uso.

Mesmo uma pessoa leiga, um usuário comum, pode realizar a leitura deste diagrama facilmente. Em função disso, é bastante comum contar com a presença do usuário no momento de elaborar o diagrama de casos de uso, uma maneira de realizar e validar o levantamento de requisitos. Além disso, para cada elipse, ou seja, para cada caso de uso pode existir uma narrativa associada, um texto que descreve as interações que devem ocorrer entre o ator e o sistema. Existem diversos padrões

para a redação das narrativas, cada empresa define seu padrão a partir de suas necessidades dentro do processo de desenvolvimento, narrativas mais ou menos detalhadas. A seguir é apresentado um exemplo de narrativa para o caso de uso Logar, ilustrado na Figura 4.1. Observe que esse modelo contém 5 partes, destacadas em negrito.

Nome: Logar - CSU001.
Objetivo: Permitir aos usuários terem acesso ao sistema
Ator Principal: todos os usuários do sistema

1. Cenário Principal

1. O caso de uso Logar inicia quando o usuário acessa o menu Login.
2. O usuário digita o login.
3. O usuário digita a senha.
4. O usuário pressiona o botão Confirmar.
5. O sistema realiza a validação dos dados do usuário.
6. Se os dados estiverem corretos carrega a página HOME.

2. Cenários alternativos

a) O usuário precisa se registrar:

1. O usuário acessa o link "registrar" de acordo com CSU002.
2. Ao final do registro o sistema retorna ao passo 2 do cenário principal.

b) O usuário esqueceu a senha:

1. O usuário acessa o link "esqueci a senha".
2. O sistema apresenta uma janela para o usuário digitar seu email de recuperação.
3. O usuário digita seu email e clica no botão "recuperar Senha".
4. O sistema valida o email e, caso esteja cadastrado, gera uma senha provisória e envia ao usuário em seu email.

5 Caso o email fornecido pelo usuário não exista, notificar o usuário com uma mensagem em tela.

6. Ao final da recuperação da senha, o sistema retorna ao passo 2 do cenário principal.

3. Cenários de Exceção

a) Falha na autenticação:

1. Caso o email ou senha fornecidos sejam inválidos, o sistema envia uma notificação em tela alertando o usuário de que ele tem n-1 tentativas de autenticação.

2. O sistema retorna ao passo 2 do cenário principal, desde que as tentativas não estejam esgotadas.

3. Se o número de tentativas tiver sido excedido, o sistema bloqueará o usuário.

4. Pré-condições

1. O usuário deve estar registrado no sistema, ou deve se cadastrar.

5. Pós-condições

1. O sistema registra a entrada do usuário no arquivo de log.

Esse é apenas um modelo possível para ser utilizado na construção de narrativas de casos de outro. A modelo inicia definindo o nome do caso de uso e sua sigla correspondente (as siglas são usadas para identificar um caso de uso de maneira única). Normalmente, existe um ator principal para cada caso de uso, no entanto, como o caso Logar é genérico e atende à todos os usuários do sistema.

Continuando a descrição de nosso modelo, a parte 1, chamada de cenário principal, contém a sequência natural de utilização da tela de login, ou seja, o passo a passo para que o usuário consiga acessar no sistema, nesse caso numerado sequencialmente do 1 ao 6. O cenário principal sempre deve conter a sequência mais comum de utilização por parte dos usuários. A parte 2, chamada de cenário alternativo, contém ações que podem ser escolhidas pelo usuário quando o cenário principal não atender às suas necessidades. Observe que existe um cenário alternativo para o usuário se registrar no sistema e um cenário alternativo no caso de esquecimento da senha. Os cenários de exceção, descritos na parte 3, contém os passos realizados pelo sistema em caso de falhas. Normalmente, os cenários de exceção estão associados a erros de validação, autenticação, entre outras checagens realizadas pelo sistema. A parte 4 contém as pré-condições, isto é, as condições necessárias que devem existir para que o usuário possa utilizar o caso de uso. Já a parte 5, referente às pós-condições, como o nome sugere, são executadas após o encerramento do caso de uso. Neste exemplo, após o usuário realizar o login será registrado sua entrada em um arquivo de log.

Como dissemos anteriormente, a elaboração do diagrama de casos de uso, juntamente com suas narrativas, é um bom começo, mas não é tudo. O diagrama de caso de uso mapeia somente as funcionalidades relacionadas ao usuário, ou seja, as funções que o usuário realiza por meio do sistema. No entanto, existem outras funcionalidades que podem ser necessárias em um sistema e que não são capturadas pelo diagrama de casos de uso. Além disso, você já teve contato com as diversas técnicas que podem ser usadas para realizar o levantamento de requisitos.

Pensando mais tecnicamente, é possível identificar diversas funcionalidades pertinentes ao sistema em si, e que não necessariamente tem relação com o usuário. Observe as seguintes necessidades: o armazenamento dos dados no banco de dados, questões de validação de usuários, criação de tokens e outras questões de segurança, a internacionalização do sistema, a usabilidade das telas, rotinas automáticas de backup, questões relacionadas a responsividade, entre tantas outras e que podem ser necessárias na criação de qualquer sistema.

Depois de levantar todos os requisitos, pode ser feita uma lista, da mesma forma que fizemos com o Product Backlog no capítulo 2. No entanto, até o momento listamos apenas os requisitos funcionais, vamos analisar os outros tipos de requisitos nas seções seguintes.

4.2. Requisitos de Qualidade

Os requisitos de qualidade, também chamados de requisitos não funcionais, se referem às questões complementares às funcionalidades. Por exemplo, você utilizaria um mecanismo de busca que levasse, em média, cinco minutos para dar uma resposta a cada pesquisa realizada? Provavelmente não, por mais eficiente que fosse o mecanismo de busca, o tempo de resposta seria muito grande. Observe que um requisito de desempenho (ou velocidade) pode estar diretamente relacionado a um requisito funcional. Portanto, os requisitos de qualidade estão relacionados aos funcionais, agregando características importantes e, às vezes, essenciais das funcionalidades.

Existe uma lista considerável de requisitos não funcionais, vamos descrever alguns deles:

- **Desempenho**: como já dissemos, o desempenho tem relação direta com a velocidade com que o sistema realiza suas tarefas. Pesquisas lentas, relatórios que demoram para ser carregados são exemplos de problemas com desempenho. Normalmente o desempenho é comprometido por problemas de hardware (processador e memória) ou problemas na lógica utilizada nas rotinas de um sistema.
- **Usabilidade**: este requisito envolve diversos aspectos e pode ser subdividido em diversos tipos (navegabilidade, acessibilidade, objetividade entre outros). De forma geral, a preocupação da usabilidade tem relação com a facilidade de uso do sistema. A sequência de passos para realizar as atividades deve estar muito clara no sistema. Sistemas que precisam de muitos manuais para serem utilizados são um indício de falta de usabilidade. A interface gráfica do usuário deve ser “gostosa” de ser usada, atraente, padronizada, contendo elementos de fácil identificação. O uso correto de ícones e atalhos pode ajudar o usuário a encontrar as funções com mais facilidade, aumentando sua produtividade. A usabilidade tem relação também com o perfil dos usuários, pois a experiência destes pode contribuir para um melhor desempenho no uso da interface gráfica.
- **Segurança**: outra característica extremamente importante para os dias atuais se refere à segurança de um sistema. Você utilizaria um sistema com uma interface bem amigável, que fosse rápido, mas invadido facilmente? Provavelmente não! Um sistema seguro é essencial para o sucesso de qualquer projeto. Dentre os principais elementos atuais usados para aumentar a segurança de um sistema podemos citar o uso da criptografia, de mecanismos de validação e recuperação de senhas, a prevenção de SQL Injection, entre outros. A definição de uma política de segurança que envolva o treinamento de usuários é extremamente importante para qualquer empresa. Usuários treinados têm menos chances de cair em golpes aplicados por meio de engenharia social, prática cada vez mais comum em todos os ambientes.
- **Portabilidade**: este requisito possui relação com o local onde o sistema será usado. Cada vez mais os usuários utilizam smartphones no lugar de computadores pessoais. Elaborar um sistema que rode apenas numa plataforma específica é correr um grande risco. Precisamos de sistemas

responsivos que funcionem em qualquer tipo de equipamento, não importa seu tamanho ou sistema operacional utilizado - normalmente eram feitos sistemas para o sistema operacional Windows, mas, a chegada de novos sistemas operacionais tem trazido muitos desafios para os desenvolvedores de sistemas. Fazer um sistema que rode em qualquer equipamento, em qualquer sistema operacional e ainda, em qualquer tipo de navegador para a Internet é um grande desafio.

- **Escalabilidade**: outra característica importante para alguns tipos de sistema, em especial os que rodam no lado do servidor (Back-end), se refere ao aumento de sua capacidade de atendimento. Por exemplo, um sistema que atende mil usuários ao mesmo tempo, continuará operando normalmente se o número de usuários aumentar para dez mil? Essa pode ser uma realidade para sistemas que operam no ambiente da Internet. A escalabilidade é facilmente alcançada em sistemas que rodam na nuvem, visto a facilidade que esses serviços disponibilizam para aumentar seu poder de processamento, memória, espaço em disco etc.
- **Internacionalização**: outra característica extremamente importante, principalmente para sistemas e aplicativos que têm o objetivo de ter alcance mundial. Por mais eficiente, bonito, seguro etc. que for um sistema, provavelmente ele não será utilizado por um chinês se não oferecer suporte a seu idioma. É o preço de termos um mundo cada vez mais globalizado. Se queremos alcance mundial devemos criar sistemas com diversos idiomas, pelo menos os principais dependendo dos objetivos do sistema.

Como afirmamos anteriormente, existem diversos outros tipos de requisitos de qualidade, pesquise mais sobre eles, pois é extremamente importante levar em consideração essas características do sistema - sua falta pode comprometer todo o trabalho desenvolvido.

4.3. Requisitos de Negócio

Os requisitos de negócio, também conhecidos por regras do negócio, se referem às funcionalidades que dependem das características particulares do ambiente onde o sistema será executado. Em outras palavras, um sistema implantado em uma loja de materiais elétricos, pode necessitar de ajustes quando usado em outra loja de materiais elétricos. Isso ocorre porque cada empresa, ou

cada negócio, possui características próprias que os diferenciam, por isso são chamadas de regras de negócio.

Dessa forma, quando se pretende que um sistema seja utilizado em diferentes empresas e instituições, deve-se pensar nas particularidades que podem existir entre eles. Por exemplo, suponha que a loja de materiais elétricos queira dar um desconto para aposentados e pensionistas. A primeira coisa a pensar é: como o sistema irá identificar se o cliente a ser atendido é aposentado? Provavelmente, esse atributo deveria estar presente no cadastro de clientes, para que o sistema possa identificar quando um cliente é aposentado. Em segundo lugar, qual seria o desconto? Onde estaria registrado esse valor (ou percentual)? Provavelmente num arquivo de configurações, pois esse valor poderia ser alterado facilmente a qualquer momento, caso a política da empresa mude. E por último, o sistema deve calcular o valor do desconto quando um cliente aposentado realizar uma compra. Ou seja, essas regras do negócio têm um impacto direto sobre o funcionamento de um sistema, uma vez que ele deve acompanhar as necessidades do negócio. E como dissemos, cada negócio poderá exigir regras e valores diferentes.

Vamos reforçar a ideia das regras de negócio considerando diferentes instituições de ensino. Como um sistema pode identificar os alunos que foram aprovados, ou reprovados? Normalmente existem dois critérios para isso: a média final na disciplina e o percentual de faltas. Focando apenas na média final, como identificar os alunos aprovados? Obviamente conhecendo a nota mínima para aprovação. É aí que entra nossa regra de negócio, pois para uma faculdade X, a nota mínima para aprovação pode ser 5,0 enquanto que para outra 6,0. Esse valor precisa estar configurado em algum lugar para que seja possível utilizar o sistema em diferentes instituições. Apesar de não ser uma regra, normalmente as regras de negócio são definidas por meio de configurações do sistema, seja uma tabela, um arquivo ou qualquer outro meio que permita oferecer uma flexibilidade ao sistema.

Portanto, da mesma forma que os requisitos funcionais e de qualidade, os requisitos de negócio devem ser levantados o quanto antes dentro do ciclo de vida do sistema. Sua falta pode gerar muito retrabalho, senão a impossibilidade de utilização do sistema. Sistemas que consideram o uso correto das regras de negócio são mais flexíveis e se adaptam mais facilmente em diferentes cenários e culturas.

4.4. Requisitos de Hardware e Software

Os requisitos de Hardware e Software são encontrados em praticamente todas as especificações de sistema e definem as configurações que os equipamentos e plataformas onde o software será executado. Se o hardware for inadequado, com baixa memória por exemplo, pode ser que o sistema não funcione ou fique muito lento. Por outro lado, se o sistema operacional for incompatível, o sistema não poderá ser instalado.

Em função dessa necessidade, ao produzir (e principalmente ao comercializar) um sistema, é necessário definir as configurações mínimas necessárias para que o software funcione adequadamente. Um exemplo de requisitos de hardware pode ser visualizado no Quadro 4.1.

- Processador Intel® Xeon® E3-1220v2 3.10 GHz, 8M Cache, Turbo, Quad Core/4T (69W);
- 8GB, UDIMM, 1600Mhz, DR, Low Volt, BCC;
- 1TB 7.2K RPM SATA 3.5" Cabled Hard Drive;
- Configuração RAID com Controladora On-Board, 1 a 4 HDs
- Mídia para backup externo (pendrive, hd externo)

Quadro 4.1 – Exemplo de Requisitos de Hardware

Como podemos visualizar no Quadro 4.1, a descrição de requisitos de hardware deve possuir todos os elementos físicos que sejam essenciais ao bom funcionamento do sistema, tais como processador, memória, espaço em disco, placa gráfica, dispositivos USB, impressoras, entre outros. Qualquer sistema deve fornecer a definição desses pré-requisitos; caso um usuário não siga as recomendações e instale o sistema em um hardware inferior, pode ser que o sistema não rode adequadamente.

Já os requisitos de software definem quais são os softwares exigidos para a máquina onde o sistema desenvolvido será instalado. Suponha que você desenvolveu um sistema do tipo Front-End (na linguagem React JS) e Back-End (na linguagem Node). Provavelmente para a aplicação Front-End o único requisito de software seja um software navegador para Internet, sem preocupação com a plataforma onde o navegador está instalado (Windows, Linux, Android, iOS etc.). Já para a aplicação Back-End, provavelmente precisará especificar a versão mínima do Node.js, do Sistema Gerenciador de Banco de Dados, certificado digital, protocolo SSL para segurança, entre outros requisitos.

Os pré-requisitos de software normalmente são simples de se definir, uma vez que o desenvolvedor conhece quais os requisitos mínimos para seu sistema rodar. Por outro lado, as especificações dos pré-requisitos de hardware são mais difíceis de elaborar. Uma maneira mais rápida de se definir essas especificações é rodar o sistema em uma máquina qualquer e verificar seu desempenho. Caso o desempenho seja satisfatório, é possível identificar as configurações da máquina e fazer um “copy paste”. No sistema operacional Windows, por exemplo, para verificar as configurações da máquina basta pesquisar por “informações do sistema”.

REFERÊNCIAS

FURGERI, S. **Modelagem de Sistemas Orientados a Objetos** – Ensino Didático. 1. Ed. São Paulo: Editora Érica, ISBN 978-8536504612, 2013.

R.S. Pressman, B.R. Maxim, B.R., **Engenharia de Software: Uma Abordagem Profissional**, 8ª edição, Ed. McGraw-Hill, ISBN 9788563308337, 2016.

W.P. Paula Filho, **Engenharia de Software: Fundamentos, Métodos e Padrões**, 3a edição, LTC, ISBN 9788521616504, 2009.

C. Larman, **Utilizando UML e Padrões**, 3a edição, ISBN 8560031529, 2007.

Capítulo 5 – Orientação a Objetos

A orientação a objetos pode ser definida como um estilo de se programar, ou uma maneira de pensar na hora de codificar um programa. Em termos mais acadêmicos, podemos dizer que a orientação a objetos representa um novo paradigma de desenvolvimento de software. Existem diversos paradigmas de desenvolvimento no mercado, dentre os quais podemos citar programação estruturada, programação declarativa, programação orientada a eventos, programação orientada a aspectos, entre outros. Cada um desses paradigmas nos remete a pensar diferente na hora de produzir o software.

A orientação a objetos surgiu na década de 1960, por meio de uma linguagem chamada Simula, que mais tarde foi nomeada de Smalltalk. Apesar de existir há bastante tempo, há mais de 50 anos, a orientação a objetos começou a ser largamente utilizada a partir do surgimento da linguagem de programação Java, em meados da década de 1990. De lá para cá, muitas empresas começaram a migrar seus sistemas para esse novo paradigma, até então o paradigma dominante era a programação estruturada.

A orientação a objetos foi concebida inicialmente para trabalhar de forma semelhante ao organismo humano, existem literaturas que comparam o funcionamento de uma célula humana com um objeto, o principal elemento da orientação a objetos. De forma bastante resumida, a orientação a objetos permite criar um sistema com diversos tipos de objetos que se comunicam entre si, isto é, trocam mensagens entre si. Se pudéssemos visualizar o sistema orientado a objetos por meio de um microscópio, veríamos diversos objetos interligados, como se fosse uma rede de comunicação.

Este capítulo aborda os principais conceitos relacionados à orientação a objetos, tais como classes e objetos, encapsulamento, herança, polimorfismo, entre outros conceitos importantes para que o estudante possa iniciar seus estudos nesse importante paradigma de desenvolvimento. Existem diversas linguagens com suporte a orientação a objetos e por isso a sintaxe utilizada nas instruções pode variar. Em função disso, esse material não adota nenhuma linguagem para implementar os conceitos da orientação a objetos. Consideramos ideal que você

utilize os conhecimentos adquiridos em linguagens de programação para implementar os conceitos aqui apresentados.

5.1. Classes e Objetos

Você já aprendeu que na orientação a objetos o sistema é composto por diversas partes pequenas chamadas objetos. Como vimos, cada objeto possui uma função bem específica dentro do software. Mas, talvez você esteja se perguntando, de onde vem os objetos? Afinal de contas, como é possível definir um objeto? Como é possível realizar a sua criação? Vamos responder essas questões.

Para criar um objeto existe a necessidade da definição de uma classe, por meio desta é que o objeto será gerado. Em outras palavras, sem a existência da classe não há possibilidade de criar objetos. A classe funciona como um modelo que contém uma estrutura que estará presente em todos os objetos criados a partir desta. Portanto, essa é a relação existente entre classes e objetos, mas vamos conhecer melhor a respeito.

Assim como uma variável, uma classe possui um nome que permite diferenciá-la dentre todas as outras, ou seja, cada classe deve ter um nome único. Se considerarmos a linguagem Java, por exemplo, uma classe deve ser definida dentro de um pacote (uma pasta). É importante observar que podem existir classes com o mesmo nome em pacotes diferentes. Apenas recordando, até o momento temos que um objeto precisa de uma classe que está definida dentro de um pacote. Existem regras para nomear pacotes, classes e objetos, mas, como existem padrões diferentes entre as linguagens, não entraremos nesta questão.

Vamos iniciar pela definição de uma classe. Uma classe contém uma estrutura formada por duas partes, os atributos e os métodos. A linguagem UML representa uma classe de acordo com a figura 5.1.

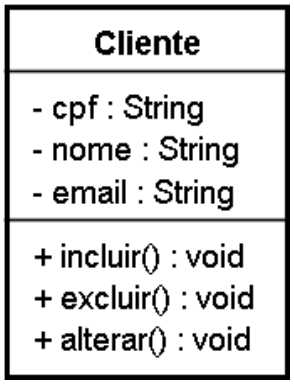

Figura 5.1 – Representação da classe Cliente

Como pode ser observado, a representação da classe possui três divisões, na parte superior temos o nome da classe, na parte central temos a definição dos atributos e na parte inferior a definição dos métodos. Essa é a representação básica para qualquer classe, mas podem existir classes que possuem apenas atributos, ou que possuem apenas métodos.

Ainda em relação à representação, podemos observar que a classe Cliente possui três atributos (cpf, nome e email) e três métodos (incluir, excluir e alterar). Observe também, que ao lado esquerdo de cada atributo existe o sinal de menos, a representação usada pela UML da visibilidade privada (veremos mais detalhes a respeito disso na seção 5.2 que trata sobre o encapsulamento). da mesma forma, ao lado de cada método existe o sinal de mais, usado pela UML para representar que o método é do tipo público.

Não é nosso objetivo entrarmos em questões mais técnicas a respeito da definição de classes e objetos, mas por meio da representação é possível verificar que uma classe possui uma estrutura e esta será idêntica quando um objeto for criado, ou seja, o objeto receberá a mesma estrutura da classe a partir da qual ele foi gerado. Dessa forma, a partir da criação de uma classe é possível gerar múltiplos objetos. A criação de uma classe é semelhante à criação de um novo tipo de dados. Assim como existe o tipo de dados inteiro, por exemplo, com a criação de nossa classe, passamos a ter o tipo cliente. Portanto, a criação de classes gera novos tipos de dados, a partir dos quais podem ser gerados os objetos.

Dentro de um sistema orientado a objetos, cada classe deve possuir um papel bem específico. Fazendo uma rápida análise de nossa classe cliente, considere que todas as funções relacionadas à clientes devem ser tratadas por essa classe. Na

prática, dentro da arquitetura de um sistema, podemos ter mais de uma classe operando em conjunto para desempenhar funções relacionadas aos clientes, mas não entraremos neste nível de detalhe.

Em relação aos objetos, é importante destacar que cada objeto pode manter valores exclusivos em seus atributos. Por exemplo, supondo que sejam criados dois objetos do tipo Cliente, objCliente1, objCliente2, então, seria possível termos dois objetos contendo o seguinte conteúdo:

- **objCliente1**: cpf=123, nome=”Sandro”, email=”sandro@gmail.com”
- **objCliente2**: cpf=456, nome=”Helena”, email=”helena@gmail.com”

Assim como uma variável, cada objeto pode ter qualquer conteúdo. A partir da classe Cliente é possível gerar milhares de objetos em que cada um mantém seu conteúdo de forma independente.

Nesse ponto vamos tratar do diagrama de classes, um diagrama da UML que permite representar classes e relacionamentos. Após realizar as fases de levantamento de dados, especificação de requisitos e demais tarefas relacionadas com as fases iniciais do ciclo de vida do sistema, um analista pode realizar a modelagem de classes. Imagine que o sistema a ser modelado se refere à locação de veículos. Se o paradigma escolhido para o desenvolvimento do sistema for o orientado a objetos, então o analista deverá pensar na estrutura do sistema em termos de classes, isto é, a partir do levantamento de dados será necessário definir as classes principais (classes chamadas de entidades), de maneira semelhante ao produzido pelo modelo de entidades e relacionamentos em termos de banco de dados.

Para exemplificar o caso da locação de veículos, considere o diagrama de classes representado na Figura 5.2.

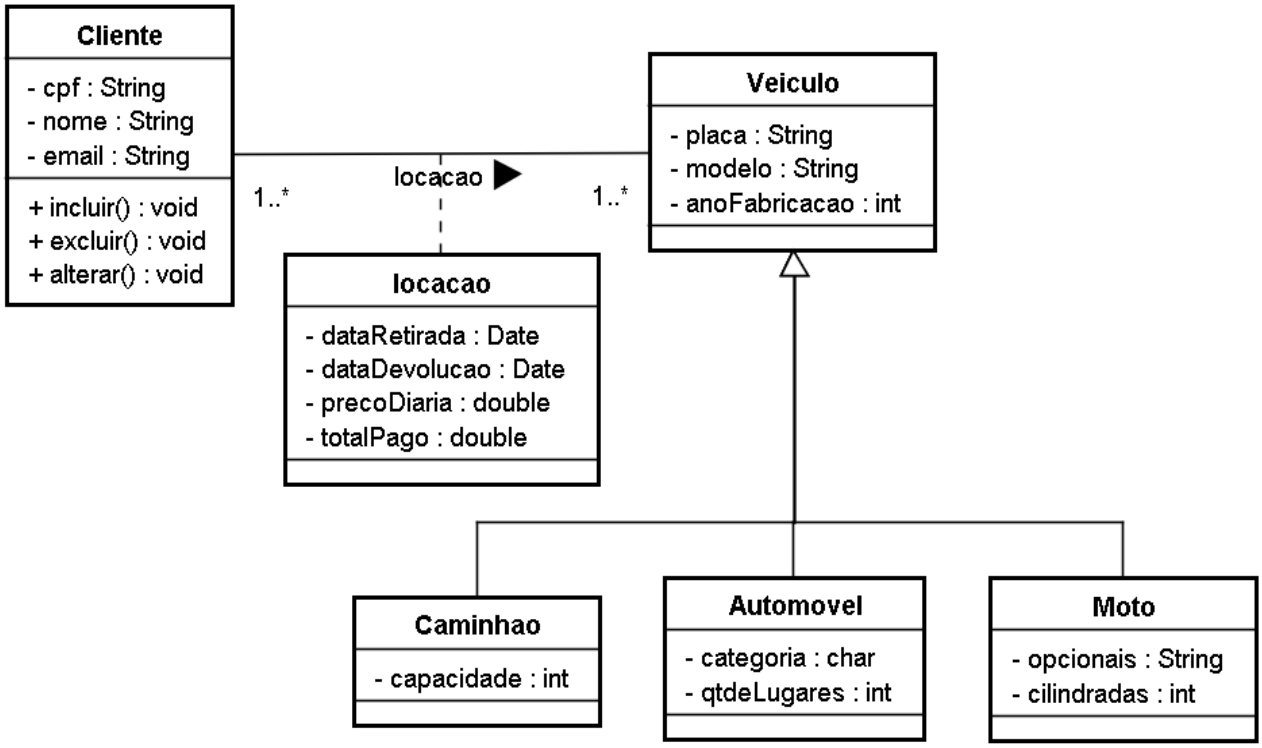

Figura 5.2 – Diagrama de Classes

O diagrama da Figura 5.2 poderia ser o resultado da modelagem das classes principais do sistema de locação de veículos. Observe a presença de nossa classe Cliente contendo atributos e métodos. As outras classes contém apenas atributos, pois nosso objetivo é apenas demonstrar um exemplo de como o sistema poderia ser modelado, pelo menos inicialmente, a partir de classes. Pelo diagrama, pode ser interpretado que um cliente realiza uma locação de veículo que pode ser do tipo caminhão, automóvel ou moto. Existem muitos detalhes relacionados ao diagrama de classes que não serão tratados aqui. Mais a frente na seção 5.4, daremos maiores detalhes a respeito do mecanismo de herança, um conceito existente na orientação a objetos, e que está representado em nosso diagrama entre as classes Caminhao, Automovel e Moto relacionadas com a classe Veiculo.

5.1.1. Relacionamentos

Como um sistema é formado por inúmeros objetos que são gerados a partir de classes, então um sistema orientado a objetos é formado por uma série de classes. Como cada classe possui uma função específica e, normalmente, os processos envolvem diversas funcionalidades, as classes precisam estar relacionadas para trabalhar em conjunto.

O relacionamento pode ocorrer entre diferentes elementos da orientação a objetos, tais como classes, objetos, pacotes, interfaces, componentes etc., entretanto, iremos focar nos relacionamentos existentes entre classes e objetos, juntamente com suas representações na UML. Os principais são:

1. Relacionamento de Dependência

Como o nome sugere, o relacionamento de dependência ocorre quando um elemento depende de outro. Esse é o tipo de relacionamento mais genérico que existe, pois a dependência pode ocorrer de inúmeras maneiras. A dependência implica que o elemento dependente precisa de um outro para realizar alguma atividade. Por exemplo, um objeto que realiza uma inclusão no banco de dados, provavelmente, será dependente de outro objeto (ou classe) do tipo conexão, ou seja, o primeiro não consegue incluir nenhum registro no banco de dados sem a existência de um objeto do tipo conexão.

A Figura 5.3 ilustra os símbolos definidos pela UML para a representação de uma dependência. Cada um destes símbolos é utilizado para uma dependência diferente. Vamos apresentar alguns exemplos para cada tipo de dependência.

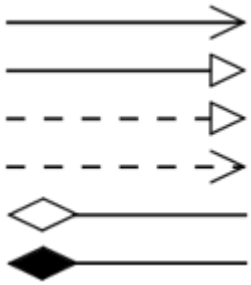

Figura 5.3 – Tipos de representação da dependência na UML

Observe na Figura 5.4, alguns tipos de dependência utilizando a linha tracejada com uma seta aberta, o tipo mais genérico de dependência representado pela UML. A parte superior (a) ilustra um relacionamento de dependência entre pacotes, a parte central (b) ilustra um relacionamento de dependência entre classes. Já a parte inferior (c) também ilustra uma dependência entre duas classes, porém, deixa mais explícito que se trata de uma dependência do tipo parâmetro por meio da inclusão de um estereótipo.

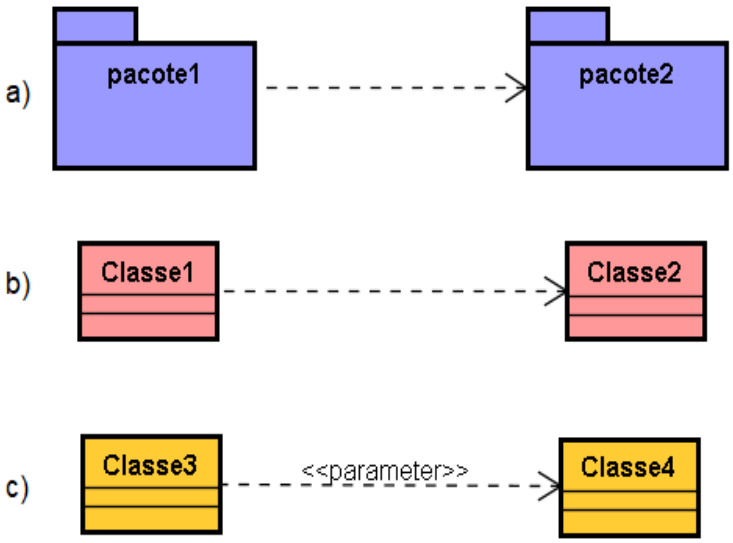

Figura 5.4 – Dependência entre pacotes e classes

2. Relacionamento de Associação

O relacionamento de associação, chamado também de comunicação, permite que dois elementos mantenham algum tipo de vínculo. Considere o diagrama da Figura 5.5 envolvendo uma associação entre as classes Torneio e Partida.

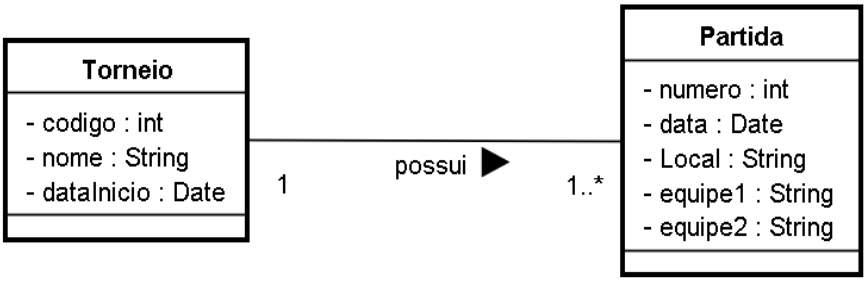

Figura 5.5 – Associação entre classes

A linha sólida representa a associação existente entre as classes Torneio e Partida. A associação pode conter um nome (no caso possui) e a definição da multiplicidade para cada lado (os valores ao lado de cada classe). A leitura da associação pode ser realizada nos dois sentidos do relacionamento. Iniciando pela classe Torneio, temos: um mesmo torneio pode possuir de uma a muitas partidas. Do lado oposto, ou seja, realizando a leitura a partir da classe Partida, temos: uma mesma partida pertence a um único torneio. A definição da multiplicidade é bastante semelhante à definição da cardinalidade, usada em modelos de bancos de dados e contém valores mínimos e máximos que mapeiam a realidade apresentada pelo

diagrama. Isso quer dizer que, dependendo da realidade modelada, os valores da multiplicidade podem ser diferentes.

O relacionamento de associação faz com que as classes se completem na construção de algo maior. Voltando ao nosso exemplo, um torneio sem partidas ficará incompleto, o mesmo vale para a partida, que sem um torneio será apenas um evento isolado. Na prática, pensando em codificação, poderá existir um objeto do tipo Partida dentro de um objeto tipo Torneio, ou vice-versa, dependendo das necessidades da aplicação.

Uma associação pode ocorrer também entre uma única classe, relacionamento chamado de **reflexivo** (associação reflexiva). Nesse caso, a representação da associação parte da classe Pessoa e chega também na própria classe Pessoa, um tipo de associação em que os objetos da classe terão uma ligação entre si. Por exemplo, considere uma classe chamada Pessoa que defina uma associação chamada “casa” (uma pessoa se casa com outra pessoa).

3.Relacionamento de Agregação

O relacionamento de agregação representa uma realidade em que uma classe agrega outras classes (na prática um objeto agrega outros objetos) para formação de um elemento maior. Existe uma relação entre o elemento TODO e os elementos PARTES. O relacionamento de agregação é representado por um diamante não preenchido como ilustra a Figura 5.6.

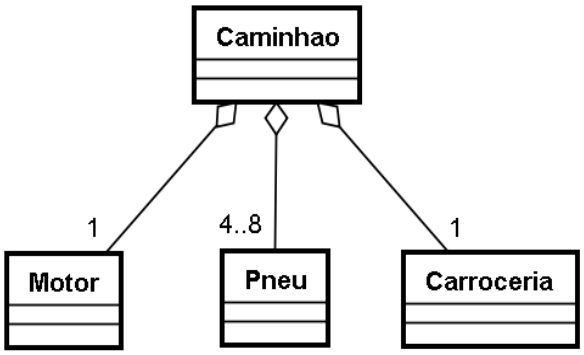

Figura 5.6 – Relacionamento de Agregação

Nesse caso a classe Caminhao está representando o TODO e as classes Motor, Pneu e Carroceria estão representando as PARTES, ou seja, um caminhão é formado por diversas partes. Observe que foi inserida a multiplicidade ao lado de

cada parte, indicando que um mesmo caminhão é formado por um motor, uma carroceria e de quatro a oito pneus.

4. Relacionamento de Composição

O relacionamento de composição também representa uma realidade em que uma classe é formada por outras classes, semelhante à agregação. Da mesma forma, existe uma relação entre o elemento TODO e os elementos PARTES, no entanto a relação é mais forte, por isso a composição é representada por um diamante preenchido, como ilustra a Figura 5.7.

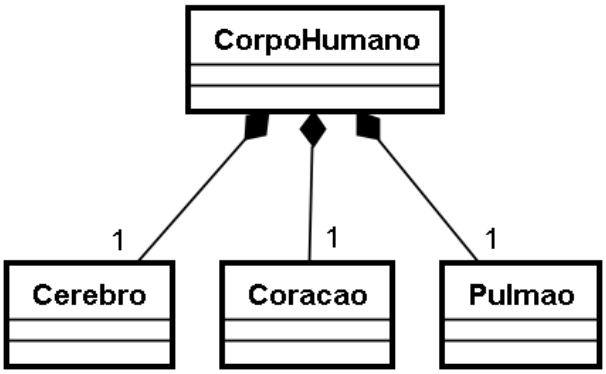

Figura 5.7 – Relacionamento de Composição

Nesse caso a classe CorpoHumano está representando o TODO e as classes Cerebro, Coracao e Pulmao estão representando as PARTES, ou seja, um corpo humano é formado por diversos órgãos. A composição implica em exclusividade, pois, um mesmo cérebro não pode fazer parte de mais de um corpo. O mesmo para os demais órgãos. Essa é a diferença em relação a agregação, na composição as PARTES pertencem exclusivamente ao TODO, se este deixar de existir, todas as PARTES também deixarão.

5.2. Encapsulamento

Ao realizar a leitura da palavra encapsulamento nos vem à mente colocar alguma coisa dentro de uma cápsula. Primariamente, o objetivo da cápsula é proteger, como um astronauta dentro de uma cápsula espacial, ou o conteúdo de um medicamento, por exemplo.

Na orientação a objetos, além do objetivo de proteger, o encapsulamento tem o objetivo também de ocultar, um termo em inglês conhecido como data hiding. Você sabe que um objeto pode conter diversas variáveis de instância (atributos), mas nem

sempre seu conteúdo deve ser acessado por outros objetos. Para ocultar o conteúdo desses atributos de outros objetos é usado o encapsulamento, ou seja, os atributos ficam encapsulados, protegidos de acesso externo. Na prática, o encapsulamento de um atributo permite que a própria classe controle seu conteúdo, impedindo que valores absurdos possam ser inseridos. Suponha, por exemplo, o atributo idade presente numa classe chamada Pessoa. A classe pode definir que a faixa de valores possíveis para esse atributo deve ser entre 0 e 120, valores diferentes são rejeitados, impedidos de serem armazenados.

O encapsulamento pode ocorrer também com os métodos de uma classe, podemos ocultar esses métodos permitindo acesso somente por meio da própria classe, ou seja, os detalhes presentes na implementação da classe ficam restritos e escondidos. Um elemento externo poderá apenas utilizar os serviços da classe, sem saber de fato como isso acontece.

Além de permitir controlar o acesso a atributos e métodos, existem outras vantagens ao usar o encapsulamento. A tendência é que o código se torne mais limpo e legível, os erros de programação diminuam, e a ampliação do código por novas atualizações ocorra mais facilmente, visto que cada classe tem controle sobre seus dados e funções bem definidas.

Um item relacionado ao encapsulamento se refere ao nível de acesso associado aos elementos de uma classe. Eles são definidos por meio de qualificadores de acesso. Esses qualificadores definem a visibilidade de um elemento (atributo ou método), ou seja, define se o elemento será visível externamente (por outros objetos ou classes). Apesar de existirem variações em função das diferentes linguagens de programação, os qualificadores mais comuns e relacionados ao encapsulamento, são os seguintes:

- **public:** é o nível sem restrições, ou seja, um atributo ou método definido como public não pode ser encapsulado, uma vez que terá visibilidade total a partir de qualquer elemento externo. Dessa forma, o uso do qualificador public é equivalente a não encapsular. Na UML, o caractere + (mais) é utilizado para representar um identificador público.
- **private:** é o nível mais restrito, apenas a própria classe pode ter acesso a atributos e métodos e representa o tipo mais utilizado no encapsulamento. Na

UML, o caractere - (menos) é utilizado para representar um identificador privado.

- **protected:** é o nível intermediário de encapsulamento em que atributos e métodos podem ser acessados pela própria classe ou por subclasses (um tipo de classe que herda características de outra classe). Na UML, o caractere # (cerquilha) é utilizado para representar um identificador protegido.

Quando os atributos ou métodos existentes em uma classe são declarados com visibilidade pública, eles são visíveis (isto é, podem ser acessados) a partir de qualquer outra classe ou objeto. No entanto, quando os atributos ou métodos são declarados como privados, apenas a própria classe em que esses elementos foram definidos é que pode realizar o acesso. Quando os atributos são definidos com visibilidade privada e seu conteúdo precisa ter acesso externo, então podem ser criados métodos públicos que se responsabilizam por realizar seu acesso. Esses métodos públicos são nomeados com os termos **get** (pegar) e **set** (definir) de forma padronizada pela maioria das linguagens orientadas a objeto.

Um objeto pode ser representado como se fosse uma célula humana, contendo uma camada interna, um núcleo onde se localizam os elementos privados, e uma camada mais externa onde se localizam os elementos públicos. Veja a representação de um possível objeto do tipo Cliente na Figura 5.8, criado a partir de nossa classe Cliente ilustrada na Figura 5.1. Observe os atributos cpf, nome e email existentes no interior do objeto. O fato de estarem no interior do núcleo demonstra que estão protegidos, ocultos do meio externo (essa é a área reservada aos elementos privados). Por outro lado, na camada mais externa, são inseridos os elementos públicos, no caso os métodos incluir, alterar e excluir. Nessa área pública poderiam ser definidos os métodos públicos do tipo get e set, para acessar e definir os valores de cada um dos atributos do objeto cliente.

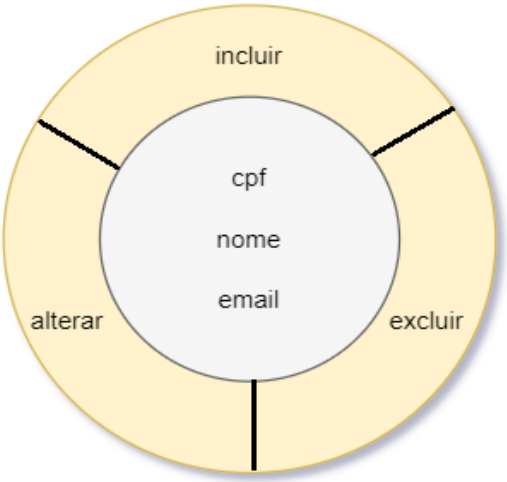

Figura 5.8 – Representação de um Objeto do tipo Cliente

5.3. Comunicação entre objetos

Na orientação a objetos, a comunicação ocorre quando os objetos trocam mensagens entre si. Mas, como isso acontece? A comunicação ocorre quando um objeto precisa de algum recurso ou serviço existente em outro objeto. Imagine que um determinado objeto chamado “mensagem”, possui um texto que deve ser enviado por e-mail. O objeto “mensagem” pode requisitar a um objeto do tipo e-mail que envie essa mensagem para alguém. A figura 5.9 ilustra a troca de mensagens entre diferentes tipos de objetos.

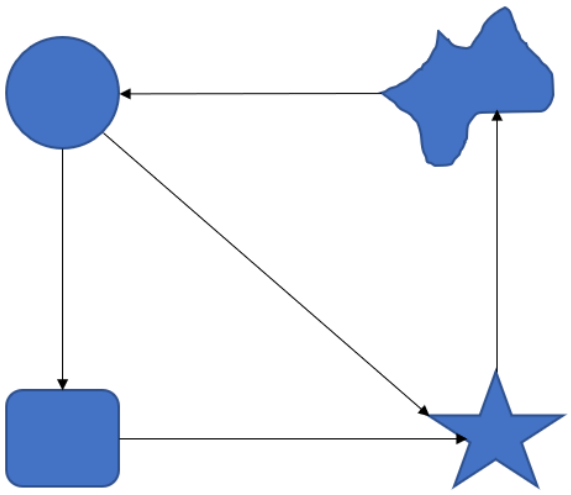

Figura 5.9 – Troca de mensagens entre objetos

Em casos de aplicações reais, é comum existir diversos tipos de objetos e também que um desses objetos tenha a necessidade de fazer uma tarefa que já esteja definida em algum outro objeto. Esse envio de mensagem ocorre por meio da chamada de um método. Pela UML, o sentido da flecha representa exatamente o

sentido da mensagem em que um objeto emissor envia uma mensagem ao objeto receptor. Se voltarmos à Figura 5.9, podemos interpretar que existe uma mensagem sendo enviada pelo objeto círculo (o emissor) a um objeto estrela (o receptor). Obviamente, se o objeto círculo está enviando uma mensagem ao objeto estrela, isso significa que este último disponibiliza o serviço requisitado.

Apesar de a comunicação ser uma característica da orientação a objetos, ela cria uma forte dependência entre os objetos envolvidos, ou seja, se o objeto receptor sofrer uma manutenção, pode ser que o objeto emissor tenha que ser alterado também. Para reduzir esse forte acoplamento, existem padrões de projeto que criam uma camada intermediária entre os dois objetos que se encarrega de realizar a comunicação. Um tipo de padrão que define um tipo de “mensageiro” que se encarrega de gerenciar as mensagens é chamado de Strategy.

A comunicação é utilizada em diversos diagramas da UML, dentre os quais podemos citar o diagrama de sequência e o diagrama de comunicação. Nesses diagramas, a mensagem é representada por diferentes tipos de flechas, cada uma com um significado diferente. A figura 5.10 ilustra os modelos de flechas utilizadas na representação de mensagens entre objetos.

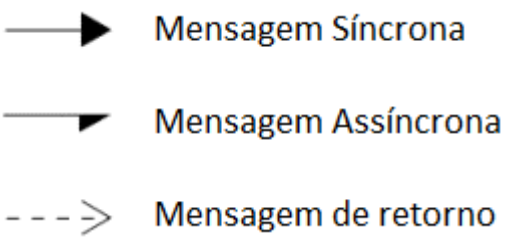

Figura 5.10 – Comunicação entre objetos na UML

A primeira flecha, se refere à uma mensagem síncrona. Esse tipo de mensagem indica que o objeto receptor precisa esperar o retorno da mensagem para continuar a execução, ou seja, enquanto o objeto receptor não terminar a execução e retornar a mensagem, o objeto emissor fica em estado de espera. Já uma mensagem assíncrona, na parte central da imagem, libera o objeto emissor para continuar a execução, mesmo sem ter obtido retorno do objeto receptor. Em outras palavras, o objeto emissor pode realizar outras operações até ser notificado pelo objeto receptor. Já a flecha da parte inferior da imagem representa uma mensagem de retorno. Normalmente, essa mensagem não é obrigatória, mas pode ser usada

para oferecer maior semântica ao processo, visto que permite deixar claro o tipo de retorno realizado pelo objeto receptor.

5.4. Herança

Já citada brevemente em seções anteriores, a herança representa o principal mecanismo para reaproveitamento de código existente na orientação a objetos. Ela gera um tipo de relacionamento que possibilita a criação de classes que herdam atributos e métodos de outras classes, a fim de evitar que o código tenha que ser escrito novamente.

A herança gera um relacionamento entre classes que faz com que uma das classes herde atributos e métodos de outra classe. Na verdade, conceitualmente dizemos que uma subclasse herda características de uma superclasse. Se considerarmos o uso de objetos, podemos dizer que um objeto recebe, ou toma para si, propriedades já existentes em outros objetos. Observe o diagrama da Figura 5.6 que ilustra uma representação da herança.

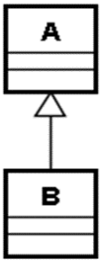

Figura 5.11 – Representação da herança na UML

Analisando o diagrama da Figura 5.11, a classe A é a superclasse, enquanto que a classe B está fazendo o papel da subclasse. Uma outra forma de realizarmos a leitura desse diagrama é: a classe B é uma classe do tipo A. Dissemos anteriormente que uma classe serve de modelo para a criação de objetos, mas, ao considerarmos o mecanismo de herança temos que uma superclasse serve de modelo para a criação de outras classes (suas subclasses).

Imagine que a classe A é uma classe chamada pessoa que contém diversos atributos, tais como código, nome, endereço etc. A classe B poderia ser uma classe chamada Cliente, que herdaria os atributos já existentes na classe Pessoa, podendo incluir novos atributos. Se, por exemplo, a classe Cliente possuísse os atributos e-

mail e telefone, então, de fato ela teria cinco atributos, os dois definidos na classe Cliente e os três herdados por meio da classe A.

Existem diversas categorias de herança: herança simples, herança múltipla, herança com vários níveis, hierárquica, entre outras. Vamos apresentar alguns exemplos de classes contendo apenas os atributos para simplificação dos diagramas, mas, é importante salientar que a herança envolve também os métodos existentes nas classes.

A Figura 5.12 ilustra o exemplo de uma herança simples em que uma classe chamada Estudante está herdando as propriedades de uma classe chamada Pessoa. Podemos considerar que este diagrama representa dois níveis, pois existe uma classe mãe (Pessoa) e uma classe filha (Estudante).

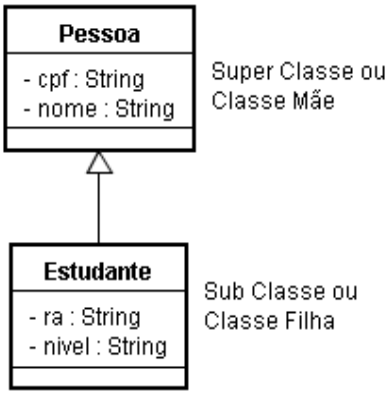

Figura 5.12 – Representação da herança na UML

A Figura 5.13 ilustra uma herança de múltiplos níveis. Observe que existe uma classe chamada Pessoa, a superclasse Funcionário que herda as propriedades da classe Pessoa, portanto, a classe Funcionário é subclasse da classe Pessoa. Na sequência, temos uma classe chamada Vendedor que herda as propriedades da classe Funcionário. Nesse caso, a classe Vendedor herda todas as propriedades de Funcionário e de Pessoa, agrupando todas as propriedades em uma única classe. Veja que não existe a necessidade de colocar as classes no sentido vertical, o que importa é o sentido da flecha.

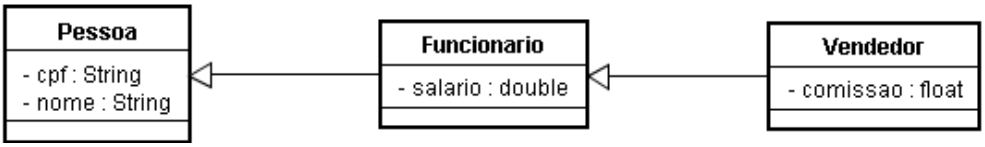

Figura 5.13 – Hierarquia de três níveis

A Figura 5.14 ilustra o exemplo de uma herança múltipla. Observe que a classe Automóvel está herdando propriedades da classe Veículo e também da classe Motor, ou seja a classe Automóvel é subclasse de duas superclasses ao mesmo tempo. Dessa forma, pela representação do diagrama, a classe Automóvel possui dez atributos, pois agrega os atributos de todas as classes.

É importante salientar que nem todas as linguagens orientadas a objeto oferecem suporte à herança múltipla.

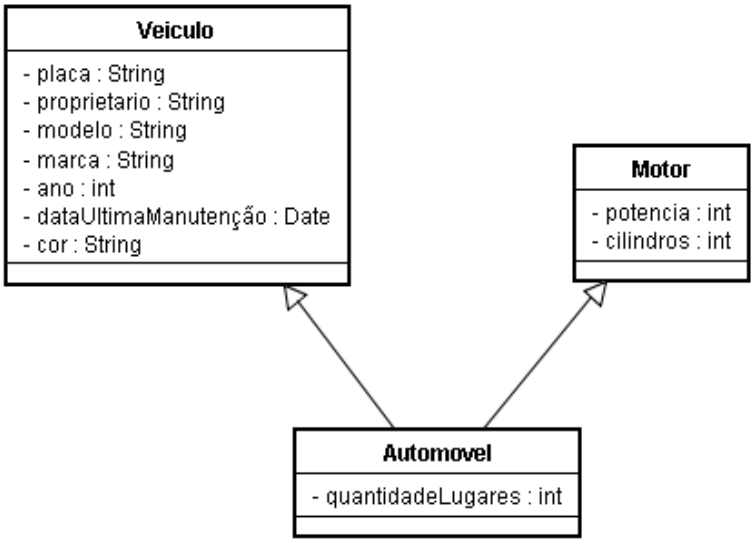

Figura 5.14 – Herança múltipla

A utilização da herança fornece vantagens e desvantagens. Como já vimos, um importante benefício da herança é o reaproveitamento de código, pois é possível reaproveitar o código de outras classes já conhecidas e testadas. Esse pode ser um grande benefício, depois diminui o tempo de desenvolvimento e os custos associados à manutenção.

Por outro lado, a herança também possui algumas desvantagens. O uso incorreto da herança pode acarretar problemas no projeto. Membros de uma superclasse que não são utilizados nas subclasses podem gerar desperdício de memória, além disso a herança causa uma forte dependência entre as classes da hierarquia, ou entre as classes derivadas como também são chamadas. Uma mudança na superclasse irá afetar todas as subclasses.

5.5. Abstração

A abstração de dados é uma propriedade em que apenas detalhes essenciais são selecionados e/ou apresentados aos usuários. Se fizermos uma analogia com um smartphone, o usuário tem conhecimento apenas das funcionalidades disponíveis a ele, detalhes sobre os componentes internos, ou mesmo o funcionamento do sistema operacional são suprimidos. De forma semelhante, quando estamos em um veículo temos conhecimento sobre os procedimentos relacionados a sua direção, mas, talvez não tenhamos a mínima noção de como o motor funciona, ou como ocorre a mudança automática das marchas. Algo semelhante ocorre na orientação a objetos, podemos criar uma classe que disponibiliza diversos métodos que serão consumidos por uma aplicação cliente. Essa aplicação cliente precisa saber como acessar e como receber resultados da classe, sem, no entanto, saber como os métodos internos funcionam.

A abstração também pode ser vista também como o processo que identifica apenas características necessárias de um objeto, e que ignora detalhes insignificantes dentro de um contexto. Por exemplo, considere que seja necessário projetar a classe Livro. Quais atributos e métodos podem ser definidos? A resposta correta é: depende do contexto onde essa classe for utilizada. Se considerarmos uma biblioteca, serão necessários determinados atributos e métodos que, com certeza, seriam diferentes se essa mesma classe Livro fosse utilizada numa livraria e ainda diferente para uma editora.

Dessa forma, a abstração permite definir o que é importante para uma classe em determinado contexto. Quando você precisar definir a estrutura para uma classe, deve primeiramente reconhecer em qual ambiente essa classe irá operar e ainda precisará identificar quais funcionalidades serão necessárias para essa classe. Uma mesma classe pode ter uma estrutura e conjunto de métodos completamente diferente quando usada em contextos diferentes.

5.6. Polimorfismo

Um dos importantes conceitos relacionados à orientação objetos se refere ao polimorfismo. O termo polimorfismo é formado por duas palavras: poli e morfo, ou múltiplas formas. Isso não quer dizer, necessariamente, que o objeto círculo se

transformará em um quadrado. Na verdade, o polimorfismo é melhor traduzido como múltiplos comportamentos. Vamos entender melhor isso.

Considere um objeto genérico chamado animal, proveniente de uma classe com o mesmo nome. Talvez você esteja pensando, mas que animal é este? A ideia do polimorfismo é que durante a execução do código, um objeto animal se comporte como qualquer tipo de animal. O objeto pode se comportar como um gato, ou um pássaro, dependendo das condições no momento de sua criação. Portanto, não estamos dizendo que um animal vai se comportar como um livro, por exemplo, mas com um tipo de animal específico de sua classe.

O que se espera de um cachorro? Obviamente, que ele lata, corra, coma, brinque e assim por diante. Já um pardal poderia piar, voar, bater asas, enfim, teria um comportamento de acordo com sua natureza. Algo semelhante pode acontecer com os objetos de software, de acordo com o conceito do polimorfismo.

Na implementação do polimorfismo é declarado um objeto genérico e, durante sua execução, esse objeto poderá adquirir comportamentos diferentes, a partir da classe da qual ele for criado. Para que o polimorfismo possa ser utilizado existem dois pré-requisitos importantes: em primeiro lugar, as classes envolvidas que servirão de base para a criação de objetos devem estar em uma hierarquia de classes, de acordo com o que vimos no conceito de herança. Considere, por exemplo, uma superclasse chamada Animal contendo as subclasses Gato, Cachorro, Pato, Rinoceronte etc.. Em segundo lugar, existe a necessidade de que todas as subclasses envolvidas possuam métodos sobrescritos. O método sobrescrito (override) ocorre quando uma subclasse reescreve um método existente na superclasse. Para facilitar a compreensão vamos analisar a Figura 5.15.

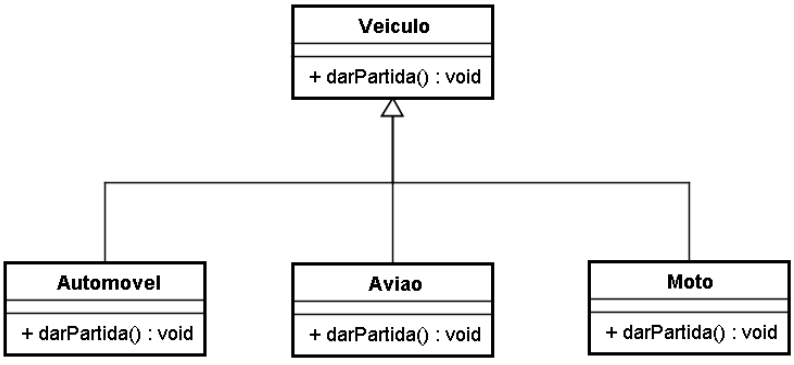

Figura 5.15 – Sobrescrita de métodos

Você já sabe que a classe Veiculo é a superclasse, enquanto as classes Automovel, Aviao e Moto são subclasses. Observe que na classe Veiculo existe um método chamado **darPartida**. Considere que este método contém apenas declarações genéricas relacionadas a dar partida em um veículo qualquer (esse método pode também estar em branco, sem nenhum código). Considere também que o método darPartida existente na classe Automovel implementa todas as etapas relacionadas a ligar um automóvel, o mesmo para a classe Aviao e também para a classe Moto, ou seja, cada classe implementa as necessidades individuais a cada tipo de veículo.

Nesse ponto você deve estar pensando: se já existe o método darPartida em Veiculo e, consequentemente, ele é herdado nas subclasses, por qual motivo existe a necessidade de escrevê-lo novamente nas subclasses? O motivo é simples: cada tipo de veículo possui etapas, ou funcionalidades, diferentes para dar partida, ou seja para ligar o motor. Portanto, a sobrescrita é necessária sempre que o método genérico existente na superclasse não atender os requisitos de funcionamento nas subclasses. Então, sobrescrever um método significa escrevê-lo novamente para atender uma necessidade individual da classe.

Uma vez definida a hierarquia apresentada na Figura 5.15, juntamente com a sobrescrita dos métodos, é possível implementar o polimorfismo. De forma resumida, a utilização do polimorfismo em uma aplicação, poderia ser da seguinte forma: declaramos um objeto do tipo Veiculo (genérico) e, durante a execução, dependendo das necessidades da aplicação, será gerado um objeto do tipo Automovel, Aviao ou Moto. Dizemos então, que um objeto do tipo Veiculo passará a se comportar como um automóvel, um avião ou uma moto. Isso é possível porque cada subclasse contém implementações diferentes em seu método darPartida.

Os benefícios do uso do polimorfismo se concentram na economia de recursos, já que um mesmo objeto pode ser usado em diferentes circunstâncias. Além disso, o sistema tende a ser mais flexível e resistente a modificações. Por outro lado, a implementação do polimorfismo pode representar aumento de complexidade no código. Vimos o polimorfismo com a utilização de classes, mas existe também o polimorfismo por meio de interfaces, assunto tratado na seção seguinte.

5.7. Interfaces

O termo interface normalmente está associado às telas do usuário - a interface gráfica, ou ainda GUI (Graphical User Interface). No entanto, não é este tipo de interface que estamos nos referindo, mas de um conceito relacionado à orientação a objetos. Em termos gerais, nosso foco se concentra no seguinte: uma interface se refere a um mecanismo que permite a comunicação entre dois objetos de software, diferente de uma interface gráfica que permite a comunicação entre o usuário e o sistema.

Fazendo uma analogia com o mundo real, em orientação a objetos uma interface define uma espécie de contrato com uma classe. Você sabe que um contrato do mundo real, de compra e venda por exemplo, contém uma série de regras que devem ser seguidas pelos envolvidos para que o objetivo final seja encontrado. Isso ocorre de maneira semelhante com classes e interfaces.

Vamos começar pela interface, lembrando que não estamos focando em nenhuma linguagem de programação, apenas apresentando sua representação por meio da UML. Observe pela Figura 5.16 que existem diferentes representações para as interfaces.

Figura 5.16 – Representação de interfaces na UML

Olhando a Figura 5.16 podemos notar que existem três representações diferentes para as interfaces. A primeira, da esquerda, bastante semelhante à representação da classe, permite identificar toda a estrutura como, por exemplo, declarações de constantes e métodos. A única diferença em relação à representação da classe é a presença de um estereótipo chamado <<interface>>. Na UML, os estereótipos são utilizados para diferenciar elementos semelhantes fornecendo maior semântica ao modelo. Observe que a única maneira de diferenciar a interface Motor de uma classe é a presença do estereótipo <<interface>>.

A representação central da Figura 5.16, cujo nome da interface é Porta, indica uma interface do tipo provedora, ou seja, um elemento que provê uma interface chamada porta (em inglês o termo utilizado para este tipo de interface é Provided Interface). Esse elemento provedor da interface pode ser uma classe. Na UML, quando uma classe implementa uma interface, dizemos que a classe **realiza** (realizes) a interface.

A representação mais à direita na Figura 5.16, ilustra dois tipos de interface ao mesmo tempo, o círculo representando a interface provedora e um semi círculo representando uma interface requerida, isto é, um elemento que precisa (ou recebe) uma interface para funcionar. Essas duas representações juntas podem ser comparadas a uma tomada do tipo macho e fêmea, ou seja, existe um elemento provedor e um elemento receptor, um elemento que fornece uma interface e um elemento que precisa de uma interface para funcionar. Esse conceito é abstrato e seria mais fácil de entender se estivéssemos o implementando por meio de uma linguagem de programação.

Vamos tratar agora do contrato que dissemos anteriormente a respeito da interface. Para isso considere o diagrama apresentado na Figura 5.17.

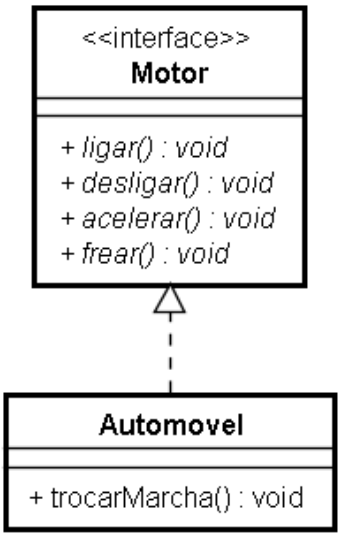

Figura 5.17 – Classe Automovel implementando a Interface Motor

Analisando o diagrama da Figura 5.17, vemos que foram declarados quatro métodos na interface Motor (ligar, desligar, acelerar e frear). Uma interface contém apenas a definição (também chamada de especificação) de métodos. Isso significa que não pode existir nenhum tipo de implementação (o método não possui código),

a interface apenas especifica o que as classes (que seguirão seu contrato) deverão conter. Todas as classes que realizarem a interface Motor deverão, obrigatoriamente, implementar os métodos por ela especificados. Como pode ser visualizado no diagrama, a classe Automovel contém o método trocarMarcha, mas não é apenas isso! Pelo fato de existir um relacionamento entre a classe e a Interface (seta com linha tracejada), definido pela UML com o nome de **realiza**, dizemos que a classe Automovel realiza a interface Motor. Numa linguagem de programação o termo correto é implementa (uma classe implementa uma interface). Em função disso, a classe Automovel é obrigada a definir todos os métodos existentes na interface, ou seja, apesar de não estar implícito no diagrama, a classe Automovel possui cinco métodos, o trocarMarcha mais os quatro especificados na interface.

Uma classe pode realizar mais de uma interface ao mesmo tempo. Essa característica está ilustrada na Figura 5.18. Observe que a classe Aviao realiza as interfaces Turbina e Motor. Isso deixa explícito no diagrama de que a classe Aviao possui seu método moverFlaps e todos os métodos especificados em Turbina e Motor.

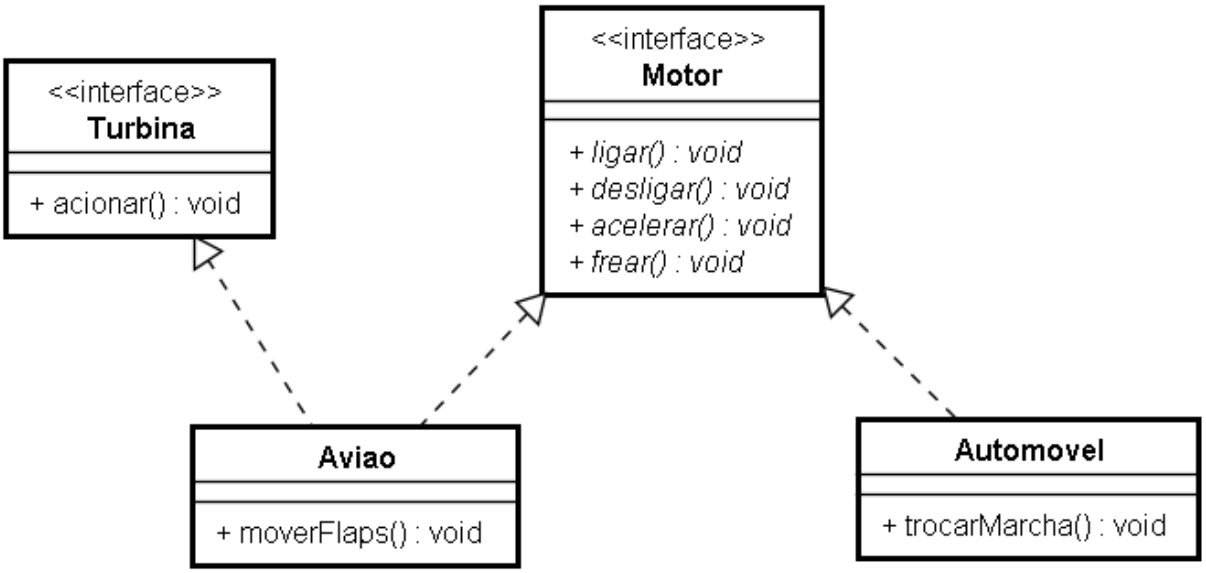

Figura 5.18 – Classe Aviao implementando duas Interfaces

O uso de interfaces garante, portanto, que uma determinada classe implemente os serviços especificados pelas interfaces. Esse é um importante mecanismo da orientação a objetos, pois permite que classes sejam substituídas por novas versões, sem que a aplicação cliente se dê conta disso! Se uma determinada aplicação utiliza um objeto do tipo Automovel, e essa classe for substituída por uma

versão mais recente, por exemplo, AutomovelEletrico, desde que esta implemente a mesma interface de Automovel, a aplicação não deverá sofrer grandes impactos, idealmente, a aplicação não deveria sentir nenhuma diferença.

REFERÊNCIAS

FURGERI, S. **Java** – Ensino Didático. 1.ed. São Paulo.: Editora Saraiva, ISBN 978-8536527260, 2018.

FURGERI, S. **Modelagem de Sistemas Orientados a Objetos** – Ensino Didático. 1. Ed. São Paulo: Editora Érica, ISBN 978-8536504612, 2013.

DEITEL, P.; DEITEL, H. **Java – Como Programar**. 8. Ed. Editora Prentice Hall – Br., 2009.

Apêndice A – ArgoUML

Esse apêndice tem o objetivo de fornecer uma visão geral sobre a utilização do ArgoUML, uma ferramenta gratuita que permite elaborar diversos diagramas da UML, dentre os quais destacamos o diagrama de casos de uso e o diagrama de classes.

ArgoUML é uma ferramenta CASE que suporta a elaboração dos diversos diagramas da UML, diagramas de requisitos SysML e os diagramas ER (Entidades e Relacionamentos). Além de permitir a elaboração e manipulação do projeto de modelagem, a ferramenta oferece suporte a transformações específicas para códigos-fonte de algumas linguagens de programação como, por exemplo, Java, PHP, Python, C++ e C#.

Este apêndice não mostra a descrição completa de todas as funcionalidades da ferramenta, mas apenas uma breve introdução de como elaborar diagramas simples. Nosso objetivo é ajudar o leitor a ter um primeiro contato com a utilização do ArgoUML.

A.1. Download e Instalação do ArgoUML

O Argo pode ser baixado em: https://argouml-tigris-org.github.io/tigris/argouml/. Ao entrar no site, é apresentado um botão de *download*, como pode ser visto na Figura A.1.

Figura A.1 - Tela para download do ArgoUML

Após o download terminar:

1. Localize o arquivo do ArgoUML que você baixou e dê duplo clique nele para iniciar a instalação.
2. O Windows pergunta se você realmente deseja executar a instalação do software; pressione o botão Executar.
3. Na tela de boas-vindas (Welcome) do assistente de instalação, clique no botão Next.
4. Na tela “Choose Components”, selecione JRE caso não possua o JAVA instalado na sua máquina. Após isso, aperte Next.
5. Defina o diretório de instalação (clicando no botão Browse...) ou mantenha o padrão clicando no botão Next.
6. A tela seguinte permite que seja definido como será a criação de atalhos no menu Start para o ArgoUML, mantenha como sugerido e clique no botão Install.
7. A ferramenta será instalada, e após clicar em Finish, mantendo a opção Run ArgoUML habilitada, executará o programa.

A.2. Conhecendo o ArgoUML

Quando executado, a ferramenta apresenta a página inicial. A Figura A.2 mostra a tela principal do Argo.

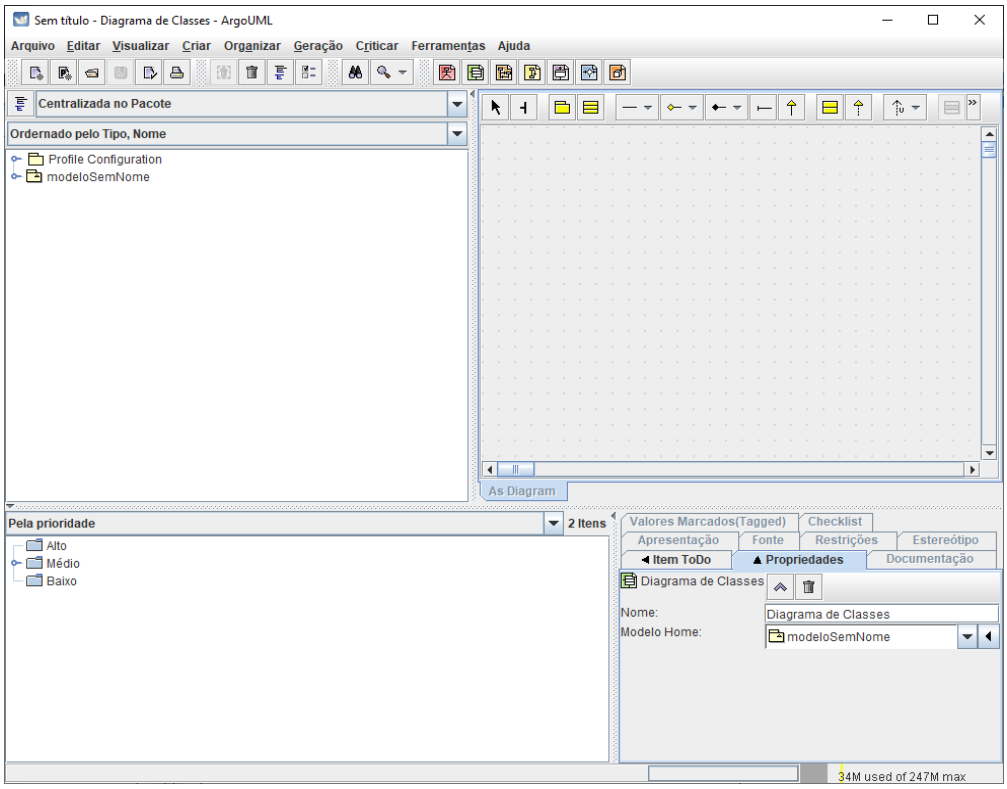

Figura A.2 – Tela inicial do ArgoUML

Na parte superior da tela principal do Argo existe um menu e uma barra de ferramentas contendo as funções mais usadas. A Figura A.3 apresenta o cabeçalho da tela principal, incluindo o menu principal e a barra de ferramentas do Argo.

Figura A.3 – Menu principal e barra de ferramentas

A Tabela A1 descreve a função de cada ícone existente na barra de ferramentas.

Ícone	Descrição
	Criar novo projeto.
	Criar novo perfil.
	Abrir projeto existente.
	Salvar projeto.
	Propriedades do projeto.

	Imprimir diagrama.
	Remover do diagrama.
	Apagar do modelo.
	Configurar perspectivas.
	Configurações do projeto.
	Procurar.
	Zoom.
	Criar novo diagrama de uso.
	Criar novo diagrama de classes.
	Criar novo diagrama de sequência.
	Criar novo diagrama de colaboração
	Criar novo diagrama de estados.
	Criar novo diagrama de atividades.
	Criar novo diagrama de distribuição.

Tabela A.1 – Função dos Ícones

Na lateral esquerda da tela principal, temos a árvore do projeto. Nela, encontramos os arquivos contidos nele hierarquicamente, como possível ver na Figura A.4.

Figura A.4 – Árvore do projeto

Na parte inferior esquerda da tela principal, encontramos um painel com 9 abas. Para este apêndice, focaremos na aba “Propriedades”. Como o próprio nome sugere, esta aba apresenta todas as propriedades de um elemento qualquer selecionado no diagrama. Além de listar as propriedades é possível editar seus valores. Alterações realizadas neste painel refletem diretamente no desenho do modelo editado. A Figura A.5 apresenta o painel Propriedades.

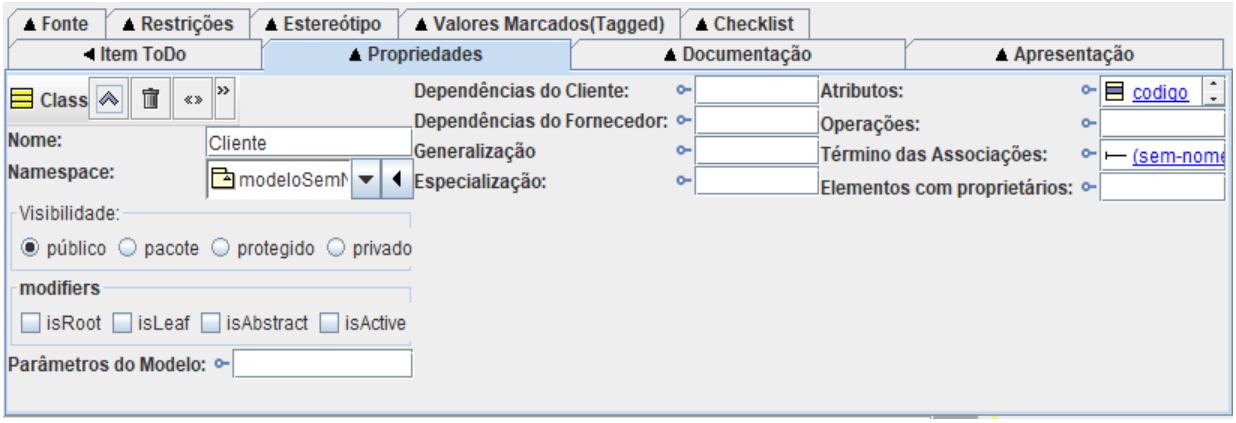

Figura A.5 – O painel Propriedades

A.3. Elaborando os Diagramas da UML

O objetivo dessa seção é demonstrar a criação de diagramas pelo Argo, mais especificamente apresentar os passos para elaborar os diagramas de casos de uso e classe.

O Argo trabalha de forma semelhante a um sistema gerenciador de banco de dados, armazenando todos os diagramas de um projeto num mesmo arquivo. Por isso, antes de iniciar a elaboração dos diagramas seguintes, vamos elaborar um projeto. Para elaborar um projeto faça o seguinte:

1. Na aba “Arquivo”, no menu superior, clique em “novo”, como demonstrado na Figura A.6. Feito isso, o programa irá carregar um novo projeto.

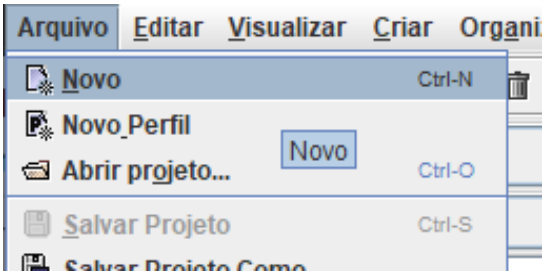

Figura A.6 – Criação de um projeto

A.3.1. Diagramas de caso de uso

Com o Argo, é possível não apenas desenhar diagramas de casos de uso como também documentar o fluxo de eventos associado a cada um deles. Apresentaremos os passos necessários para que você possa elaborar o diagrama da Figura A.11. Vejamos:

1. Para se criar o diagrama de casos de uso acesse o caminho pelo menu superior: Criar => Novo diagrama de uso. Ou simplesmente clique no botão destacado na Figura A.7.

Figura A.7 – Criação do diagrama de caso de uso

2. Surgirá o painel “Propriedades” que dará acesso a uma janela que contém diversas especificações referentes ao diagrama. Para dar nome ao diagrama, localize o painel Propriedades e clique na caixa Nome. Nela, defina o nome que aparece na Figura A.8.

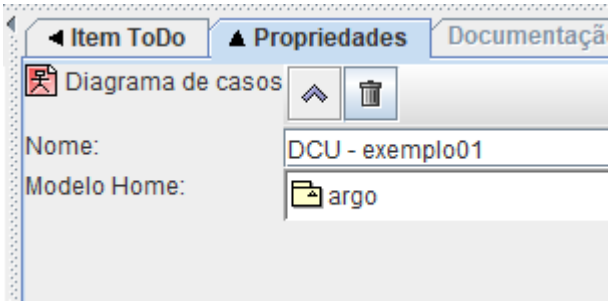

Figura A.8 – Definindo o nome do diagrama

No Argo, ao se criar qualquer diagrama, surgirá a janela de propriedades, que dará acesso a todas as especificações do diagrama. Da mesma forma, surge um painel dos elementos da UML associados ao diagrama, isto é, para cada diagrama surgem diferentes elementos a ele associados. Observe, por exemplo, a Figura A.9 que contém os diversos elementos que podem ser usados na elaboração do diagrama de casos de uso.

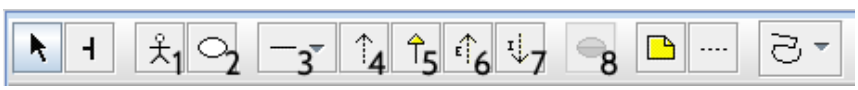

Figura A.9 – Elementos do Diagrama de Casos de Uso

A seguir faremos uma pequena descrição do significado e de como utilizar alguns elementos que se encontram no painel da Figura A.9. Cada uma das opções disponíveis nela (numeradas de 1 a 8) se refere a possíveis relacionamentos que um ator pode ter com outros elementos da UML.

As opções disponíveis são:

1. Cria um ator.
2. Cria um caso de uso.
3. Cria uma nova associação.
4. Cria uma nova dependência.
5. Cria uma nova generalização.
6. Cria uma nova extensão.
7. Cria uma nova inclusão.
8. Cria um ponto de extensão.

Para inserir um elemento, basta clicar no seu respectivo ícone e clicar na área de desenho onde deseja incluí-lo.

3. Continuando nossa descrição para elaboração do diagrama de caso de uso, arraste um Ator para o diagrama criado no item 2. Ao soltar o elemento na área de desenho surgirá o homem palito, como na Figura A.10. Por padrão, o ator não possui nome, mas é importante sempre nomear os atores. No painel “Propriedades”, defina o nome do ator como Gerente.

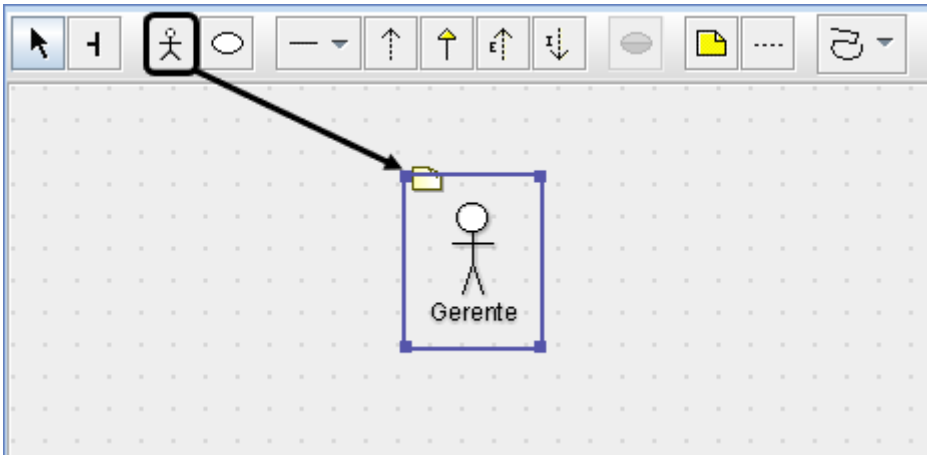

Figura A.10 – Inclusão de um ator

4. Para finalizarmos nosso diagrama, arraste três elipses a partir do ator gerente e defina os nomes que aparecem na Figura A.11.

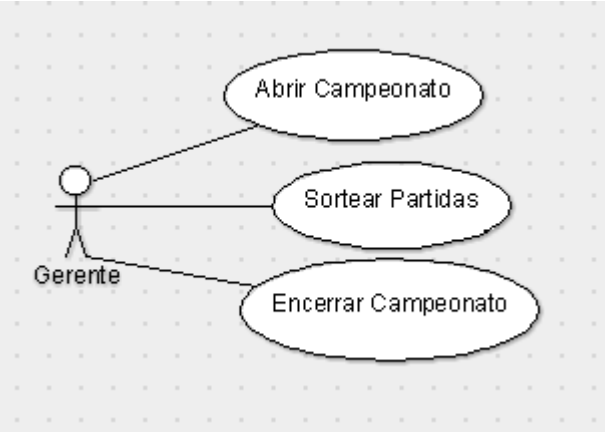

Figura A.11 – Diagrama de casos de uso

Nosso diagrama já está pronto, mas vamos descrever a inserção de um caso de uso da mesma forma que fizemos com o ator, isto é, arrastando o componente a partir da paleta de componentes (e não a partir do ator como fizemos no item 4).

Para inserir um caso de uso, crie um novo diagrama, conforme Figura A.12 e arraste uma elipse para ele. Ao soltar o elemento na área de desenho surgirá a elipse referente a representação do caso de uso.

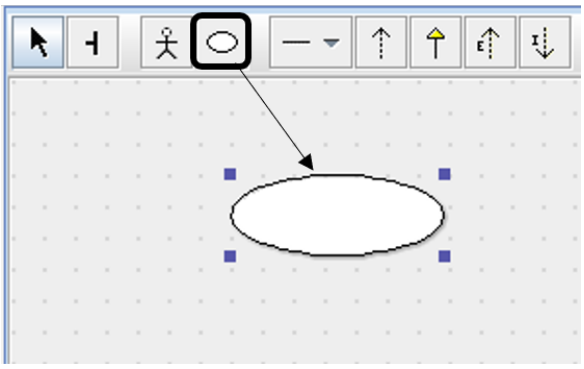

Figura A.12 – Inclusão de um caso de uso

Cada um dos elementos disponíveis na Figura A.9 (numeradas de 1 a 8) se referem a possíveis relacionamentos que um caso de uso pode realizar com outros elementos da UML, da mesma forma que descrevemos para a inserção de um ator.

A partir do Caso de Uso selecionado é possível relacioná-lo a:

1. Um novo ator.

2. A outro caso de uso por meio de uma inclusão.
3. A outro caso de uso por meio de uma extensão.
4. A outro caso de uso por meio de uma generalização.
5. A outro caso de uso por meio de uma colaboração.
6. A uma nota explicativa.
7. A um recurso genérico.

Como dissemos anteriormente, não é nosso objetivo esgotar as funcionalidades existentes no Argo, mas apenas demonstrar como elaborar alguns diagramas. A Figura A.13 ilustra diversos tipos de relacionamentos existentes entre atores e casos de uso.

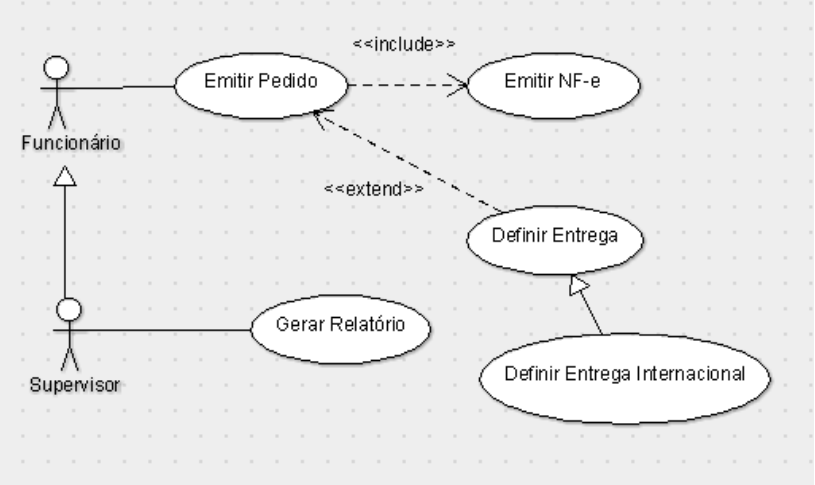

Figura A.13 – Relacionamentos mais comuns em diagramas de casos de uso

Os relacionamentos representados na Figura A.13 são:

a) **Generalização**: entre Supervisor e Funcionário e entre Definir Entrega e Definir Entrega Internacional;

b) **Associação**: entre Funcionário e Emitir Pedido e entre Supervisor e Gerar Relatório;

c) **Inclusão**: entre Emitir Pedido e Emitir NF-e;

d) **Extensão**: entre Definir Entrega e Emitir Pedido;

A.3.2. Diagrama de Classes

Nesta seção, vamos apresentar os passos para a elaboração de um diagrama de classes.

1. Para se criar um diagrama de classe, vá no menu superior => Criar => Novo diagrama de classes. Ou clique no botão “Novo Diagrama de Classes” na barra de ferramentas. Digite o nome DCU – exemplo01 no painel Propriedades. Para criar um novo diagrama de classes, também é possível clicar no ícone ilustrado na Figura A.14.

Figura A.14 – Ferramenta para criação de um diagrama de classes

2. Surgirá a janela por meio da qual podemos desenhar nosso diagrama. Arraste o elemento Classe (Ilustrado na Figura A.14) para a área do diagrama criado, como ilustrado na Figura A.16. Ao soltar o componente na área de desenho surgirá o painel Propriedades com diversas opções. Da mesma forma que fizemos anteriormente com o elemento Ator no diagrama de casos de uso, vamos focar apenas nas opções que aparecem na Figura A.15. Por padrão, uma classe nova é inserida sem nenhum nome, então a nomeie como Cliente.

Figura A.15 – Elementos do Diagrama de Casos de Uso

Cada opção numerada na Figura A.15 representa:

1. Criar um novo pacote.
2. Inserir nova classe.
3. Inserir nova associação.
4. Inserir nova agregação.
5. Inserir nova composição.
6. Realizar fim de associação.
7. Nova generalização.
8. Inserir interface.
9. Nova realização.
10. Novo uso.
11. Inserir atributo.
12. Inserir operação.
13. Nova classe de associação.
14. Novo sinal.

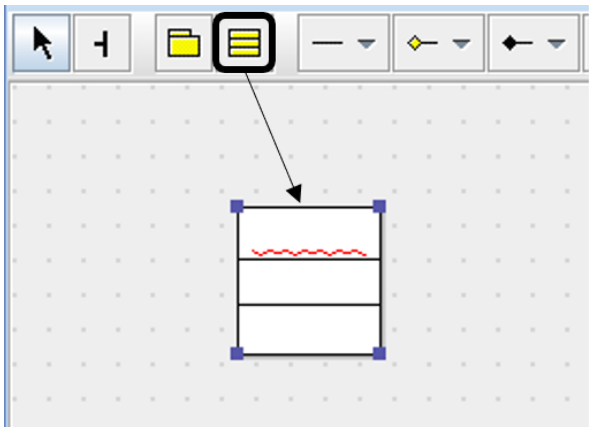

Figura A.16 – Inclusão de uma Classe.

A Figura A.16 ilustra a inclusão de uma classe por meio do ArgoUML.

3. Da mesma forma, insira outras três classes ao diagrama: Compra, Item e Produto.

4. Após isso vamos definir os atributos de cada classe. Você deverá inserir todos os atributos que aparecem na Figura A.21. Para isso siga o seguinte caminho: clique com o botão direito sobre a classe e escolha “Adicionar” – Novo atributo. Uma caixa de texto surgirá dentro da classe e você pode digitar o nome do atributo seguido por dois pontos (:) e seguido pelo tipo do atributo (Exemplo: “**atributo: Integer”**). Outra maneira de inserir um atributo na classe selecionada é pelo botão Novo atributo do painel Propriedades. Então, é possível inserir o nome, o tipo, entre outras propriedades. Podemos observar as duas formas de inserção de atributos de uma classe na Figura A.17.

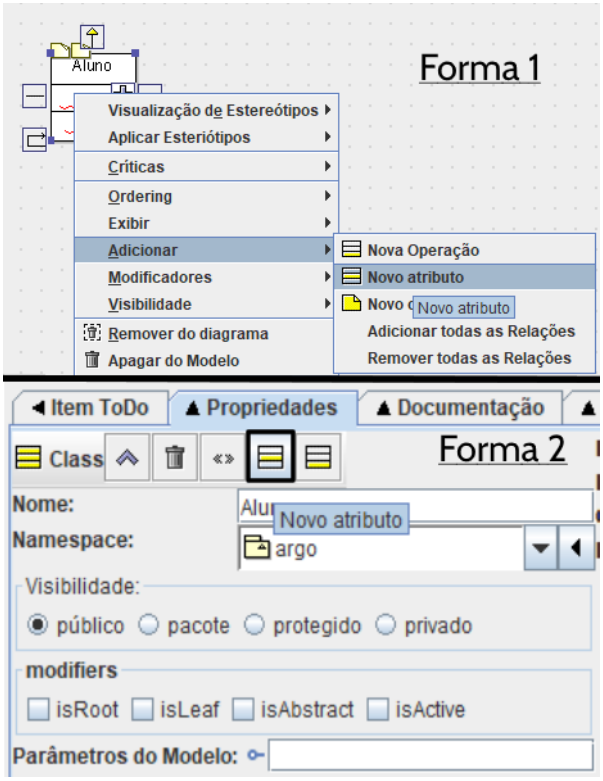

Figura A.17 – Definindo Atributos em Diagrama De Classe.

5. Após a inserção de todos os atributos das classes, passamos para os relacionamentos. Em nosso caso usaremos o relacionamento de associação. Para isso, basta clicar no botão “Nova Associação”, na barra de ferramentas, e traçar a linha entre as duas classes que deseja associar, como ilustrado na Figura A.18.

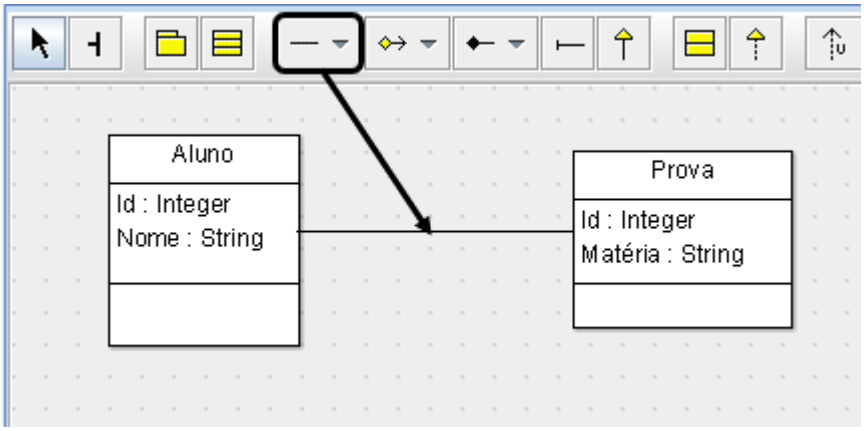

Figura A.18 – Relacionamento de Associação em Diagrama De Classe.

6. Feito o relacionamento entre as classes falta inserir a multiplicidade entre elas. Para isso, clique sobre a linha do relacionamento de associação existente entre as classes Cliente e Compra. Então, dê um clique duplo na caixa “Conexões”, como ilustrado no Passo 1 da Figura A.19. Assim, no painel Propriedades, aparecerão

novas opções. Em “Tipo”, se seleciona a classe para aplicar a multiplicidade, e em “Cardinalidade”, está a checkbox para ativá-la e o campo para selecionar o valor, como ilustrado no Passo 2 da Figura A.19. O mesmo processo deve ser repetido para os demais relacionamentos existentes em nosso diagrama da Figura A.21.

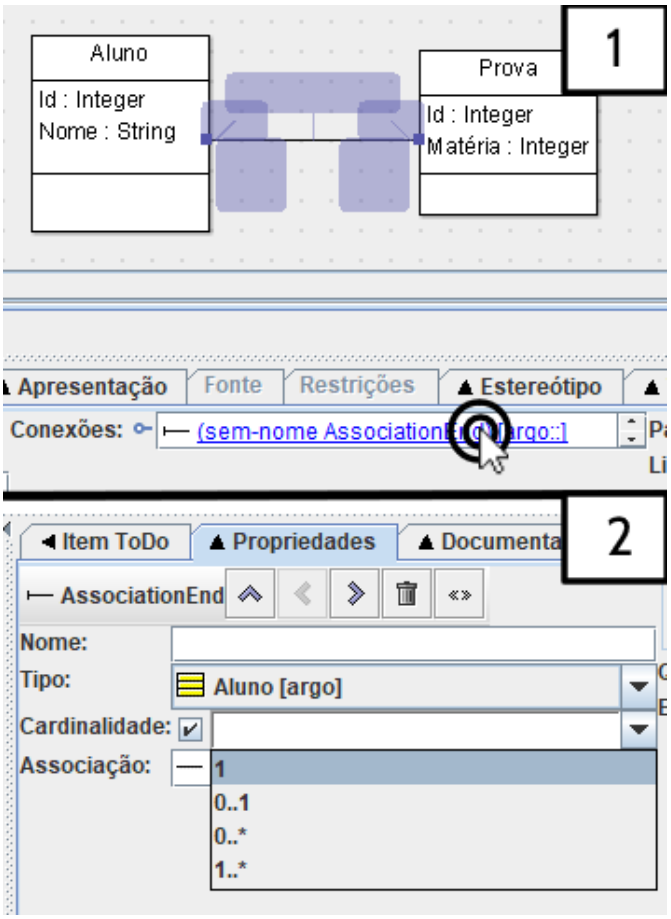

Figura A.19 – Definindo a Multiplicidade

7. Outra coisa necessária é inserir o nome dos relacionamentos. Para isso basta dar duplo clique sobre o relacionamento que se deseja nomear e inserir o nome na caixa de texto que aparece. Digite o nome dos relacionamentos que aparecem na Figura A.21. Para mostrar a direção de um relacionamento, clique com o botão direito do mouse sobre o relacionamento e selecione a direção na seção “Navegabilidade”, como pode ser visualizado na Figura A.20. Repita esse processo para todos os relacionamentos.

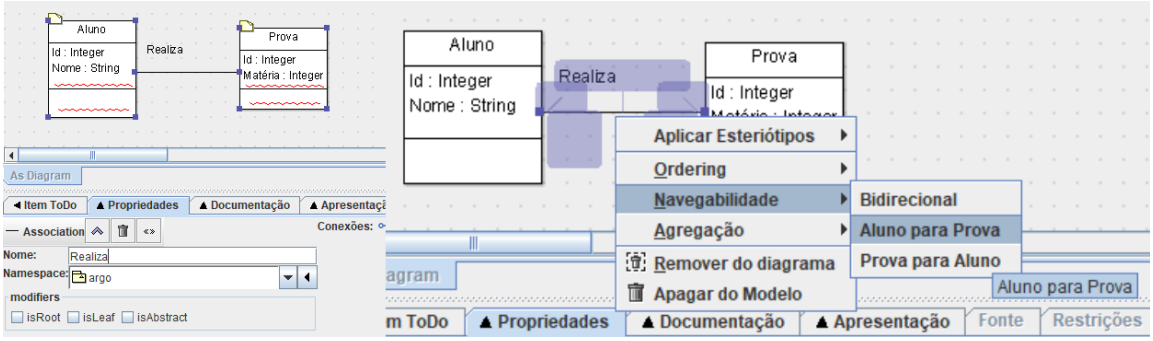

Figura A.20 – Nome e Direção de um Relacionamento

8. Com isso nosso diagrama deve estar completo. Confira seu diagrama com o que aparece na Figura A.21. Caso esteja faltando algo, realize a correção.

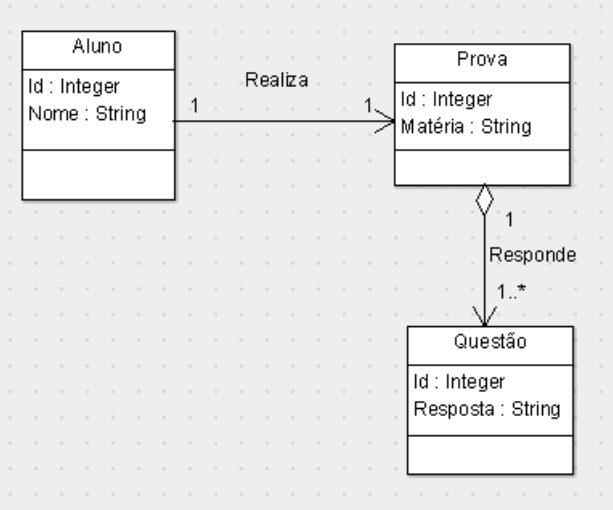

Figura A.21 – Diagrama de Classe