

# ***ENGENHARIA DE SOFTWARE I***

## **CAPÍTULO 3 - COMO GARANTIR QUE TODOS OS PRODUTOS DESENVOLVIDOS POR UMA FÁBRICA DE *SOFTWARE* TENHAM UM PADRÃO DE QUALIDADE?**

Mauricio Antonio Ferste

INICIAR



## **Introdução**

Na era do conhecimento, o espaço da tecnologia tem se confundido com o espaço social, pois tornou-se parte intrínseca do cotidiano. Foi um caminho de transformações, desde que os engenheiros começaram a se dedicar aos primeiros computadores construídos, todo o aprimoramento, de materiais, de aproveitamento de energia, de sofisticação de *software* e de programação, até chegar ao que hoje conhecemos como equipamentos normais para a vida pessoal e profissional de qualquer pessoa, inserida nesse contexto.

Houve também uma mudança de mentalidade, neste período. Antes, os processos estavam sendo construídos do zero, não havia estruturas replicáveis e claras de dados e tudo era experimento, vindo de iniciativas individuais, o que gerava

problemas quando os responsáveis abandonavam, por alguma razão, a equipe de desenvolvimento.

Atualmente, os processos já estão mais consolidados, e uma das características dessa era é o compartilhamento de conhecimento, que exige o estabelecimento de definições, envolvendo qualidade e reuso. Há uma larga escala de diferentes *softwares*, plataformas, e perfis distintos de desenvolvedores, sem contar as diferentes arquiteturas.

Por exemplo, como faríamos para desenvolver um sistema para uma escola, desde o início, definindo sua documentação com o cliente? Em qual formato passar esta documentação para o desenvolvedor? Como fazer a implementação, que arquitetura ele deve usar, como codificar? Como saber se ele está desenvolvendo de uma forma moderna? São muitos passos para chegar a um produto com qualidade.

Vamos, então, para os estudos!

## 3.1 Padrões de arquitetura

Para desenvolver um *software*, temos de ter em mente que é fundamental estruturar a base do sistema e, para construir algo, tem de se pensar na fundação. Em termos de *software*, esta fundação seria formada por vários blocos, assim como em uma construção, cada conexão e tijolo tem uma função e uma forma de uso.

Os engenheiros de *software*, Mary Shaw e David Garlan (1996), definiram que “a arquitetura é a forma de utilizar componentes básicos”, e, com esta afirmação, criaram toda uma noção sobre componentes, estilos e módulos básicos, que influenciariam os pesquisadores depois deles. Tanto que Roger Pressman, um dos maiores autores na área declarou que “todo seu trabalho conseguiu, por fim, elevar em muito o nível da arquitetura no desenvolvimento de *software*” (PRESSMAN, 2016, p. 234).

A teoria que veremos a seguir deve se desdobrar, para que depois, na prática, você possa ter em mente como organizar a arquitetura e reutilizá-la em seus projetos de sistemas.

## VOCÊ SABIA?

Um termo que usaremos muito neste capítulo será “abstrair”, palavra que vem de abstração e significa isolar o conceito do elemento propriamente dito, do latim “*abstracione*”, que significa “separação”. A locução “fazer abstração de” significa não levar em consideração, não dar importância, não dar crédito, não fazer deferência. Abstrato é tudo que não é concreto ou resulta de abstração. É o que só existe na ideia, no conceito. É o que possui alto grau de generalização, que opera unicamente com noções (PRESSMAN, 2016).

Dentre as principais arquiteturas conhecidas, iniciaremos o trabalho em um fluxo histórico, partindo das arquiteturas mais antigas e deixando para o fim, as arquiteturas mais recentes, para facilitar o entendimento, estabelecendo uma linha de evolução. Não que as mais antigas sejam mais fáceis, mas, as mais recentes fazem uso de vários conceitos de arquiteturas antigas, o que torna necessário conhecê-los.

### **3.1.1 Arquitetura em camadas**

Neste padrão, que é aberto e não define uma estrutura, deve ser considerada uma hierarquia entre camadas, nas quais, individualmente é ofertado um serviço. Normalmente, as camadas comunicam-se entre si, as superiores, com as imediatamente inferiores, em ambos os sentidos.

Entre as camadas, são definidos conectores, que são as estruturas com as quais se comunicam, camada a camada, estabelecidos, normalmente, por algum protocolo para a comunicação. Não existe um padrão para definir uma arquitetura em camadas, tem de existir o entendimento de que elas significam uma divisão, como vemos na figura a seguir, segundo a sugestão de Sommerville (2011).

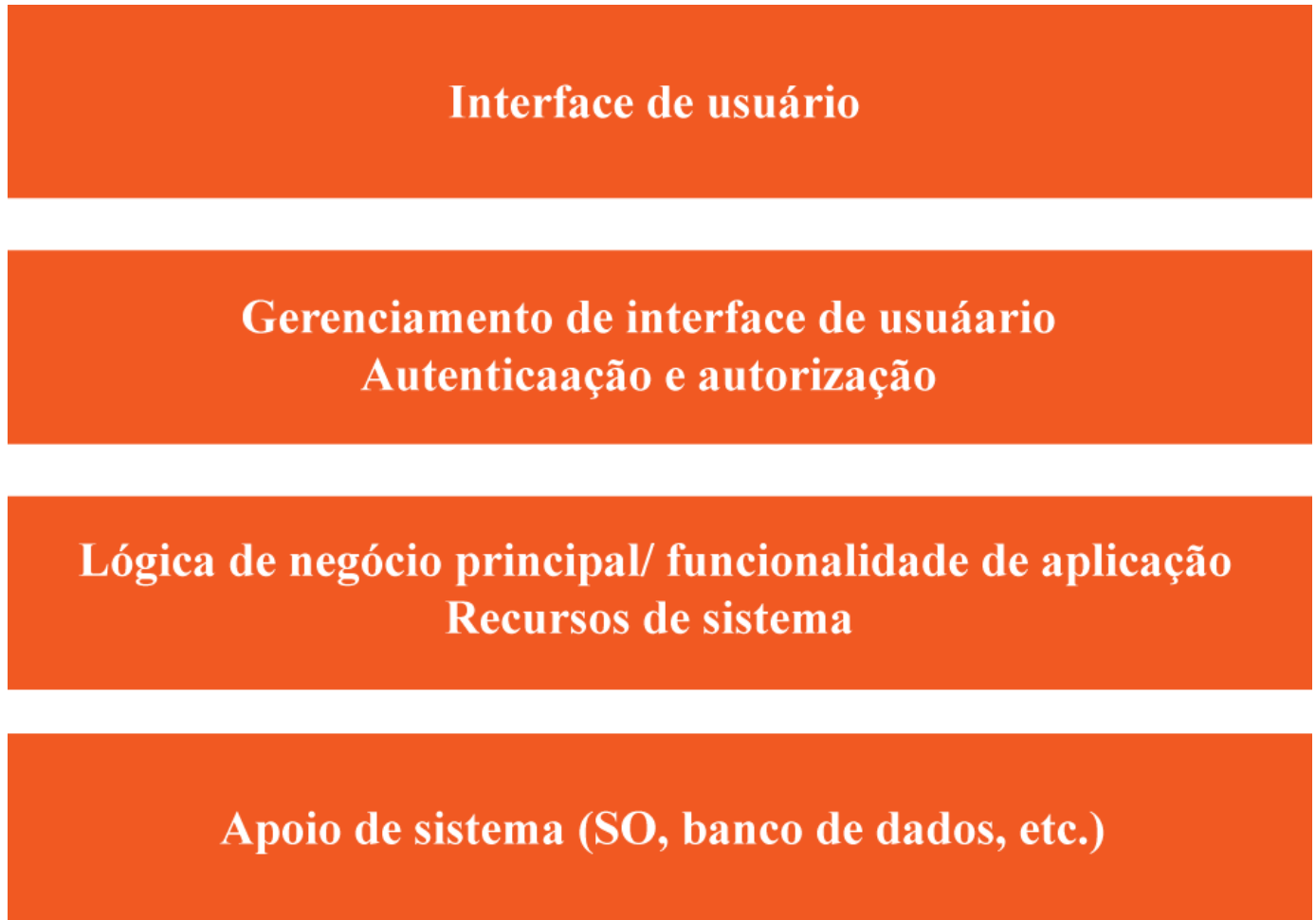


Figura 1 - Arquitetura Genérica em camadas. Fonte: SOMMERVILLE, 2011, p. 111.

Pode-se entender que existem, principalmente, quatro camadas neste modelo (PRESSMAN, 2016):

- **interface de usuário:** parte do sistema responsável pela apresentação das informações para o usuário, permite navegação e forma o que é chamado de camada de apresentação;
- **gerenciamento de interface de usuário, autenticação e autorização:** código responsável por validar acesso, autenticar usuários, ou seja, segurança no acesso da informação;
- **lógica de negócio principal:** código referente ao negócio do sistema, é aqui que as regras de negócio, referentes ao objetivo fim do sistema, são implementadas;
- **apoio de sistema:** esta é uma camada focada em arquitetura, permite abstrair toda a informação de arquivos e de bancos de dados além de segurança.

Embora a divisão trate de quatro camadas, as camadas de apoio e de lógica de negócio, acabam sendo fundidas em somente uma camada, pois tratam da mesma estrutura do sistema que controla os dados.

A abordagem em camadas, em especial três, é muito utilizada na atualidade, pois se adapta facilmente às características das plataformas atuais. Nelas, normalmente, existe uma interface que é controlada e ligada ao dado. Entendeu? Três partes ou camadas.

Esta abstração permite que uma camada seja tratada para a tela, independentemente do resto, com suas características de desenvolvimento, como por exemplo, uso de HTML e CSS, que são linguagens, ou pseudolinguagens práticas para a criação de telas, que podem ser elaboradas por um profissional especialista, independentemente da camada de negócio, que pode ser tratada por outro profissional, especialista, por sua vez, em tratar o dado, guardá-lo e validá-lo, finalizando a integração entre estas camadas que são unidas por outra, que é a de controle.

---

## VOCÊ SABIA?

Em sistemas operacionais e banco de dados, também é utilizado o termo camadas para implementações com divisões de implementação, como, por exemplo: camada de visão, ou telas, camada de modelo de dados e camada de acesso a dados.

A arquitetura em camadas traz características de uso, que se classificam entre positivas e negativas. As positivas são:

- divide o desenvolvimento em camadas, criando diferentes níveis de abstração, sendo que, normalmente, é mais abstrato quanto mais alto for o nível;
- permite o reúso, camada a camada, permite ganho de desempenho também, pois cada camada é especialista em um assunto;

- permite um crescimento e manutenção contínuas, inclusive facilita solucionar erros, pois isola os problemas em suas respectivas camadas.

E as negativas:

- embora pareça simples, dividir um problema em camadas é uma tarefa que pode exigir, muitas vezes, uma maturidade maior do que se espera, principalmente no momento de encontrar a camada certa para cada componente.
- podem existir problemas relacionados com performance, pois, devido à divisão em partes, as conexões entre elas geram, muitas vezes, transformações de dados e sobretrabalhos indesejados (PRESSMAN, 2016).

Quando a estrutura das camadas é bem elaborada, é possível, inclusive, trocar uma camada por um componente externo ou algo assim, este é um conceito amplamente difundido, nem sempre de fácil aplicação, mas possível, devido ao desacoplamento que deve existir entre cada camada (RICHARDS, 2015).

### 3.1.2 Model – View – Controller

Presente, de alguma forma, em praticamente todos os padrões, sua criação original foi estabelecida pelo cientista da computação Trygve Reenskaug (1979), que trabalhava com Smalltalk, uma linguagem de programação, na Xerox. Entenda que é um padrão de arquitetura e não um *design pattern*, entretanto não é exatamente também um padrão arquitetural, mas sim uma evolução do padrão de arquitetura em camadas, tornando-se na verdade uma componentização em três partes interconectadas, o modelo, a visão e o controle descritos brevemente a seguir, conforme seu criador, Reenskaug (1979).

**Modelo:** conjunto de os dados, e camadas de acesso a eles, que representa, também, a lógica de negócio da aplicação. O Modelo é independente da arquitetura da aplicação e da forma como ele será mostrado.

**Visão:** podem existir várias visões para um modelo, especificam como os resultados são mostrados para os usuários em telas, relatórios ou outras formas de interação.

**Controlador:** responsável por conectar as visões aos modelos, traduzem as interações com as visões feitas pelo usuário.

---

## VOCÊ QUER LER?

O MVC considera conceitos de reúso, com base em desacoplamento. Historicamente, o MVC foi definido por Tygve Reenskaug (1979), e os principais documentos da época, que também trazem informações sobre orientação a objetos e UML, podem ser encontrados no site: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html> (<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>).

---

Esta arquitetura é hoje muito utilizada em várias linguagens como C#, Java, Ruby, PHP, Javascript. Cada linguagem tem seu próprio *framework* implementando este padrão, como vemos na figura a seguir.

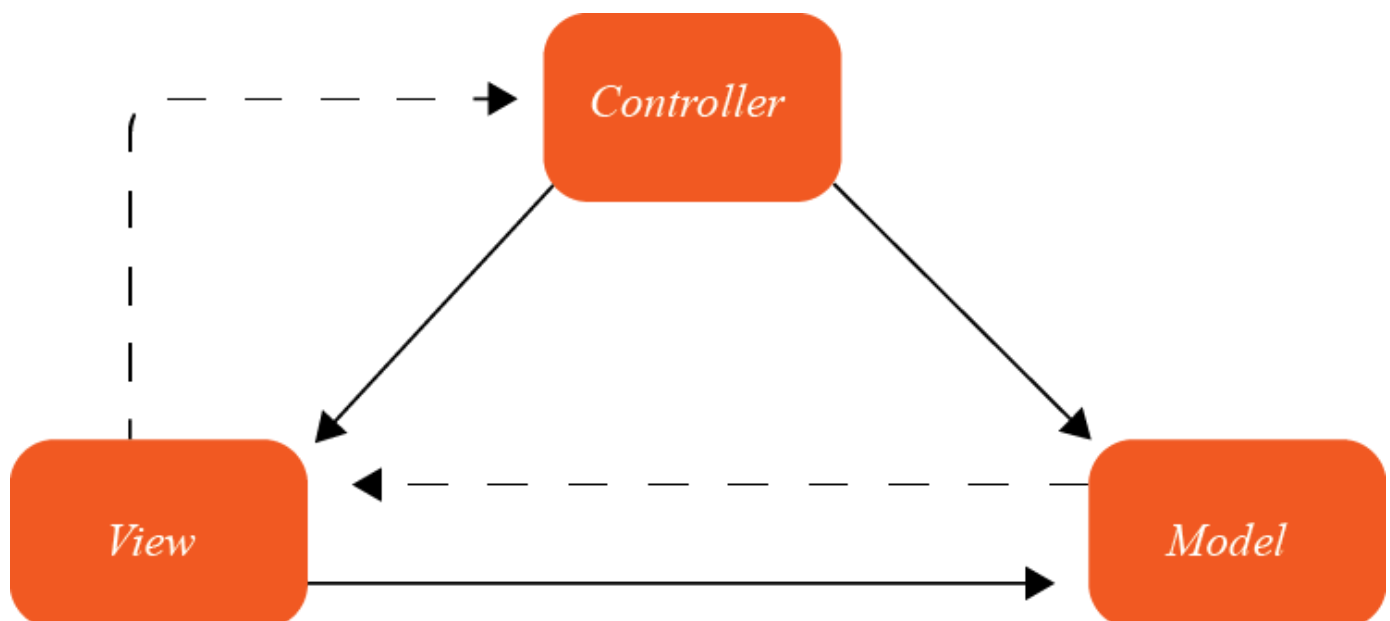


Figura 2 - Visão atual de como é o MVC, de acordo com Sommerville. Fonte: SOMMERVILLE, 2011, p. 109.

Após o MVC firmar-se no mercado, surgiu uma série de variações como MVP (*Model-View-Presenter*) ou MVVM (*Model-View-View-Model*), todas usuais no mercado atual, sendo que, cada uma destas, valeria um trabalho completo. Estes padrões são amplamente utilizados nos sistemas modernos, como Android, iOS e Windows, portanto, seu aprendizado é essencial para desenvolver *software* para qualquer plataforma.

### **3.1.3 Arquitetura orientada a eventos - EDA (Event Driven Architecture)**

Considerando o avanço da informática, e de sua infraestrutura, as soluções evoluíram para sistemas que não são mais centralizados, ditos monolíticos. Atualmente, a tendência é a de existir uma infraestrutura distribuída, que tem característica de processamento paralelo, como internet, processamentos multinúcleo e *clusters* de computadores, ou seja, “em sistemas distribuídos, seus componentes executam em computadores diferentes e as chamadas são feitas pela rede, de componente para componente” (SOMMERVILLE, 2011, p. 252).

O contexto de descentralização de processamento, iniciou a construção de uma arquitetura nova, e a resposta foi a arquitetura orientada a eventos. O que é isto?

Bem, entenda que, ao contrário das arquiteturas anteriores, nesta assume-se que parte da solução produz eventos enquanto outra parte os consome, e isto tem de ser claro na solução. Pode-se traçar um paralelo com a situação conhecida como Produtor/Consumidor, cada um especialista e específico em sua função. Fundamentalmente, o modelo orientado a eventos alavancou soluções de IoT (*Internet of Things*, em português, Internet das Coisas), pois a fonte dos eventos pode ser externa ao sistema.

Este fato é positivo, pois a arquitetura é flexível, permite que vários subsistemas processem os mesmos eventos em tempo real, propiciando uma maior escalabilidade e uma enorme integração, já que diferentes tecnologias podem processar os eventos gerados.

Obviamente que todo avanço tecnológico tem seu custo e nessa arquitetura, podem-se considerar dois problemas principais maiores. Primeiro, se um dos dois “lados” independentes deixar de funcionar, todo o sistema para. De fato, é um preço a se pagar por um sistema distribuído e que permite tanta flexibilidade.



A forma de desenvolver soluções nesta arquitetura também mudou, pois, logicamente, se criada uma aplicação que utilizará dados de, por exemplo, correios e bancos, em eventos separados, não necessariamente os dois serão processados juntos, portanto o consumidor dos eventos passa a ter de tratar estes retornos em paralelo e em tempos distintos. Interessante não?

### **3.1.4 Arquitetura de microsserviços**

A verdade é que o mundo deriva para IoT (Internet das Coisas) e utilizar uma arquitetura que, embora mais complexa, permita atingir e servir o maior número de sistemas com escalabilidade, é tentador. E isso ganhou ainda mais força com a publicação do artigo de Martin Fowler (2014), que trata das características básicas, pois existe muita confusão na diferenciação entre microsserviço e serviço (SOA):

- anteriormente, os dados eram tratados internamente da mesma forma, agora não, cada microsserviço possui um banco de dados;
- existe uma entrega mais rápida de recursos e melhor utilização;
- a orquestração de chamadas, antes, necessitava de um sistema ESB, visto acima, com a implementação de serviços baseados em REST. Essas chamadas foram simplificadas (diferente da arquitetura anterior, que utilizava protocolos WS-\*, ou com acesso ao banco de dados do serviço vizinho).
- a segurança aumentou, pois, como cada microsserviço possui um Gateway, seu controle é mais fácil.

Com microsserviços, um serviço que é compilado e opera em escala para atingir clientes em novas regiões geográficas (por exemplo), pode ser recompilado e alterado sem ter de alterar outra parte consumidora, sem avisar outra equipe (na prática é sempre bom manter contato).

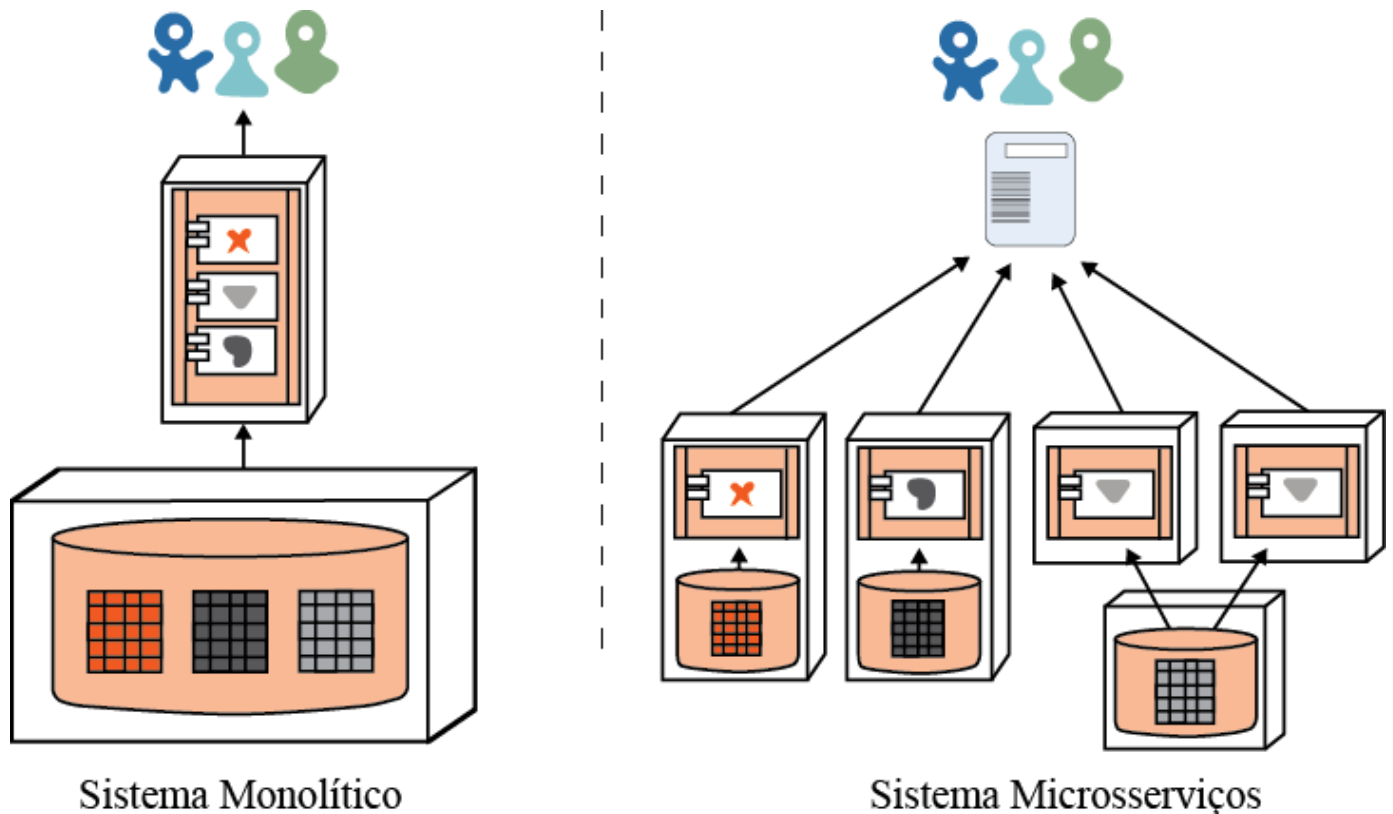


Figura 3 - No sistema microserviços, ao contrário do monolítico, os serviços funcionam, por exemplo, em servidores diferentes, ou em processos exclusivos. Fonte: FOWLER, 2014.

A arquitetura de microserviços não é para qualquer situação, pois, embora permita uma escalabilidade elevada, necessita de diversos tratamentos, caso, por exemplo, de um serviço que esteja inoperante. Como qualquer tecnologia, essa deve ser muito analisada antes de ser adotada, pois é complexa, já que vários controles tem de ser implementados para monitorar as chamadas entre os diferentes serviços, como vimos na figura acima, permite o trabalho em paralelo, mas exige mais controle e demanda, mais cuidados quanto à configuração.

### **3.1.5 Arquitetura baseada em espaços / arquitetura em nuvem (Cloud Architecture)**

Nuvem é um termo um tanto abstrato para as pessoas em geral, pois refere-se à situação na qual o sistema, infra, ou todo o sistema operacional, permanece fora da empresa e não um *data-center* interno. O conceito de “nuvem” refere-se a uma arquitetura que não é somente interna da aplicação, mas afeta a forma como a informação é disponibilizada para o usuário, ou cliente. Vamos entender isso a seguir, nas três formas mais comuns que são tratadas comercialmente, ou seja, na verdade, as camadas, nesse contexto, são muito mais abstrações, utilizadas pelos envolvidos para tomar as decisões sobre implementação (BECK, 2011).

### **IaaS: Infraestrutura como Serviço (*Infrastructure as a Service*)**

Significa usar uma plataforma como serviço. Esta plataforma de infraestrutura envolve tudo relacionado com um servidor virtualizado, normalmente o usuário, ou cliente, paga para uma empresa fornecer esta estrutura e acessa a mesma por meio de um endereço especificado. Lá pode, muitas vezes, coordenar o uso em disco, memória, número de servidores e vários outros fatores críticos que, antes, eram internalizados na empresa. Exemplos mundialmente conhecidos incluem Amazon *Web Services*, Microsoft Azure, e Google *Compute Engine* (BECK, 2011).

### **PaaS: Platform como Serviço (*Platform as a Service*)**

Este é o ambiente mais indicado para desenvolvedores, pois nele é disponibilizada toda uma plataforma, com infraestrutura mais aberta, um serviço que permite instalar ou formatar, desde o tipo de sistema operacional até a interface (BECK, 2011)

### **SaaS: Software como Serviço (*Software as a Service - SaaS*)**

Neste formato os aplicativos ou sistemas desejados são disponibilizados para o usuário final na íntegra, como um sistema completo. Por exemplo, toda a plataforma do Google, com aplicativos de *e-mail* e arquivos, é um exemplo de SaaS, da mesma forma, pode ser visto o Facebook, plataforma da Microsoft *online* e outros. Um desenvolvedor pode fornecer todo um sistema de ERP pronto, *online*, nessa plataforma, de maneira que seus usuários finais não se preocupariam com infraestrutura. Na figura abaixo, é demonstrada a estrutura em nuvem, toda a infraestrutura envolvida é, muitas vezes, despercebida, pois existe um barramento de serviços que é oferecido ao usuário.

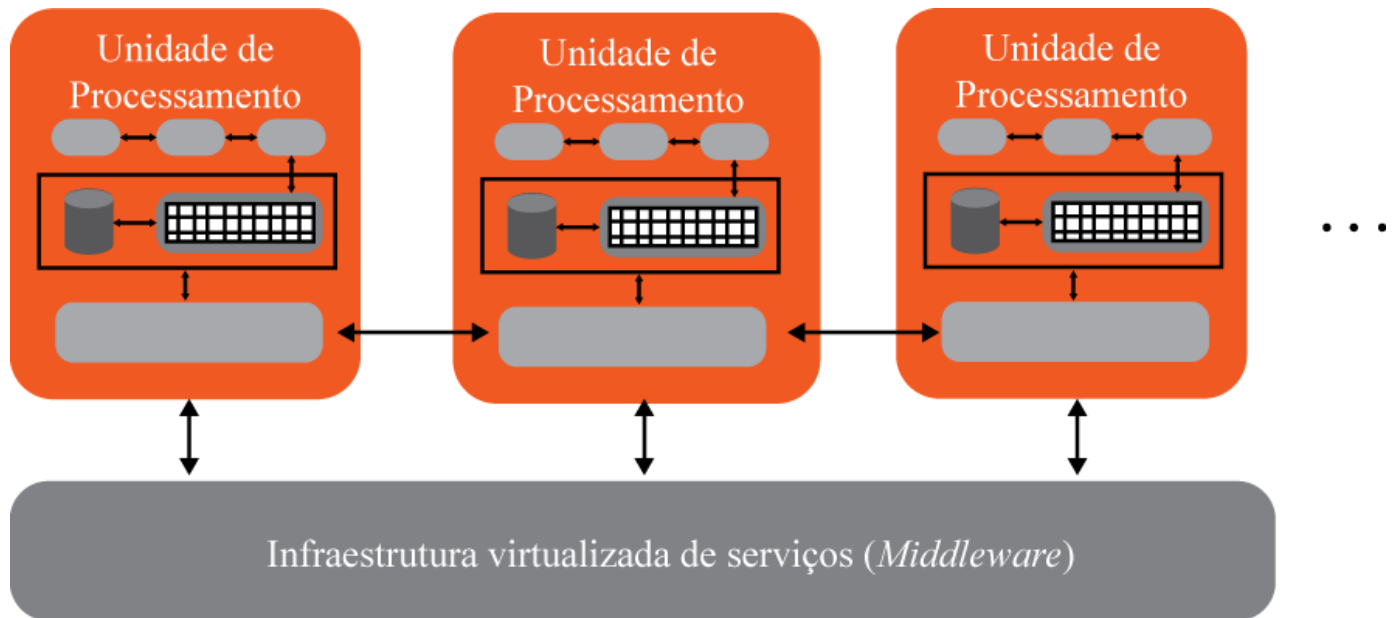


Figura 4 - Exemplo de arquitetura em nuvem. Fonte: Elaborado pelo Autor, 2018.

Assim, para escolher qual o modelo de nuvem deve ser adotado, devemos considerar: mais ou menos controle; desenvolver, ou não; usar um sistema pronto; enfim, tudo dependerá do seu interesse como desenvolvedor, ou como usuário.































	Camadas	Eventos	<i>Microkernel</i>	Microserviços	Nuvem
Agilidade					
Implantação					
Testabilidade					
Performance					
Escalabilidade					
Desenvolvimento					

Figura 5 - Comparação entre os diferentes tipos de arquitetura. Fonte: RICHARDS, 2015, p. 46.

Como podemos notar na figura acima, o uso de diferentes arquiteturas fornecem diferentes características, enquanto que a arquitetura em camadas, mais clássica, permite fácil testabilidade e um desenvolvimento mais rápido, na medida em que mais estruturas de serviços são consideradas, mais complicada fica a arquitetura, seu desenvolvimento e os testes, mas também, mais distribuída fica a aplicação com mais performance e escalabilidade. É uma decisão complexa, entretanto, é

inegável que sistemas distribuídos são a quase totalidade do perfil em desenvolvimento, devido à possibilidade de acesso mais fácil, inclusive incluindo a Internet das Coisas (IoT).

### **3.1.6 Sistemas de processamento**

Por sistema de processamento entenda que falamos de sistemas que tratam tarefas rotineiras dentro da empresa que são automatizadas, para que se mantenha seus processos em funcionamento e seja competitiva. Vamos listar algumas definições destes que são alguns sistemas utilizados em diferentes níveis organizacionais da empresa (SOMMERVILLE, 2011).

**Sistema de processamento de dados:** tem como objetivo processar grandes quantidades de dados estruturados, normalmente processa dados em lotes e segue um modelo denominado entrada-processo-saída. São sistemas mais utilizados em corporações, como sistemas de ERP, que se enquadram nesse tipo.

**Sistema de processamento de linguagens:** são específicos e utilizados para analisar e traduzir uma linguagem para outra, como compiladores, por exemplo. Processam pseudocódigos e os transformam em linguagens compreensíveis pela máquina (interfaces humano-computador).

**Sistema de Processamento de Transações (SPT):** são sistemas de suporte para atividades do dia a dia da organização, que servem o nível operacional da empresa. Normalmente tem um grau de “inteligência” baixo, pois são utilizados para atividades diárias ou normais de uma empresa, como controle de estoque, atendimento a clientes e fluxo de materiais.

**Sistema de Informações Gerenciais (SIG):** para uso gerencial, normalmente agrupam dados, geram relatórios mais complexos e totalizam por meio de parâmetros passados pelo usuário. Estão em um nível superior ao SPT, no sentido de fornecer informações táticas.

**Sistemas de Apoio à Decisão (SAD):** são sistemas complexos, que implementam, necessariamente, conceitos relacionados com inteligência artificial, no sentido de apoiar alguma decisão. Normalmente tem cunho estratégico e um alto valor comercial. Atualmente são desenvolvidos pela chamada Ciência de Dados, uma das áreas de desenvolvimento que mais cresce no mundo.

## 3.2 Orientação a objetos, projeto e implementação com UML

Desenvolver um projeto na prática, envolve conhecimento das definições, vistas anteriormente, da estrutura arquitetural do projeto, porém, para codificar, o caminho envolve outros conceitos sobre abstração de dados, classe e objetos, logo iniciaremos o caminho pelo qual entenderemos o que é orientação a objetos, a razão de existir e sua aplicação.

---

### VOCÊ QUER LER?

Para a exemplificação em código dos contextos explicados neste capítulo, buscando trazer uma proximidade maior com situações profissionais, utilizamos códigos em linguagem Java, quando necessário. Embora sejam mostrados somente blocos pequenos de código, se desejar mais informação sobre a linguagem, sobre como baixar e instalar, ou tirar dúvidas, recomendamos dois sites (JAVA, 2018; JAVA2S, 2018): < (<http://www.java2s.com/>)<http://www.java2s.com/> (<http://www.java2s.com/>)> (vários exemplos de código); < (<https://www.java.com/>)[https://www.java.com](https://www.java.com/) ([https://www.java.com](https://www.java.com/))> (onde baixar *jvm*, e ver detalhes sobre linguagem).

---

A seguir, vamos entender como desenvolver uma solução prática. Vamos considerar um exemplo: precisamos desenvolver um sistema em uma sala de aula, que denote o professor que dará a aula, os alunos presentes e matriculados no curso e suas relações. Quais os passos para realizar tal atividade?

### 3.2.1 Análise e projeto

No início do projeto, existe todo um processo de entender o problema, o “negócio”, e antes de implementá-lo, devemos definir a construção. São duas fases similares, porém distintas, segundo (PFLEEGER, 2004).

A primeira é a análise. Considere que é uma fase com alto grau de abstração, muito próxima da visão do cliente, que gera uma série de diagramas, mas não em nível de implementação, muito útil para sanar dúvidas e entender como cada funcionalidade é utilizada. Os principais diagramas gerados aqui, são os casos de uso e o diagrama de classes de todo o sistema (diagrama de domínio).

A segunda fase é o projeto. Considera a mesma ideia de análise, porém note que o termo “*projeto*” remete já para o desenvolvimento, entende? Os diagramas consideram mais uma nomenclatura que pode tranquilamente ser passada para o implementador, a fim de que ele realize o desenvolvimento, sem ter de realizar muitas consultas para o analista. Os diagramas são principalmente os de classe e sequência, focando no código e com termos que já nomearão estruturas dentro do sistema, definindo o que o implementador deve codificar.

---

## VOCÊ O CONHECE?

Edsger W. Dijkstra (1930-2002) foi o criador da programação estruturada. Antes da programação orientada a objetos, os códigos seguiam a vertente chamada de estruturada. Nela, o código era simplesmente organizado em estruturas lógicas básicas (laços, condicionais) e sua divisão e estrutura principal ficava ao critério do desenvolvedor. Orientação a objetos utiliza programação estruturada, mas cria uma divisão e abstração de dados mais próxima do mundo real. Dijkstra, como normalmente é chamado, elaborou um grande número de diagramas para a época, além de blocos de código.

---

Em nossa fase de análise podemos mencionar que temos em nosso problema, professor e alunos, que tanto professor como aluno terão vários atributos como nome, documento, endereço e outros; ambos terão de cumprir suas funções que são os métodos de realizar atividades na aula. Agora entenderemos como analisar o problema e transformar as estruturas em unidades que são mais que estruturadas, são classes e objetos.

### 3.2.2 Estruturas básicas do paradigma orientado a objetos



Existem algumas estruturas básicas que temos de compreender ao iniciar nosso trabalho, vamos falar sobre elas agora e, para isso, as listamos a seguir.

**Classe:** é a estrutura principal da orientação a objetos. Deve ser imaginada como uma abstração de um dado, como uma forma, que estabelece como um objeto (coisa, algo, um processo, ou ser vivo) atua no mundo real. Deve definir suas características que são chamadas de atributos, e suas ações, chamadas de métodos, ela é usada para modelar a aplicação. Já os objetos são instâncias da classe e determinam cópias desta “forma” na memória (PFLEEGER, 2004).

Por exemplo, podemos imaginar que, em uma sala, temos como classes *Professor* e *Alunos*, mas suas instâncias poderiam ser vistas como os alunos na sala, digamos 30 ou 40 e um professor, que seria uma instância única.

<pre><b>public class</b> Aluno {     <b>int</b> matricula;     <b>String</b> nome;     <b>String</b> endereco;     <b>Disciplinas</b>[] disciplinas;      <b>public void</b> realizaAtividadeAula();     <b>public void</b> registraEntradaAula();     <b>public void</b> registraSaidaAula(); }</pre>	<pre><b>public class</b> Professor {     <b>Long</b> cpf;     <b>String</b> nome;     <b>String</b> endereco;     <b>Disciplinas</b>[] disciplinas;      <b>public void</b> realizarAulaDia();     <b>public void</b> registraEntradaAula();     <b>public void</b> registraSaidaAula(); }</pre>
--	--

Figura 6 - Duas classes de nosso sistema, professor e aluno, que denotam os atributos (variáveis) e ações (métodos). Fonte: Elaborado pelo autor, 2018.

Na descrição das classes *aluno* e *professor*, no exemplo dado, existem algumas características que são atributos, como, por exemplo, nome e matrícula, e algumas atividades realizadas por eles, que são os métodos, como vimos na figura acima.

### 3.2.3 Princípios fundamentais para implementação orientada a objetos

Utilizada corretamente, a orientação a objetos para o desenvolvimento do sistema, dado principalmente a características listadas pelos pesquisadores Rumbaugh, Jacobson e Booch (2005), pode-se diminuir consideravelmente o custo e o retrabalho no desenvolvimento. Vamos conhecer as características a seguir.

**Modularidade:** possibilidade de criar pacotes, ou módulos separadamente, como blocos.

**Encapsulamento:** vem de encapsular ou separar em partes, de uma forma isolada, para tornar o *software* flexível e de manutenção mais fácil, pois deverá o código ficar dividido em partes menores e mais legíveis. Também vai definir o nível de acesso aos dados, devido ao encapsulamento, pode-se dizer se o acesso a determinado atributo ou método é: privado, ou seja, só pode ser acessado dentro da classe; ou público, visível para fora da classe.

**Herança:** mecanismo que permite criar classes, a partir de outras já existentes, reaproveitando as características existentes da classe dita “pai” ou “superclasse”, na classe “filha” ou “subclasse”. Promove um grande reúso de código existente.

**Polimorfismo:** depende totalmente da herança, utilizando polimorfismo podemos redefinir na subclasse um método na superclasse, ou seja, na classe pai, um método realiza algo e na classe filha, outra coisa.

Vamos ver um exemplo:

```

public class Pessoa {
    String nome;
    String endereco;
    Disciplinas[] disciplinas;
    public void registraEntradaAula();
    public void registraSaidaAula();
    public void realizarAtividade();
}

public class Aluno extends Pessoa{
    int matricula;
    public void realizarAtividade();
}

public class Professor extends Pessoa{
    Long cpf;
    public void realizarAtividade();
}

```

Figura 7 - Utilizando refatoração de dados, consideramos usar herança para definir uma classe Pessoa, que manterá os dados comuns de Aluno e Professor, o método “realizarAtividade” é polimórfico. Fonte:

Elaborado pelo autor, 2018.

Utilizando herança e polimorfismo, note na figura acima, que o que é comum para aluno e professor, passou para uma classe superior chamada *Pessoa*, os métodos “realizarAtividadeAula” e “realizarAulaDia”, foram substituídos por um método “realizarAtividade”, comum a Professor e Aluno, mas que realizam ações diferentes (Polimorfismo).

---

## VOCÊ SABIA?

Duas das principais características desejáveis da orientação a objetos são coesão e acoplamento, sendo coesão, o quanto uma estrutura ou classe realiza o que deve fazer, e acoplamento considera o quanto uma classe é ligada à outra. O ideal em um sistema orientado a objetos é que ocorra alta coesão e baixo acoplamento.

A procura e aperfeiçoamento no estudo de orientação a objetos e seu emprego no desenvolvimento, garantirá, para o desenvolvedor, um sistema com menos erros e com manutenção mais simples, além de permitir que seja mantido por outras

equipes mais facilmente, são conceitos já antigos, mas que são atualizados todos os dias.

### 3.2.4 UML

A linguagem UML (*Unified Modeling Language*) destina-se a documentar as diferentes versões do sistema, desde a especificação e construção, e pode ser usada em todas as etapas de desenvolvimento. Foi definida inicialmente por Jim Rumbaugh e Grady Booch ao combinarem dois métodos populares de modelagem orientada a objeto: Booch e OMT (*Object Modeling Language*).

Mais tarde, Ivar Jacobson, o criador do método *Objectory*, uniu-se aos dois (formando os famosos três amigos), para a concepção da primeira versão da linguagem UML (*Unified Modeling Language*). Ela apresenta diversos diagramas, que permitem representar a estrutura lógica e física do *software* orientado a objetos e a descrição de seu comportamento, ou seja, trata-se de uma linguagem visual para documentação de projeto e padrões de *software* (RUMBAUGH; JACOBSON; BOOCH, 2005). Vamos utilizar alguns dos diagramas UML para descrever a arquitetura lógica do *software* da sala de aula, segundo Sommerville (2011).

---

## VOCÊ SABIA?

No *site* <<https://www.omg.org/spec/UML/>> (<https://www.omg.org/spec/UML/>) você pode encontrar as definições e toda a história da UML, as diferentes versões, estratégias para uso da linguagem. Essa é a referência oficial, na qual se encontram as últimas versões, como a 2.5.1, por exemplo, foi criada em dezembro de 2017 (UML, 2018).

Os Diagramas da UML estão divididos em Estruturais e Comportamentais.

### Diagramas estruturais

Diagramas estruturais tratam o aspecto do ponto de vista do sistema, quanto aos seus componentes e classes. Sobretudo tenta definir a parte estática do sistema, seu esqueleto, as estruturas e tipos principais, mas não como elas se comunicam.

Como aspectos estáticos de um sistema de *software*, podem ser citados itens como classes, interfaces, colaborações, componentes. E é representado por vários diagramas, dentre os quais, consideramos os diagramas abaixo, como principais, pelo seu uso mais ostensivo (RUMBAUGH; JACOBSON; BOOCH, 2005).

## Diagrama de classes

Uma classe é essencialmente um modelo, a partir do qual, qualquer número de objetos pode ser derivado. Ela não existe como um objeto por si só, mas define as propriedades (ou atributos) que um objeto terá e as operações que podem ser executadas pelo objeto.

Como pode ser visto na figura a seguir, a classe é definida por um retângulo, dividido em três seções por linhas horizontais. A seção superior contém o nome da classe, a seção intermediária contém os atributos de classe.

Existem sinais para indicar se os atributos e métodos de classes são públicos ou privados. A seção inferior contém os métodos que podem ser executados pela classe, são as ações que podem ser realizadas. Classificar os atributos e as operações da classe UML como privados, protegidos, ou públicos reflete a maneira como as variáveis de classe e os métodos, nas linguagens de programação orientadas a objetos são declarados (MEDEIROS, 2004).

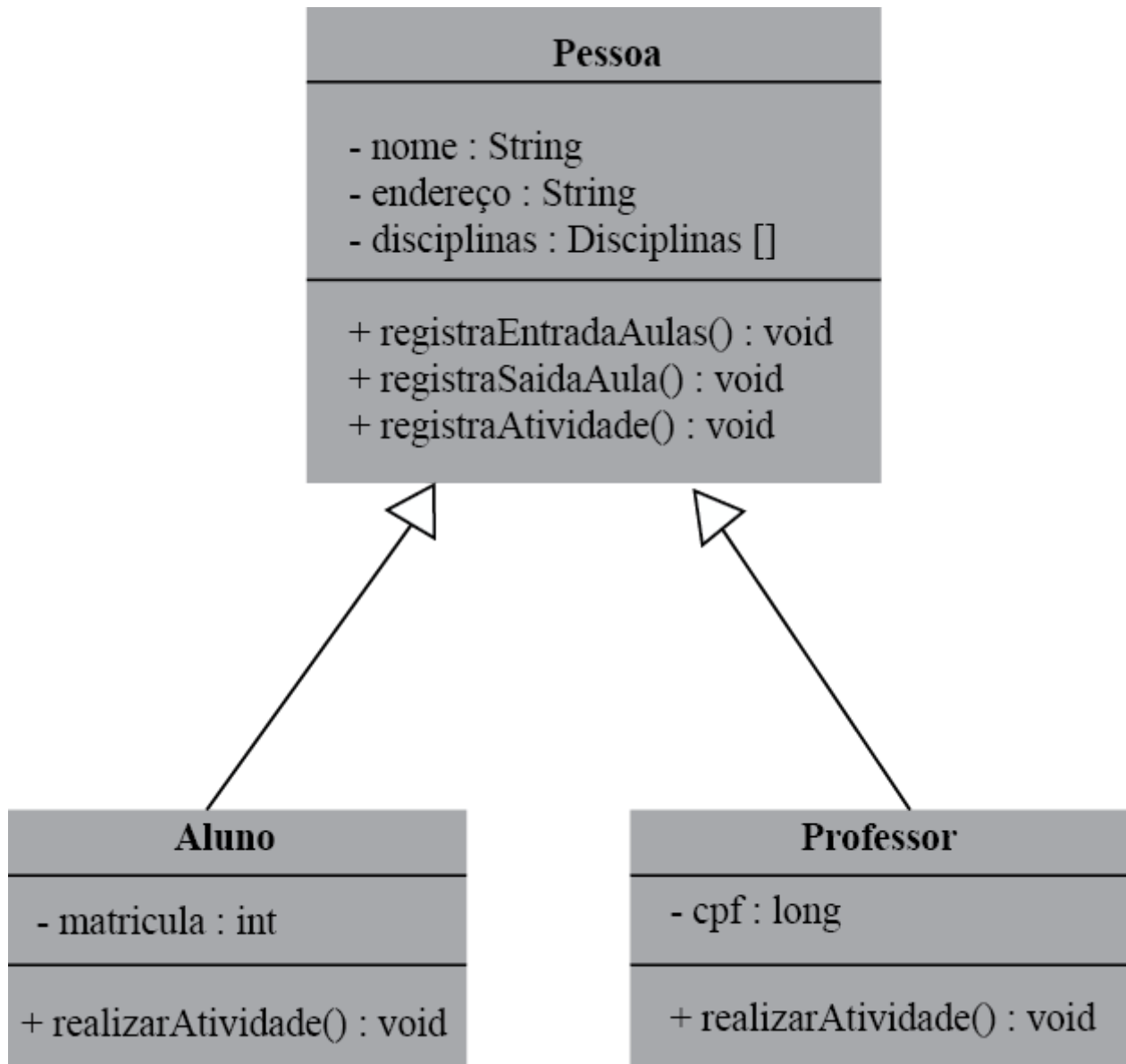


Figura 8

- Exemplo de diagrama de classes, note que Professor e Aluno, como mencionado no código acima, tem relacionamento com Pessoa. Fonte: Elaborado pelo autor, 2018.

## Diagrama de objetos

Diagramas de objetos são muito semelhantes aos diagramas de classes, um objeto é uma instância de uma classe. Ambos, diagrama de classes e diagramas de objetos, representam uma visão estrutural estática de um sistema. Este diagrama é menos utilizado, às vezes pode ser útil para demonstrar a maneira como os objetos interagem em um sistema "em tempo de execução" (ou seja, enquanto o sistema está realmente em execução), como na figura a seguir, isto porquê o diagrama de classes trata somente da visão totalmente estática do sistema, antes de ser executado, pois a classe é como uma forma, representa as características do sistema, de forma estática.

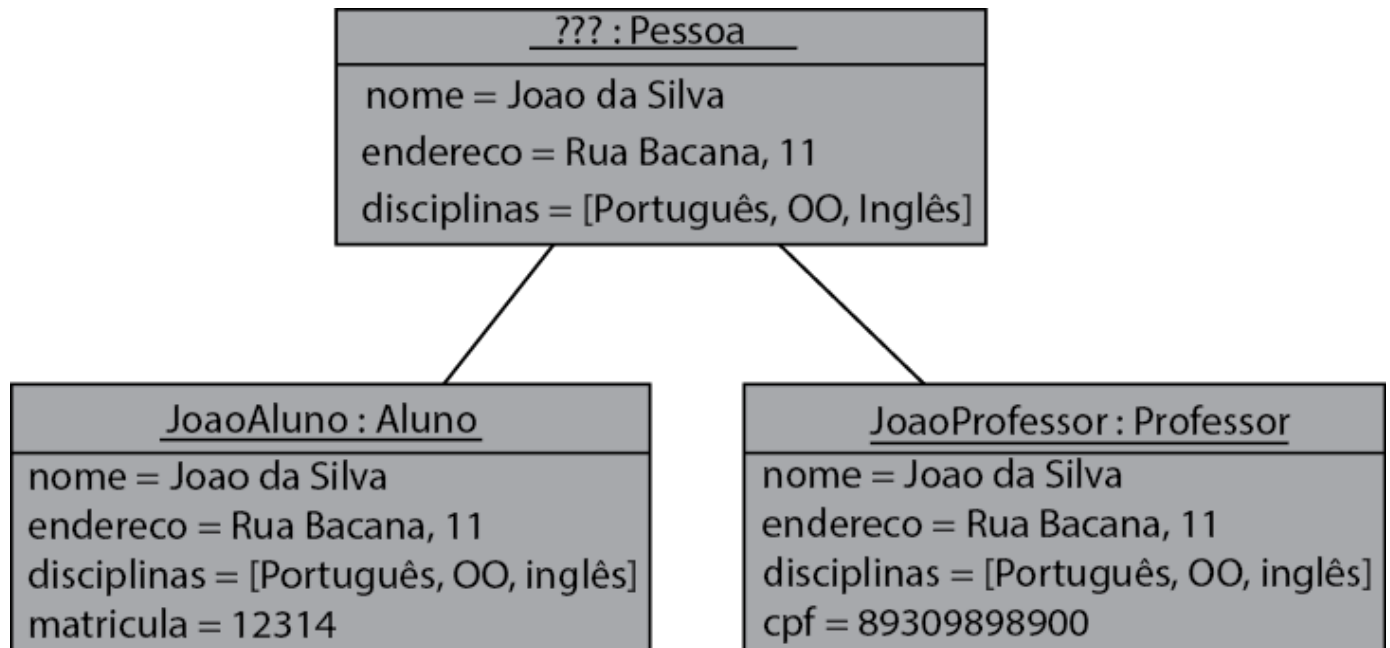


Figura 9 - Exemplo de diagrama de objetos, o professor ou aluno, quando vêm de pessoa, trazem os dados. Fonte: Elaborado pelo autor, 2018.

O diagrama de objetos pode ser usado para mostrar como uma pequena parte do sistema funciona, quando isso não é imediatamente óbvio, ao olhar para um diagrama de classes. Enquanto um ícone de classe representa todas as instâncias, de uma classe específica, que existem, existiram ou existirão em um sistema, um ícone de objeto representa exatamente uma instância de uma classe, em um determinado momento, com seus atributos normalmente preenchidos com exemplos (RUMBAUGH; JACOBSON; BOOCH, 2005).

## Diagramas comportamentais

Os diagramas comportamentais, ao contrário dos estruturais, não tem o objetivo de mostrar somente a estrutura, mas a dinâmica que nela existe, quando são transferidos dados e mensagens entre as classes.

## Diagrama de Casos de Uso

Utilizado pelos desenvolvedores e analistas para, logo no início do projeto de *software*, determinar o escopo do projeto, do ponto de vista do usuário, muito abstrato, contém muitas informações de negócios e descreve as principais funcionalidades do sistema, como pode ser visto na figura a seguir. Conhecido por

ser utilizado logo no início do projeto de *software*, pois é muito utilizado para documentar o sistema do ponto de vista do usuário. Seu nível de abstração é muito alto e escreve as principais funcionalidades do sistema e a interação dessas funcionalidades com os usuários do sistema. Não tem características técnicas, mas, por outro lado, é base para toda a criação de vários diagramas que vimos anteriormente como o diagrama de classes, que só pode ser feito corretamente após a definição dos casos de uso (MEDEIROS, 2004).

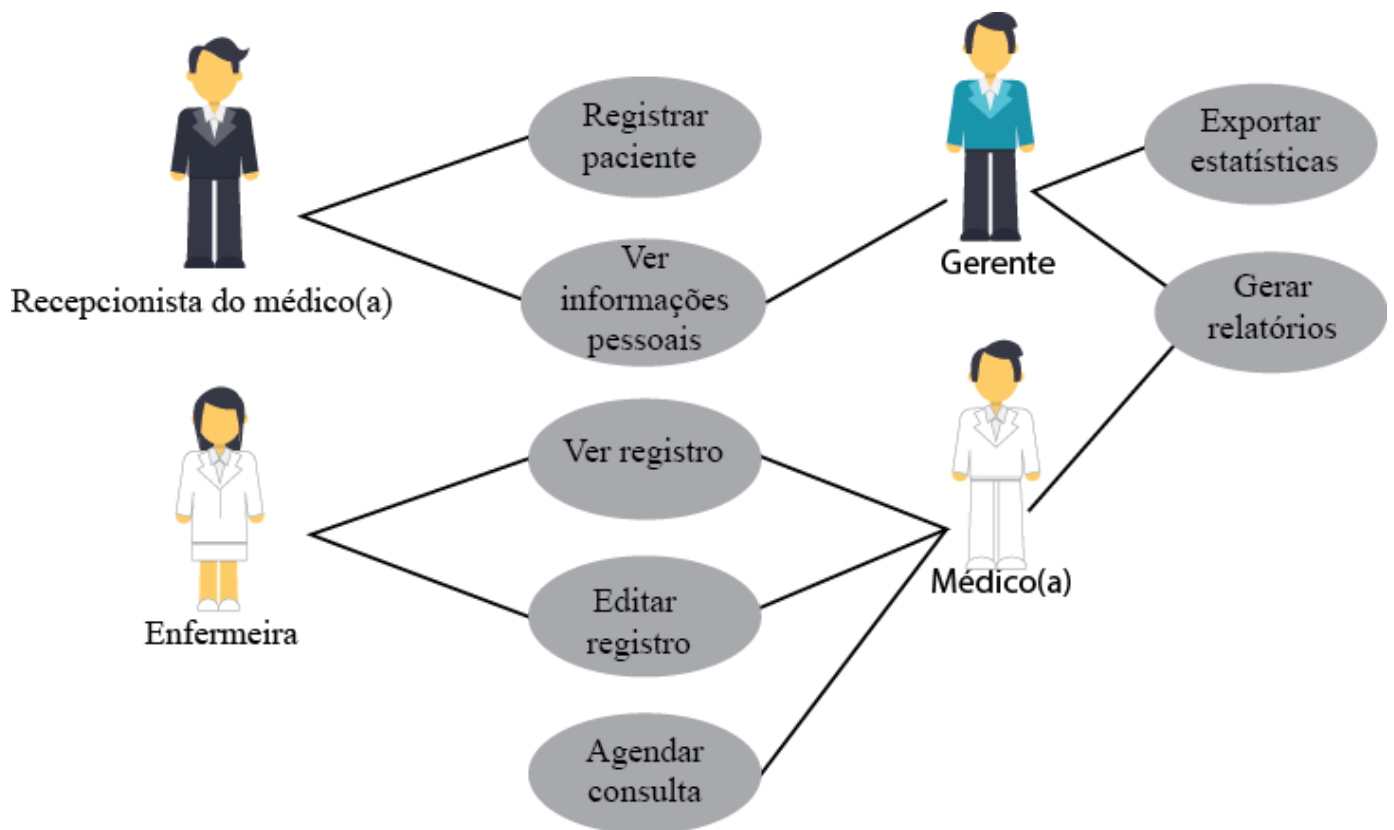
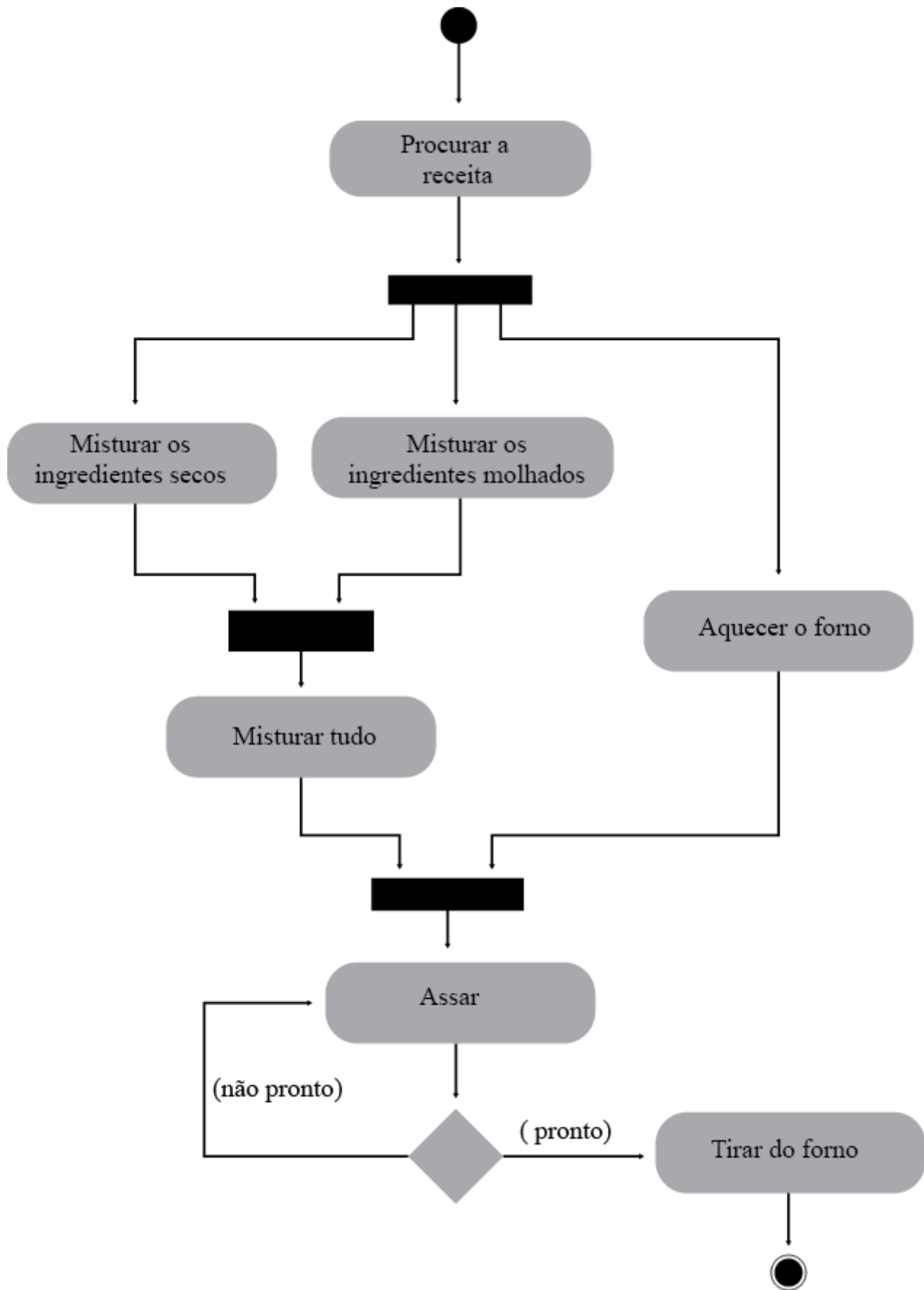


Figura 10 - Exemplo de Diagrama de Caso de Uso. Fonte: SOMMERVILLE, 2011, p. 75.

## Diagrama de atividades

O diagrama de atividades representa fluxo e processamento. É muito útil para o entendimento de como os dados fluirão pelo sistema e como as decisões são tomadas dentro dele. O diagrama pode ter diferentes graus de abstração, pois como é utilizado para definir algum processo dentro do sistema, pode servir para detalhar alguma dinâmica de um Caso de Uso, algum processo de uma classe, algum momento específico, baseado em algum fluxo (MEDEIROS, 2004).





Figura

11 - Exemplo de diagrama de atividades. Fonte: PRESSMAN, 2016, p. 882.

## Diagrama de sequência

Dentre os diagramas de interação (sequência, geral interação e comunicação) é o principal, mostra a ordem temporal em que as mensagens são trocadas entre os objetos, ou seja, quais condições devem ser satisfeitas e quais métodos devem ser disparados dentre os objetos envolvidos e, em que ordem, durante um processo.

Dessa forma, determinar a ordem na qual os eventos ocorrem, as mensagens que são enviadas, os métodos que são chamados e como os objetos interagem entre si, dentro de um determinado processo é o principal objetivo deste diagrama. É um diagrama complexo de fazer e manter, pois depende muito do código implementado (projeto), como vemos na figura a seguir. As chamadas, ou mensagens, são trocadas entre as classes implementadas, mostrando qual dado foi enviado, ou método foi disparado, ou seja, qualquer alteração de chamada entre as classes significa, na prática, uma alteração deste artefato (MEDEIROS, 2004).

Sistema de informações meteorológicas

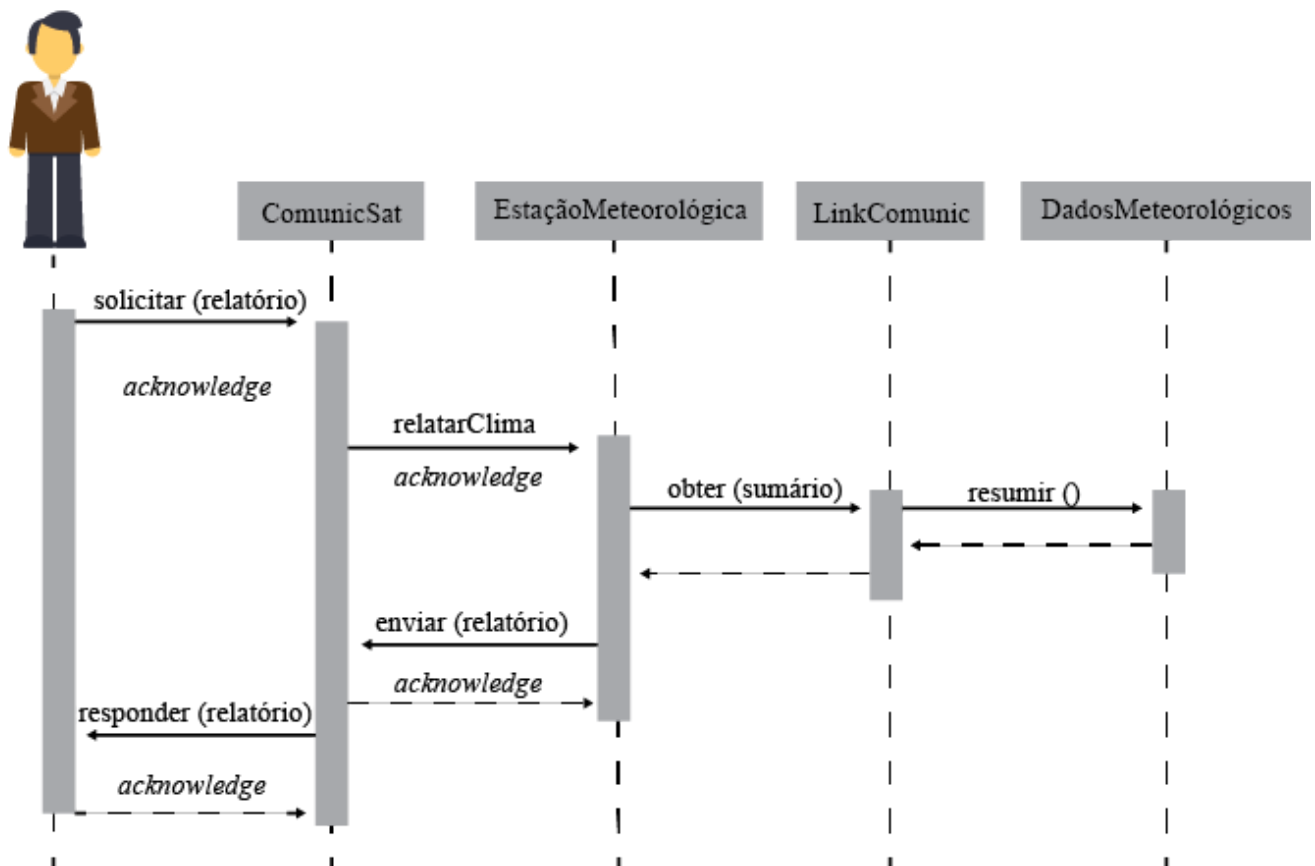


Figura 12 - Exemplo de diagrama de sequência. Fonte: SOMMERVILLE, 2011, p. 154.

Normalmente, o diagrama de sequência envolve um ator, o boneco visto na figura acima, que dispara o evento inicial de tela em um sistema. Este evento ou mensagem, vai de classe em classe, muitas vezes, até as classes de domínio, ou a classe responsável por persistir o dado. Depois o dado retorna para o nível do ator que iniciou o processo.

## io

apanhar um caso sobre uma empresa que desenvolveu um *software* para gestão de pessoas e folha de . O produto era um *software* de prateleira, que é vendido pronto, ou seja, não pode ser modificado e tem lido como um todo. O *software* foi um sucesso de vendas, mas quando a empresa cresceu, por alguma e uma queda de qualidade. Novos desenvolvedores tiveram de ser contratados e aspectos importantes careciam de maiores informações.

ndo o produto atingiu um tamanho de ERP, que atendia desde folha de pagamento até o setor financeiro, ção do sistema tornou-se praticamente impossível. Isso porque o sistema passou a ser mantido e por uma equipe de cinco a seis desenvolvedores, e sem eles, ninguém mais sabia alterar o código com

envolvedor saísse do projeto, acarretava anos de adequação e treinamento para um novo, ou seja, mais io.

ue o criador original do sistema teve a ideia de fazer o que se chama de engenharia reversa do sistema, intratar analistas para redigir todos os diagramas da UML, para documentar o quanto achasse necessário. o conhecimento passou a ser registrado, não estava mais somente na cabeça do desenvolvedor, o que ia transmissão. Com isso, compreendemos que a maior riqueza de um sistema não é o código, mas sua ção.

Vimos uma mostra dos principais diagramas, mas existem vários outros para várias finalidades, como de componentes, por exemplo.

## 3.3 Padrões de projetos

*Design patterns*, ou, daqui para frente, padrões de projetos, são formas preestabelecidas de resolver um problema já conhecido (BECK, 2011). De forma menos técnica, é usar uma solução já estudada e documentada, para o problema

que está na sua frente. Normalmente essas soluções são notoriamente conhecidas, comuns a várias equipes de desenvolvimento e podem ser repetidas. Com o uso de padrões é criada uma forma comum de falar sobre uma implementação.

Sobre os padrões em si, é importante entender que não existe também uma fórmula mágica, ou uma “bala de prata”, os padrões são reconhecidos por serem amplamente utilizados com sucesso. Não significa que nunca houve fracassos, mas, seu uso, embora implique em um risco significativamente menor para o projeto de *software*, não garante o sucesso completo. Em sua implementação, tampouco um padrão deve ser seguido cegamente, ele é uma abstração, um protótipo que dita linhas gerais e que tem de ser refinado na construção do sistema.

### **3.3.1. Principais características para utilizar os padrões de projetos**

Os padrões de projeto utilizam principalmente da experiência de especialistas em desenvolvimento que já foram comprovadas, para definir um vocabulário comum a todos os desenvolvedores, e resolver problemas específicos. Eles não são estratégias de implementação e, sim, blocos de código comprovados para reúso.

### **3.3.2 Atributos de padrões de projeto**

As características de um padrão de projeto variam muito, conforme sua utilização, quem o criou e seus objetivos. Existem, na verdade, várias vertentes, embasadas em autores diferentes, como, por exemplo, padrões para *web*, ou padrões básicos, como os definidos pelo GoF (em inglês, *Gang of Four*, traduzido como gangue dos quatro) veja no leia mais.

---

## **VOCÊ QUER LER?**

Até hoje, o livro mais famoso sobre padrões, na história, foi escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, a chamada gangue dos quatro. O livro, denominado “*Design Patterns: elements of reusable object-oriented software*”, definia, na época, uma série de padrões para desenvolvimento e é, até hoje, o trabalho mais importante da área.

## 1. Padrões de criação (*creational*)

- *Abstract Factory*: uma classe abstrata, que encapsula métodos de uma fábrica, *factory*.
- *Builder*: cria diferentes representações, dentro de um mesmo processo de criação.
- *Factory Method*: instancia objetos, mantendo isoladas as classes concretas da sua forma de criação.
- *Prototype*: gera tipos arbitrários, de acordo com parâmetros passados.
- *Singleton*: garante que, para uma classe específica, só possa existir uma única instância, para todo o sistema, a qual é acessível de forma global.

## 2. Padrões de estrutura (*structural*)

- *Bridge*: cria uma “camada” interna, pois desacopla a implementação da interface.
- *Composite*: realiza uma composição, ou seja, agrupa elementos de forma hierárquica.
- *Adapter*: considerando a necessidade de se ter interfaces de dados compatíveis, permite que dois objetos se comuniquem, as utilizando.
- *Decorator*: “decora” um objeto, ou seja, atribui características adicionais de forma dinâmica, com se fosse uma extensão.
- *Facade*: muito utilizado, gera uma interface unificada para um subsistema, permitindo expor, ou não, partes de seu sistema ou subsistema para um sistema externo.
- *Flyweight*: gera compartilhamento para dar suporte a vários objetos, de forma mais prática e mais integrável.
- *Proxy*: gera um objeto que será representante para passar dados, ou acessar outro objeto de forma a realizar algum tipo de ação neste segundo, normalmente acesso.

## 3. Padrões de comportamento (*behavioral*)

- *Iterator*: cria uma forma de percorrer uma estrutura.
- *Memento*: se necessário guardar um estado, o memento salva o estado de um objeto para que depois seja restaurado.

- *Observer*: observa uma situação, permite uma relação de dependência 1:N e, quando um objeto muda seu assunto, os objetos que estão observando são notificados.
- *State*: como se fosse um semáforo, permite ao objeto alterar seu comportamento, de acordo com alguma situação interna.
- *Chain of responsibility*: gera um barramento e evita dependências entre objetos, pois todos consultam e se comunicam por este barramento.
- *Command*: associa uma interface a uma ação específica.
- *Interpreter*: é um interpretador de linguagem natural para uma funcionalidade.
- *Mediator*: é um objeto que atua como interface entre dois outros objetos, tratando suas mensagens.
- *Strategy*: gerencia toda uma estrutura à parte do sistema, como se fosse um subsistema.
- *Template method*: gera um esqueleto para definir estruturas para uma subclasse.
- *Visitor*: uma classe independente que vai acessar métodos específicos em outras estruturas do sistema.

## 3.4 Implementação de *software*

Neste tópico, consideraremos os passos essenciais para gestão de processos de implementação de *software*. Entenda este contexto por todas as atividades necessárias para construir o código, algumas mais evidentes e conhecidas, mas falaremos mais das atividades específicas, que estão inseridas em todo o processo de desenvolvimento do produto.

### 3.4.1 Implementação de *software* como fase de um projeto

Como é entendida dentro das fases de projeto: pensar na forma de desenvolver, torna a compreensão e a mudança no *software* mais fácil e, provavelmente, reduzirá os custos de evolução e aplicação de boas técnicas de engenharia de *software*.

Dependendo do tamanho do projeto, segundo Sommerville (2011), a fase de implementação representa e atua durante todo o projeto, principalmente em projetos menores, já em maiores, acaba sendo melhor definida e mais facilmente gerenciada.

Geralmente, é preferível investir o maior esforço na implementação de um sistema, do que correr riscos de mudanças futuras por erros. Geralmente adiciona uma nova funcionalidade, após a liberação, é mais caro por várias razões como: estabilidade da equipe, que pode ter mudado, más práticas de desenvolvimento cujos problemas podem ter de ser resolvidos depois, qualificação da equipe, pois, normalmente, a equipe que desenvolve, tende a ser mais experiente que a equipe que realiza manutenção, e também a tecnologia, que pode ter decaído, ou não ser mais perfeitamente conhecida.

### **3.4.2 Engenharia de software para implementação**

#### **Reúso**

O reúso é uma das características mais desejáveis de um sistema, entretanto complexa, pois depende muito da fase de requisitos. Isso mesmo! Requisitos.

Definir quando um requisito pode ser um componente, ou utilizar um componente já pronto não é uma tarefa fácil, posteriormente, é possível, por reengenharia, mudar parte do código para reutilizar um componente, mas também é uma tarefa complexa.

Normalmente, pode-se adquirir um conjunto de *softwares* para realizar atividades reutilizáveis para tela, acesso a dados e outras funções, ou criar um repositório próprio, com seus blocos de código reutilizáveis. A verdade é que, com o reúso, os projetos tornam-se menores, os códigos mais seguros, pois utilizam partes comuns e conhecidas de código, por isso mais testadas (SOMMERVILLE, 2011).

#### **Gestão de configuração**

A gestão de configuração trata de todas as atividades para administrar alterações, e elas ocorrerão, e muito. Atualmente existem vários modelos abertos para tal, baseados, por exemplo, em GIT (*Global Information Tracker*). Em determinado momento na linha de evolução do seu sistema, será necessário retornar a uma versão anterior de *software*, para verificar o que foi alterado, pesquisar e comparar

entre diferentes versões ou, até mesmo recuperar um código perdido, e isto tem de ser feito de forma organizada, utilizando um sistema, não somente utilizando *backups* na máquina ou cópias esporádicas.

Atualmente, em um sentido mais amplo, existe o conceito de DevOps (*Development and Operations*), que considera que, mesmo durante a implementação, deve existir uma extensa integração entre os desenvolvedores e os responsáveis por infraestrutura.

## **Framework de aplicação**

O uso de um *framework* para implementar uma estrutura padrão, assim como o reúso, é muito útil, pois retira do desenvolvedor, a necessidade de iniciar um trabalho, desde a primeira linha de código. Os *frameworks* tornaram-se mais populares com o surgimento de interfaces gráficas e, em muitos casos, permitem criar toda a estrutura do aplicativo integrando desde a tela, até o banco de dados. Em muitas vezes também, os desenvolvedores “herdam” características de classes do *framework* para suas aplicações, utilizando o cabedal visto anteriormente para orientação a objetos.

Existe uma infinidade de *frameworks* disponíveis no mercado, basta uma pesquisa para encontrar diversas opções. Mas aqui, novamente, vale o princípio da “bala de prata”, pois um *framework* vai resolver apenas parte do problema, cabe ao engenheiro, arquiteto, ou implementador, pesar por experiência, ou por exemplos, o que é mais desejável.

Existem exemplos de *frameworks* livres, feitos por grandes empresas, como partes de projetos como Mozilla, OpenOffice, KDE, NetBeans e Eclipse.

---

## **VOCÊ QUER LER?**

Muitas vezes não está claro como realizar um processo de desenvolvimento mais formalmente, e embora haja uma boa quantidade deles, não é demais ler mais alguns processos, abaixo encontra-se o processo de desenvolvimento para Java (da empresa Oracle, a proprietária), é bem extenso por ser mais técnico e menos abstrato, vale a referência para estudos futuros e implementações futuras também (SOUZA, 2014). Leia mais: <<http://www.oracle.com/technetwork/pt/articles/adf/desenvolver->



apps-com-oracle-adf-2225340-ptb.html

(<http://www.oracle.com/technetwork/pt/articles/adf/desenvolver-apps-com-oracle-adf-2225340-ptb.html>)>.

## XP

Iniciativas baseadas em *Extreme Programming*, também devem ser levadas com muita seriedade. Programação em pares por exemplo, é uma técnica na qual, mais de um desenvolvedor trabalha em conjunto intercalando o trabalho para implementar o código, parece perda de tempo? Não! Definitivamente.

Adotando alguma técnica de programação em par, trará um compartilhamento de conhecimento entre os desenvolvedores e fará com que eles compartilhem a informação. Procure mais!

Este é só um exemplo, pois se verificar no diagrama a seguir, que é típico do movimento de programação XP, verá que existem várias atividades inerentes ao desenvolvimento que, por si só, renderiam todo um trabalho, mas não deixe de pesquisar mais sobre *design* simples, ritmo sustentável, metáforas de pequenas versões.

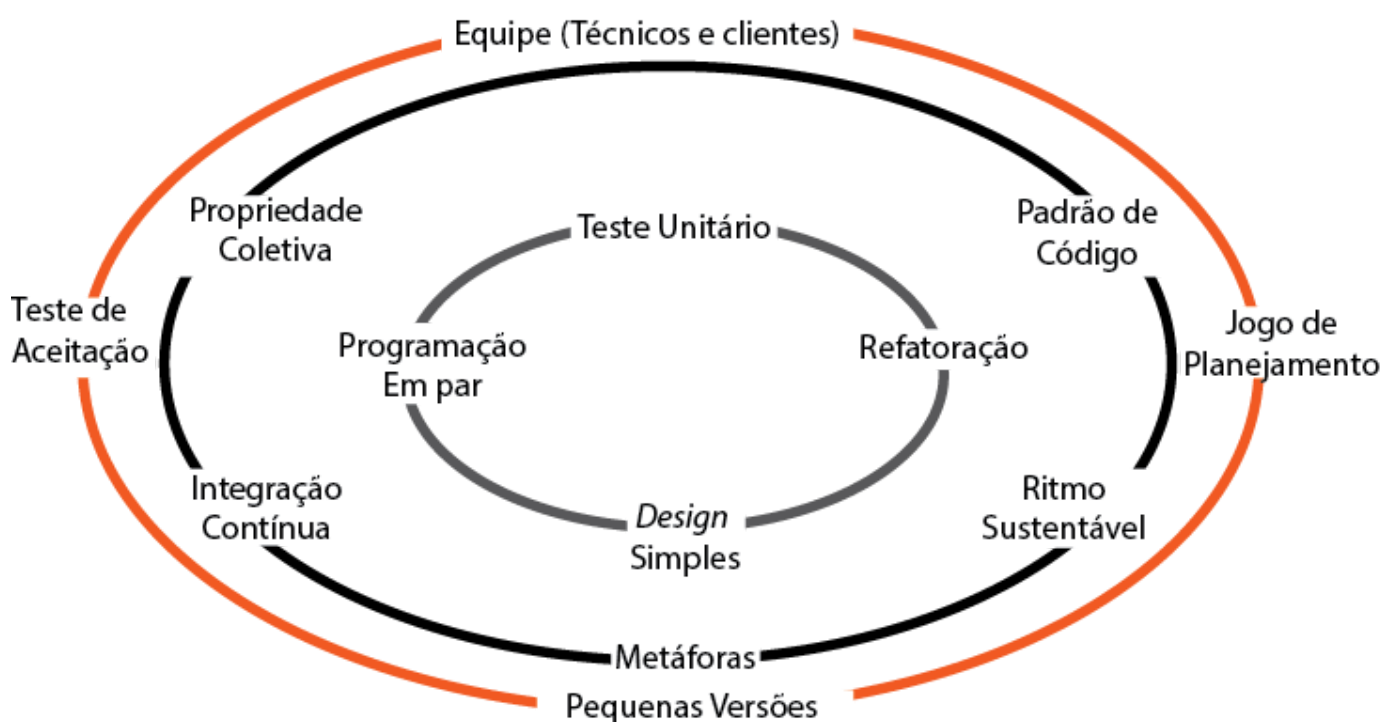


Figura 13 - Estrutura de técnica XP, que Kent Beck comparou com uma cebola, pelas camadas. Fonte: BECK, 2011, p. 4.

Assim como qualquer metodologia, existem vantagens e desvantagens no uso do XP, o uso intensivo de técnicas de interação entre grupos de programadores, gera uma forte sensação de que o código está sempre sendo revisto e atualizado. Entretanto, muitos desenvolvedores não gostam da ideia de ter seu código constantemente alterado e criticado por todo o grupo. Mas, é necessário ter em mente que codificar é uma atividade intelectual e existem muitas características sensíveis nesse processo. Para nós, como seres humanos, e para uma equipe, é importante sempre usar de muito diálogo e tentar manter os desenvolvedores à vontade.

### 3.4.3 Desenvolvimento open source

Livre não é grátis! Se a questão é contrapor ou defender esta vertente entenda que os dois termos não são sinônimos, nem próximo disso, deve-se entender que o preceito principal de *software* livre é, como o próprio nome diz, a possibilidade de tratar com níveis para liberdade para uso, ajustá-lo à sua maneira e adaptá-lo. Talvez, melhor que *software* livre, seria chamar de aberto, como propõe Eric Raymond, em seu livro “A Catedral e o Bazar”. Ao contrário de livre, *software* aberto é uma evolução mais comercial para esta vertente, que trata não em virtude de preço, mas sim, com direito de uso do código fonte (FERSTE, 2017).

Atualmente, o uso do *software* aberto, ou livre, é inevitável para qualquer empresa ou pessoa, seja ela produtora ou usuária. Sua barreira de entrada, em qualquer plataforma, é praticamente zero e, normalmente, qualquer *software* aberto está apenas a um clique de distância, em geral basta acessar um *site* e fazer *download*.

---

## VOCÊ QUER VER?

Por liberdade, tem de se entender também liberdade sobre dados e informação e não somente no desenvolvimento de *software*. O filme *Snowden, herói ou traidor* (FITZGERALD; STONE, 2016) trata deste ponto. Quando Edward Snowden vai a público divulgar uma série de informações de espionagem, ninguém acreditava na seriedade das acusações, mas assim que publicou uma série de documentos sigilosos, comprovando os atos, sua vida mudou, e em prol de suas convicções, Snowden perdeu sua liberdade para defender cidadãos comuns.

---

## Modelos de trabalho com *software* livre consolidado

Hoje, certamente, é possível afirmar com grande possibilidade de acerto que, pelo menos 30% dos *softwares* em seu computador sejam livres, soluções como navegadores, editores de texto, plataformas de desenvolvimento, mesmo também tecnologias são muito utilizadas sem nenhum retorno ao seu desenvolvedor.

Isto se deve ao fato de que as licenças permitem vários modelos de negócio e distribuição, alguns um pouco ultrapassados, mas muitos persistem, demonstrando que várias formas de trabalho podem ocorrer, utilizando essa vertente.

Atualmente, é uma tendência a venda de serviços e licenças mais baratas para versões mais simples de *software*. Muito comum também é a criação de variações de adaptações realizadas por empresas menores, utilizando código aberto que, inicialmente, não atendia a um determinado problema, mas, após um processo de manutenção adaptativa passa a atender várias necessidades para empresas de uma região.

Fomentar o uso de *software* livre implica em incentivos para novas empresas desenvolverem produtos mais adaptados ao uso regional, que o *software* proprietário, cujas configurações permitem poucas adequações além da língua. Além disso, basicamente, nos proprietários, os polos de enriquecimento tendem a ser centralizados.

# Síntese

Neste capítulo descrevemos, de uma forma concisa, os principais conceitos conhecidos em engenharia de *software*, aplicados para os passos de desenvolvimento, chamados de implementação.

Neste capítulo, você teve a oportunidade de:

- distinguir sobre o uso de arquiteturas de *software*;
- aprender conceitos sobre orientação a objetos e uso de padrões para desenvolvimento;
- esclarecer regras gerais para serem seguidas durante a implementação;

- contextualizar o que é *software* aberto e livre e a necessidade de utilizá-lo.



◀ Clique para baixar o conteúdo deste tema.

# Bibliografia

BECK, K.; WARD, C. **Using Pattern Languages for Object-Oriented Programs**. Submitted to the OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming. Technical Report No. CR-87-43, September 17, 1987. Disponível em: <<http://c2.com/doc/oopsla87.html>>. Acesso em: 3/8/2018.

BECK, K. **Padrões de Implementações**. Porto Alegre: Bookman, 2011.

FERSTE, M. Liberdade para *software*, princípios, fatos, pessoas ilustres e você. **Revista Eletrônica Conhecimento Iterativo**, 2017.

FOWLER, M. **Microservices**. 2014. Disponível em: <<https://www.martinfowler.com/articles/microservices.html>>. Acesso em: 3/8/2018.

JAVA. **Java e você, faça download hoje**. Portal Java, Oracle. 2018. Disponível em: <<http://www.java.com> (<http://www.java.com>)>. Acesso em: 3/8/2018.

JAVA2S. **Java2s**. Portal Java, 2018. Disponível em: <<http://www.java2s.com> (<http://www.java2s.com>)>. Acesso em: 3/8/2018.

MEDEIROS, E. **Desenvolvendo Software com UML 2.0 Definitivo**. São Paulo: Pearson Makron Books, 2004.

PFLIEGER, S. L. **Engenharia de Software - Teoria e Prática**. 2. ed. São Paulo: Pearson Addison Wesley. 2004.

PRESSMAN, R. **Engenharia de Software**. 8. ed. Porto Alegre: Bookman, 2016.

PRESSMAN, R.; MAXIM, B. R. **Software Engineering**: a practitioner's approach. 8. ed. MC Graw Hill, 2015.

REENSKAUG, T. M. H. **MVC** (<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>) - **XEROX PARC 1978-79**. Publicado em 22 March 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html> (<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>)>. Acesso em: 3/8/2018.

RICHARDS, M. **Software Architecture Patterns**. Sebastopol (CA): O'Reilly, 2015.

RUMBAUGH J; JACOBSON I; BOOCH, G. **The Unified Modeling Language Reference Manual**. 2. ed. São Paulo: Pearson Education, 2005.

SHAW, M.; GARLAN, D. **Software Architecture**: perspective on an emerging discipline. New Jearsey: Pearson, 1996.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Education, 2011.

SOUZA, W. **Começando a desenvolver aplicações utilizando Oracle ADF**. Portal Oracle. Postado em Junho 2014. Disponível em: <http://www.oracle.com/technetwork/pt/articles/adf/desenvolver-apps-com-oracle-adf-2225340-ptb.html> (<http://www.oracle.com/technetwork/pt/articles/adf/desenvolver-apps-com-oracle-adf-2225340-ptb.html>)>. Acesso em: 3/8/2018.

FITZGERALD, K.; STONE, O. **Snowden, herói ou traidor**. Direção: Oliver Stone. Produção: Moritz Borman; Eric Kopeloff; Philip Schulz-Deyle; Fernando Sulichin. USA. Open Road Films, 2016.

UML. **Object Management Group**. Disponível em: <https://www.omg.org/spec/UML/> (<https://www.omg.org/spec/UML/>)>. Acesso em: 3/8/2018.