Paulino Ng

Conceitos básicos de Orientação Objeto

Brasil

2019, v-0.3.5

Lista de ilustrações

Lista de tabelas

Sumário

1 Conceitos Básicos de Orientação a Objetos

Você precisa entender os conceitos de orientação a objetos, OO, antes de aplicá-los com sucesso no desenvolvimento de sistemas. Como as técnicas OO emergiram, em parte, de áreas da engenharia de software, da inteligência artificial e da modelagem da informação, muitas dessas técnicas podem parecer já familiares para quem já trabalha na área de desenvolvimento de software. Não permita que esta familiaridade o torne complacente – você ainda terá vários conceitos novos para aprender.

Antes de começar com a orientação a objetos, relembre os principais paradigmas de programação disponíveis para o desenvolvimento de software.

- Programação imperativa
- Programação estruturada
- Programação funcional
- Programação orientada a objetos
- Programação por prototipagem
- Programação por aspectos

A **programação imperativa** é o modelo de programação que melhor segue a maneira de um processador executar as instruções. Dizemos que ela é imperativa pois os programas parecem dar ordens (comandos, instruções) para o processador executar e ele executa as instruções uma após a outra. Nesse modelo, o programador projeta seus programas como sequências de instruções a serem executadas no processador e o programador deve conhecer como funciona um processador, a arquitetura do computador, o sistema operacional e o modelo acesso à memória para melhor escrever seus programas.

A **programação estruturada** é um modelo de programação um pouco mais avançado do que o puramente imperativo. Nele, sequências de instruções são agrupadas em blocos, alguns blocos formam subprogramas chamados de funções ou procedimentos. Estruturas de controle de fluxo de instruções permitem a execução dos blocos de instruções. A instrução *GOTO*, ou *jump*, não é permitida em linguagens estruturadas, desde a publicação de um artigo do Dijkstra mostrando como o uso de desvios torna os programas mais suscetíveis a erros de programação. A programação estruturada faz forte uso de subprogramas e a principal maneira de reutilizar software é através de bibliotecas de funções e procedimentos. Wirth popularizou a decomposição funcional, ou projeto *top-down*, usado neste modelo de programação. Esta forma de projeto mostra-se adequada para pequenos projetos de software, mas é inadequada para projetos de maior porte e projetos com requisitos que mudam com frequência.

A **programação funcional** é um modelo de programação bastate antigo, basta lembrar que a linguagem LISP foi criada mais ou menos na mesma época que o Fortran e o Cobol. Neste modelo, os programas são funções. Semelhante às funções matemáticas, a valor dos argumentos das funções determinam o resultado do cálculo das funções. Neste modelo de computação, não existe o conceito de memória e estado de processamento. É um modelo muito útil para o processamento paralelo, já que não há interferência com variáveis de memória compartilhadas, o que elimina as dificuldades provenientes da programação concorrente nos outros modelos de programação.

A **programação orientada a objetos** ...

A **programação por prototipagem** ...

A **programação por aspectos** ...

Estes paradigmas, estilos de programação, são todos equivalentes no sentido em que todos devem permitir a resolução dos mesmos problemas com um computador, uma máquina de Turing. A

diferença está na facilidade de exprimir conceitos computacionais abstratos usados para a resolução dos problemas.

Estes diferentes paradigmas são apropriados para resolver diferentes tipos de problemas e desenvolvimentos. A programação imperativa, tão criticada por muitos desenvolvedores, pode ser útil no desenvolvimento de pequenas aplicações e bibliotecas que precisem de alto desempenho. Atualmente, a programação funcional voltou a receber uma grande atenção. A programação com JavaScript chama a atenção para a programação por prototipagem. Algumas linguagens de programação dão suporte a um ou vários paradigmas. A orientação a objetos é um dos principais paradigmas para a reutilização de código.

Os programadores novatos, muitas vezes, se desencorajam com a burocracia da programação orientada a objetos. Isto se deve principalmente à baixa complexidade dos problemas que lhes são propostos para resolver. Os novatos não entendem a vantagem de descobrir erros na compilação em vez de encontrá-los na execução. A resolução de problemas simples geralmente se beneficiam do uso de bibliotecas, especialmente as de entradas e saídas de dados, mas não há a necessidade de criar suas próprias bibliotecas (onde está o reuso de código). A orientação a objetos é muito importante para o projeto e implementação de soluções para problemas mais complexos. A orientação a objetos permite modelar melhor o domínio do problema, do sistema e da solução. O que resulta em uma decomposição mais natural do código e um melhor reuso dele. Um aspecto extremo deste reuso é a aplicação de padrões de projeto, *design patterns*, onde percebe-se que determinados problemas, não só de computação, são encontrados e reencontrados com frequência. No lugar de tentar reinventar uma solução, pode-se reutilizar uma solução já estada e aprovada. A análise e modelagem deste tipo de euso, muitas vezes, usa a modelagem OO.

Um outro diferencial da programação orientada a objetos é o tratamento de erros. Na programação OO, a estrutura *try-catch* permite isolar o código para o processamento sem erros do código para o processamento das situações com erro. Na programação imperativa usual, este tratamento é feito com o uso de estruturas condicionais que tornam a legibilidade e manutenção do código bastante complexa.

A seguir veremos:

- Uma rápida introdução aos conceitos de OO;
- Conceitos OO do ponto de vista estruturado;
- Os diagramas da linguagem unificada de modelagem (UML 2);
- Objetos e classes;
- Atributos e operações;
- Abstração, encapsulamento e ocultação de informação (*information hiding*);
- Herança;
- Persistência;
- Relacionamentos;
- Colaboração;
- Polimorfismo;
- Interfaces;
- Componentes; e
- Padrões.

1.1 Uma rápida introdução aos conceitos de OO

Nesta seção são vistos alguns conceitos fundamentais das técnicas de desenvolvimento OO. Programadores experientes com tecnologias estruturadas já conhecem muitos destes conceitos, outros, porém, são novos. Por exemplo, os conceitos que estão no núcleo de OO como encapsulamento, acoplamento e coesão vêm da engenharia de software. Estes conceitos são importantes porque eles revelam boas práticas de projeto independentemente da tecnologia que está sendo usada. Não se engane, entretanto, não é porque você já viu alguns dos conceitos e os pratica que você já está fazendo OO, mas você já está realizando bons projetos. Enquanto fazer bons projetos é uma grande parte da orientação a objetos, ainda há muito mais.

Os conceitos de OO parecem muito simples. Mas, os conceito em que se baseiam as técnicas estruturadas também pareciam simples, os programadores experientes sabem que o desenvolvimento estruturado é na verdade difícil. Assim como havia mais do que uns poucos conceitos simples no paradigma estruturado, há mais coisas no paradigma OO do que os conceitos básicos que veremos. Assim como leva tempo para se tornar um bom desenvolvedor estruturado, também leva tempo para se tornar um bom desenvolvedor OO. Dentre os diversos paradigmas de programação, o que pode parecer mais simples é o da programação funcional, pois nele usamos basicamente funções recursivas para obter iterações e composição de funções para obter funções mais complexas. É, entretanto, longe de ser simples a resolução de problemas complexos com a programação funcional. Para lhe dar um gostinho do conteúdo deste texto, alguns dos principais conceitos e termos que serão apresentados são resumidos na tabela 1.

Tabela 1 – **Conceitos e termos da Orientação a Objetos**

Termo	**Descrição**
Abstração	As características essenciais de um item (pode ser uma classe ou uma operação)
Acoplamento	O grau de dependência entre dois itens
Agregação	Relacionamento entre duas classes ou componentes definidas por *é parte de*
Hierárquia de agregação	Um conjunto de classes relacionadas através da agregação
Associação	Um relacionamento entre duas classes ou objetos
Atributo	Algo que uma classe conhece (dados/informação)
Cardinalidade	O conceito de *quantos?*
Classe	Uma abstração em software de objetos similares, um padrão a partir do qual objetos são criados
Classe abstrata	Uma classe que não pode ter objetos instanciados dela
Classe concreta	Uma classe que pode ter objetos instanciados dela
Classificador	Um termo de UML para uma coleção de instâncias que têm algo em comum. Isto inclui classes, componentes, tipos de dados e casos de uso
Coesão	O grau de relacionamento de uma unidade encapsulada (tal como um componente ou uma classe)
Colaboração	Classes que trabalham juntas (colaboram) para cumprir as suas responsabilidades
Componente	Uma unidade coesa de funcionalidade que pode ser desenvolvida, entregue e composta por outros componentes para construir uma unidade maior
Composição	Uma forma forte de agregação na qual o *todo* é completamente responsável pelas suas partes e cada objeto *parte* está associado apenas a um objeto *completo*
Encapsulamento	O agrupamento de conceitos relacionados num único item, tal como uma classe ou um componente
Estereótipo	Um uso comum de um elemento de modelagem

Continua na próxima página

Tabela 1 – continuação da página anterior

Termo	Descrição
Herança	Relacionamento definido por *é um* ou *é como* (ou *parece com*)
Herança múltipla	A herança direta de mais de uma classe
Herança simples	A herança direta de apenas uma classe
Hierárquia de heranças	Um conjunto de classes relacionadas por heranças
Instância	Um objeto que é um exemplo de uma classe específica
Instanciar	Criar objetos a partir da definição de classes
Interface	Uma coleção de uma ou mais assinaturas de operações que define um conjunto coeso de comportamentos
Mensagem	Um pedido por informação ou para realizar uma ação
Mensageria (ou caixa postal)	Um processo de colaboração entre objetos pelo envio de mensagens uns para os outros
Método	Um processo implementado por uma classe que realiza uma ação de valor (similar a uma função na programação estruturada)
Objeto	Uma pessoa, lugar, coisa, evento, conceito, tela ou relatório baseado numa definição de classe
Objeto persistente	Um objeto salvo numa memória permanente
Objeto transitório	Um objeto não salvo numa memória permanente
Ocultação de informação	A restrição ao acesso externo de atributos
Opcionalidade	O conceito de *Você precisa ter isto?*
Padrão de projeto	Uma solução reusável para um problema comum levando em conta forças relevantes
Persistência	O armazenamento de objetos em memória permanente, por exemplo, em arquivos, bancos de dados, etc.
Perspectivas	Os objetos podem ser vistos sob três perspectivas: Do conceito, da especificação e da implementação
Polimorfismo	A capacidade de diferentes objetos responderem a uma mesma mensagem de maneiras diferentes, permite que um objeto interaja com um outro sem conhecer o seu tipo exato
Propriedade	Em UML 2, um valor nomeado (com nome), por exemplo, atributos e associações, inclusive composição, denotando uma característica de um elemento (tal como, uma classe, ou um componente). Em C#, a combinação de um atributo com seus getter e setter
Refatoração	Modificar a implementação e/ou a definição de componentes sem afetar outros componentes de um sistema já em funcionamento
Sobreescrita	Redefinir atributos e/ou métodos em subclasses tais que eles sejam diferentes da definição na superclasse
Subclassse	Uma classe que herda de uma outra
Superclase	Uma classe da qual uma outra herda.

Fonte: Fortemente baseada em (AMBLER, 2004)

1.2 Conceitos de OO do ponto de vista estrutural

Provavelmente, você se assustou com a lista de conceitos da Tabela 1. Se você está estudando para um teste, supõe-se que a lista seja útil para você. A maioria de nós procura por uma maneira de aprender os conceitos OO facilmente. Antes de ver as explicações detalhadas, vamos descrever quatro conceitos básicos de OO rapidamente, numa linguagem que pode ser familiar para você, a terminologia estruturada.

Classe Uma classe é uma abstração de software de um objeto, efetivamente, um padrão a partir do qual os objetos são criados. Se você tiver experiência com bancos de dados, você pode pensar

numa classe como uma tabela, embora tenha mais em classes do que isto conforme você verá mais tarde. A definição de uma classe descreve o layout, incluindo tanto os dados, quanto as funcionalidades, dos objetos a serem criados a partir deles. Observe que eu disse, tanto os dados, quanto as funcionalidades. Diferente de uma tabela, que define apenas dados, uma classe define ambos, dados (atributos) e código (operações/métodos). Por enquanto, uma boa maneira de pensar a respeito de uma classe é que ela é a combinação de uma definição de tabela e a definição do código fonte que acessa os dados. Infelizmente, esta é uma visão muito orientada a dados de objetos que ignora a principal força dos objetos que é a capacidade de responder a pedidos de ações.

Objeto Um objeto é uma construção de software que espelha um conceito do mundo real, por exemplo, uma pessoa, lugar, coisa, evento, conceito, tela ou relatório. Objetos são tipicamente (mas nem sempre) nomes próprios, substantivos. Se uma classe pode ser pensada como uma tabela, um objeto pode ser pensada como a ocorrência de um registro. De novo, esta é uma visão muito orientada a dados. Por exemplo, um objeto estudante pode fazer muitas das coisas que um estudante *real* pode, por exemplo, fornecer seu nome, calcular quantos anos tem e requerer a inscrição numa disciplina. Numa aplicação estruturada, cada estudante seria representado como um registro na tabela de dados Estudante. Numa aplicação orientada a objetos, cada estudante seria representado por um objeto na memória. A principal diferença é que onde registros de estudantes só têm dados, os objetos estudantes têm ambos, dados (atributos) e funcionalidades (métodos). Do ponto de vista de projeto, é conveniente entender os objetos como entidades com responsabilidades, estas responsabilidades incluem as informações que o objeto detem e as ações que ele pode realizar.

Atributo Um atributo é equivalente a um elemento de dados num registro. Do ponto de vista de programação, também faz sentido pensar num atributo como uma variável local aplicável a apenas um objeto. Os objetos são responsáveis pelas suas próprias informações que devem ser guardadas nos atributos dos objetos. Existem informações que não são de responsabilidade dos objetos individualmente, mas de todos os objetos de um mesmo tipo/classe, nesse caso, temos atributos de classe. Quando cada objeto, chamado de instância de uma classe, tem seu próprio atributo, este atributo é chamado de atributo de instância.

Método Um método pode ser pensado tanto como uma função ou um procedimento. Métodos acessam e modificam os atributos de um objeto. Melhor ainda, métodos podem fazer toda uma série de coisas que nada têm a ver com os atributos. Alguns métodos retornam um valor (como as funções), enquanto outros provocam efeitos colaterais, como impressão ou armazenamento em memória permanente, ou mudam o valor de um atributo, dizemos que o método muda o estado do objeto, ou de outro objeto associado. Um método útil deve ou retornar um valor, ou ter um efeito colateral significativo. Em terminologia OO, dizemos que os objetos trocam mensagens entre eles. Uma mensagem de um objeto para um outro requer uma ação de responsabilidade deste último. A maneira de "enviar" a mensagem é o primeiro objeto chamar o método do segundo objeto. Este é o modelo da maioria das linguagens de programação para aplicações locais. Em sistemas distribuidos, é possivel que a chamada de um método efetivamente provoque o envio de uma mensagem de um objeto local para um objeto remoto.

Atributos e métodos de uma classe são chamados de membros de uma classe de forma geral. Algumas características se aplicam tanto a atributos, quanto a métodos, por isso o conceito de membro da classe é útil.

(FOWLER, 2004) descreve três perspectivas diferentes no processo de desenvolvimento OO que ajudam a determinar as classes:

Perspectiva Conceitual Esta perspectiva "representa os conceitos do domínio estudado ... um modelo conceitual deve ser obtido sem, ou com pouca, ligação com o software que vai implementá-lo ..." Ela responde à pergunta: "*Pelo que sou responsável?*"

Perspectiva da Especificação "Agora olhamos para o software, não para a implementação, mas para as interfaces." Ela responde à pergunta: "*Como sou usado?*"

Perspectiva da Implementação Estamos na visão do próprio código. "Esta é provavelmente a perspectiva mais usada, mas de muitas maneiras, a perspectiva da especificação é melhor." Ela responde à pergunta: "*Como cumpro minhas responsabilidades?*"

(SHALLOWAY; TROTT, 2005) afirma que a melhor maneira de projetar as classes é pela determinação das suas responsabilidades. O projeto de software antigamente usava basicamente a perspectiva de implementação que é mais próxima aos programadores, mas os projetos resultantes são pouco flexíveis e difíceis de serem mantidos. Em software, os requisitos dos projetos mudam com frequência, esta maneira de projetar não se adapta bem às mudanças nos requisitos. É melhor projetar pela perspectiva da especificação, pelas interfaces. Esta perspectiva resulta em projetos mais adaptáveis às mudanças de requisitos. Os padrões de projeto também usam esta maneira de especificar as soluções. (GAMMA et al., 1994) usa com frequência classes abstratas nos seus esquemas de soluções.

1.3 Os diagramas de UML 2

A *Unified Modeling Language*, UML, é uma linguagem visual (isto é, ela é uma notação gráfica de diagramas com semântica) usada para criar modelos de programas. Neste contexto, o termo *modelos de programas* significa representações em diagramas dos programas que mostram os relacionamentos entre os objetos no código. A tabela 2 ilustra o uso dos diagramas UML em diferentes momentos do desenvolvimento OO.

Tabela 2 – Os Diagramas de UML 2 e seus propósitos

Quando você está	**Diagramas UML a usar**
Na fase de análise	Diagramas de Caso de Uso, que envolvem entidades interagindo com o sistema (usuários e outros sistemas) e os pontos de função que você precisa implementar. Diagramas de Atividade, cujo foco é o fluxo de trabalho do domínio do problema (o verdadeiro espaço onde as pessoas e outros agentes estão trabalhando, a área de trabalho do programa) e não o fluxo da lógica do programa.
Na observação das interações dos objetos	Diagramas de Interação, que mostram como os objetos interagem entre eles. Por lidar com casos específicos e não com situações gerais, eles demonstram serem úteis tanto para verificar os requerimentos, quanto para verificar os projetos. O diagrama de interação mais popular é o Diagrama de Sequência.
Na fase de projeto	Diagramas de Classe, que detalham os relacionamentos entre as classes.
Na observação das mudanças de *comportamento* em função do estado atual do objeto	Diagramas de Máquina de Estado, que detalha os diferentes estados em que o objeto pode estar e suas transições.
Na fase de distribuição	Diagramas de Distribuição, que mostram como módulos diferentes serão distribuidos.

Fonte: (SHALLOWAY; TROTT, 2005)

Entender os 14 diagramas de UML 2 ilustrados na figura 1 e resumidos na tabela 3 é importante para a compreensão da análise e do desenvolvimento OO. Existem 3 classificações para os diagramas

UML[1]:

***Diagramas de Comportamento*:** Este é o tipo de diagrama que representa características comportamentais de um sistema ou processo de negócio. Isto inclui diagramas de atividade, máquina de estado e caso de uso, assim como os 4 diagramas de interação.

***Diagrams de Interação*:** Este é um subconjunto de diagramas comportamentais que enfatizam a interação entre objetos. Isto inclui os diagramas de comunicação, visão simplificada de interação, sequência e temporização.

***Diagramas Estruturais*:** Este tipo de diagrama representa os elementos estáticos de uma especificação que não mudam com o tempo. Isto inclui os diagramas de classe, estrutura composta, componente, distribuição (*deployment*), objetos, pacotes e perfil.

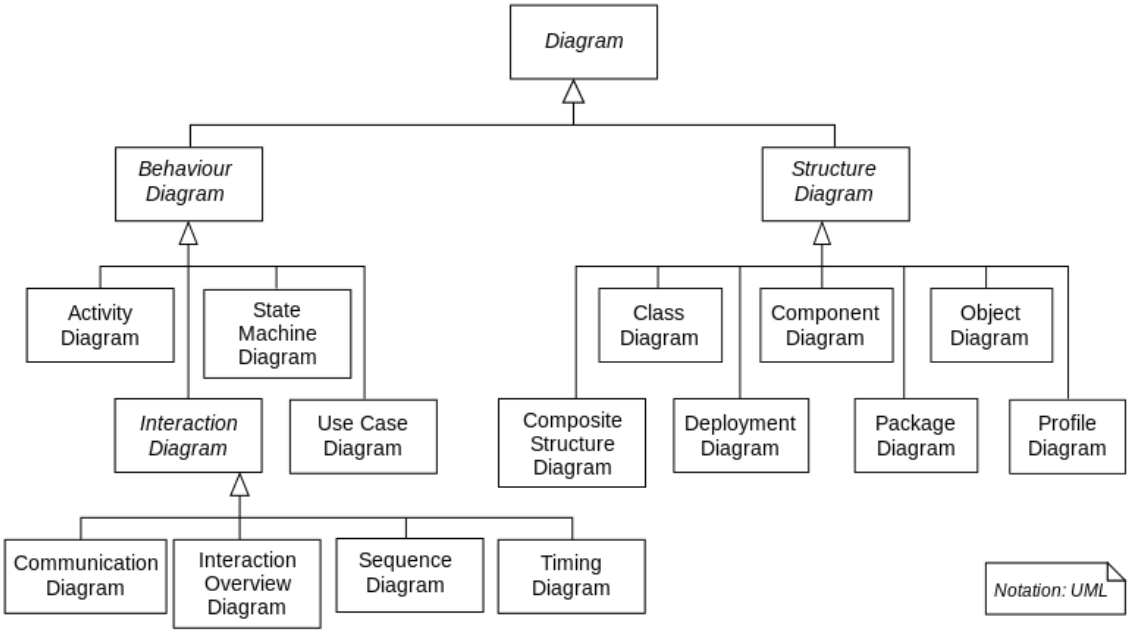

Figura 1 – Hierárquia dos diagramas UML

Fonte: (WIKIPEDIA, 2018b)

Tabela 3 – Os Diagramas de UML 2

Diagrama	**Descrição**
Diagrama de Atividade	Representação de alto nível de processos de negócios, incluindo fluxos de dados ou modelagem da lógica complexa dentro de um sistema
Diagrama de Caso de Uso	Mostra casos de uso, atores e seus relacionamentos
Diagrama de Classe	Mostra uma coleção de elementos de modelagem estáticos tais como classes e tipos, seus conteúdos e seus relacionamentos
Diagrama de Componente	Representa os componentes, inclusive seus interrelacionamentos e interfaces públicas, que compõem uma aplicação, sistema ou empresa

Continua na próxima página

[1] Os diagramas UML 2 deste trabalho foram feitos usando a ferramenta Dia, (LARSSON, 2012).

Tabela 3 – continuação da página anterior

Diagrama	Descrição
Diagrama de Comunicação	Mostra instâncias de classes, seus interrelacionamentos e o fluxo de mensagens entre elas e tipicamente foca na organização estrutural de objetos que enviam e recebem mensagens; eram chamados de diagramas de colaboração no UML 1
Diagrama de Distribuição	Mostra a arquitetura de execução dos sistemas, inclusive os nós, tanto os ambientes de execução de HW, quanto de SW, e o *middleware* que os conecta
Diagrama de Estrutura Composta	Representa a estrutura interna de um classificador (tal como uma classe, componente ou caso de uso), inclusive os pontos de interação do classificador com outras partes do sistema
Diagrama de Máquina de Estado	Descreve os estados em que um objeto ou uma interação podem estar, assim como as transições entre os estados. Anteriormente, era chamado de diagrama de estados, diagrama do mapa de estados ou diagrama de transição de estados
Diagrama de Objetos	Representa objetos e seus relacionamentos num instante determinado, tipicamente, um caso especial de um diagrama de classes ou de comunicação
Diagrama de Pacotes	Mostra como elementos de modelagem são organizados em pacotes, assim como as dependências entre os pacotes
Diagrama de Perfil	Opera no nível do metamodelo para mostrar estereótipos como classes com o estereótipo «stereotype» e perfis como pacotes com o estereótipo «profile». A relação de extensão (linha sólida com ponta de seta fechada e preenchida) indica qual elemento metamodelo um determinado estereótipo está estendendo. Este diagrama não existia na UML 1
Diagrama de Sequência	Modela a lógica sequencial, na verdade, a ordenação temporal das mensagens entre classificadores
Diagrama Simplificado de Interação	Uma variação do diagrama de atividade, que resume o fluxo de controle dentro de um sistema ou de um processo de negócios, onde cada nó/atividade no diagrama pode representar um outro diagrama de interação
Diagrama de Temporização	Representa a mudança de estado ou condição de uma instância, de um classificador ou de um papel (*role*) ao longo do tempo. Tipicamente, usado para mostrar a mudança de estado de um objeto no tempo ao responder a um evento externo.

Fonte: (AMBLER, 2004)

1.4 Objetos e Classes

O paradigma OO é baseado em construir sistemas a partir de itens chamados de *objetos*. Um objeto é uma pessoa, lugar, coisa, conceito, tela ou relatório. Uma classe generaliza/representa uma coleção de objetos similares e é efetivamente um padrão a partir do qual cria-se objetos. Num sistema universitário, Clara é um objeto estudante, ela está matriculada em diversos objetos disciplinas e ela trabalha para obter um objeto diploma de um objeto curso. Num sistema bancário, Clara é um objeto cliente. Ela tem um objeto conta para o qual ela pode preencher objetos cheques. Num sistema de controle de inventário, cada item do inventário é um objeto, cada entrega é um objeto e cada cliente é um objeto.

No mundo real, você tem objetos, logo, você precisa deles como um conceito para refletir precisamente o espaço do seu problema. Entretanto, no mundo real, objetos são frequentemente similares a outros objetos. Estudantes compartilham qualidades similares (eles fazem as mesmas

coisas, eles são descritos da mesma forma), disciplinas compartilham qualidades semelhantes, itens de inventórios, contas bancárias, ... compartilham qualidades semelhantes entre eles. Enquanto você pode modelar (e programar) todos os objetos individualmente, isto exige um trabalho enorme. É preferível definir o que é ser um estudante uma vez, definir disciplina uma vez, definir item de inventório uma vez, ... É por isso que você precisa do conceito de classe. Os objetos são exemplos/instâncias da classe.

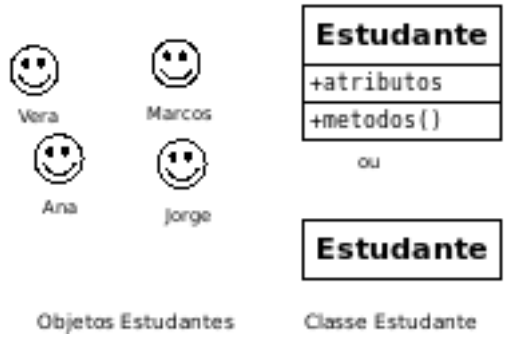

Figura 2 – Objetos *Estudante* e duas maneiras de modelar a classe *Estudante*

A figura 2 ilustra como temos objetos estudantes e como podemos modelar a classe *Estudante*. Ela também mostra a notação padrão para modelar uma classe usando UML. Classes são tipicamente modeladas por uma das duas maneiras: um retângulo com uma lista dos atributos e métodos da classe ou apenas um retângulo. Existem razões para modelar classes de uma maneira ou da outra. Por um lado, listar os atributos e métodos pode ser bastante útil, permite aos leitores do modelo entender o projeto numa única visualização. Por outro lado, listar os atributos e métodos pode poluir seus diagramas e dificultar a leitura. Neste texto, serão usadas ambas as técnicas, os métodos e atributos só serão listados onde forem apropriados.

Nome de classe é tipicamente um substantivo no singular. O nome de uma classe pode ter uma ou duas palavras e deve descrever precisamente a classe usando terminologia comum ao domínio do negócio onde o problema está inserido. Se você tem dificuldades para dar nome a uma classe, ou você precisa entendê-la melhor, ou podem ser várias classes que você combinou erroneamente. Você deve modelar classes com nomes tais como *Estudante*, *Professor* e *Disciplina* e não como *Estudantes*, *Pessoas que ensinam disciplinas* ou *COMP101*. Pense da seguinte forma, no mundo real, você diria "Sou um estudante" e não "Sou um estudantes".

Classes também podem representar conceitos para os quais não existe um substantivo, como o processo de retirada de um livro de uma biblioteca.

Quando software OO está rodando, objetos são instanciados (criados/definidos) a partir de classes. Dizemos que um objeto é uma instância de uma classe e instanciamos estes objetos das classes.

Uma das maiores dificuldades no desenvolvimento de software OO é a escolha das classes que devem ser implementadas para resolver um problema. Um método mais antigo diz que devemos identificar os nomes (substantivos) do domínio do problema e as ações que estes nomes executam para resolver o problema. Uma maneira mais atualizada de encontrar as classes é encontrar as responsabilidades para a solução do problema. No início da programação OO era comum encontrar classes que faziam muitas coisas, porque representavam objetos complexos, hoje, prefere-se classes pequenas que são responsáveis por um serviço, ou poucos serviços.

1.5 Atributos e Operações/Métodos

Classes têm responsabilidades, as coisas que elas sabem e fazem. Atributos são as coisas que as classes sabem, métodos são as coisas que elas fazem. O paradigma da orientação a objetos é baseado nos conceitos de que sistemas devem ser construídos a partir de objetos e estes objetos têm tanto dados, quanto funcionalidade. Atributos definem os dados, enquanto os métodos definem a funcionalidade.

Quando você define uma classe, você deve definir os atributos que ela tem, assim como, os seus métodos. Em UML 2, um atributo é um tipo de propriedade, como são relacionamentos tais como associação e composição. A definição de um atributo é direta. Você define seu nome, talvez seu tipo (se ele é um número, um string, uma data cronológica, etc.). Linguagens fracamente tipadas como Smalltalk (ou Python) permitem que você use atributos da maneira que você quiser e, portanto, não exigem que você defina o tipo deles. Linguagens fortemente tipadas como Java e C++, entretanto, insistem que você defina o tipo de um atributo antes de você poder usá-lo. Você pode ainda escolher se você quer indicar alguma regra de negócio ou restrição aplicável ao atributo, tais como valores válidos que o atributo pode ter, vide figura 3.

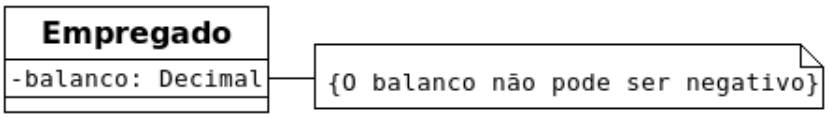

Figura 3 – Exemplo de classe com atributo limitado

A definição de um método é mais simples: você define a lógica para ele, do mesmo modo que você codifica uma função ou um procedimento. O termo *método* é usado diferentemente de como a UML usa – o que está sendo mostrado, na verdade, são as operações da classe e os métodos são as implementações dessas operações. Esta visão de método é consequência da influência do Smalltalk, que usa o termo método dessa maneira. Não se preocupe com isto, a terminologia de objetos é notoriamente inexata. Mais para a frente detalharemos melhor a especificação de métodos. Ao vermos o conceito de colaboração, você verá que colaboração é feita exclusivamente através de métodos. Por ora, retenha que métodos fazem estas duas coisas: eles retornam um valor e/ou eles fazem alguma coisa, isto é, eles provocam um efeito de borda. Em computação, efeito de borda (*side effect*) significa que uma modificação é realizada por este método na memória e esta modificação é visível por outros métodos, isto é, fora deste método.

Na figura 4, você vê dois tipos diferentes de objetos: um *estudante* e uma *disciplina*. Ambos objetos, sabem e fazem algumas coisas e você deseja registrar isto nos seus modelos, como você pode observar na figura 5. Estou usando a notação de classe com 3 seções neste caso: a seção do topo para o nome, a seção do meio para a lista dos atributos e a seção de baixo lista os métodos.

A figura 5 expõe 2 tipos diferentes de atributos: atributos de instância que são aplicáveis a um único objeto e atributos estáticos que são aplicáveis a todas as instâncias de uma classe. Atributos estáticos têm o nome sublinhado, atributos de instância não são sublinhados. Por exemplo, *nome* é um atributo de instância na classe Estudante. Cada estudante individual tem um nome, por exemplo, um estudante tem como nome "José da Silva", enquanto uma outra estudante pode ter o nome "Maria de Oliveira". Pode até acontecer de dois estudantes diferentes terem o mesmo nome, embora sejam duas pessoas diferentes.

Por outro lado, *proximoNumeroEstudante* é um atributo estático (também chamado de atributo da classe) que é aplicável à classe Estudante, não especificamente a uma instância determinada. Este

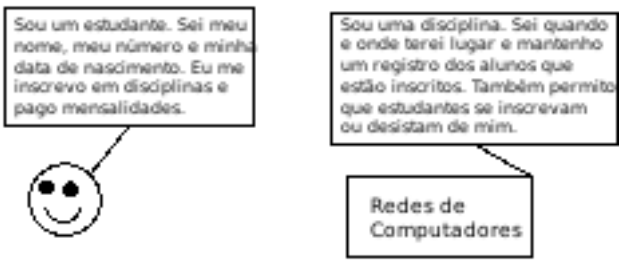

Figura 4 – Objetos no mundo real

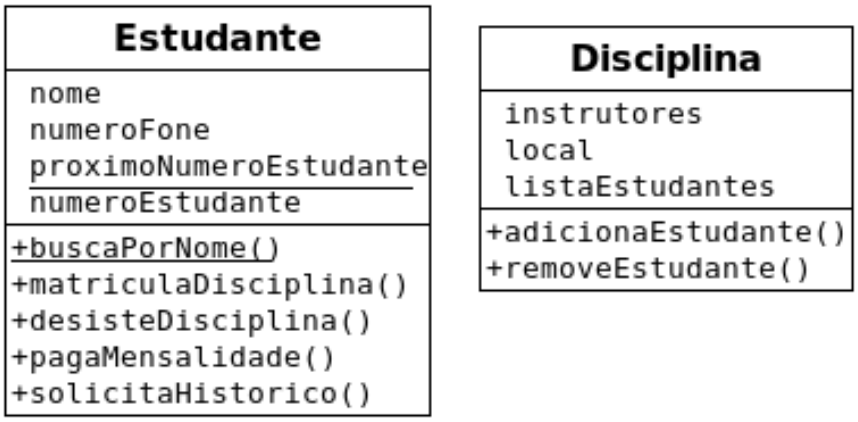

Figura 5 – Classes *Estudante* e *Disciplina*

atributo é usado para guardar qual é o valor que será usado como número de estudante pelo próximo estudante a ser cadastrado. Quando um novo estudante ingressa na escola, seu número de estudante é dado pelo valor atual de *proximoNumeroEstudante*, que será incrementado para garantir que o número de todos os estudantes é único.

De modo similar, existe o conceito de método de instância e método estático (ou de classe): Métodos de instância operam sobre uma única instância, enquanto métodos estáticos operam potencialmente sobre *todas* as instâncias de uma classe[2]. Na figura 5, observe que na classe *Estudante* os métodos *matriculaDisciplina()* e *desisteDisciplina()* são métodos de instância, coisas que um estudante individual faria. Ela também tem um método estático *buscaPorNome()*, que fornece o comportamento de procurar os estudantes cujo nome atendem um critério de busca, é um método que opera sobre todas as instâncias da classe.

1.6 Abstração, Encapsulamento e Ocultação de Informação

Em vez de dizer que determinamos o que uma classe *sabe* e *faz*, dizemos que *abstraímos* a classe. Em vez de dizermos que projetamos como a classe realizará essas coisas, dizemos que *encapsulamos* elas. Em vez de dizer que projetamos a classe restringindo o acesso aos seus atributos, dizemos que *escondemos*, ocultamos, a informação.

1.6.1 Abstração

O mundo é um lugar complicado. Para lidar com esta complexidade, formamos abstrações das coisas dele. Por exemplo, considere a abstração de uma pessoa. Do ponto de vista da universidade, ela precisa saber o nome, endereço, número de telefone/celular, RG, CPF e formação acadêmica da pessoa. Do ponto de vista da polícia, eles precisam saber o nome, endereço, número de telefone, peso, altura, cor do cabelo, cor dos olhos, etc. Ainda é a mesma pessoa, apenas são abstrações diferentes, dependendo da aplicação de interesse.

Abstração é um problema da análise que lida com o que a classe sabe e faz. Sua abstração deve incluir as responsabilidades, os atributos e os métodos de interesse da aplicação em vista – ignore o resto. Esta é a razão para a abstração de um estudante incluir o nome e o endereço da pessoa, mas provavelmente não o seu peso e altura. Sistemas OO abstraem apenas o que precisam para resolver o problema em vista. As pessoas costumam dizer que abstração é o ato de pintar uma caixa em torno de algo: você está identificando o que ele faz e não faz. Algumas pessoas também dizem que abstração é o ato de definir a interface de algo. De qualquer modo, você está definindo o que a classe sabe e faz.

[2] Um método estático não precisa de uma instância para operar, então ele pode ser executado mesmo se nenhuma instância ainda foi criada. Um cuidado que novatos devem ter, é que na implementação de um método de classe não é possível acessar atributos de instância.

1.6.2 Encapsulamento

Embora o ato de abstrair nos diga que precisamos armazenar o nome e endereço de um estudante, assim como ser capaz de matricular estudantes nas disciplinas, ele não nos diz como vamos fazer isto. Encapsulamento lida com o problema de como você pretende modularizar as caraterísticas de um sistema. No mundo orientado a objetos, você modulariza sistemas em classes, que, por sua vez, modularizam em métodos e atributos. Dizemos que encapsulamos comportamento numa classe ou encapsulamos funcionalidade num método. Encapsulamento é uma questão de projeto que lida com a funcionalidade está compartimentalizada dentro de um sistema. Você não deve precisar saber como algo está implementado para poder usá-lo. A consequência do encapsulamento é que você pode construir qualquer coisa da maneira que você quiser e mais tarde você pode mudar a implementação e não afetar outros componentes dentro do sistema (desde que a interface para aquele componente não mude).

As pessoas dizem que o encapsulamento é como pintar a caixa de preto: você está definindo como algo pode ser usado, mas você não está dizendo ao resto do mundo como você vai fazê-lo. Em outras palavras, você está escondendo os detalhes da implementação de um item dos usuários deste item. Por exemplo, pense num banco. Como ele registra as informações de uma conta? Num *mainframe*, num *minicomputador*, num *PC* ou na *nuvem*? Qual banco de dados ele usa? Qual sistema operacional? Não importa, porque ele encapsulou a maneira como realiza as operações na conta. Você apenas vai a um caixa e inicia a transação que você deseja. Pela ocultação dos detalhes de como são implementadas as contas, seu banco pode mudar a implementação a qualquer momento sem afetar a maneira como os serviços lhe são oferecidos. De acordo com (SHALLOWAY; TROTT, 2005), o encapsulamento deve ser pensado como qualquer tipo de ocultação, que veremos a seguir.

1.6.3 Ocultação de Informação

Para simplificar a manutenção de suas aplicações, você quer restringir o acesso aos seus atributos de dados e alguns métodos. A ideia básica é esta: se uma classe deseja informações de uma outra, ela deve requisitá-la e não simplesmente pegá-la. Quando você reflete a respeito, isto é exatamente o que acontece no mundo real. Se você quer saber o nome de alguém, o que você faria? Você perguntaria para a pessoa ou roubaria sua carteira e olharia no RG dela? Ao restringir o acesso aos atributos, você previne que outros programadores escrevam código fortemente acoplado. Quando o código é fortemente acoplado, uma mudança numa parte do código força a modificação de outra, e outra, e assim por diante.

(SHALLOWAY; TROTT, 2005) diz que, em geral, se oculta dados, mas que podemos ocultar também:

- Implementações;
- Classes Derivadas;
- Detalhes de projeto; e
- Regras de Instanciação.

1.7 Herança

Similaridades sempre existem entre classes diferentes. Duas ou mais classes frequentemente compartilham os mesmos atributos e/ou mesmos métodos. Como você não quer ter de reescrever o mesmo código repetidas vezes, você deseja um mecanismo que tire vantagem destas semelhanças. Herança é este mecanismo. Herança modela os relacionamentos “é um” e “é como”, o que permite que você reuse dados e códigos existentes facilmente.

Por exemplo, estudantes têm nomes, endereços e números de telefone e eles dirigem veículos. Ao mesmo tempo, professores também têm nomes, endereços e números de telefones e dirigem veículos. Sem dúvida, você pode desenvolver as classes para estudantes e professores e fazê-las executarem. De

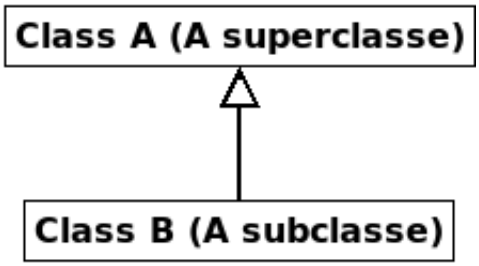

Figura 6 – A notação de modelagem UML para a herança

fato, você poderia até desenvolver a classe *Estudante* primeiro e, quando ela estiver rodando, fazer uma cópia dela, chamá-la de *Professor* e fazer as modificações necessárias. Enquanto, esta é uma maneira muito direta de resolver o problema, ela não é perfeita. E se houver um erro no código original do estudante? Agora, você tem de corrigir o erro em dois lugares, o dobro do trabalho. O que acontece se você precisar mudar a maneira de lidar com o nome, digamos que no lugar de ter no máximo 30 caracteres, precisa de 40 caracteres? Agora, você tem de fazer a mesma modificação em dois lugares, o que é chato, besta e custoso (em termos de tempo). Não seria bom se só tivesse uma cópia do código para desenvolver e fazer manutenção?

Isto é exatamente o que a herança se propõe a fazer. Com herança, você define uma nova classe que encapsula as similaridades entre os alunos e professores. Esta nova classe teria os atributos *nome*, *endereço* e *numeroTelefone* e o método *dirigeVeiculo*. Como você precisa dar nome a todas as suas classes, você precisa se perguntar o que esta coleção de dados e funcionalidade descrevem. Neste caso, provavelmente o nome *Pessoa* cabe bem. Este tipo de modificação, partindo da classe *Estudante*, encontrando as similaridades, ou comunalidades, com a classe *Professor*, para criar a classe *Pessoa*, é um tipo de *refatoração* bastante comum no desenvolvimento de software.

Quando você tiver a classe *Pessoa* definida, você fará com que *Estudante* e *Professor* herdem dela. Você pode dizer que *Pessoa* é a superclasse de *Estudante* e *Professor* e que *Estudante* e *Professor* são subclasses de *Pessoa*. Tudo que uma superclasse sabe ou faz, a subclasse sabe ou faz sem que seja necessário escrever mais código. Na verdade, para este exemplo, você teria de escrever duas linhas de código, uma dizendo que *Estudante* é subclasse de *Pessoa* e outra dizendo que *Professor* é uma subclasse de *Pessoa*. Porque *Pessoa* tem um nome, endereço e número de telefone, ambos, *Estudante* e *Professor*, também, têm estes atributos. Porque *Pessoa* tem a capacidade de dirigir um veículo, as classes *Estudante* e *Professor* também têm. As subclasses também são ditas derivadas da superclasse. Também dizemos que a subclasse é uma especialização da superclasse. Uma subclasse tem todos os atributos da superclasse. Ela também herda o comportamento, isto é, os métodos da superclasse. A subclasse, porém, pode modificar o comportamento, ela pode sobrescrever os métodos herdados, para que eles tenham um comportamento mais adequado com as características que são próprias à subclasse.

1.7.1 Modelagem de Herança

A figura 6 ilustra a notação de modelagem UML para a herança, uma linha com uma seta com a ponta oca saindo da subclasse e apontando para a superclasse. A maneira de ler o diagrama é "B herda de A". Em outras palavras, B é uma subclasse direta de A e A é a superclasse direta de B.

A figura 7 mostra como você modelaria a hierarquia de herança da classe *Pessoa*. Em geral, chamamos de hierarquia de classes de *Pessoa*, simplesmente. Observe que o nome da classe *Pessoa* está em itálico, isto indica que ela é *abstrata*, enquanto as classes *Professor* e *Estudante* são *concretas*.

1.7.2 Dicas e Técnicas para Herança

As dicas e técnicas a seguir devem lhe ajudar a aplicar hereditariedade de uma forma mais efetiva:

Figura 7 – Modelagem do conceito de que *Professor* e *Estudante* herdam de *Pessoa*

Procure semelhanças: Sempre que houver semelhança em duas ou mais classes, seja semelhança nos atributos, seja nos métodos, então, você provavelmente tem uma oportunidade para aplicar a herança.

Procure por classes existentes: Quando você identifica uma nova classe pode ser que você já tenha uma classe semelhante a ela. Algumas vezes, você pode herdar diretamente de uma classe existente e apenas acrescentar código para as diferenças entre as duas. Por exemplo, suponha que seu sistema de informação universitária também precisa suportar administradores da universidade. A classe *Pessoa* já tem muitas características que uma classe *Administrador* precisa, logo você deve considerar se *Administrador* deve herdar de *Pessoa*.

Siga a regra da sentença: Uma das frases a seguir deve ter sentido: "Uma subclasse é um tipo da superclasse" ou "Uma subclasse é como uma superclasse". Por exemplo, faz sentido dizer que um estudante é um tipo de pessoa e um dragão é como um pássaro. Não faz sentido dizer que um estudante é um tipo de veículo, ou é como um veículo, então, a classe *Estudante* provavelmente não deve herdar de *Veiculo*. Se nenhuma das sentenças faz sentido, então você deve ter encontrado um relacionamento de composição ou de associação.

Evite herança de implementação: Desenvolvedores novatos na orientação a objetos têm uma tendência a errar na aplicação da herança, em geral, num esforço para reusar tanto quanto possível (uma boa motivação). Herança é o conceito mais excitante do repertório de objetos deles, então eles o usam tanto quanto podem. O problema é com o que se chama de herança de implementação ou de conveniência: A aplicação da herança quando a regra da sentença não se aplica e a única justificativa é a de que a subclasse precisa de uma ou mais características da superclasse, a aplicação da herança é mais conveniente do que *refatorar* suas classes. Uma boa regra é reconsiderar a aplicação da regra de herança "é como", porque esta é uma justificativa mais fraca.

Herda tudo: A subclasse deve herdar tudo da superclasse, este conceito é chamado de *herança pura*. Se não, o código torna-se mais difícil de entender e de fazer a manutenção. Por exemplo, digamos que a classe B herda de A. Para entender B, você precisa entender o que é A e as características que são acrescentadas em B. Se você começar a remover funcionalidade, você precisa entender também o que B **não é**. Isto é muito trabalho e logo a manutenção desse sistema se torna um pesadelo.

1.7.3 Herança Simples e Múltipla

Quando uma classe herda de apenas uma outra classe, dizemos que temos uma *herança simples*. Quando uma classe herda de duas ou mais classes, temos *herança múltipla*. Lembre-se: A subclasse herda todos os atributos e métodos de suas superclasses.

Nem todas as linguagens dão suporte à herança múltipla. C++ é uma das poucas que dá, linguagens como Java, Smalltalk, C# e Ruby não dão. O ponto é que se sua linguagem de programação não oferece herança múltipla, não use herança múltipla na modelagem. Herança simples resulta

em hierarquia de herança que é sempre uma árvore. Herança múltipla resulta em grafos e algumas situações com nomes iguais vindo de superclasses diferentes são muitos complexas. O livro (ELLIS; STROUSTRUP, 1990) comenta sobre algumas das dificuldades enfrentadas para a implementação da herança múltipla no C++.

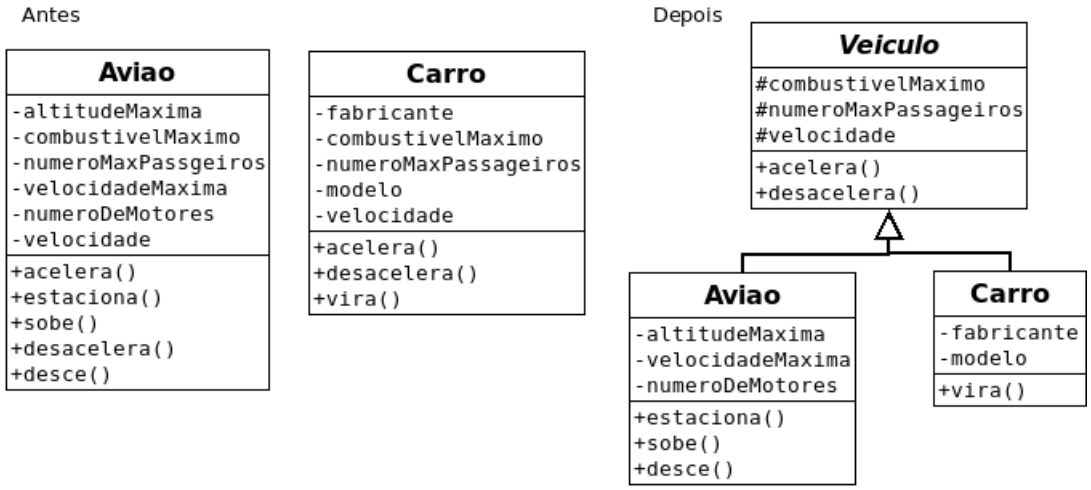

Figura 8 – Criação da hierárquia de classes de veículo

Na figura 8, vemos várias semelhanças entre aviões e carros. Ambos têm um número de passageiros, um nível máximo de combustível e ambos podem aumentar ou diminuir a velocidade. Para se aproveitar dessas semelhanças, você pode criar uma nova classe chamada de *Veiculo* e fazer *Aviao* e *Carro* herdar dela. Dizemos que as classes *Veiculo*, *Aviao* e *Carro* formam uma hierarquia de heranças, também chamada de hierarquia de classes. A classe no topo de uma hierarquia de classes (neste caso *Veiculo*) é chamada de raiz ou *classe raiz*. Em Smalltalk e Java, a classe *Object* é a classe raiz de todas as outras classes, direta ou indiretamente.

Observe que existe o método *vira()* em *Carro* e o método *inclina()* em *Aviao*. Virar e inclinar fazem a mesma coisa. Você poderia definir um método *vira()* em *Veiculo* e fazer *Aviao* e *Carro* herdá-lo. Isto implicaria que você teria de requerer que usuários de aviões, isto é, os pilotos, teriam de mudar a terminologia que eles costumam usar. Realisticamente, isto não funcionaria. A melhor solução é manter *vira()* em *Carro* e *inclina()* em *Aviao*.

No Antes: da figura 9, você quer criar uma nova classe *Dragao*. Você já tem as classes *Passaro* e *Lagarto*. Um dragão *é como* um pássaro, pois ambos voam. Um dragão também *é como* um lagarto, ambos têm garras e escamas. Porque dragões têm características de pássaros e lagartos, no Depois: da figura 9 temos a classe *Dragao* herdando tanto de *Passaro*, quanto de *Lagarto*. Este é um exemplo de um relacionamento "é como" (ou "parece com"): um dragão é como um pássaro e, também, é como um lagarto.

É importante perceber que a regra de sentença "é como" não é perfeita. Por exemplo, faz sentido dizer que uma sequóia é como um arranha-céu já que ambos são altos, têm raízes no chão e balançam ao vento. Entretanto, eles não devem compartilhar uma relação de herança. Também faz sentido dizer que um faixa preta é um artista de artes marciais, mas não faz sentido fazer uma classe *Roupa* herda da classe *Pessoa*.

Observe que o método *come()* foi listado na classe *Dragao*. Embora todos os três tipos de criaturas comam, eles comem cada um de maneira e coisas diferentes. Pássaros comem sementes, lagartos comem insetos e dragões comem cavaleiros em armaduras reluzentes. Porque a maneira de comer dos dragões é diferente da do pássaro e do lagarto, é necessário *redefinir* (sobrescrever) a definição do método *come()*. A ideia geral é que uma subclasse precisa sobrescrever um atributo ou um método sempre que ela usa aquele dado ou executa aquele método de uma maneira diferente da sua superclasse[3].

[3] A programação de jogos é um dos poucos domínios em que herança múltipla faz muito sentido.

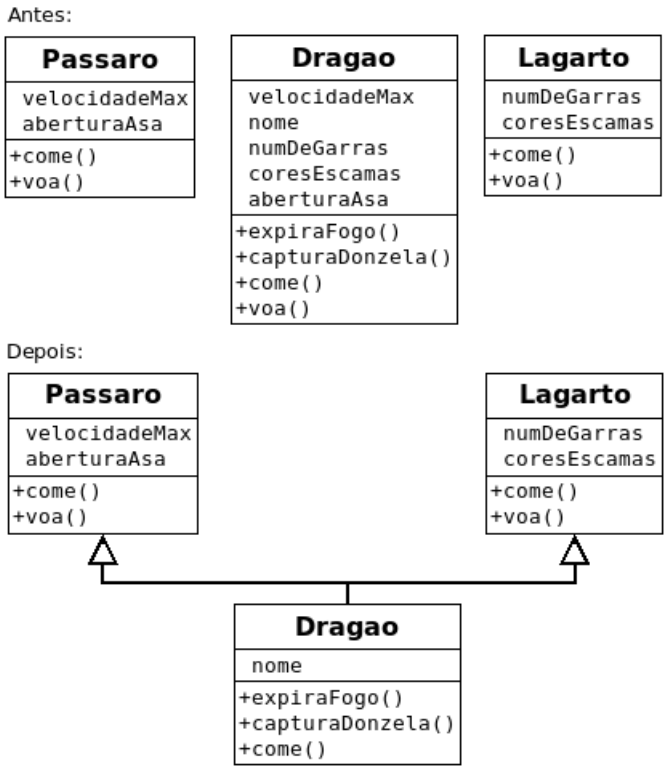

Figura 9 – Um exemplo de herança múltipla

1.7.4 Classes Abstratas e Concretas

Na figura 8, a classe *Veiculo* está marcada como abstrata (o nome está em itálico, em versões mais antigas de UML, você poderia usar a restrição *{abstract}*), enquanto *Aviao* e *Carro* não são. Dizemos que *Veiculo* é uma classe abstrata, enquanto *Aviao* e *Carro* são classes concretas. A diferença principal entre classes abstratas e concretas é que objetos podem ser instanciados (criados) de classes concretas, mas não de classes abstratas. Por exemplo, no domínio do seu problema, você tem aviões e carros, mas não tem nada que é apenas um veículo (se algo não é um avião ou um carro, você não se interessa por ele). Isto significa que seu software instanciará objetos aviões e carros, mas não objetos veículos. Classes abstratas são modeladas quando você precisa criar uma classe que implementa características comuns a duas ou mais classes. Observe que ao usar padrões de projeto, muitos deles definem interfaces que são classes abstratas. No modelamento é normal projetar classes pelas suas responsabilidades e essas classes não serem instanciáveis até que se defina uma forma de implementação concreta para estas responsabilidades.

Não é necessário se preocupar com se uma classe é abstrata quando estamos modelando, assumimos que faremos a coisa certa quando vier a hora de implementar a classe, o que numa equipe de projeto ágil pode ser minutos depois de modelá-la. Usar itálico para indicar que uma classe é abstrata é uma escolha infeliz do pessoal da OMG. Como (FOWLER, 2004) disse, quando você está modelando num quadro negro ou num papel (lembre-se que a prática de modelagem ágil diz: *use a ferramenta mais simples*), você precisa usar a notação antiga, *{abstract}*, ou *«abstract»*. Se você realmente pretende que uma classe seja abstrata, a maioria das linguagens OO permite que se declare que uma classe é abstrata e ela nunca poderá ser instanciada.

Enquanto (AMBLER, 2004) não se preocupa com o fato da classe ser abstrata na modelagem, (SHALLOWAY; TROTT, 2005) dá uma forte enfase no uso de classes abstratas para representa a ideia de comunalidade e a definição da interface de programação (API - Application Programming Interface) entre os componentes de software. Na prática, devemos observar as classes e na medida que o desenvolvimento for avançando, com o surgimento da necessidade de mais classes, devemos considerar se classes diferentes apresentam responsabilidades semelhantes e se não faz sentido unificá-las em uma

mesma classe abstrata. Devemos, entretanto, evitar os excessos que levam à herança de implementação. O uso de padrões de projeto pode auxiliar a determinar quais são as classes abstratas interessantes na solução de um problema. As alterações necessárias para usar a comunalidade de diversas classes e forma uma interface unificada por uma classe abstrata é uma forma de refatoramento que permite reduzir, em geral, acoplamentos.

1.8 Persistência

Persistência foca na questão de como fazer com que os objetos estejam disponíveis para uso futuro pelo software. Em outras palavras, como salvar objetos na memória permanente. Para fazer um objeto ser permanente, você deve salvar os valores de seus atributos na memória permanente (tais como num banco de dados ou num arquivo) e também qualquer informação necessária para manter seus relacionamentos (agregação, herança e associação) nos quais está envolvido. Em outras palavras, você precisa salvar as propriedades apropriadas na memória permanente. Além de salvar objetos, persistência também se preocupa em recuperá-los e apagá-los.

Do ponto de vista de desenvolvimento, existem dois tipos de objetos: **objetos persistentes** que continuam por aí e **objetos transitórios** que se vão. Por exemplo, um *Cliente* é uma classe persistente. Você quer salvar objetos clientes em algum tipo de memória permanente de modo que você possa trabalhar com eles de novo no futuro. Uma tela de edição de um cliente, entretanto, é um objeto transitório. Sua aplicação cria uma tela de edição de cliente, mostra-a e livra-se dela depois que o usuário terminou de editar os dados do cliente.

As seguintes dicas e técnicas devem ajudar a entender e aplicar o conceito de persistência melhor:

1. **Classes do negócio/ do domínio do problema são geralmente persistentes.** Você naturalmente precisa manter um registro permanente (ou persistente) de instâncias de classes do mundo real como *Estudante*, *Professor* e *Disciplina*.

2. **Classes da interface usuário são geralmente transitórias.** Classes da interface de usuário (telas e relatórios) são, em geral, transitórios. Telas são criadas e exibidas quando necessário e, então, quando não estão mais em uso, elas são destruídas (removidas da memória). Objetos relatórios são criados, eles adquirem os dados de que precisam, manipulam os dados e imprimem os dados. Depois disto, o objeto relatório é destruído. Note que, às vezes, você pode precisar manter um registro (*log*) da impressão do relatório e de quem mandou fazer a impressão, o registro (*log*) neste caso deve ser permanente.

3. **Você precisa armazenar tanto os atributos quanto as associações.** Quando um objeto é escrito no disco, você obviamente precisa armazenar os valores dos seus atributos. Entretanto, você também deve armazenar informações sobre seus relacionamentos/associações com os quais o objeto está envolvido. Por exemplo, Bia está cursando Bio-medicina e Enfermagem, então você precisa se assegurar que quando você for armazenar o objeto Bia no disco, o software vai registrar que ela está matriculada nestas duas disciplinas.

1.9 Relacionamentos

No mundo real, os objetos têm relacionamentos uns com os outros. Os relacionamentos entre os objetos são importantes porque nos ajudam a definir como os objetos interagem uns com os outros. Por exemplo, estudantes *se matriculam* em disciplinas, professores *ensinam* disciplinas, criminosos *roubam* bancos, políticos *beijam* bebes e capitães *comandam* espaçonaves. *Se matriculam*, *ensinam*, *roubam*, *beijam* e *comandam* são todos verbos que definem associações entre objetos. Você quer identificar e potencialmente documentar estes relacionamentos; assim, você pode ganhar um melhor entendimento de como os objetos interagem uns com os outros. A figura 10 resume as notações para relacionamentos entre duas classes.

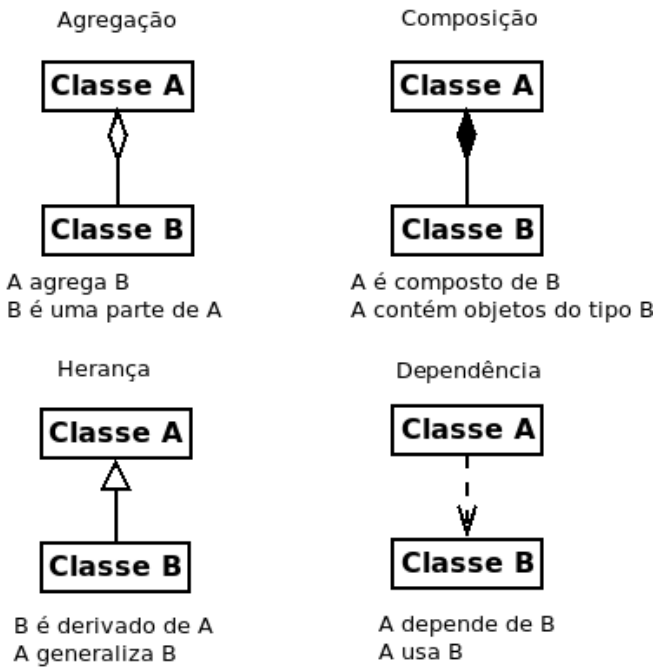

Figura 10 – Resumo dos diagramas de relacionamento.

Fonte: (AHMED; UMRYSH, 2002)

Você não precisa apenas identificar os relacionamentos entre as classes, você deve também descrever o relacionamento. Por exemplo, não é suficiente saber que os estudantes se matriculam em disciplinas. Quantas disciplinas os estudantes podem cursar? Nenhuma, uma ou várias? Além disso, relacionamentos são vias com circulação nos dois sentidos: Não apenas os estudantes se matriculam em disciplinas, mas também disciplinas têm estudantes matriculados. Isto leva a questões como quantos estudantes podem ser matriculados numa dada disciplina e é possível ter uma disciplina sem nenhum estudante? Isto implica em que você deve também identificar a cardinalidade e a opcionalidade de um relacionamento. Cardinalidade representa o conceito "quantos", opcionalidade representa o conceito "se você deve ter algo". É importante observar que a UML escolheu combinar os conceitos de opcionalidade e cardinalidade num único conceito de *multiplicidade*.

Considere Opcionalidade e Cardinalidade Separadamente
A experiência mostra que é melhor considerar separadamente estes dois conceitos: Para cada direção de uma associação é importante se a associação deve existir (opcionalidade) e quantos podem possivelmente existir (cardinalidade)? Por exemplo, considere a associação "ensina" entre professores e disciplinas, devemos perguntar:

- Um professor deve ensinar uma disciplina, ou pode haver professor que não ensina nenhuma disciplina?
- Quantas disciplinas um professor pode ensinar no máximo?
- Toda disciplina deve ser ensinada por um professor, ou é possível que alguém que não é professor a ensine?
- Mais de um professor pode ensinar numa disciplina? Quantos?
- Existem disciplinas que exigem que mais de um professor as ensinem?

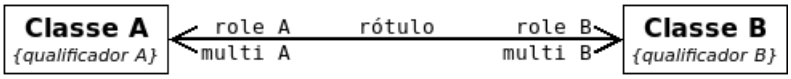

Figura 11 – Notação simplificada da modelagem de associações em diagramas de classes em UML

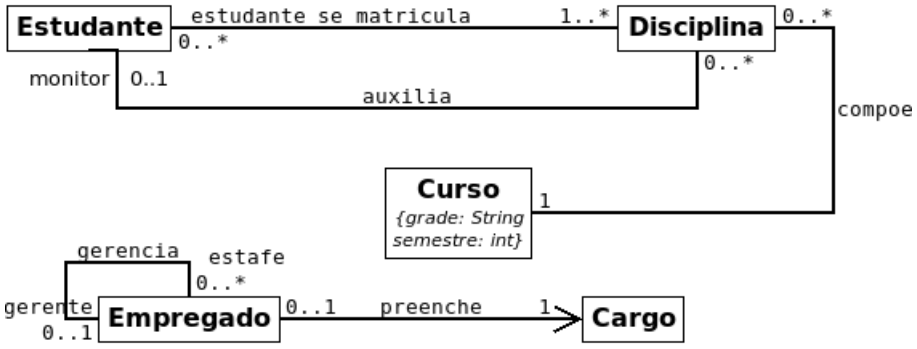

Figura 12 – Diversos exemplos de associação

1.9.1 Associações

Uma associação é um relacionamento persistente entre duas ou mais classes ou objetos. Quando você modela associações em diagramas de classes de UML, você as mostra como uma linha fina conectando duas classes, como você pode ver na figura 11. Associações podem facilmente ficar bastante complexas; Consequentemente, você pode exibir várias coisas a respeito deles nos seus diagramas. A figura 12 mostra os itens comuns a modelar numa associação:

Direcionalidade: Setas com ponta vazada indicam a direcionalidade da associação. Quando existe uma seta, a associação é unidirecional: Ela só pode ser percorrida numa direção (na direção da seta). Quando não tem seta ou tem dupla seta, a associação é bidirecional: Ela pode ser percorrida em ambos sentidos. Embora você deva indicar com as duas setas, é prática comum deixar de colocar as setas nas associações bi-direcionais. Exemplos das duas notações para associações bidirecionais estão na figura 12. Neste texto, adotaremos a prática da modelagem ágil – exiba os modelos de maneira simples – nas associações bidirecionais usaremos linhas sem setas. Recomenda-se a escolha de uma convenção e mantê-la.

Rótulo: O rótulo que é opcional, é tipicamente composto por uma ou duas palavras descrevendo a associação. A leitura do nome de uma classe, do rótulo e do nome da outra classe deve produzir uma frase que descreva o relacionamento, por exemplo, *Professor ensina Disciplina*. Evite rótulos genéricos como *tem* ou *comunica com*, tanto quanto possível.

Multiplicidade: A multiplicidade da associação é rotulada em ambas pontas da linha, uma multiplicidade para cada direção. A tabela 4 mostra os principais indicadores de multiplicidade que podem ser usados.

Papel: O papel (*role*), o contexto que cada objeto assume na associação, pode ser indicado na ponta da associação.

Qualificador: Um qualificador é um valor que seleciona um objeto do conjunto de objetos relacionados através de uma associação. Por exemplo, na figura 12, a associação *compõe* de *Disciplina* para *Curso* é qualificada pela combinação dos atributos *grade* e *semestre* (potencialmente implementada pela *Disciplina*). Qualificadores são opcionais e na prática raramente modelados[4].

[4] Tanto que a ferramenta Dia usada para criar os nossos diagramas não possui a opção de incluí-los, a notação usada no diagrama foi a de uma *restrição*.

Tabela 4 – Indicadores de Multiplicidade do UML

Indicador	**Significado**
0..1	Zero ou um
1	Apenas um
0..*	Zero ou mais
1..*	Um ou mais
n	Apenas n (onde $n > 1$)
0..n	Zero a n (onde $n > 1$)
1..n	Um a n (onde $n > 1$)

Observe o diagrama de classes da figura 12, que mostra diversas classes e as associações entre elas. Eis como você leria as associações:

- Um estudante se matricula em uma ou mais disciplinas;
- Uma disciplina tem 0 ou mais estudantes matriculados;
- Um estudante, como monitor, pode auxiliar em zero ou mais disciplinas;
- Uma disciplina pode ter 0 ou um estudante que age como monitor;
- Uma disciplina compõe um curso;
- Um curso é composto por zero ou mais disciplinas;
- Um empregado possui um cargo;
- Um cargo pode ser preenchido por um empregado (alguns cargos podem não está preenchidos);
- Um empregado pode ser gerenciado por um um outro empregado, o gerente (o presidente da companhia é o único sem um gerente); e
- Um empregado gerencia zero ou mais empregados.

Diversos pontos importantes estão mostradas na figura 12. Primeiro, observe que é possível ter mais de uma associação entre duas classes: As classes *Estudante* e *Disciplina* têm as associações *se_matricula* e *auxilia* entre elas. Você se interessa nestes dois relacionamentos entre estas classes para o seu sistema de informação da universidade, logo, você precisa modelar ambas as associações.

Segundo, é válido que a mesma classe se envolva nas duas pontas de uma associação; Isto é chamado de associação recursiva ou auto-associação (Fowler (2004)). Um exemplo perfeito disto é a associação *gerencia* que a classe *Empregado* tem consigo mesma. A maneira de ler isto é a de que um empregado qualquer pode gerenciar outros e ele, ou ela, pode ser gerenciado(a) por outro empregado.

Terceiro, algumas vezes, o sentido de uma associação é único, como pode ser visto na figura 12 pela associação *preenche* entre *Empregado* e *Cargo*. A implicação é que o objeto empregado sabe o cargo que preenche, mas o objeto cargo não precisa saber qual empregado o está preenchendo. Esta é uma associação dita unidirecional, uma associação que só é percorrida numa única direção. Se você não tiver um requisito para que se percorra uma associação nos dois sentidos, por exemplo, cargos não precisam colaborar com objetos empregados, então use uma associação unidirecional. Veremos que associações unidirecionais requerem menos trabalho para serem implementadas.

1.9.2 Modelagem do Desconhecido

Não importa o quanto é bom o trabalho que você realiza, pode-se com quase certeza absoluta garantir que você perdeu algum detalhe dos relacionamentos dos seus objetos. Então, o que pode ser feito é: *Faça e torça para que dê certo?* Óbvio que não. Você vai voltar aos seus usuários e perguntar para eles como estão as coisas, não? O problema é lembrar de voltar aos seus usuários. A solução: Marcar como *desconhecido agora* colocando um ponto de interrogação na parte do relacionamento em que você tem uma dúvida. Por exemplo, na figura 13, você acredita que zero ou mais empregados podem preencher um cargo. Você sabe que um cargo pode não está preenchido, mas não tem certeza se um cargo pode ser preenchido por uma ou mais pessoas. Existe compartilhamento de emprego na organização? Existem cargos genéricos como Faxineiro, que é preenchido por várias pessoas? Ou é realmente, uma pessoa por cargo, como estamos mostrando atualmente? Porque você não sabe com certeza ainda, você marca o relacionamento com um ponto de interrogação, indicando que você deve retorna para ele mais tarde e checar seu *chute educado*. Observe que UML não inclui pontos de interrogação como parte da notação, é apenas uma técnica que Ambler descobriu ser muito útil na prática – idealmente, você não deve adivinhar ou supor coisas sem falar antes com quem está adquirindo o sistema (o *stockholder*), mas se você precisar, então marque sua suposição no seu trabalho e procure resolver a dúvida o quanto antes.

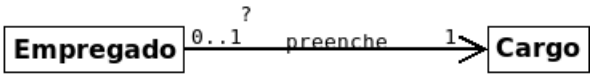

Figura 13 – Indicação de desconhecido

1.9.3 Como Associações São Implementadas

Associações são mantidas pela combinação de atributos e métodos. Os atributos armazenam as informações necessárias para manter o relacionamento e os métodos atualizam os atributos. Por exemplo, a classe *Estudante* da figura 12 potencialmente tem um atributo *matriculado_em*, talvez, um *array*, que é usado para guardar em quais objetos *Disciplina* o estudante está matriculado. A classe *Estudante* pode também ter métodos como *adicionaDisciplina()* e *removeDisciplina()* para adicionar ou remover objetos disciplinas do *array*. A classe Disciplina teria um atributo similar *estudantes* e métodos *adicionaEstudante()* e *removeEstudante()* para manter a associação no sentido oposto. Tudo isto tem uma implicação importante: assim como atributos e métodos são herdados, associações também são.

Na figura 12, a associação unidirecional *preenche* entre *Empregado* e *Cargo* é mais fácil de implementar porque você só precisa percorrer num único sentido: Do *Empregado* para o *Cargo*. Logo, *Empregado* tem um atributo *cargo* e métodos *setCargo()* e *getCargo()* para implementar a associação. Nada precisa ser acrescentado à classe *Cargo*, porque não existe a necessidade de objetos cargos colaborarem com objetos empregados. Logo, não há a necessidade de acrescentar código para implementar a associação no outro sentido.

Não mostre atributos e métodos para implementar associações
É normal supor que existem atributos e métodos para implementar associações, você não precisa mostrá-los nos seus diagramas de classes. Considere estabelecer convenções para os nomes destes atributos e métodos. Tipicamente, o nome atributo deve representar o papel do objeto e o nome dos métodos deve representar a ação sobre os atributos como *adicionaNomeAtributo()* e *removeNomeAtributo()* para associações múltiplas e *setNomeAtributo()* e *getNomeAtributo()* para associações simples.

1.9.4 Propriedades

Em UML 2, uma propriedade é um valor nomeado que denota uma característica de um elemento (tal como uma classe ou um componente). Atributos e associações, inclusive composição, são propriedades. Por exemplo, o nome de um estudante é uma propriedade importante da classe *Estudante*.

O fato de estudantes se matricularem em disciplinas também é uma propriedade importante da classe *Estudante*. Logo, faz sentido atributos e relacionamentos serem propriedades de classes. Lembre-se, também, que a implementação de associações se faz, em parte, através de atributos.

1.9.5 Agregação e Composição

Algumas vezes, um objeto é feito de outros. Por exemplo, um avião é feito de fuselagem, asas, motores, trem de pouso, *flaps*, ... Uma entrega contém um ou mais pacotes. Uma equipe de projeto constitui-se de dois ou mais empregados. Estes são exemplos de agregação, que é representada pelo relacionamento *é parte de*. Um motor é parte de um avião, um pacote é parte de uma entrega e um empregado é parte de uma equipe.

Composição é um tipo forte de agregação no qual o "todo" é completamente responsável pelas partes e cada objeto "parte" só é associado a um objeto "todo". Por exemplo, em qualquer instante, um motor só está associado a um único avião. Além disso, nenhum outro objeto iria colaborar com o objeto motor, além do seu objeto avião. Por exemplo, objetos passageiros do avião não podem pedir para aumentar diretamente a velocidade de um motor.

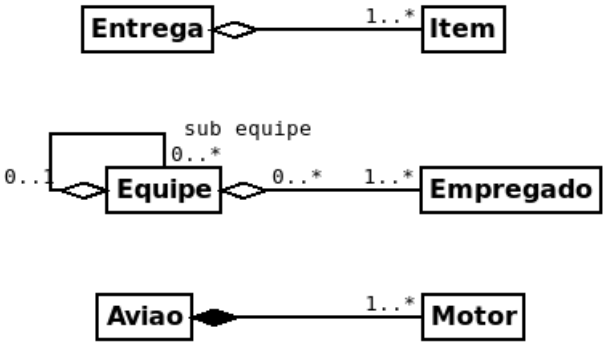

Figura 14 – Exemplos de agregação

Em UML 1, você podia modelar agregação e composição, mas no UML 2, a notação da agregação caiu. Embora isto pareça uma boa ideia, porque muita gente confundia os dois conceitos, suspeita-se que agregação continue a ser usada em diagramas de classe de UML por mais algum tempo, a agregação é modelada desde 1997, quando a UML se tornou popular. A figura 14 mostra exemplos de ambos – a agregação era exibida como uma seta com uma ponta em losângulo vazio e composição com um losângulo cheio. O losângulo está conectado à classe toda. Agregação é simplesmente um tipo de associação. Você ainda tem de modelar as multiplicidades e os papéis. Deixar de indicar a multiplicidade de uma associação é permitido, mas não é recomendável. Por exemplo, na figura 14, a multiplicidade não é indicada, nem para entregas, nem para aviões, nestes casos, a multiplicidade implícita é 1.

Assim como associações são vias de duplo sentido, também as agregações são. Além disso, as associações de agregação/composição mostradas na figura 14 são lidas de maneira similar:

- Um item é parte de uma e apenas uma entrega;
- Uma entrega é composta por um ou mais itens;
- Um motor é parte de um e apenas um avião;
- Um avião tem um ou mais motores;
- Um empregado pode ser parte de uma ou mais equipes;
- Uma equipe é constituída de um ou mais empregados;
- Qualquer equipe pode fazer parte de uma equipe maior; e

- Uma equipe pode ser constituída de subequipes menores.

É importante conhecer a antiga notação para agregação. Mesmo que não a usemos mais.

As dicas a seguir ajudam a modelar adequadamente a composição:

1. **Aplique a regra da sentença:** O primeiro teste é que a frase "a parte *é parte do* todo" deve fazer sentido. Por exemplo, faz sentido dizer: Um motor *é parte de* um avião. Entretanto, não faz sentido dizer que um empregado *é parte de* um cargo, ou um cargo *é parte de* um empregado, por isso que é mais apropriado usar uma associação como na figura 12. Se a frase não faz sentido, a composição não é apropriada. Herança ou associação devem ser usadas nesse caso.
2. **O todo deve gerenciar as partes:** O segundo teste, supondo que o primeiro tenha sido aprovado, é se o todo gerencia as partes. Por exemplo, um avião deve gerenciar os motores; você não deseja que um passageiro no avião seja capaz de manipular os motores.
3. **Você deve se interessar pela parte:** Um objeto pode fazer parte no mundo físico, mas você pode não estar interessado nele, logo, não o modele. Por exemplo, um sistema de manutenção de aeronaves pode ser interessante para os registros de cada motor, guardando as informações de manutenção dos motores. Por outro lado, o sistema de controle de tráfego aéreo não tem informações interessantes para os motores. Logo, um motor não vai aparecer no diagrama de classes do sistema de controle de tráfego aéreo.
4. **Mostre a multiplicidade e os papéis:** Assim como você mostra a multiplicidade e os papéis numa associação, você deve mostrá-los numa composição.
5. **Composição é herdada:** Associações de composição, assim como, associações comuns, são implementadas por atributos e métodos, portanto, podem ser herdadas.

1.9.6 Dependências

Existem dois tipos de relacionamentos: persistente e transitório. A diferença principal é a de que associações persistentes devem ser salvas, enquanto relacionamentos transitórios têm natureza temporária e não precisam ser salvos.

Relacionamentos persistentes são permanentes, ou, pelo menos, semi-permanentes, por natureza. Um relacionamento de objetos é persistente se a informação para mantê-lo é salva em memória permanente. Por exemplo, o relacionamento *se_matricula* entre estudantes e disciplinas é persistente. Isto é uma informação de negócio importante que deve ser armazenada no disco. O relacionamento *ensina* entre professores e disciplinas é persistente pelo mesmo motivo. Todos os relacionamentos que vimos até agora são persistentes.

Associações transitórias são temporárias por natureza. Elas não são salvas no armazenamento permanente. Relacionamentos transitórios envolvem, geralmente, mas não sempre, pelo menos um objeto transitório, tal como uma tela, ou um relatório. A razão para isto é simples: Se você não precisa persistir um objeto, então, você não precisa persistir seus relacionamentos.

Relacionamentos transitórios existem entre objetos por uma razão apenas: Os objetos precisam colaborar entre eles. Para um objeto colaborar com outro objeto, ele precisa conhecê-lo. Isto significa que deve existir, ou um relacionamento entre os objetos, ou um relacionamento do tipo "parte-de" entre os dois objetos. Quando não há uma associação persistente entre dois objetos, mas eles colaboram um com o outro, você modela uma relação de dependência entre as duas classes.

A figura 15 mostra um relacionamento de dependência – modelada pela linha tracejada com uma seta aberta – entre as classes *Estudante* e *TelaEdicaoEstudante*, representando o relacionamento transitório entre um objeto tela de edição de estudante usado para atualizar o objeto estudante. A tela de edição obtém a informação atual do objeto estudante, mostra no modo de edição e atualiza a informação nova quando termina a edição. O relacionamento transitório entre o objeto tela de edição e o objeto estudante existe enquanto a informação do estudante é exibida na tela. Quando a tela é

Figura 15 – Dependência entre duas classes.

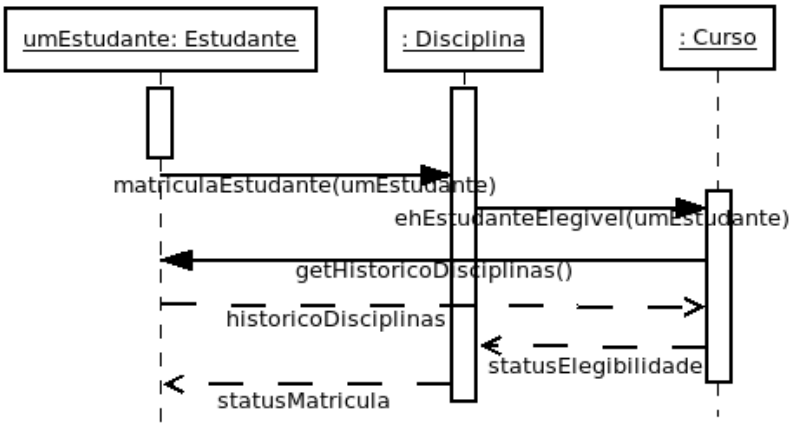

Figura 16 – Diagrama de sequência mostrando mensagens entre objetos.

fechada, o relacionamento deixa de existir (a tela também não existe mais) e os objetos transitórios provavelmente já foram destruídos.

Associações transitórias podem também ocorrer entre objetos persistentes. Por exemplo, na figura 16, uma associação transitória implícita existe entre o objeto estudante e o objeto curso (o estudante é passado como um parâmetro para o curso, para saber se ele atende as condições de pré-requisitos). Esta associação não foi ilustrada na figura 12 porque o conceito não foi introduzido naquele momento. Por consistência, as associações transitórias também precisam ser modeladas.

1.10 Colaboração

Classes precisam, em geral, trabalhar juntas para cumprirem suas responsabilidades. Na verdade, são os objetos, as instâncias das classes que trabalham juntas. Colaboração entre objetos ocorre quando um objeto pede ao outro informações ou para o outro realizar algo. Por exemplo, um avião colabora com seus motores para poder voar. Para o avião ir mais rápido, ele pede para os motores girarem mais rápido. Quando um avião precisa desacelerar, os motores precisam desacelerar. Se o avião não colaborasse com seus motores, ele não voaria.

Objetos colaboram uns com os outros pelo envio de mensagens. Uma mensagem é tanto um pedido para fazer alguma coisa, quanto um um pedido por informações. Mensagens são modeladas por diagramas de sequência e diagramas de comunicação (diagramas de colaboração em UML 1.x) em UML. A figura 16 mostra um diagrama de sequência simples. Nele, você vê como um estudante requer a matrícula numa disciplina: O objeto disciplina de sua parte manda uma mensagem para o objeto curso com o qual está associado para saber se o estudante pode se matricular na disciplina (checar pré-requisitos, por exemplo). A figura 17 mostra o mesmo exemplo, mas na forma de diagrama de comunicação.

Vejamos alguns detalhes sobre o diagrama de sequência, as caixas no topo do diagrama representam classificadores, no caso, mais precisamente, objetos. As linhas tracejadas penduradas nas caixas são chamadas de *linha de vida* dos objetos e representam o intervalo de tempo de vida do objeto

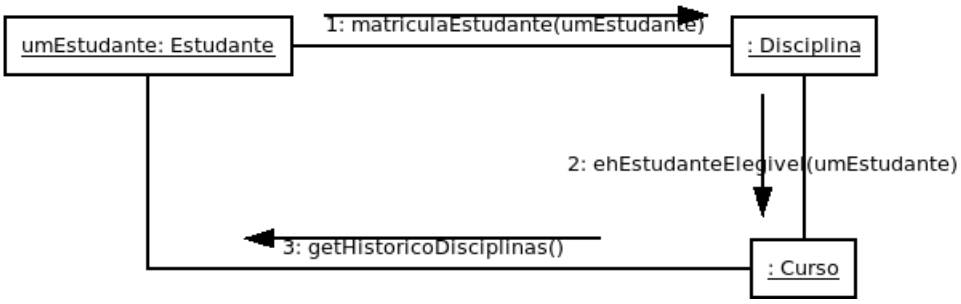

Figura 17 – Diagrama de comunicação exibindo as mensagens.

no cenário modelado. Objetos são rotulados no formato *nome: classe* onde o *nome* é opcional (objetos que não receberam nomes no diagrama são chamados de *objetos anônimos*). A instância de *Estudante* recebeu um nome porque ela é usada como parâmetro numa mensagem, já as instâncias de *Disciplina* e *Curso* não precisam ser referenciadas em nenhuma parte do diagrama e, portanto, podem ser anônimas. As mensagens são indicadas por setas rotuladas, o rótulo é a assinatura do método. Valores de retorno são indicados opcionalmente por seta com linha tracejada e um rótulo indicando o valor de retorno. Valores de retorno estão representados na figura 16, mas não na 18. Em metodologias ágeis, devemos manter os diagramas simples e assumir que o valor de retorno volta. O sequenciamento das mensagens é dado pela ordem das próprias mensagens, começando no topo a esquerda do diagrama.

Diagramas de comunicação têm uma notação similar aos diagramas de sequência. Objetos são indicados da mesma maneira, embora eles sejam conectados por linhas de associação sem rótulos. As mensagens são indicadas com setas de novo, embora não conectem os objetos (não há linhas de vida nos diagramas de comunicação/colaboração). A sequência de mensagens é opcionalmente indicada colocando um número na frente do nome da mensagem. Na figura 17, vemos a mesma sequência de mensagens que a da figura 16.

As dicas e técnicas a seguir devem ajudar a modelar de maneira efetiva as colaborações:

1. **Deve ter algum tipo de relacionamento.** A única maneira de um objeto poder mandar um mensagem para um outro é se o conhece antes. No mundo real, você não pode pedir ajuda a alguém se você não sabe como contactá-lo. O mesmo princípio se aplica a objetos. Deve existir um associação, talvez uma agregação, entre as duas classes para que as instâncias (objetos) sejam capazes de colaborar.

2. **Deve ter um método correspondente no objeto alvo.** Colaborações são implementadas através de chamadas de métodos, isto implica que o método deve existir no objeto alvo para ele ser chamado. Para um objeto poder receber um pedido de informação, ele deve ter um método que retorna esta informação. Por exemplo, na figura 16, a mensagem *ehEstudanteElegivel()* é enviada para o objeto curso, logo na classe *Curso* deve haver uma implementação do método *ehEstudanteElegivel()*. Se desejamos que um objeto faça algo, ele deve ter um método que o faça. De modo similar, na figura 16, observe que a mensagem *matriculaEstudante()* é enviada a objetos disciplinas, logo a classe *Disciplina* deve ter um método *matriculaEstudante()*.

3. **Pode haver um valor de retorno.** Se a colaboração é um pedido de informação, então deve haver o retorno de um valor (a informação pedida). Este fato deve ser incluída na documentação do método e, opcionalmente, será indicada no diagrama de sequência como uma linha tracejada. Valores de retorno não são modelados em diagramas de colaboração porque eles tendem a poluir os diagramas.

4. **Pode ter ou não parâmetros.** Algumas mensagens têm parâmetros, outras não. Lembre-se de que mensagens são efetivamente uma chamada de método (função). Assim como funções têm parâmetros, métodos também têm. Por exemplo, na figura 16, um objeto estudante passa

a si mesmo como um parâmetro quando chama a mensagem *matriculaEstudante()* do objeto *Disciplina*. Na figura 18, o objeto entrega não precisa passar qualquer parâmetro quando pede ao objeto item o seu preço.

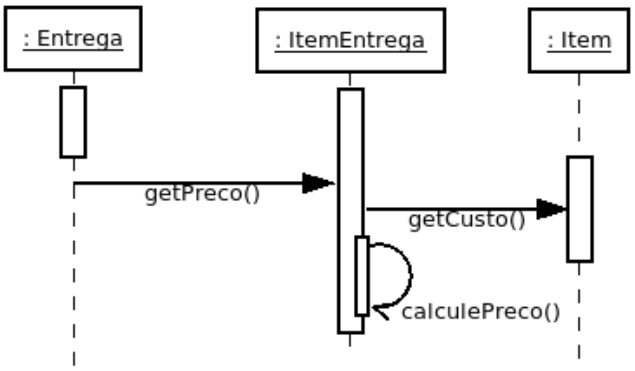

Figura 18 – Diagrama de sequência.

5. **Mensagens mostram a colaboração, não o fluxo de dados.** Mensagens são pedidos. Elas não são fluxos de dados. Diagramas de processo (diagramas de fluxo de dados) do mundo estruturado mostram fluxos de dados, que são movimentos de dados de uma parte do sistema para outra.

6. **Algumas vezes o alvo precisa colaborar.** O receptor de uma mensagem pode não ser capaz de atender a um pedido completamente e pode precisar colaborar com outros objetos para cumprir sua responsabilidade. Por exemplo, na figura 16, o objeto disciplina precisou interagir com o objeto curso do qual ela faz parte para matricular um estudante. Isto é perfeitamente válido.

7. **Cada método deve fazer alguma coisa.** É importante que cada objeto colaborador deve sempre fazer algo, não apenas encaminhar a mensagem para outro objeto, algo chamado de uma passarela (*passthrough*). Passarelas resultam em *código espaguete* que pode ser difícil para dar manutenção. Algumas vezes, *delegação*, um padrão de projeto, faz sentido, mas, enquanto você for um principiante, é melhor evitar.

8. **Um objeto pode colaborar consigo mesmo.** Objetos frequentemente se enviam mensagens para obter informações e/ou para fazer alguma coisa. Isto é como uma função chamando uma outra em uma linguagem procedural como o C. Na figura 18, o objeto *ItemEntrega* se manda uma mensagem para calcular o preço (provavelmente, o preço do item vezes o número de itens a ser entregue).

1.11 Acoplamento

Acoplamento é uma medida do quanto dois itens, tais como classes ou métodos, estão relacionados entre eles. Quando uma classe depende de uma outra, dizemos que elas estão acopladas. Quando uma classe interage com uma outra, mas não conhece nada sobre os detalhes de implementação da outra, dizemos que elas são fracamente acopladas. Quando uma classe se baseia na implementação (isto é, tem acesso direto aos atributos da outra), dizemos que elas estão fortemente acopladas.

Na discussão de como *Estudante* pode implementar o método *se_matricula()*: Ele pode acessar diretamente o atributo *listaDeEstudantes* de *Disciplina*, ou pode enviar uma mensagem para objetos *Disciplina* pedindo para matricular o estudante na disciplina. Acessando diretamente e mudando o atributo *listaDeEstudantes* pode economizar alguns ciclos de CPU e rodar mais rápido, mas assim que a implementação daquele atributo mudar (passando de um *array* para uma *lista encadeada*, por exemplo), seu código da classe *Estudante* também vai ter de mudar. Isto não é muito bom. O problema

básico é que quando duas classes são fortemente acopladas, uma mudança numa, requer uma mudança na outra. Isto, por sua vez, pode requerer uma mudança em outra classe, e em outra, ... Acoplamento forte é uma das causas para a manutenção de código ser complexa. O que deveria ser uma troca simples de manutenção pode levar meses para estabilizar, se é que vai ficar estável. É incrível a quantidade de código que existe que ninguém quer tocar com medo de quebrá-lo. Por exemplo, lembre-se do ano 2000 (bug do milênio, que não era no milênio).

Frequentemente, os desenvolvedores são seduzidos pelo lado negro da força e decidem escrever código fortemente acoplado. Isto só se justifica quando você está desesperado em reduzir todos códigos redundantes do seu programa. Por exemplo, *drivers* de bancos de dados são fortemente acoplados aos sistemas de arquivos do sistema operacional onde o banco de dados está instalado. Se você puder salvar uns microssegundos no acesso aos dados, isto pode ser significativo ao acessar centenas de milhares de objetos.

1.12 Coesão

Coesão é uma medida do quanto um item, tal como uma classe ou um método, faz sentido. Uma boa medida da coesão de algo é quanto tempo leva para descrevê-lo numa frase: quanto mais tempo levar, menos coeso ele é. Você deseja projetar classes e métodos com alto grau de coesão. Em outras palavras, você deseja classes e métodos cuja descrição seja bem clara.

Um método é bastante coeso se ele faz uma coisa e apenas esta coisa. Por exemplo, na classe *Estudante*, você teria métodos para matricular estudantes em disciplinas e remover estudantes de disciplinas. Cada um desses métodos faze uma coisa e apenas esta coisa. Você poderia escrever um método que faz as duas coisas e chamá-lo, por exemplo, de *mudaStatusDisciplina()*. O problema com esta solução é que o código ia ficar mais complexo do que com os dois métodos separados. Isto significa que seu código ia ficar mais difícil de ser entendido e, portanto, de fazer a manutenção ia ficar mais complicada. Lembre-se de que você deve simplificar a manutenção, não torná-la mais complexa.

Nomes de métodos indicam o grau de coesão
O nome de um método, em geral, indica o quanto ele é coeso. Sempre que você encontrar uma combinação forte verbo/complemento para ser usada como nome de método, há grandes chances do método ser fortemente coeso. Por exemplo, leve em consideração métodos como *getNome()*, *printNome()*, *matriculaDisciplina()* e *desisteDisciplina()*. Verbos tais como get, print, matricular e desistir são todos muito fortes. Leve em consideração *mudaStatusDisciplina()*, agora. Mudar é tão forte ou tão explícito quanto as palavras matricular e desistir? Provavelmente, não, mudar é mais geral.

Uma classe altamente coesa representa um tipo de objeto e só um tipo de objeto. Por exemplo, para o sistema de informação da universidade, modelamos professores e não empregados. Apesar de um professor ser na realidade um empregado, eles são diferentes de outros tipos de empregados. Por exemplo, professores fazem coisas diferentes de faxineiros, que fazem coisas diferentes de secretárias, que fazem coisas diferentes do pessoal administrativos, etc. Podemos escrever facilmente uma classe *Empregado* genérica, capaz de lidar com todas as funcionalidades realizadas por todos os empregados da universidade. Entretanto, esta classe pode se tornar rapidamente pesada e difícil de manter. Uma solução melhor seria definir uma hierarquia de classes com *Professor*, *Faxineiro*, *Secretaria*, *Administrativo*, etc. Por causa das similaridades existentes entre estas classes, você criaria uma nova classe abstrata chamada *Empregado*, que herdaria de *Pessoa*. As outras classes, inclusive *Professor*, herdam de *Empregado*. A vantagem disto é que cada classe representa um tipo de objeto. Se aparecerem mudanças para serem feitas sobre os faxineiros, você faz diretamente na classe *Faxineiro*. Você não precisa se preocupar com o efeito destas mudanças no código dos professores.

1.13 Polimorfismo

Um objeto pode ser de um ou vários tipos. Por exemplo, um objeto José da Silva pode ser um estudante, um administrativo e até um professor. Há problemas para os outros objetos o tipo de pessoa

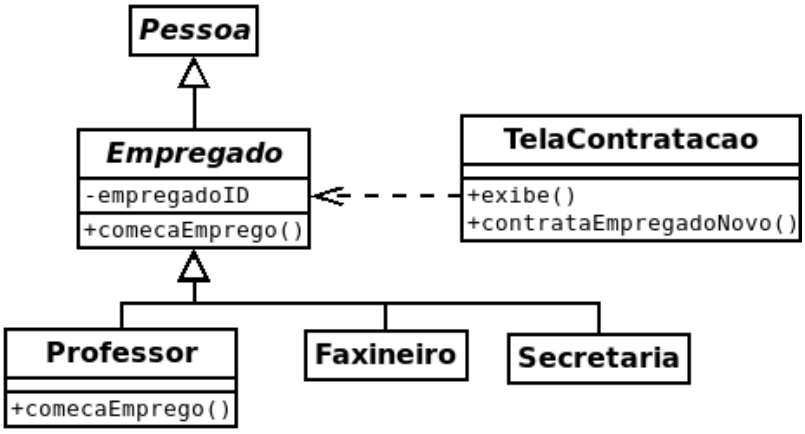

Figura 19 – Implementação parcial do sistema de recursos humanos da universidade.

que o José é? Reduz significativamente o esforço de desenvolvimento se outros objetos no sistema pudessem tratar os objetos pessoas da mesma maneira independente do tipo específico de cada pessoa. O conceito de polimorfismo diz que você pode tratar instâncias de várias classes do mesmo jeito dentro do seu sistema. A consequência disto é que você pode mandar uma mensagem para um objeto sem saber de antemão o tipo dele e o objeto ainda vai "fazer a coisa" certa, pelo menos do ponto de vista dele.

1.13.1 Polimorfismo na Universidade

Considere um exemplo um pouco mais realístico de polimorfismo explorando o projeto de como a universidade lida com a contratação de pessoal modelada na figura 19. Há uma maneira padrão para a universidade contratar pessoal: Uma vez contratada uma pessoa, ela é incluída no plano de saúde da universidade e ganha um crachá. Quando um professor é contratado, o mesmo processo é seguido, mas, além disso, uma vaga de estacionamento é atribuída para o novo professor se houver vaga disponível, senão ele entra na fila de espera por vagas de estacionamento.

Se o método *comecaEmprego()* for implementado na classe *Empregado*, ele implementa o comportamento necessário para adicionar uma pessoa ao plano de saúde da universidade e a geração do crachá. O método *comecaEmprego()* na classe *Professor* terá de ser sobrescrito. Ele chama o método da classe *Empregado*, pois professores também são inclusos no plano de saúde da universidade e recebem crachás, mas, além disso, reserva uma vaga de estacionamento.

Por ser capaz de enviar a mensagem *comecaEmprego()* para qualquer tipo de empregado, não há a necessidade de usar complicadas instruções condicionais no método *contrataEmpregadoNovo()* do objeto tela. Este método não precisa mandar uma mensagem *comecaProfessor()* para objetos professores, *comecaFaxineiro()* para objetos faxineiros, etc. Ele só precisa enviar *comecaEmprego()* para qualquer tipo de empregado e o objeto fará a coisa certa. Como resultado, você pode acrescentar novos tipos de empregados (como o *Administrativo*) e não precisa mudar o objeto tela. Isto também significa que a classe *TelaContratacao* é fracamente acoplada à hierarquia de classes, o que permite a extensão do sistema facilmente.

É importante mencionar que o projeto da figura 19 não é um muito bom. Uma maneira melhor de fazer é refatorar a hierárquica usando o padrão de projeto *Papéis interpretados* ilustrado na figura 20. Este padrão separa o conceito de alguém ser uma pessoa e o papel que ela interpreta no trabalho. Esta separação permite que uma pessoa interprete mais de um papel no trabalho.

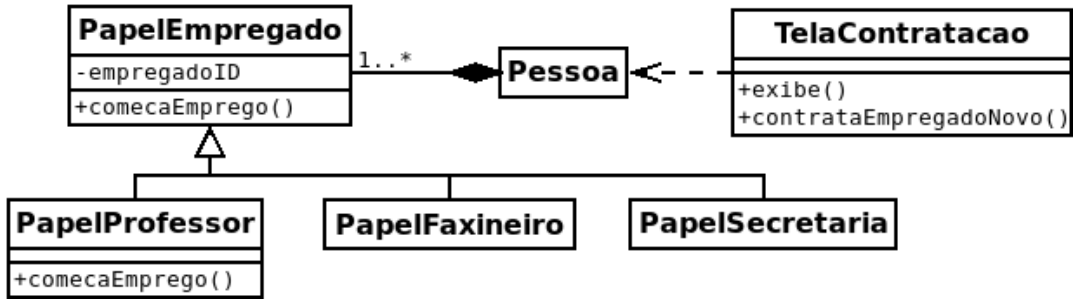

Figura 20 – Sistema de recursos humanos da universidade refatorado.

1.14 Interfaces

Uma interface é a definição de uma coleção de um ou mais métodos e zero ou mais atributos. Interfaces idealmente definem um conjunto de comportamentos coesos. Interfaces são implementadas por classes e componentes. Para implementar uma interface, uma classe ou componente deve incluir os métodos definidos pela interface. Por exemplo, a figura 21 ilustra que a classe *Estudante* implementa as interfaces *Serializavel*[5] e *Buscavel*. Para implementar a interface *Buscavel*, *Estudante* deve definir o método *busca()* com o parâmetro *critério*. Qualquer classe ou componente pode implementar zero ou mais interfaces e uma ou mais classes podem implementar qualquer interface. Interfaces são usadas para promover a consistência dentro de seus modelos e código fonte.

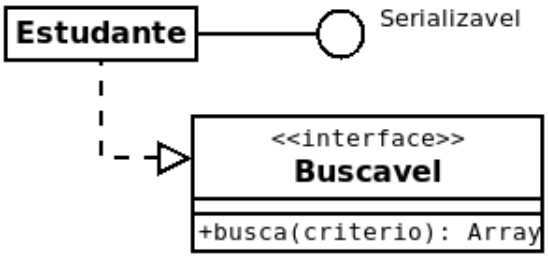

Figura 21 – As duas notações para interfaces em UML.

Interfaces servem para garantir acoplamentos fracos. Eles permitem a uma classe participar de um conjunto comum de funcionalidades sem que nenhuma outra classe tenha que saber algo mais além dela implementar a interface. Um objeto GUI, por exemplo, pode exibir uma lista de estudantes que atendem um critério de busca sem conhecer nada da classe *Estudante*, exceto que ela implementa a interface *Buscavel* e a interface *Descrição* que tem um método que apresenta o estudante. A mesma GUI pode ser usada para realizar a mesma ação com os professores, se a classe *Professor* também implementar estas duas interfaces. De modo análogo, se a classe *Disciplina* também implementar estas interfaces, a GUI também pode ser usada para exibir uma lista de disciplinas que atendam ao critério de busca. A GUI não precisa saber nada sobre a classe, só que ela implementa as interfaces. Isto mostra um fraco acoplamento entre a GUI e os objetos das classes que ela exibe.

Em linguagens como Java e C#, interfaces fazem parte do mecanismo de tipagem. Isto é, a classe *Estudante* da figura 21 tem pelo menos três tipos: *Estudante*, *Serializavel* e *Buscavel*. Ela também tem o tipo de todas as suas superclasses. Por exemplo, se *Estudante* herda de *Pessoa* e ela implementa a interface *Printable*, então os objetos estudantes também são do tipo *Pessoa* e *Printable*. O mecanismo de tipo da sua linguagem de implementação afetará a extensão do polimorfismo suportado.

[5] Observe que estamos traduzindo *Serializable* por Serializavel, embora não achemos que este termo reflita adequadamente a noção do que esta interface permite que seja feito com os objetos que a implementam. A ideia é que um objeto serializável é um objeto que pode ser convertido para um formato textual que permita a transmissão do objeto serializado através da rede para um processo remoto ou local. O objeto serializado também pode ser armazenado em uma memória permanente e recuperado depois.

A figura 21 mostra ainda que existem duas notações em UML para indicar que algo implementa uma interface: A notação da circunferência com linha contínua (vulgarmente chamada de pirulito) da interface *Serializavel* e a notação em caixa usada para a interface *Buscavel*. A primeira tem a vantagem de ser mais compacta, enquanto a segunda fornece mais detalhes. A caixa da interface *Buscavel* é a mesma da classe com o estereótipo *interface*. Estereótipo é um mecanismo para definir extensões comuns e consistentes na notação UML. A seta tracejada de *Estudante* para *Buscavel* é uma relação *realiza* em UML. Isto significa que *Estudante* implementa (realiza) a interface *Buscavel*.

1.15 Componentes

Um componente é uma unidade modular, extensível de distribuição independente que tem interface(s) contratualmente especificadas e dependências explicitamente definidas, se houver. Idealmente, componentes devem ser modulares, extensíveis e abertas. Modularidade implica em que um componente contém tudo que ele precisa para cumprir suas responsabilidades. Extensibilidade implica em que um componente pode ser melhorado para cumprir outras responsabilidades além daquelas para as quais foi originalmente projetada. E aberta implica que ele possa funcionar em várias plataformas e interagir com outros componentes através de uma única interface de programação.

Componentes, como classes, implementam interfaces. As interfaces de um componente definem os seus pontos de acesso. Componentes são tipicamente implementadas como coleções de classes, idealmente, classes que formam um subconjunto coeso dos seus sistemas completos. Componentes são tipicamente pesados e podem até ser pensados como classes grandes ou mesmo subsistemas. Por exemplo, um banco de dados pode ser um componente, ou uma coleção de classes de negócio/domínio que implementam os comportamentos requeridos pelas pessoas dentro da sua aplicação pode ser um componente.

Diagramas de Componentes mostram os componentes de software que compõem um pedaço maior de software, suas interfaces e suas inter-relações. Vamos considerar um componente como sendo um item grande usado nas operações diárias do seu negócio – por exemplo, um subsistema comum, um sistema comercial padrão (COTS - commercial off-the-self system), uma aplicação OO ou uma aplicação antiga empacotada. De muitas maneiras, um diagrama de componentes é simplesmente um diagrama de classes numa escala maior, apenas menos detalhada.

A figura 22 mostra um exemplo de um diagrama de componente usado para modelar uma arquitetura de negócio para a universidade. As caixas representam componentes – neste caso, as interfaces de usuário que o pessoal usa para interagir com os sistemas da universidade, os componentes de negócio tais como *Instalacoes* ou componentes técnicos como o componente de *Seguranca* ou banco de dados.

Na figura 22, você pode observar que os componentes UI (interface usuário) têm dependências nas interfaces dos componentes de negócio. Ao fazer os componentes UI dependentes das interfaces e não dos próprios componentes, você torna possível substituir os componentes com diferentes implementações desde que eles implementem as interfaces dadas. Isto reduz acoplamento. De modo similar, é possível que os componentes de negócio dependam uns dos outros. Por exemplo, o componente *Disciplina* é acoplado ao componente *Horario*.

Uma notação interessante da UML 2 é o soquete. Um soquete é o semicírculo em torno do símbolo pirulito, como pode ser visto com as interfaces *ControleAcesso* e *Persistencia* com as componentes de *Seguranca* e *Persistencia*, respectivamente. Este símbolo é usado para indicar que alguma coisa requer a existência de uma interface. Por exemplo, as quatro componentes de negócios (*Instalacoes*, *Estudante*, *Disciplina* e *Horario*) todas requerem a existência de alguma coisa que implemente a interface *ControleAcesso*.

Observe a consistência da notação entre diagramas de classes e diagramas de componentes. Ambos usam exatamente a mesma notação para relacionamentos de dependência, vide figura 15. Esta é uma característica boa da UML: A notação é consistente. Existem uns poucos detalhes da UML que não são consistentes, mas, na maioria dos casos, cada conceito é modelado da mesma maneira através

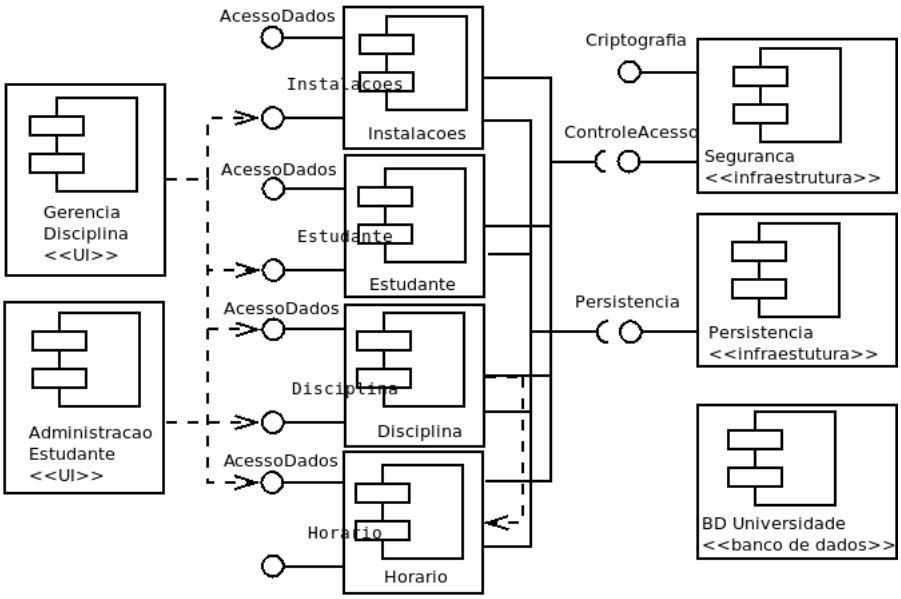

Figura 22 – Diagrama de componentes da UML 2.

dos diferentes diagramas.

1.16 Padrões

Com a prática, você vai sentir que você repetidamente está resolvendo os mesmos problemas. Se você não resolveu pessoalmente um certo problema, é muito provável que alguém ou resolveu este problema, ou um muito similar. O problema em que você está trabalhando pode ser simples ou complexo, mas, geralmente, já foi trabalhado antes. Não seria bom ser capaz de encontrar uma solução facilmente, ou, pelo menos, uma solução parcial, para o problema? Pense em quanto tempo e esforço poderiam ser poupados se você tivesse acesso a uma biblioteca de soluções para problemas de desenvolvimento de sistemas comuns. É disto que tratam os padrões (*patterns*).

Um padrão é uma solução para um *problema comum* levando em conta as forças relevantes, apoiando efetivamente o reuso de técnicas e procedimentos comprovados de outros desenvolvedores. Existem diferentes tipos de *padrões*, incluindo *padrões de análise*, *padrões de projeto* e *padrões de processos*. Padrões de análise descrevem uma solução para problemas comuns encontrados na análise da lógica de negócios/domínio de uma aplicação, padrões de projeto descrevem uma solução para problemas comuns encontrados no projeto de sistemas e padrões de processos se aplicam a questões de processos de software. O livro (GAMMA et al., 1994) é a referência de base clássica que popularizou o uso de padrões em software.

Por exemplo, é comum descobrir classes em sua aplicação que devem ter apenas uma instância. Por exemplo, deve haver apenas uma janela de edição para um dado tipo de alteração, ou alguma informação de configuração só pode ser armazenada num único local, ou você precisa manter valores constantes num único lugar. Nestes casos todos, você precisa de uma única instância da classe em questão – uma única instância de uma caixa de diálogo, uma única instância da informação de configuração e uma única instância das constantes. Este problema é resolvido pelo padrão *Singleton*, um padrão de projeto que mostra como apenas uma instância de uma classe pode existir. A figura 23 mostra o diagrama de classe de um *Singleton*. Um atributo estático existe para acessar a instância única, se ela existe. O método *create()*, talvez melhor chamado de *getSingleton()*, permite acessá-lo e criá-lo se necessário. *Singleton* é um padrão simples e muito usado.

Existem padrões muito complexos. A maioria dos padrões são baseados em três ou mais classes. O principal desafio com os padrões é que quando você aprende sobre eles, tudo parece ser um padrão e,

Figura 23 – Padrão de projeto *Singleton*

às vezes, isto é verdade. O perigo é que você pode exagerar na construção do seu software, aumentando a dificuldade de manutenção, apesar de reduzir o tempo para implementar os requisitos. Para evitar, atenuar, este problema, o modelamento ágil (AM) inclui a prática de *aplicar gentilmente os padrões*, onde você introduz os padrões pela *refatoração* do seu projeto lentamente no lugar de aplicá-los quando você acha que precisa deles. (SHALLOWAY; TROTT, 2005) considera que um bom momento para aprender padrões de projeto é quando se está aprendendo projetos com orientação a objetos.

1.17 O que foi aprendido

Neste capítulo, você foi apresentado aos principais conceitos do paradigma da orientação objeto e à notação básica da UML 2 para modelar alguns destes conceitos. Os conceitos de OO são numerosos e, às vezes, complexos. Não se preocupe, é normal se sentir sobrecarregado no início, mas com a prática você passa a dominá-los.

1.18 Questões de Revisão

1. Discorra sobre a diferença entre herança e composição. Quais são as vantagens e desvantagens de cada um deles. Você pode implementar um com o outro?

2. Como uma classe se assemelha com uma tabela de banco de dados? Como elas se diferem? Como estas diferenças e similaridades justificam modelos de classes e modelos de dados?

3. Discorra sobre a diferença entre associação e composição. Quais são as vantagens e desvantagens de cada um?

4. Quando você deve aplicar herança? Quando não? Forneça exemplos de quando herança é apropriada e quando não é.

5. Compare e oponha os conceitos de acoplamento e coesão. Como eles se relacionam, se é que se relacionam?

6. Descreva o relacionamento entre o polimorfismo e a tipagem.

7. Através da Internet, pesquise sobre tecnologias de bancos dedados, incluindo bancos de dados relacionais, bancos de dados de objetos, bancos de dados relacionais-objetos, bancos de dados XML, bancos de dados hierárquicos e bancos de dados em redes. Forneça pelo menos um exemplo de cada. Compare as tecnologias, listando as vantagens, desvantagens e uma indicação de quando você deve usar cada um.

8. Como interfaces reduzem o acoplamento de sistemas OO? Como elas podem aumentar? Por que?

2 Padrões de Projeto

Neste capítulo veremos um resumo das ideias apresentadas em (SHALLOWAY; TROTT, 2005) e (GAMMA et al., 1994) sobre padrões de projetos. Shalloway e Trott acreditam que é possível aprender orientação objeto simultaneamente com os padrões de projeto. E é o que tentamos fazer neste texto. Não veremos os padrões com a minúcia desses livros, apenas vamos apresentar alguns dos aspectos mais essenciais. Os leitores mais interessados em padrões de projeto devem ler pelo menos estes dois textos.

Um padrão tem quatro elementos essenciais:

1. O *nome do padrão* permite distinguir rapidamente o padrão de outros e ter uma noção da aplicação e o que é o padrão. Encontrar bons nomes pode ser difícil, mas provavelmente podemos aplicar também a regra da coesão aqui: Se não há uma boa maneira de nomeá-lo, há uma boa chance do padrão não está correto. Óbvio existem exceções para esta regra e, às vezes, precisamos criar um nome novo (uma palavra nova) para representar o conceito.

2. O *problema* descreve quando aplicar o padrão. Ele explica o problema e seu contexto (as forças relevantes).

3. A *solução* descreve os elementos que compõe o projeto, seus relacionamentos, responsabilidades e colaborações. A solução não descreve um projeto concreto específico, ou uma implementação.

4. As *consequências* descrevem os resultados e as limitações de aplicar o padrão. As consequências para software em geral se preocupam com as limitações de tempo de execução e espaço de memória. Elas também podem tratar de problemas com a linguagem de programação e a implementação. As consequências de um padrão incluem aspectos do impacto sobre a flexibilidade do sistema, extensibilidade e portabilidade da solução.

(GAMMA et al., 1994) classifica os padrões de projetos do ponto de vista do propósito em três categorias:

1. **Criação:** Padrões de criação lidam com a criação de objetos.

2. **Estrutura:** Padrões de estrutura lidam com a composição de classes e objetos.

3. **Comportamento:** Padrões de comportamento caracterizam as maneiras como as classes e objetos interagem e distribuem responsabilidade.

Os padrões também podem ser classificados de acordo com o escopo de aplicação do padrão: Se o padrão se aplica sobre as classes ou sobre os objetos. Padrões de classes lidam com os relacionamentos entre as classes e suas subclasses. Estes relacionamentos são estabelecidos através de herança, portanto são estáticos, fixados durante a compilação. Os padrões de objetos lidam com o relacionamento entre os objetos, que podem mudar durante a execução, portanto são mais dinâmicos.

As oito características básicas na descrição dos padrões de acordo com Gamma et al. (1994) são:

Nome: Todos os padrões devem ter um único nome que os identifica.

Propósito: O propósito do padrão.

Problema: O problema que o padrão tenta resolver.

Solução: Como o padrão fornece uma solução ao problema no contexto em que ele se encaixa.

Participantes e Colaboradores: As entidades envolvidas no padrão.

Consequências: As consequências de usar o padrão. Investiga as forças relevantes no emprego do padrão.

Implementação: Como o padrão pode ser implementado. Observação: Implementações são manifestações concretas de padrões, não devem ser entendidas como o padrão em si.

Estrutura Genérica: Um diagrama UML que mostra a estrutura típica do padrão.

2.1 Catálogo de Padrões

Começaremos apresentando um catálogo de padrões de projetos dos dois livros, (GAMMA et al., 1994) e (SHALLOWAY; TROTT, 2005). Na exposição de alguns padrões, veremos algumas aplicações deles. Em geral, não aplicamos os padrões de maneira isolada, mas, para resolver problemas práticos, aplicamos diversos padrões. Não detalharemos todos os padrões e não recomendamos decorar todos os padrões para tentar a todo custo utilizá-los. O bom senso deve sempre prevalecer e aplicar o padrão apenas se ele se mostrar interessante. Neste aspecto, é importante que a maioria das ferramentas de desenvolvimento de software auxiliem na refatoração dos programas. Um desenvolvimento pode ser feito sem utilizar um padrão de projeto e mais tarde pode-se perceber que é interessante a aplicação de um padrão para aumentar o desacoplamento, ou para tornar o projeto mais coeso, ou alguma outra característica interessante que o padrão de projeto nos forneça, nesse caso aplica-se uma refatoração.

Apesar da *gang de 4*[1] apresentar os padrões em ordem alfabética, procuraremos estudar os padrões pelos mais simples e familiares. Já apresentamos o padrão Singleton no capítulo anterior, página 31. Além disso, procuramos deixar próximos os padrões que estão correlacionados. Os padrões que serão apresentados rapidamente são:

- O Padrão Fachada
- O Padrão Adaptador
- O Padrão Fabrica Abstrata
- O Padrão Método Fabrica
- O Padrão Observador
- O Padrão Composição
- O Padrão Iterador
- O Padrão Visitante
- O Padrão Estratégia
- O Padrão Decorador
- O Padrão Método Padrão
- O Padrão *Singleton*
- O Padrão Trava Dupla
- O Padrão Ponte
- O Padrão Construtor
- O Padrão Protótipo

[1] (GAMMA et al., 1994)

- O Padrão Repositório de Objetos
- O Padrão Cadeia de Responsabilidade
- O Padrão Comando
- O Padrão Peso Pena
- O Padrão Interpretador
- O Padrão Mediador
- O Padrão Recordação
- O Padrão Procurador
- O Padrão Estado

2.1.1 O Padrão Fachada (*Facade*)

2.1.1.1 Propósito

Fornece uma interface unificada para um conjunto de interfaces num subsistema. Fachada define uma interface de alto nível que torna o subsistema mais fácil de usar.

2.1.1.2 Problema

Você precisa usar apenas um subconjunto de um sistema complexo, ou você só precisa interagir com o sistema de uma maneira específica.

2.1.1.3 Solução

A Fachada apresenta uma nova interface para o cliente de um sistema existente usar.

2.1.1.4 Participantes e colaboradores

Ela apresenta uma interface simplificada para o cliente que torna o uso do subsistema mais fácil.

2.1.1.5 Consequências

A Fachada simplifica o uso do subsistema requerido. Entretanto, como a Fachada não é completa, algumas funcionalidades podem não está disponíveis para o cliente.

2.1.1.6 Implementação

Defina uma nova classe (ou classes) que tem a interface requerida. Faça esta classe usar o sistema existente.

2.1.1.7 Estrutura Genérica

A figura 24 mostra a estrutura do sistema antes e depois de aplicar o padrão da Fachada.

O padrão da Fachada pode ser usado para reduzir o número de objetos com os quais um objeto cliente tem de lidar, além da simplificação da interface.

2.1.2 O Padrão Adaptador (*Adapter*)

2.1.2.1 Propósito

Converte a interface de uma classe em uma outra interface esperada pelos clientes. Adaptador permite que classes trabalhem juntas, o que não seria possível por incompatibilidade de interfaces.

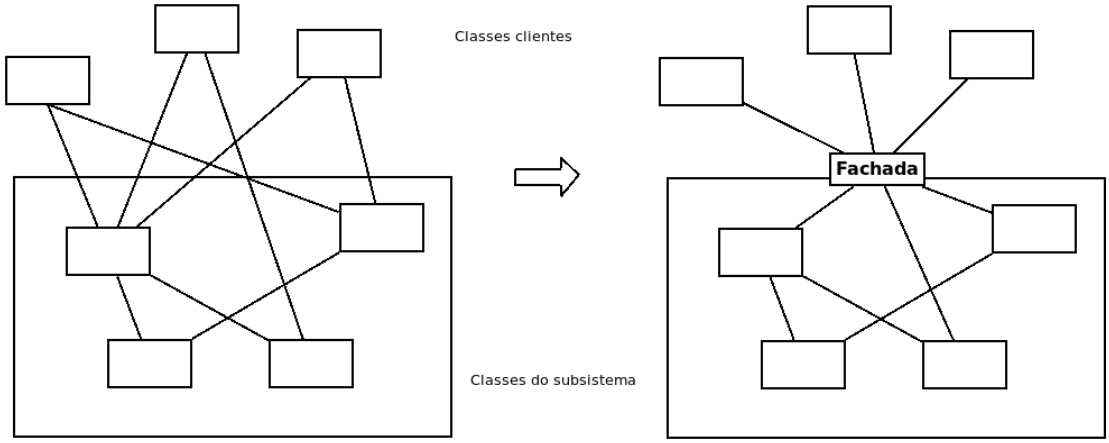

Figura 24 – Estrutura Genérica do Padrão Fachada

2.1.2.2 Problema

Um sistema tem os dados e o comportamento corretos, mas a interface errada. Tipicamente usado quando você tem de fazer algo derivado de uma classe abstrata.

2.1.2.3 Solução

O *Adaptador* fornece um embrulho (uma roupagem) com a interface desejada.

2.1.2.4 Participantes e colaboradores

O *Adaptador* adapta a interface de um *Adaptado* para casar com aquela do *Objetivo* do Adaptador (a classe de quem ela deriva). Isto permite ao *Cliente* usar o *Adaptado* como se ele fosse do tipo do *Objetivo*.

2.1.2.5 Consequências

O padrão Adaptador permite o uso de objetos pré-existentes em novas estruturas de classes sem ficar limitado às suas interfaces antigas.

2.1.2.6 Implementação

Contém a classe existente numa outra classe. A nova classe tem a interface requerida e chama os métodos pela classe contida.

2.1.2.7 Estrutura Genérica

A estrutura genérica do Padrão Adaptador é ilustrado na figura 49.

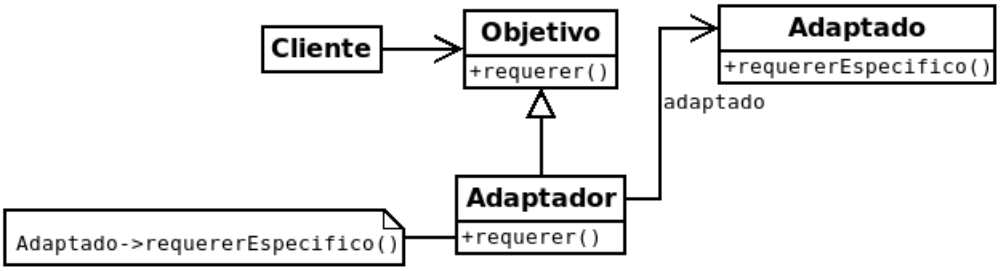

Figura 25 – Estrutura Genérica do Padrão Adaptador

Existem dois tipos de padrões de Adaptadores:

Padrão Adaptador de Objeto: O padrão de Adaptador, geralmente, exemplificado é o Adaptador de Objeto que consiste em um objeto adaptador contendo um objeto adaptado.

Padrão Adaptador de Classe: Um outro modo de implementar o padrão Adaptador é com herança múltipla. Neste caso, temos um padrão Adaptador de Classe.

É comum, projetistas inexperientes não saberem diferenciar os dois padrões: Fachada e Adaptador. (SHALLOWAY; TROTT, 2005) fornecem a seguinte tabela para compará-los:

Tabela 5 – Comparação entre os padrões Fachada e Adaptador

	Fachada	**Adaptador**
Há classes pré-existentes?	Sim	Sim
Há uma interface para a qual devemos projetar?	Não	Sim
Um objeto precisa ter comportamento polimórfico?	Não	Provavelmente
Uma interface mais simples é necessária?	Sim	Não

Fonte: (SHALLOWAY; TROTT, 2005)

2.1.2.8 Exemplo de Adaptador

A figura 26 mostra como uma classe *CirculoEspc* é adaptada para uma classe *Circulo*. Se o código da *CirculoEspc* fosse acessível, ela poderia herdar de *Circulo*, mas, se a classe faz parte de uma biblioteca externa proprietária, não será possível alterar a hierarquia de classes. Outra razão para a classe *CirculoEspc* não ser derivada de *Circulo* é que este tipo de projeto tem alto grau de acoplamento, herança tem alto grau de acoplamento, agregação mantêm um grau de acoplamento menor, tornando o projeto mais flexível para mudanças.

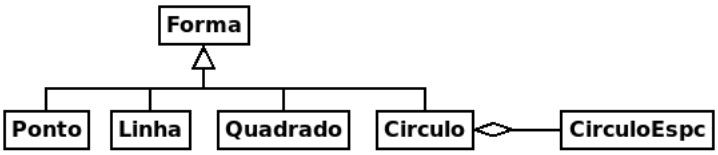

Figura 26 – Adaptação de CirculoEspc com Circulo

2.1.3 O Padrão Fabrica Abstrata (*Abstract Factory*)

O padrão da Fabrica Abstrata é um dos padrões de projeto que procuram abstrair o processo de instanciação. Este tipo de padrão ajuda a tornar um sistema independente de como seus objetos são criados, compostos e representados. Ele esconde como instâncias das classes são criadas e são postas juntas.

2.1.3.1 Propósito

Fornece uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar as suas classes concretas.

2.1.3.2 Problema

Famílias de objetos relacionados precisam ser instanciadas.

2.1.3.3 Solução

Coordena a criação de famílias de objetos. Fornece uma maneira para retirar as regras de como realizar a instanciação do objeto cliente que está usando estes objetos criados.

2.1.3.4 Participantes e Colaboradores

A FabricaAbstrata define a interface de como criar cada membro da família de objetos requeridos. Tipicamente, cada família é criada usando sua própria FabricaConcreta.

2.1.3.5 Consequências

O padrão isola as regras de quais objetos usar da lógica de como usar estes objetos.

2.1.3.6 Implementação

Defina uma classe abstrata que especifica quais objetos devem ser feitos. Implemente, então, uma classe concreta para cada família.

2.1.3.7 Estrutura Genérica

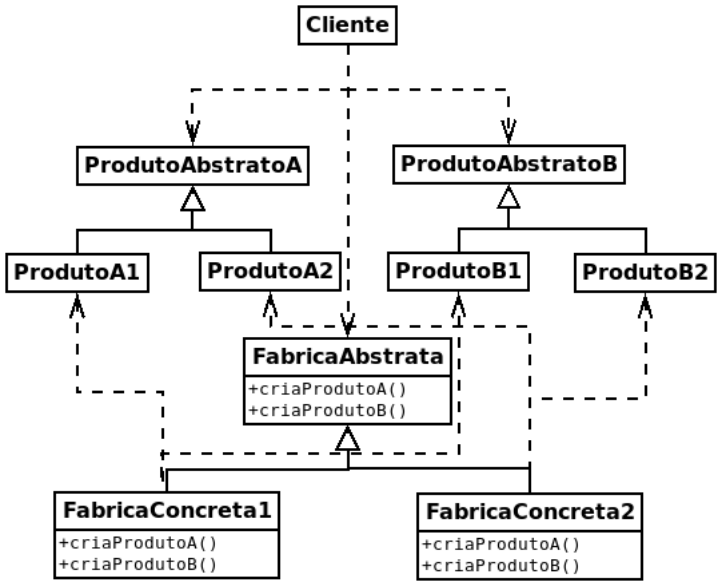

Figura 27 – Estrutura Genérica do Padrão Fabrica Abstrata

2.1.4 O Padrão Método Fabrica (*Factory Method*)

2.1.4.1 Propósito

Define uma interface para a criação de um objeto, mas deixa que uma subclasse decida qual classe instanciar. Adia a instanciação para as subclasses.

2.1.4.2 Problema

Uma classe precisa instanciar uma derivação de uma outra classe, mas não sabe qual. O *Factory Method* permite que uma classe derivada tome a decisão.

2.1.4.3 Solução

Uma classe derivada toma a decisão de qual classe instanciar e como.

2.1.4.4 Participantes e colaboradores

Produto é a interface do tipo de ojeto que a *Factory Method* cria. *Creator* é a interface que define a *Factory Method*.

2.1.4.5 Consequências

Clientes precisarão derivar a classe Creator para ter um *ProdutoConcreto*.

2.1.4.6 Implementação

Use um método na classe abstrata que é abstrato. O código da classe abstrata refere-se a este método quando precisa instanciar um objeto contido , mas ainda não sabe qual objeto particular é necessário.

2.1.4.7 Estrutura Genérica

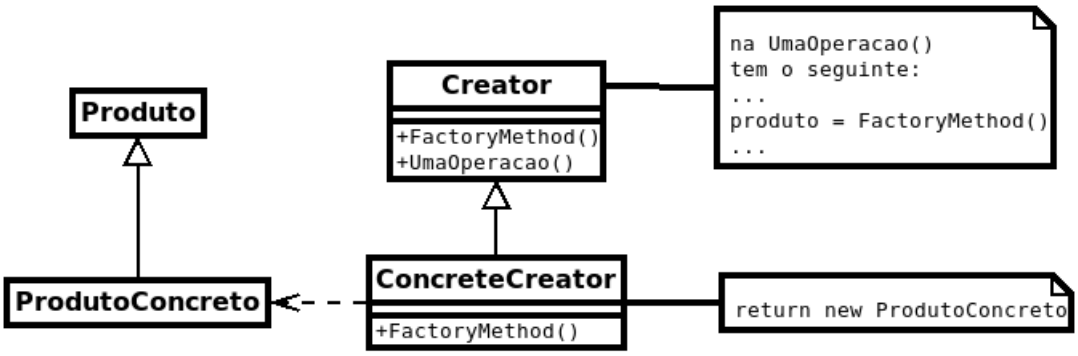

Figura 28 – Estrutura Genérica do padrão Método Fabrica.

O padrão Método Fabrica é normalmente usado na definição de *frameworks*. Isto porque frameworks existem num nível abstrato. Geralmente, eles não sabem e não deveriam se preocupar com a instanciação de objetos específicos. Eles precisam adiar as decisões sobre objetos específicos para usuários de *frameworks*.

2.1.5 O Padrão Observador (*Observer*)

2.1.5.1 Propósito

Define uma dependência um-para-muitos entre objetos tal que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

2.1.5.2 Problema

Um efeito colateral de particionar um um sistema numa coleção de classes cooperativas é a necessidade de manter a consistência entre objetos correlacionados. Você não deseja obter consistência aumentando o acoplamento entre as classes, porque isto reduz a reusabilidade delas.

2.1.5.3 Solução

Use o padrão **Observador** nas seguintes situações:

- Quando uma abstração tem dois aspectos, cada um dependente do outro. Encapsular estes aspectos em objetos diferentes permite que você modifique e reuse cada um deles de modo independente.

- Quando uma mudança num objeto requer a mudança em outros objetos e você não sabe quais objetos necessitam serem modificados.

- Quando um objeto deve ser capaz de notificar outros objetos sem ter de fazer suposições sobre os outros objetos.

2.1.5.4 Participantes e colaboradores

2.1.5.5 Consequências

2.1.5.6 Implementação

2.1.5.7 Estrutura Genérica

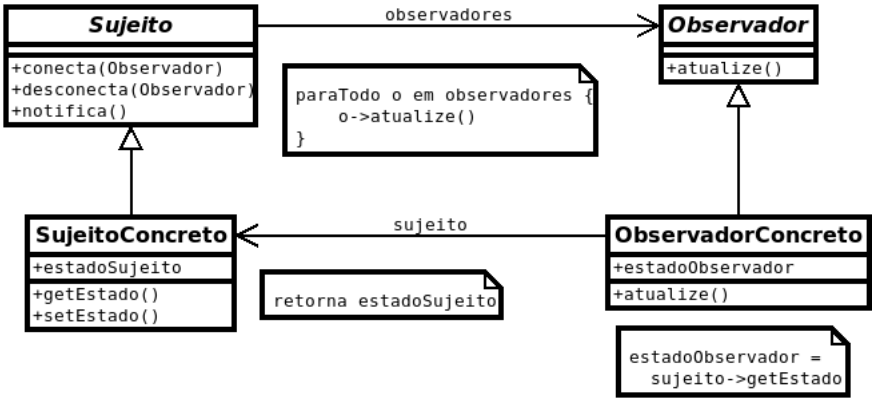

Figura 29 – Estrutura Genérica do Padrão Observador

2.1.6 O Padrão Composição (*Composite*)

2.1.6.1 Propósito

2.1.6.2 Problema

2.1.6.3 Solução

2.1.6.4 Participantes e colaboradores

2.1.6.5 Consequências

2.1.6.6 Implementação

2.1.6.7 Estrutura Genérica

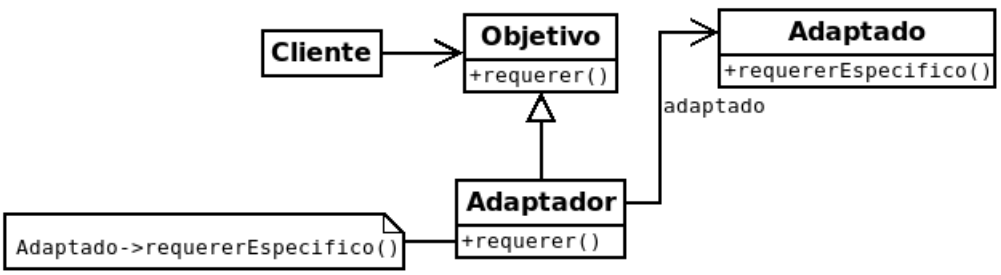

Figura 30 – Estrutura Genérica do Padrão Adaptador

2.1.7 O Padrão Iterador (*Iterator*)

2.1.7.1 Propósito

2.1.7.2 Problema

2.1.7.3 Solução

2.1.7.4 Participantes e colaboradores

2.1.7.5 Consequências

2.1.7.6 Implementação

2.1.7.7 Estrutura Genérica

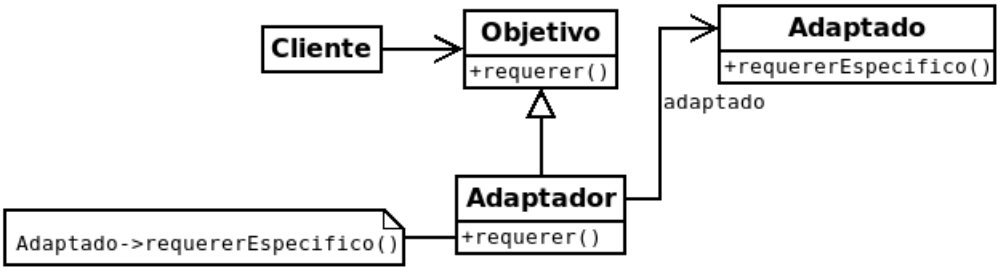

Figura 31 – Estrutura Genérica do Padrão Adaptador

2.1.8 O Padrão Visitante (*Visitor*)

2.1.8.1 Propósito

2.1.8.2 Problema

2.1.8.3 Solução

2.1.8.4 Participantes e colaboradores

2.1.8.5 Consequências

2.1.8.6 Implementação

2.1.8.7 Estrutura Genérica

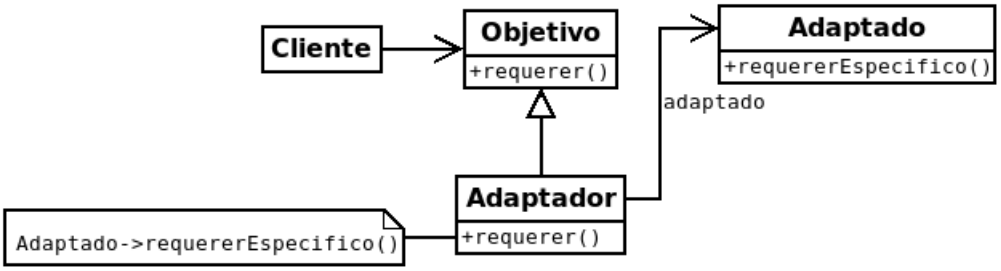

Figura 32 – Estrutura Genérica do Padrão Adaptador

2.1.9 O Padrão Estratégia (*Strategy*)

2.1.9.1 Propósito

Permite que você use regras de negócios ou algoritmos diferentes dependendo do contexto nos quais eles ocorrem.

2.1.9.2 Problema

A seleção de um algoritmo que precisa ser aplicado depende do cliente que faz a chamada ou dos dados sobre os quais ele age. Se você só tiver uma regra que não muda, você não precisa do padrão Estratégia.

2.1.9.3 Solução

Separa a seleção do algoritmo da implementação do algoritmo. Permite que a seleção seja feita baseada no contexto.

2.1.9.4 Participantes e colaboradores

Estratégia especifica como os diferentes algoritmos são usados. *EstratégiasConcretas* implementa estes diferentes algoritmos. *Contexto* usa uma *EstratégiaConcreta* especifica com uma referência do tipo *Estratégia*. Estratégia e Contexto interagem para implementar o algoritmo escolhido. (As vezes, *Estratégia* deve pedir *Contexto*.) O *Contexto* direciona pedidos dos seus clientes para *Estratégia*.

2.1.9.5 Consequências

O padrão Estratégia define uma família de algoritmos. O uso de condicionais e switches (casos) podem ser eliminados. Você pode invocar todos os algoritmos do mesmo jeito. (Eles devem todos ter a mesma interface.) A interação entre *EstratégiasConcretas* e *Contexto* pode requerer a adição de métodos que obtêm o estado do *Contexto*.

2.1.9.6 Implementação

2.1.9.7 Estrutura Genérica

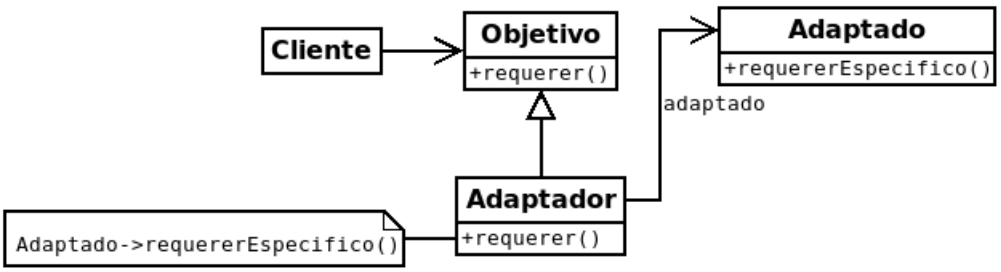

Figura 33 – Estrutura Genérica do Padrão Adaptador

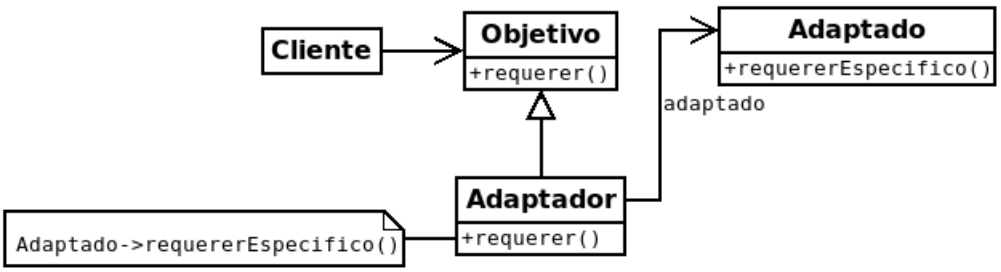

Figura 34 – Estrutura Genérica do Padrão Adaptador

2.1.10 O Padrão Decorador (*Decorator*)

2.1.10.1 Propósito

2.1.10.2 Problema

2.1.10.3 Solução

2.1.10.4 Participantes e colaboradores

2.1.10.5 Consequências

2.1.10.6 Implementação

2.1.10.7 Estrutura Genérica

2.1.11 O Padrão Método Padrão (*Template Method*)

2.1.11.1 Propósito

2.1.11.2 Problema

2.1.11.3 Solução

2.1.11.4 Participantes e colaboradores

2.1.11.5 Consequências

2.1.11.6 Implementação

2.1.11.7 Estrutura Genérica

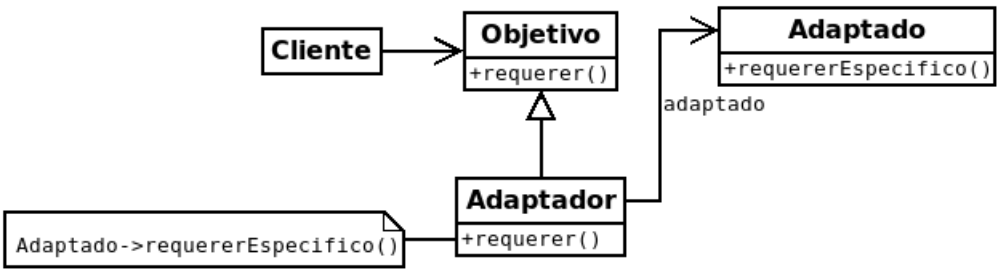

Figura 35 – Estrutura Genérica do Padrão Adaptador

2.1.12 O Padrão *Singleton*

2.1.12.1 Propósito

2.1.12.2 Problema

2.1.12.3 Solução

2.1.12.4 Participantes e colaboradores

2.1.12.5 Consequências

2.1.12.6 Implementação

2.1.12.7 Estrutura Genérica

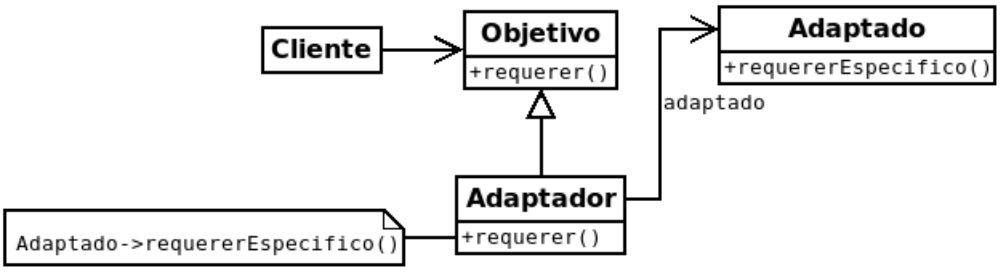

Figura 36 – Estrutura Genérica do Padrão Adaptador

2.1.13 O Padrão Trava Dupla (*Lock Double Checked*)

2.1.13.1 Propósito

2.1.13.2 Problema

2.1.13.3 Solução

2.1.13.4 Participantes e colaboradores

2.1.13.5 Consequências

2.1.13.6 Implementação

2.1.13.7 Estrutura Genérica

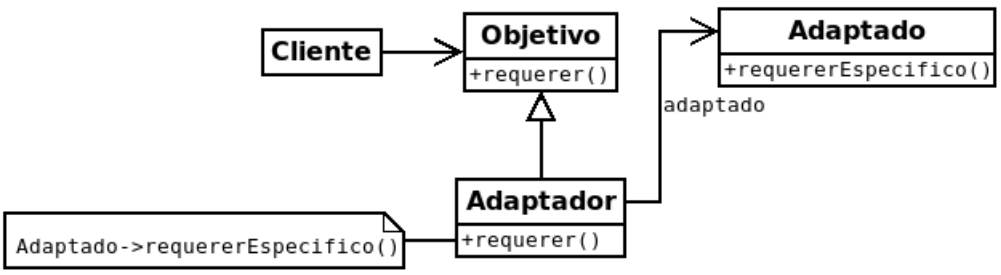

Figura 37 – Estrutura Genérica do Padrão Adaptador

2.1.14 O Padrão Ponte (*Bridge*)

2.1.14.1 Propósito

2.1.14.2 Problema

2.1.14.3 Solução

2.1.14.4 Participantes e colaboradores

2.1.14.5 Consequências

2.1.14.6 Implementação

2.1.14.7 Estrutura Genérica

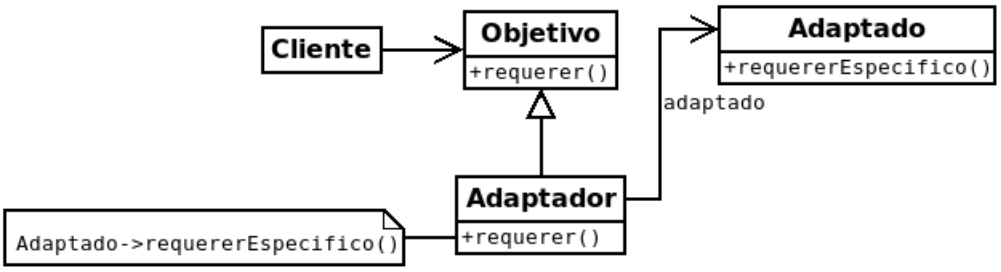

Figura 38 – Estrutura Genérica do Padrão Adaptador

2.1.15 O Padrão Construtor (*Builder*)

2.1.15.1 Propósito

2.1.15.2 Problema

2.1.15.3 Solução

2.1.15.4 Participantes e colaboradores

2.1.15.5 Consequências

2.1.15.6 Implementação

2.1.15.7 Estrutura Genérica

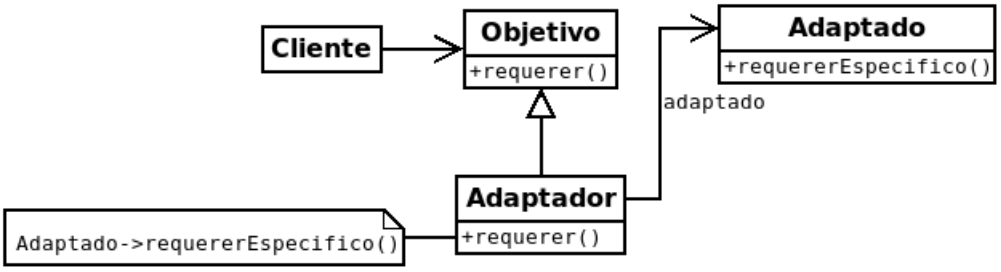

Figura 39 – Estrutura Genérica do Padrão Adaptador

2.1.16 O Padrão Protótipo (*Prototype*)

2.1.16.1 Propósito

2.1.16.2 Problema

2.1.16.3 Solução

2.1.16.4 Participantes e colaboradores

2.1.16.5 Consequências

2.1.16.6 Implementação

2.1.16.7 Estrutura Genérica

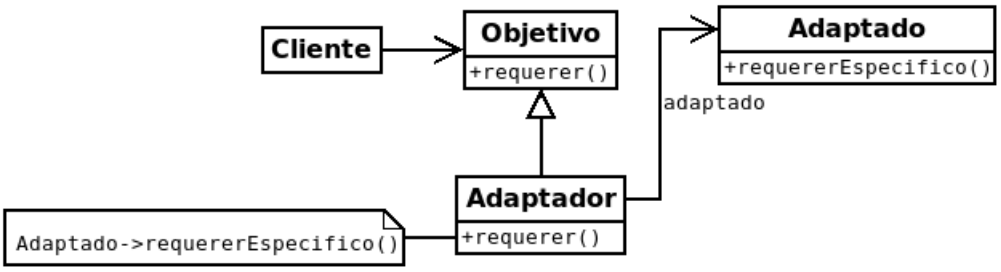

Figura 40 – Estrutura Genérica do Padrão Adaptador

2.1.17 O Padrão Repositório de Objetos (*Object Pool*)

2.1.17.1 Propósito

2.1.17.2 Problema

2.1.17.3 Solução

2.1.17.4 Participantes e colaboradores

2.1.17.5 Consequências

2.1.17.6 Implementação

2.1.17.7 Estrutura Genérica

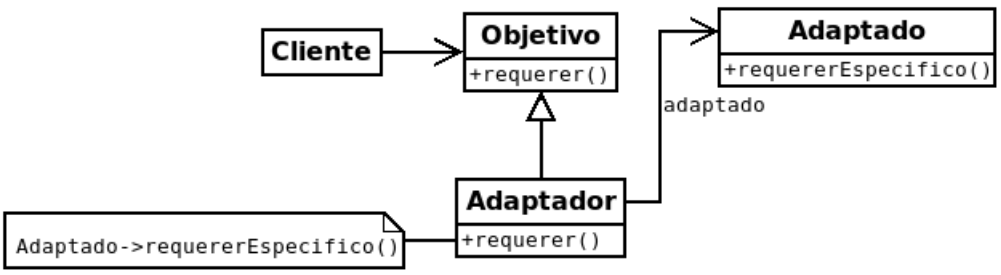

Figura 41 – Estrutura Genérica do Padrão Adaptador

2.1.18 O Padrão Cadeia de Responsabilidade (*Chain of Responsability*)

2.1.18.1 Propósito

2.1.18.2 Problema

2.1.18.3 Solução

2.1.18.4 Participantes e colaboradores

2.1.18.5 Consequências

2.1.18.6 Implementação

2.1.18.7 Estrutura Genérica

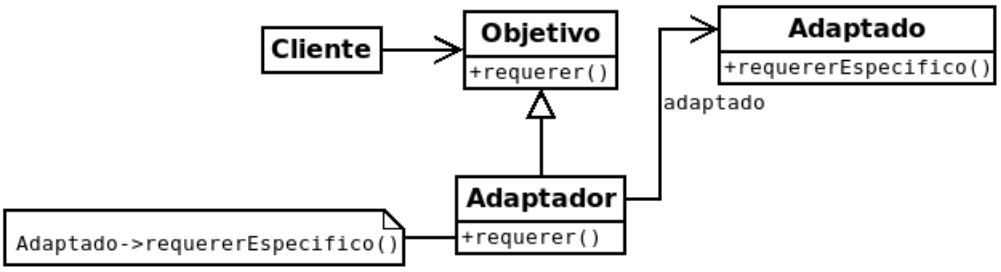

Figura 42 – Estrutura Genérica do Padrão Adaptador

2.1.19 O Padrão Comando (*Command*)

2.1.19.1 Propósito

2.1.19.2 Problema

2.1.19.3 Solução

2.1.19.4 Participantes e colaboradores

2.1.19.5 Consequências

2.1.19.6 Implementação

2.1.19.7 Estrutura Genérica

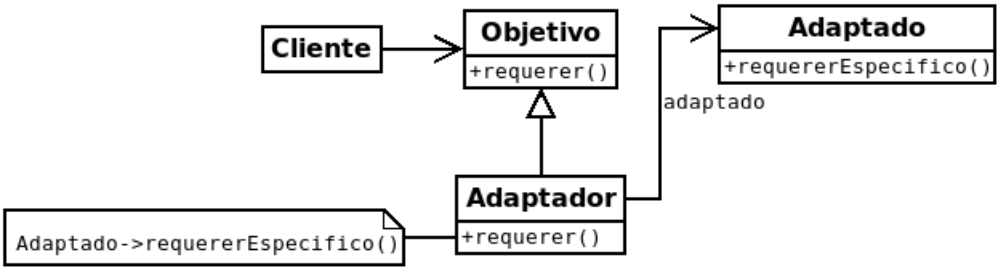

Figura 43 – Estrutura Genérica do Padrão Adaptador

2.1.20 O Padrão Peso Pena (*Flyweight*)

2.1.20.1 Propósito

2.1.20.2 Problema

2.1.20.3 Solução

2.1.20.4 Participantes e colaboradores

2.1.20.5 Consequências

2.1.20.6 Implementação

2.1.20.7 Estrutura Genérica

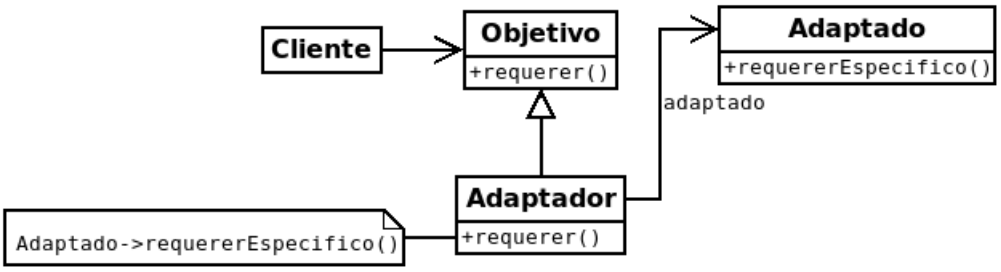

Figura 44 – Estrutura Genérica do Padrão Adaptador

2.1.21 O Padrão Interpretador (*Interpreter*)

2.1.21.1 Propósito

2.1.21.2 Problema

2.1.21.3 Solução

2.1.21.4 Participantes e colaboradores

2.1.21.5 Consequências

2.1.21.6 Implementação

2.1.21.7 Estrutura Genérica

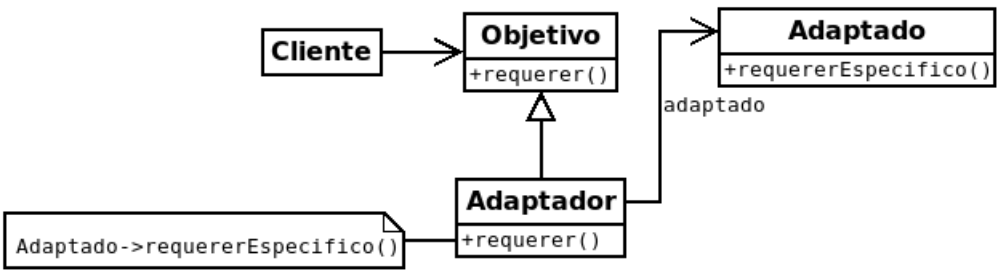

Figura 45 – Estrutura Genérica do Padrão Adaptador

2.1.22 O Padrão Mediador (*Mediator*)

2.1.22.1 Propósito

2.1.22.2 Problema

2.1.22.3 Solução

2.1.22.4 Participantes e colaboradores

2.1.22.5 Consequências

2.1.22.6 Implementação

2.1.22.7 Estrutura Genérica

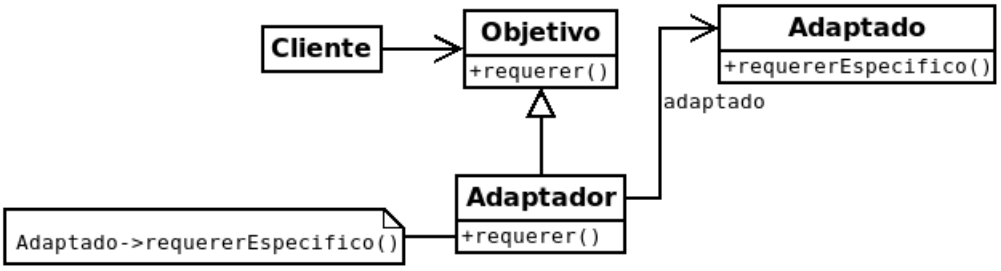

Figura 46 – Estrutura Genérica do Padrão Adaptador

2.1.23 O Padrão Recordação (*Memento*)

2.1.23.1 Propósito

2.1.23.2 Problema

2.1.23.3 Solução

2.1.23.4 Participantes e colaboradores

2.1.23.5 Consequências

2.1.23.6 Implementação

2.1.23.7 Estrutura Genérica

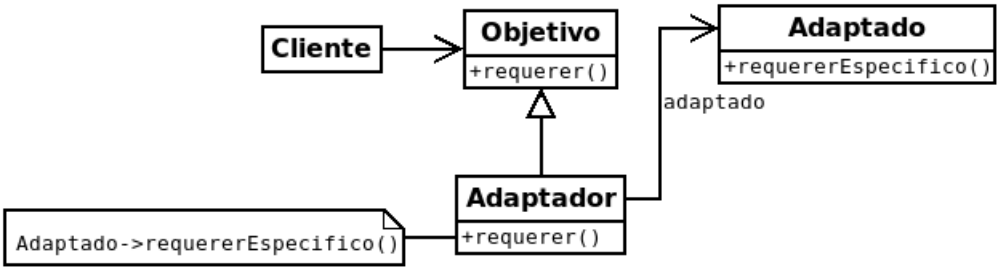

Figura 47 – Estrutura Genérica do Padrão Adaptador

2.1.24 O Padrão Procurador (*Proxy*)

2.1.24.1 Propósito

2.1.24.2 Problema

2.1.24.3 Solução

2.1.24.4 Participantes e colaboradores

2.1.24.5 Consequências

2.1.24.6 Implementação

2.1.24.7 Estrutura Genérica

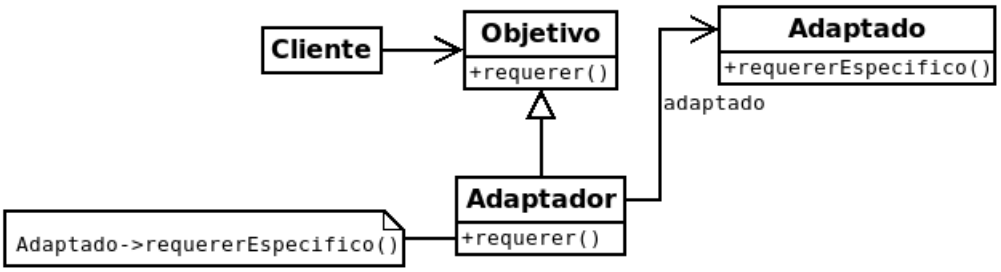

Figura 48 – Estrutura Genérica do Padrão Adaptador

2.1.25 O Padrão Estado (*State*)

2.1.25.1 Propósito

2.1.25.2 Problema

2.1.25.3 Solução

2.1.25.4 Participantes e colaboradores

2.1.25.5 Consequências

2.1.25.6 Implementação

2.1.25.7 Estrutura Genérica

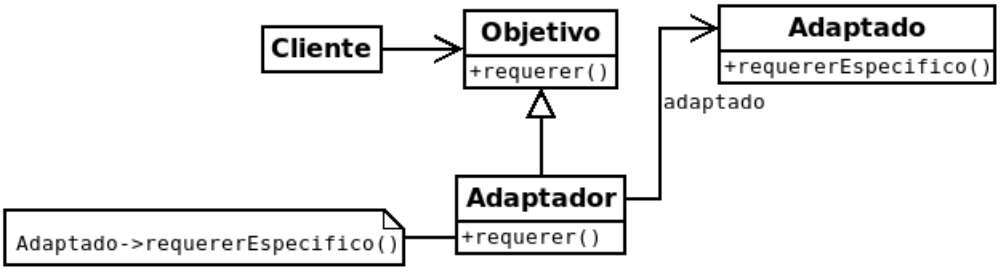

Figura 49 – Estrutura Genérica do Padrão Adaptador

2.2 O Processo de Pensar em Padrões

De acordo com Gamma et al. (1994), o processo de pensar em padrões segue os passos:

1. **Identifique os padrões:** Encontre os padrões no domínio do problema;

2. **Analise e aplique os padrões:** Para o conjunto de padrões a ser analisado, realize os passos a) até d):

    a) **Ordene os padrões pelo contexto de criação:** Ordene os padrões de acordo com como eles criam contexto para cada um dos outros padrões. A ideia é a de que um padrão vai criar contexto para um outro, não que dois padrões vão criar contexto um para o outro.

    b) **Selecione cada padrão e expanda o projeto:** Usando sua ordenação, selecione o próximo padrão da lista e use-o para um projeto conceitual de alto nível.

    c) **Identifique padrões adicionais:** Identifique qualquer padrão adicional que surja durante a sua análise. Adicione-o ao conjunto de padrões a serem analisados.

    d) **Repita:** Repita para o conjunto de padrões que ainda não foram integrados ao projeto conceitual.

3. **Adicione detalhes:** Adicione os detalhes a medida que se tornarem necessários ao projeto. Expanda as definições das classes e dos métodos.

(SHALLOWAY; TROTT, 2005) dizem que para aplicar os padrões aos seus projetos é necessário que conheça bem o domínio do problema. Além disso, pensar em padrões nem sempre é aplicável, mas análise de comunalidade e variabilidade é geralmente mais usável. Os autores mostram como aplicar os passos no projeto de um CAD/CAM.

3 Metodologias de desenvolvimento

Existem diversas metodologias para o projeto de software OO, veremos neste capítulo uma rápida introdução a algumas destas metodologias. As chamadas metodologias de desenvolvimento visam determinar a arquitetura do software e encontrar as classes/objetos que resolvem um determinado problema. Estas metodologias são, geralmente, selecionadas pelas empresas de desenvolvimento e os programadores/projetistas devem se adaptar à metodologia da empresa. Os gerentes de desenvolvimento que selecionam as metodologias têm diversos critérios para realizar a escolha: experiência da equipe, disponibilidade de ferramentas dentro da empresa, requerimentos dos clientes finais, ... Algumas vezes, a escolha se faz por motivos não totalmente objetivos. Não vamos aqui discutir os méritos e deméritos das metodologias para não entrar em batalhas, muitas vezes infindáveis e infrutíferas.

(AHMED; UMRYSH, 2002) diz que um processo de desenvolvimento de software fornece orientação sobre como desenvolver software com sucesso. Essa orientação pode abranger todo o espectro de atividades associadas ao desenvolvimento do software. O processo pode se manifestar sob a forma de abordagens comprovadas, melhores práticas, diretrizes, técnicas, sequenciamento de atividades, etc.

Seja formal ou informal, o processo de desenvolvimento de software utilizado tem impacto profundo no sucesso de um projeto de software. Uma abordagem de tentativa e erro pode funcionar bem para um pequeno projeto, mas pode levar ao caos em um grande projeto e impactar o cronograma geral. Da mesma forma, um processo de desenvolvimento de software burocrático pode levar à frustração e ao atolamento até mesmo a melhor das equipes.

(SHALLOWAY; TROTT, 2005) diz que frequentemente os projetistas novatos em OO são aconselhados a olhar para o domínio do problema e "identificar os substantivos presentes" e criar objetos representando-os e encontrar os verbos relacionando estes substantivos (isto é, as ações deles) e implementá-los através de métodos dos objetos. Este processo focado em substantivos (nomes das coisas) e verbos leva, tipicamente a hierarquias de classes maiores do que as desejáveis. Eles recomendam a análise de coisas em comum, comunalidade, e coisas que variam, variabilidade, como a principal ferramenta para a criação de objetos. Os conceitos de comunalidade e variabilidade foram apresentados por Couplien[1].

A figura 50 mostra as relações entre a análise de comunalidades e variabilidades, perspectivas e classes abstratas. Ao olhar o que os objetos devem fazer (perspectiva conceitual), determinamos como chamá-los (perspectiva de especificação) e obtemos as classes abstratas e seus métodos. Ao implementar estas classes, garante-se que a API fornece informação suficiente para uma implementação adequada e um bom desacoplamento.

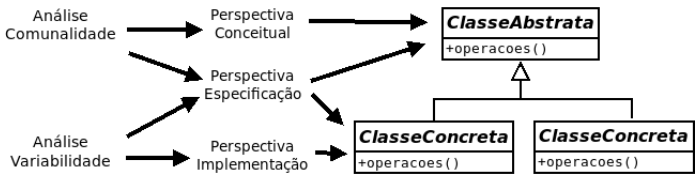

Figura 50 – Relação entre Análise de Comunalidade e de Variabilidade, Perspectivas e Classes abstratas

Fonte: (SHALLOWAY; TROTT, 2005)

A tabela 6 apresenta alguns conceitos e termos usados em desenvolvimento de Software.

[1] Couplien, J., *Multi-Paradigm Design for C++*, Boston: Addison-Wesley, 1998, pp. 60, 64.

Tabela 6 – Conceitos e Termos de Desenvolvimento de Software

Termo	Descrição
Caso de Uso	Um caso de uso é uma metodologia usada na análise do sistema para identificar, esclarecer e organizar os requisitos do sistema. Um caso de uso é composto por um conjunto de possíveis sequências de interações entre sistemas e usuários em um ambiente específico e relacionadas a um objetivo específico. O caso de uso deve conter todas as atividades do sistema que tenham importância para os usuários [2].
Desenvolvimento em Cascata	Processo de desenvolvimento que segue estritamente as etapas: Levantamento de requisitos, Análise e Projeto do SW, Codificação, Teste e Manutenção.
Estória de Usuário	Uma história de usuário é uma ferramenta usada no desenvolvimento de software Ágil para capturar uma descrição de um recurso de software de uma perspectiva do usuário final. A história do usuário descreve o tipo de usuário, o que eles querem e por quê. Uma história do usuário ajuda a criar uma descrição simplificada de um requisito[3].
Framework	Quadro de trabalho, em tradução ao pé da letra. Um framework é uma coleção de programas que facilitam o desenvolvimento de uma aplicação. De acordo com (AHMED; UMRYSH, 2002), existem dois tipos de framewoks: um baseado em biblioteca e outro que fornece uma base a ser extendida pelo programador.
Metodologia de Desenvolvimento de Software	Processo de desenvolvimento de software baseado em práticas bem definidas.
Metodologia Ágil	Processo de desenvolvimento de Software que procura reduzir o ciclo de entrega de incremental do software, privilegiando a interação com o cliente e a programação.
Processo de Desenvolvimento	Técnicas e procedimentos práticos para o desenvolvimento de Software.
RUP	Processo Unificado da Rational - *Rational Unified Process.*
Scrum	Uma metodologia ágil.
XP	eXtreme Programming, uma metodologia ágil.

Fonte: (AHMED; UMRYSH, 2002) com alguns acréscimos.

A exposição das metodologias começa com o processo unificado que de certa maneira representou o fim da guerra das metodologias, não que ela seja a metodologia vencedora. Apenas, o processo unificado ofuscou as outras e não temos mais uma disputa tão pronunciada de metodologias deste que ela foi publicada, ela é o resultado da unificação de metologias pelos 4 amigos: Philippe Kruchten, Ivar Jacobson, Jim Rumbaugh e Grady Booch. Conforme veremos, o principal representante da metodologia é chamada de RUP, Rational Unified Process, e varições dela são usadas por muitas empresas. Inicialmente, a metodologia era muito centrada na criação de documentos e o desenvolvimento do software só ocorria depois da escrita deles. Alguns desenvolvedores criaram metodologias chamadas de ágeis que pretendem reduzir a burocracia antes da programação. 15 destes desenvolvedores lançaram o Manifesto pelo Desenvolvimento de Software Ágil, (Beck et al. (2018)), eles enunciaram 12 princípios básicos:

1. Nossa maior prioridade é satisfazer o cliente através da entrega antecipada e contínua de software

[2] https://searchsoftwarequality.techtarget.com/definition/use-case
[3] https://searchsoftwarequality.techtarget.com/definition/user-story, acessado em 12/07/2019

valioso.

2. Mudanças de requisitos são bem-vindas, mesmo nas etapas finais do desenvolvimento. Processos ágeis aproveitam mudanças para oferecer vantagem competitiva ao cliente.
3. Entregar software de trabalho com frequência, de algumas semanas a alguns meses, com uma preferência para a escala de tempo mais curta.
4. Clientes e desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte de que precisam e confie neles para fazer o trabalho.
6. O método mais eficiente e eficaz de transmitir informações para e dentro de uma equipe de desenvolvimento é conversa cara a cara.
7. O software de trabalho é a principal medida de progresso.
8. Processos ágeis promovem o desenvolvimento em ritmo sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
9. Atenção contínua à excelência técnica e um bom projeto aumentam a agilidade.
10. Simplicidade é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz, em seguida, se sintoniza e ajusta seu comportamento de acordo.

Veremos uma rápida introdução à metodologia Scrum na seção 3.2 para exemplificar as metodologias ágeis. E concluiremos o capítulo citando outras metologias ágeis.

3.1 Processo Unificado da Rational - RUP

O Processo Unificado da Rational - IBM é um framework de processo para o desenvolvimento de software. O processo é iterativo e possui 4 fases distintas com 6 atividades principais de desenvolvimento representadas na figura 51. A figura não representa 3 atividades complementares.

As fases são:

1. Incepção
2. Elaboração
3. Construção
4. Transição

As atividades de desenvolvimento são:

1. Modelamento do negócio
2. Requisitos
3. Análise e Projeto
4. Implementação
5. Teste

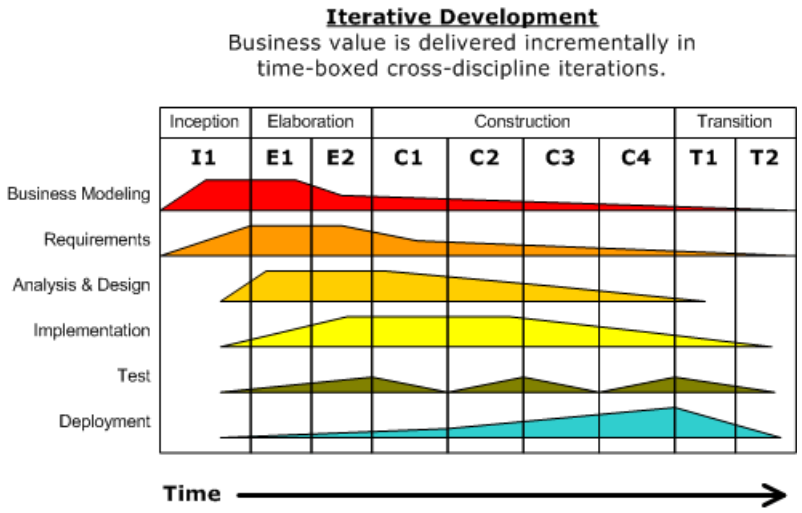

Figura 51 – Diagrama das etapas do RUP

Fonte: Wikipedia (2018a)

6. Distribuição

As atividades complementares são:

1. Configuração e Gerenciamento de Mudanças
2. Gerenciamento do Projeto
3. Ambiente

As atividades são desenvolvidas em etapas iterativas, ao fim de cada etapa iterativa existe a entrega de produtos do desenvolvimento, as iterações são ilustradas na figura por I1, E1, E2, C1, C2, ..., CN, T1 e T2.

3.2 Breve Introdução ao Scrum

3.2.1 O Que É Scrum?

Scrum é um framework projetado para ajudar pequenas equipes unidas a desenvolver produtos complexos. Resultado do trabalho de um punhado de engenheiros de software trabalhando juntos no final do século XX, scrum ganhou força no setor tecnológico, mas não é necessariamente técnico e pode ser facilmente adaptado para outras indústrias. Os autores Sims e Johnson (2012) dizem que você pode adaptá-lo para construir uma ratoeira melhor, ou para administrar a divisão de *marketing* de uma companhia de cachorrinhos de estimação, ou até para escrever um livro, como eles fizeram.

Uma equipe scrum consiste, tipicamente, de cinco a nove pessoas que trabalham juntas em curtos intervalos de intensa atividade chamadas de arrancadas (*sprints*), com bastante tempo embutido para revisão e reflexão. Um dos mantras de scrum é “inspecionar e adaptar” e equipes scrum são caracterizadas por um foco intenso na melhoria contínua do processo delas e do produto. Nesta seção veremos uma introdução muito rápida da terminologia do scrum: os vários papéis, artefatos e eventos que ocorrem num ciclo de arrancada (*sprint cycle*).

3.2.2 Papéis

O Scrum conhece três papéis distintos para os membros da equipe:

Proprietário do Produto O proprietário do produto (*product owner*) é responsável pela maximização do retorno que o negócio traz ao investimento (ROI - Return-Of-Investment). Ele é o único que controla a ordem, a prioridade, da lista de pendências (*backlog itens*) da equipe. Ele garante que a equipe entende totalmente os requisitos. Ele registra os requisitos na forma de estórias de usuário (*user stories*), por exemplo: "Como um <papel - tipo de usuário>, quero <uma característica>, de modo que possa <fazer algo>" e coloca-as na lista de pendências. Cada uma destas estórias de usuário, quando completada, aumentará o valor do produto.

Mestre Scrum O Mestre Scrum age como um *coach* (treinador), guiando a equipe para níveis mais altos de coesão, auto-organização e desempenho. Enquanto a equipe entrega o produto, o Mestre Scrum entrega uma equipe de alto desempenho e auto-organizada. O Mestre Scrum ajuda a equipe a aprender e aplicar scrum e outras práticas ágeis relacionadas. O mestre scrum está sempre disponível para ajudar a equipe na remoção de obstáculos que estejam bloqueando a execução do trabalho. O mestre scrum não é o chefe, ele é um membro da equipe que se destaca pelo seu conhecimento e suas responsabilidades.

Membro de Equipe Equipes scrum de alto desempenho são altamente colaborativas, elas também são auto-organizadas. Os membros da equipe têm total autoridade sobre como o trabalho é realizado. A equipe decide por si quais são as ferramentas a serem usadas e quais membros irão trabalhar em quais tarefas. Se o negócio precisa de estimativas de cronograma, são os membros da equipe que criam as estimativas. Uma equipe scrum deve possuir todas as capacitações necessárias para criar um produto entregável. Ocasionalmente, pode ser que um membro tenha de trabalhar fora da sua especialidade para que um item da lista de pendências (uma estória de usuário) saia do estado "progredindo" para "feito". Esta é uma mudança de mentalidade do "faço meu serviço" para "faço o serviço".

O proprietário do produto:

- detém a visão do produto;
- representa os interesses do negócio;
- representa os clientes (*stockholders*);
- é dono da lista de pendências do produto (*product backlog*);
- ordena, prioriza, os itens da lista de pendências do produto;
- cria critérios de aceitação para os itens da lista de pendências do produto; e
- está disponível para responder às dúvidas dos outros membros da equipe.

O papel do Mestre Scrum é ser:

- especialista e orientador de scrum;
- treinador;
- quebrador de barreiras; e
- facilitador.

O papel de Membro da Equipe:

- é ser responsável por completar as estórias do usuário para incrementar o valor do produto;
- se auto-organizar para conseguir que todo o trabalho necessário seja feito;

- criar suas próprias estimativas de cronogramas;
- decidir "como fazer o trabalho"; e
- fugir do pensamento "não é meu serviço".

3.2.3 Artefatos do Scrum

3.2.3.1 A Lista de Pendências do Produto (*Product Backlog*)

A lista de pendências do produto é uma lista acumulativa de itens entregáveis desejados para o produto. Isto inclui características, consertos de erros (*bug fixes*), alterações na documentação e qualquer outra coisa que seja significativa e valiosa para produzir. Embora pendência seja um termo correto para um item da lista, as equipes de scrum preferem usar o termo estória de usuário para reforçar a noção de que construímos produtos para satisfazer as necessidades dos usuários.

Cada item da lista deve incluir as seguintes informações:

- Quais usuários vão se beneficiar da estória;
- Uma rápida descrição da funcionalidade desejada (o que precisa ser construído);
- A razão para a estória ser preciosa (por que precisamos realizá-la);
- Uma estimativa de quanto trabalho será necessário para realizar a estória; e
- Critérios de aceitação que ajudarão a saber se a implementação está correta.

3.2.3.2 A Lista de Pendências da Arrancada (*Sprint Backlog*)

A lista de pendências da arrancada é a lista de coisas a fazer na arrancada, período fixo de tempo de trabalho. Diferente da lista de pendências do produto, ela tem tempo de vida finito: a duração da arrancada. Ela inclui todas as estórias e tarefas associadas que a equipe se compromete a realizar nesta arrancada. Estórias são entregas e podem ser pensadas como unidades de valor. Tarefas são coisas que precisam ser feitas, para entregar as estórias, logo, tarefas podem ser vistas como unidades de trabalho. Uma estória é algo que uma equipe entrega; Uma tarefa é um pedaço de trabalho que alguém faz. Cada estória normalmente requer muitas tarefas.

3.2.3.3 Gráficos de Queimada

Um gráfico de queimada (*burn chart*) nos mostra a relação entre o tempo e o escopo. O tempo está no eixo horizontal X e o escopo está no eixo vertical Y. Um gráfico de queimada nos mostra quanto de escopo uma equipe conseguiu realizar num período de tempo. Cada vez que algo é completado, a linha no gráfico se move para cima. Um gráfico de queimada nos mostra o que falta fazer. Em geral, esperamos que o trabalho restante diminua com o tempo à medida que a equipe termina tarefas. Às vezes, o trabalho restante muda de repente, quando escopo é adicionado ou retirado. Estes eventos aparecem como linhas verticais no gráfico de queimada.

3.2.3.4 Quadro de Tarefas

Para todos os integrantes da equipe Scrum acompanharem o andamento do trabalho, um quadro de tarefas é colocado numa sala usada por todos os integrantes. Isto evita que alguma parte importante do trabalho seja esquecida.

O quadro de tarefas mais simples tem três colunas: A fazer, Fazendo e Terminada. As tarefas são deslocadas através das colunas do quadro, fornecendo visibilidade para quais tarefas foram terminadas, quais estão sendo feitas e quais ainda não começaram. Esta visibilidade ajuda a equipe a ver a situação atual e se adaptar conforme a necessidade. O quadro também ajuda os clientes a verem o progresso que a equipe está fazendo. Com a ideia de usar os artefatos mais simples, o quadro é uma superfície

onde podemos colar *post-it*s com as tarefas escritas neles. Estes post-its são colocados nas 3 colunas conforme a evolução das atividades.

Definição de Terminada
Terminada é uma palavra fantástica, quando a equipe consegue que uma estória de usuário seja terminada, é hora de festejar. Mas, às vezes, há uma certa confusão sobre o que exatamente *terminada* significa. Um programador pode dizer que alguma coisa está terminada quando o código está escrito. Um testador pode pensar que terminada significa que todos os testes passaram. O pessoal de operações pode pensar que terminada significa que os programas foram carregados nos servidores de produção. Uma pessoa do comercial pode pensar que terminada significa que podemos vender aos clientes e que está pronta para uso. Esta confusão sobre o que *terminada* significa pode causar problemas, quando, por exemplo, o vendedor pergunta por que a equipe ainda está trabalhando numa estória que o programador disse estar terminada há duas semanas. Para evitar confusão, as boas equipes de scrum devem ter sua definição de como a palavra *terminado* se aplica a uma estória de usuário. Elas decidem juntas que coisas precisam ser concluídas antes da equipe declarar que uma estória foi terminada. A definição da equipe pode incluir coisas como: código escrito, código revisto, aprovação nos testes unitários, documentação escrita, assinatura do proprietário do produto, ... Esta é a lista de coisas, que a equipe concorda em fazer sempre antes de declarar que uma estória está terminada, é a definição da equipe do que significada estória *terminada*. A equipe deve imprimir a definição de terminada como uma lista a ser checada próxima do quadro de tarefas. Quando a equipe acha que uma *estória* está *terminada*, verifica-se que todos os itens da lista foram concluídos.

3.2.4 O Ciclo da Arrancada (*Sprint Cycle*)

O ciclo da arrancada consiste de várias reuniões, chamadas de cerimônias, para satisfazer as pessoas que não gostam da palavra reunião:

- planejamento da arrancada;
- scrum diário;
- hora da estória;
- revisão da arrancada e
- retrospectiva.

É uma questão de ritmo

O ciclo da arrancada é o ritmo fundamental do processo scrum. Como quer que você chame seu período de desenvolvimento: Uma arrancada (*sprint*), um ciclo, ou uma iteração. Você está sempre falando da mesma coisa: Um *período fixo* de tempo no qual você mastiga pedaços do seu projeto e termina-os antes de morder mais. No final da arrancada, você mostrará algum software funcionando.

Quanto mais frequente forem as entregas da equipe de incrementos do produto, maior a liberdade do negócio para decidir quando e o que entregar. Observe que existem duas decisões separadas a serem tomadas aqui:

O produto tem potencial para ser comercializado? Isto é, a qualidade é boa o suficiente para que seja comercializado? Todas as estórias atuais terminaram? Esta é uma decisão da equipe.

Tem sentido comercializar o que temos no momento? Existe valor agregado suficiente para levar o produto atual para o mercado? Esta é uma decisão de negócios.

Adicionalmente, quanto maior a frequência com que a equipe produz incrementos entregáveis do produto, maior a frequência de retornos para a equipe, o que alimenta o importante ciclo de

inspecionar-e-adaptar. Quanto mais curto o ciclo da arrancada, com maior frequência a equipe produz valor para o negócio.

Por volta do inicío dos anos 2010, as equipes de scrum trabalhavam com arrancadas de duas semanas e muitas equipes começavam a trabalhar com arrancadas de uma semana. Muitos dos escritos originais de scrum assumiam arrancadas com um duração de um mês e naquela época, isto parecia curto.

A tabela da figura 52 esboça o mapeamento das cerimônias que você deve agendar para uma arrancada de uma semana. A duração das cerimônias são uma sugestão inicial, as equipes devem adaptar para suas próprias características.

Agenda diária para uma arrancada de uma semana				
Segunda	Terça	Quarta	Quinta	Sexta
Planejamento	Levante	Stand-up	Stand-up	Stand-up
da Arrancada	15min	15min	15min	15min
2hrs				
				Revisão da
				Arrancada
			Hora de	1/2hr
			Estória	Retrospectiva
			1hr	90min

Figura 52 – Agenda diária para um Sprint de uma semana

3.2.4.1 Cerimônia de Planejamento da Arrancada

A cerimônia de planejamento da arrancada marca o início da arrancada. O objetivo da primeira parte da cerimônia é o comprometimento da equipe para a entrega dos produtos da arrancada. Na segunda parte da cerimômia, a equipe identifica as tarefas que devem ser completadas para as entregas baseadas nas estórias de usuários. Recomenda-se uma cerimônia de planejamento de uma a duas horaas de duração por semana de desenvolvimento.

Parte Um: O que faremos?

A meta da primeira parte da cerimônia de planejamento da arrancada é ter um conjunto de estórias que a totalidade da equipe se compromete a entregar no fim da arrancada. O *proprietário do produto* lidera esta parte da cerimônia.

Uma a uma, na ordem decrescente de prioridade, o proprietário do produto apresenta as estórias que ele gostaria que a equipe completasse durante a arrancada. A medida que cada estória for apresentada, os membros da equipe a discutem com o proprietário do produto e reveem critérios de aceitação para ter certeza que eles têm um entendimento único do que se espera. Então, os membros da equipe decidem se eles podem se comprometer na entrega da estória no final da arrancada. Este processo se repete para cada estória até que a equipe sinta que não pode mais se comprometer com mais trabalho. Observe a separação de autoridade, o proprietário do produto decide quais são as estórias a serem consideradas, mas são os membros da equipe que realizam o trabalho que decidem quanto trabalho eles podem fazer.

Parte Dois: Como faremos?

Na fase dois da cerimônia de planejamento da arrancada, a equipe começa a decompor as estórias selecionadas em tarefas. Lembre-se que as estórias são os produtos entregáveis: Coisas que os contratantes, usuários e clientes querem. Para entregar uma estória, os membros da equipe terão de terminar as tarefas. Tarefas são coisas como: Obtenha mais entradas dos usuários; projete uma nova

tela; adicione colunas a um banco de dados; escreva texto de ajuda (help); Faça a tradução do menu para as localidades alvo; execute os scripts de entrega.

O proprietário do produto deve estar disponível pelo menos durante metade da cerimônia para responder às questões. A equipe pode também precisar ajustar a lista de estórias para a qual ela está se compromentendo, já que na fase de identificação das tarefas, os membros da equipe podem percerber que eles se comprometeram com um excesso de estórias ou com um número insuficiente.

O resultado da cerimônia de planejamento da arrancada é o *sprint backlog* (a lista de pendências da arrancada), a lista de todas as estórias que a equipe se compromete a entregar, com as tarefas associadas. O proprietário do produto concorda em não pedir estórias adicionais durante a arrancada, a menos que a equipe especificamente solicite mais. O proprietário do produto também concorda em está disponível para responder questões sobre as estórias, negociar o escopo delas e fornecer orientações sobre o produto até as estórias serem aceitáveis e forem consideradas terminadas.

3.2.4.2 Scrum Diário

A cerimônia de scrum diário, chamada às vezes de *stand-up meeting*, é:

Diária: A maioria das equipes escolhe fazer esta cerimônia no início do dia de trabalho. Você pode adaptá-la para as preferências da sua equipe.

Breve: Um ponto desta cerimônia é desencorajar as discursões e os desvios que tornam as reuniões um inferno. O scrum diário deve sempre durar no máximo 15 minutos.

Direta: Cada participante rapidamente compartilha:

- Quais tarefas completei desde o último scrum diário;
- Quais tarefas espero completar até o próximo scrum diário; e
- Quais obstáculos estão me retardando.

O objetivo desta cerimônia é inspecionar e adaptar o trabalho dos membros da equipe para que as estórias comprometidas pela equipe sejam completadas com sucesso. A inspeção acontece na cerimônia, a adaptação pode ser feita depois dela. Isto significa que a equipe não precisa resolver os problemas na cerimônia, apenas trazer à torna as questões e decidir quais membros da equipe vão se ocupar delas é, normalmente, o suficiente. Lembre-se, a cerimônia é breve.

3.2.4.3 Hora de Estória

Nesta cerimônia, você discutirá e melhorará as estórias na lista de pendências do produto, *product backlog*, que contém todas as estórias para as futuras arrancadas. Observe que não são as estórias da arrancada atual, estas estão nas pendências da arrancada, *sprint backlog*. Recomenda-se que a cerimônia dure uma hora por semana, toda semana, independente da duração da sua arrancada. Nesta cerimônia, a equipe trabalha com o proprietário do produto para:

Definir e Redefinir Critérios de Aceitação

Cada estória de usuário na lista de pendências do produto deve incluir uma lista de critérios de aceitação. Estas são condições testáveis para aprovado/reprovado que nos ajudam a saber quando uma estória está implementada como se pretendia. Algumas pessoas pensam nelas como exemplos de aceitação: Os exemplos que a equipe vai usar para mostrar que uma estória está terminada.

Tamanho da Estória

Durante a cerimônia do tempo de estória, a equipe vai atribuir (ou estimar) um tamanho para as estórias que ainda não tiverem seus tamanhos estimados. O tamanho é uma adivinhação da equipe sobre a quantidade de trabalho que uma estória necessita para ser completada.

Divisão da estória

Estórias no topo da lista de pendências do produto precisam ser pequenas. Estórias pequenas são fáceis para todos entenderem e fáceis para a equipe completar em um curto espaço de tempo. Estórias mais no fundo da lista de pendências do produto podem ser maiores e menos bem definidas. Isto implica em que temos de quebrar estórias maiores em estórias menores a medida que as estórias sobem na lista. Enquanto o proprietário do produto pode fazer esta quebra por conta própria, o tempo de estória é a oportunidade dele ter ajuda de todo a equipe para esta atividade.

A cerimônia do tempo de estória não é uma *cerimônia oficial* do scrum. Mas as equipes de scrum de alto desempenho costumam usá-la.

3.2.4.4 Revisão da Arrancada

Este é o final público da arrancada, convide todos os interessados (*stockholders*) para esta cerimônia. É a oportunidade da equipe mostrar seus sucessos, as estórias que cumpriram a definição de terminada da equipe. Esta também é uma oportunidade para os interessados verem como o produto melhorou com a arrancada.

Se existirem estórias que a equipe se comprometeu a terminar, mas não conseguiu, este é o momento para compartilhar esta informação com os interessados. O evento principal desta cerimônia é, obviamente, mostrar as estórias terminadas. Sem dúvidas, os interessados darão um retorno e idéias que o proprietário do produto e a equipe usarão na fase de inspeção-e-adaptação do produto.

Esta cerimônia não é uma reunião para tomar decisões. Não é onde decidimos se uma estória está terminada, isto é feito antes. Não é quando decidimos sobre o que faremos em seguida, na próxima arrancada, isto é feito na cerimônia de planejamento da arrancada. Quão longa deve ser a cerimônia de revisão da arrancada? Recomenda-se que seja agendada por meia hora a uma hora por semana de desenvolmento.

3.2.4.5 Retrospectiva

Enquanto a revisão da arrancada é o término público da arrancada, a equipe ainda tem mais uma cerimônia: A retrospectiva. Scrum foi projetado para ajudar as equipes a inspecionar e adaptar continuamente, resultando em desempenho cada vez melhor e felicidade. A retrospectiva, que tem lugar no final de cada arrancada, é um tempo dedicado para a equipe se focar no que foi aprendido durante a arrancada e como este aprendizado pode ser usao para melhorar a equipe. Recomenda-se uma a duas horas por semana de desenvolvimento.

Diferente da análise *post mortem*, o objetivo de uma retrospectiva nunca é gerar uma longa lista de lavagem das coisas que funcionaram e das que deram errado, mas identificar uma ou duas mudanças de estratégia para a próxima arrancada. Ela serve para melhorar o processo.

3.2.4.6 Término Anormal da Arrancada

Em scrum, o acordo básico entre a gerência e a equipe é que a gerência não vai mudar os requisitos durante uma arrancada. Ainda assim, às vezes, algo acontece que invalida tudo no plano de uma arrancada – o negócio é vendido, uma tecnologia disruptiva entra no mercado, um competidor faz um movimento. A decisão de terminar uma arrancada mais cedo é fundamentalmente uma decisão de negócios, logo o proprietário do produto é quem pede o término anormal de uma arrancada.

Se o proprietário do produto decide terminar precipitadamente uma arrancada, a equipe deve voltar ao estado anterior à arrancada para não trabalhar com modificações incompletas. Realizar uma cerimônia de retrospectiva é particularmente importante após um término anormal para ajudar a equipe a aprender com a experiência.

3.2.4.7 Inspecione e Adapte

Então, por que desenvolvemos o trabalho em ciclos curtos? Para aprender. Experiência é o melhor professor e o ciclo de scrum é projetado para lhe fornecer várias oportunidades de receber retornos –

dos clientes, da equipe, do mercado – e aprender com eles. O que você aprende ao trabalhar num ciclo, prepara você para o planejamento do próximo ciclo. Em scrum, chamamos isto de *inpecionar-e-adaptar*. Você pode chamá-lo de *melhoria contínua*. De qualquer modo, é uma coisa boa.

3.3 Outras metodologias ágeis

3.3.1 *Full Test Driven Development* - FTDD

3.3.2 *EXtreme Programmig* - XP

3.3.3 *Agile Modeling Driven Development* - AMDD

4 Codificação em Java

Veremos neste capítulo alguns exemplos de código Java para os conceitos vistos até o presente e mais alguns novos bastante usados.

4.1 Exemplo de MVC - *Model-View-Controller*

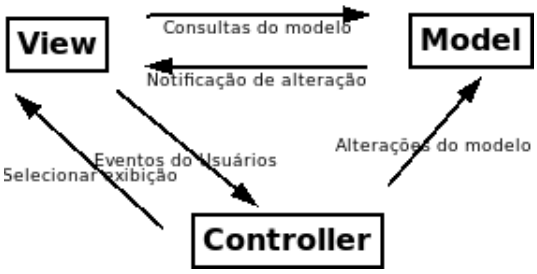

Figura 53 – Arquitetura *Model-View-Controller*

Fonte: Ahmed e Umrysh (2002)

O exemplo a seguir vem de Linden (2004). Nele mostramos um exemplo simples da arquitetura MVC usada para aplicações com GUI, interface usuário gráfica. A figura 53 mostra os princípios desta arquitetura. O *modelo* na arquitetura encapsula a lógica de negócio da solução do problema resolvido pela aplicação. A *visão* contém o código que exibe o estado do modelo para o usuário e a interface para o usuário interagir com a aplicação. O *controlador* recebe os eventos de entrada do usuário que vêm pela interface com o usuário, a *visão*. Estes eventos podem modificar o que está sendo exibido ou como está sendo exibido pela *visão*. Estes eventos do usuário também podem provocar mudanças no estado do *modelo*. Mudanças no estado do *modelo* podem gerar notificações de alterações para a *visão*.

O programa é simplesmente um relógio em Java. A figura 54 mostra o diagrama de classes da aplicação. Compare com o diagrama de dependência da figura 15.

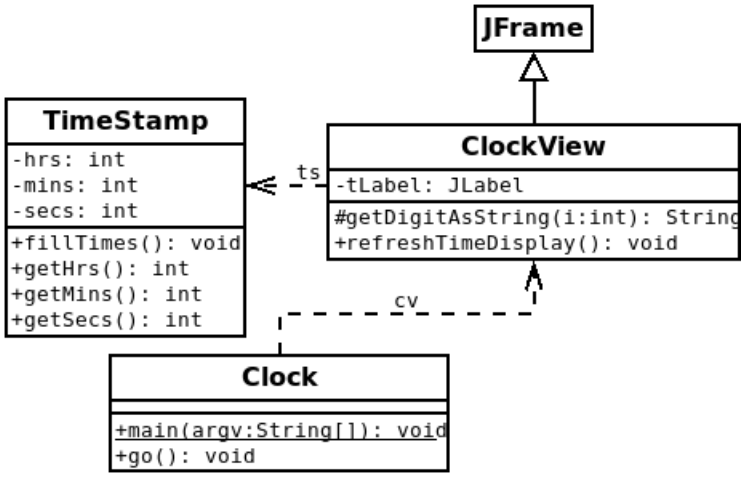

Figura 54 – Diagrama de classes do relógio digital.

```
public class TimeStamp {
```

```
    private int hrs;
    private int mins;
    private int secs;

    public void fillTimes() {
        java.util.Calendar now;
        now = java.util.Calendar.getInstance();
        hrs = now.get(java.util.Calendar.HOUR_OF_DAY);
        mins = now.get(java.util.Calendar.MINUTE);
        secs = now.get(java.util.Calendar.SECOND);
    }
    public getHrs() { return hrs; }
    public getMins() { return mins; }
    public getSecs() { return secs; }
}
```

A classe *TimeStamp* tem como atributos a hora, os minutos e os segundos e um método que atualiza os valores dos atributos com a ajuda da classe *java.util.Calendar* do sistema. A *TimeStamp* modela a "lógica de negócio" de um relógio. A classe *Calendar* do pacote *java.util* é uma classe que vem na biblioteca padrão distribuída com o JDK, kit de desenvolvimento do Java. Observe que os atributos estão ocultos dos usuários da classe *TimeStamp*, os valores deles podem ser obtidos através dos métodos *getters* dos atributos. Se mais tarde, for decidido que se deseja mais alterar estes atributos, pode se reprogramar os métodos *getters* de acordo com as mudanças e os usuários da classe não precisam saber que os atributos foram modificados. Dizemos que isolamos os usuários da classe *TimeStamp* das modificações nos atributos ocultos. Precisamos, entretanto, manter o contrato de que os métodos *getHrs()*, *getMins()* e *getSecs()* continuam a fornecer as horas, os minutos e os segundos.

```
public class ClockView extends javax.swing.JFrame {
    private javax.swing.JLabel tLabel = new javax.swing.JLabel();

    public ClockView() {
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setSize(95,45);
        getContentPane().add(tLabel);
        refreshTimeDisplay();
    }

    protected String getDigitAsString(int i) {
        String str = Integer.toString(i);
        if (i<10) str = "0" + str;
        return str;
    }

    public void refreshTimeDisplay() {
        TimeStamp ts = new TimeStamp();
        ts.fillTimes();
        String display = getDigitAsString(ts.getHrs()) + ":"
                    + getDigitAsString(ts.getMins()) + ":"
                    + getDigitAsString(ts.getSecs());
        tLabel.setText("  " + display);
        tLabel.repaint();
    }
}
```

A classe *ClockView* é a nossa classe de visualização. Ela herda da classe *javax.swing.JFrame*. O pacote *javax.swing* é parte da nova biblioteca do Java para a criação de interfaces gráficas para usuários para aplicações de *desktop*. Ao herdar de *JFrame*, um objeto *ClockView* serve de base para a janela onde estará o relógio. O único elemento gráfico que é adicionado ao painel da janela é um *JLabel*. Um objeto *JLabel* permite a exibição de um texto. Para transformar a informação do objeto TimeStamp em texto a ser exibido (variável *display*), a classe ClockView tem o método auxiliar *getDigitAsString(int)* que converte um inteiro em texto. Observe que não foram tomados cuidados para garantir que o inteiro só tem 2 dígitos. Como sabemos que os dados vem da TimeStamp corretamente, não é necessário fazer este tipo de checagem.

```
public class Clock {
    public static void main(String[] argv) {
        new Clock().go();
    }
    public void go() {
        ClockView cv = new ClockView();
        cv.setVisible(true);
        try {
            for (;;) {
                cv.refreshTimeDisplay();
                Thread.sleep(500);
            }
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

A classe *Clock* é a controladora da aplicação. Como o relógio digital é muito simples, tudo que ela tem é o método *main()* que repete para sempre (*loop* infinito) a atualização da tela do relógio e espera por meio segundo (dormindo). A espera é importante para liberar o processador do computador para outras aplicações. A razão de esperar por meio segundo e não por um segundo é que o erro de exibição no caso da espera de um segundo pode ser de até um segundo. No pior caso o programa leu as horas com o objeto TimeStamp justo quando ia mudar o segundo e mostra o valor anterior, só após o tempo de espera de 1s é que a tela vai ser atualizada. Com meio segundo de espera, o erro máximo é de 0,5s.

4.2 Programação J2EE

Referências

AHMED, K. Z.; UMRYSH, C. E. *Desenvolvendo Aplicações Comerciais em Java com J2EE e UML*. Rio de Janeiro: Editora Ciência Moderna Ltda., 2002. ISBN 85-7393-240-6. Citado 4 vezes nas páginas 18, 53, 54 e 65.

AMBLER, S. W. *The Object Primer: Agile Model-Driven Development with UML 2.0*. 3rd. ed. New York: Cambridge University Press, 2004. ISBN 978-0-521-54018-6. Disponível em: <www.cambridge.org/9780521540186>. Citado 3 vezes nas páginas 4, 8 e 16.

BECK, K. et al. *Manifesto for Agile Software Development*. 2018. Acessado em 04/10/2018. Disponível em: <http://agilemanifesto.org/>. Acesso em: 4 out 2018. Citado na página 54.

ELLIS, M. A.; STROUSTRUP, B. *The Annotated C++ Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co.,Inc, 1990. ISBN 0-201-51459-1. Citado na página 15.

FOWLER, M. *UML Distilled*. 3rd. ed. Reading, MA: Addison Wesley Longman, 2004. Citado 3 vezes nas páginas 5, 16 e 20.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co.,Inc, 1994. ISBN 0-201-63361-2. Citado 5 vezes nas páginas 6, 31, 33, 34 e 50.

LARSSON, A. *Dia - Diagram Tool*. 2012. <http://live.gnome.org/Dia>. <http://live.gnome.org/Dia>, acessada em 13 ago 2018. Citado na página 7.

LINDEN, P. van der. *Just Java 2*. Santa Clara, California: Prentice-Hall, 2004. ISBN 0-13-148211-4. Citado na página 65.

SHALLOWAY, A.; TROTT, J. R. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. 2nd. ed. Boston, MA, USA: Addison-Wesley, Pearson Education, Inc., 2005. ISBN 0-321-24714-0. Citado 9 vezes nas páginas 6, 12, 16, 32, 33, 34, 37, 51 e 53.

SIMS, C.; JOHNSON, H. L. *Scrum: a Breathtakingly Brief and Agile Introduction*. Los Angeles, CA: Dymaxicon, 2012. Citado na página 56.

WIKIPEDIA. *Rational Unified Process*. 2018. Acessado em 04/10/2018. Disponível em: <https://en.wikipedia.org/wiki/Rational_Unified_Process>. Acesso em: 4 out 2018. Citado na página 56.

WIKIPEDIA. *Unified Modeling Language*. 2018. Wikipedia. Acessado em 23/08/2018. Disponível em: <https://en.wikipedia.org/wiki/Unified_Modeling_Language>. Acesso em: 23 ago 2018. Citado na página 7.