Design Patterns, Estilos e Padrões Arquiteturais

Arquitetura de Software

Gladston Junio Aparecido

2021

Design Patterns, Estilos e Padrões Arquiteturais

Arquitetura de Software

Gladston Junio Aparecido

Sumário

Capítulo 1. Introdução à Arquitetura de Software

Arquitetura de Software é uma das principais disciplinas da Engenharia de Software. Os produtos resultantes de seus processos contribuem de forma substancial para a garantia dos atributos de qualidades de um software. Não é por menos que o papel de Arquiteto de Software tem sido cada vez mais almejado por profissionais envolvidos nas mais diversas fases do ciclo de vida de um software.

Mas o que representa a arquitetura de um software? A definição de arquitetura é muito subjetiva, uma vez que diversos fatores influenciam as entregas e atividades previstas. A norma ISO/IEC/IEEE 42010 define arquitetura como conceitos e propriedades fundamentais de um sistema considerando seu ambiente, seus elementos, seus relacionamentos e seus princípios de projeto e evolução. Existem alguns pontos chaves nessa definição que estão presentes na maioria das definições de arquitetura de software:

- **Arquitetura é a parte fundamental do software**: a arquitetura é o alicerce do sistema e as decisões tomadas na concepção e evolução da arquitetura tem alto impacto e são difíceis de mudar. “Arquitetura é sobre as coisas importantes. Seja o que for.” (GAMMA et al., 1994).

- **Um software está situado em um ambiente**: a arquitetura deve sempre considerar o ambiente ao qual o software está inserido. Isso inclui não apenas os computadores que compõem o *runtime* mas também as características da equipe, das metodologias utilizadas, as demais aplicações ao qual ele se integra etc.

- **Arquitetura engloba os elementos do software e seus relacionamentos**: ao longo deste módulo, estudaremos diversos princípios e práticas de Arquitetura de Software com focos distintos e independentemente do nível da análise, em sua maioria, tratamos de três questões que resumem o que representa uma arquitetura. Essas questões estão relacionadas a:

  – Como definir os elementos de um software;

– Quais as responsabilidades de cada um desses elementos;

– Como esses elementos se relacionam.

Os elementos de um software no qual um projeto de arquitetura de software deve considerar, dependem do nível da análise. Em determinados momentos um arquiteto deve analisar a arquitetura do ponto de vista de distribuição física e em outros momentos, o foco da análise deve ser definir papeis para artefatos de código fonte. A Figura 1 descreve os principais tipos de elementos de um software.

Figura 1 – Tipos de elementos de uma Arquitetura de Software.

	Example Elements	Example Relations
Module	class, package, layer, stored procedure, module, configuration file, database table	uses, allowed to use, depends on
Component and Connector	object, connection, thread, process, tier, filter	call, subscribe, pipe, publish, return
Allocation	server, sensor, laptop, load balancer, team, Owen (a person), Docker container	runs in or on, responsible for, develops, stores, pays for

Fonte: KEELING (2017).

Embora a arquitetura de um software seja abstrata, ela deve ser descrita por documentos organizados por visões focadas nos interesses dos diferentes *stakeholders* do projeto. Os temas abordados nesta seção e outros tópicos importantes sobre a disciplina Arquitetura de Software podem ser conferidos nos livros INGENO (2018) e MALVEAU (2000).

Arquiteto de Software

Como Arquiteto de Software, um profissional tem a responsabilidade de tomar decisões importantes capazes de maximizar inúmeros atributos de qualidade de um software. Dentre esses atributos, se destacam a manutenibilidade, interoperabilidade,

segurança e performance. O arquiteto deve sempre se preocupar em reduzir os possíveis *gaps* existente entre especificação e implementação.

Em algumas equipes a figura de arquiteto é permanente e desempenhada por um arquiteto dedicado ou um time de arquitetos. Em outras esse papel é exercido por diferentes membros da equipe. Fatores como complexidade do negócio e do sistema, tamanho das equipes, cultura organizacional são alguns dos fatores que levam as organizações a adotarem ou não o papel dedicado de arquiteto de software.

KEELING (2017) destaca que, enquanto desempenha o papel de arquiteto, o profissional tem uma posição única na equipe. Embora não seja gerente ou coordenador, o arquiteto decide quando e como o software pode ser entregue e determina se os requisitos estratégicos foram atendidos. Embora seja um papel técnico, enquanto arquiteto, o profissional atua mais na definição de projetos do que na definição de algoritmos. Por fim, um arquiteto não é profissional de infraestrutura, mas projeta topologias, monitora aplicações e garante a operabilidade do sistema.

Para entregar os resultados esperados para todas as responsabilidades descritas acima, o profissional deve ter uma visão holística do negócio e de tecnologia, com habilidades de negociação, liderança e principalmente, de comunicação. Um arquiteto deve ser capaz de discutir inúmeros detalhes técnicos com a equipe e posteriormente apresentá-los para a gerência sem perder a essência da mensagem.

O conteúdo e as referências bibliográficas deste módulo discutem vários aspectos técnicos e habilidades requeridas em um arquiteto. Além disso, HOHPE (2020) apresenta *insights* interessantes sobre o trabalho de um arquiteto.

Padrões na Arquitetura de Software

Conforme discutido nas seções anteriores, a arquitetura de um software envolve definições das partes elementares de um sistema e, portanto, espera-se que todas elas sejam fundamentadas em práticas de engenharia. Um arquiteto deve saber identificar características relevantes de problemas e identificar soluções efetivas.

Para auxiliar esse processo existem na literatura diversas propostas de padrões que podem ser adaptados e replicados no projeto de um software.

A prática de definir e aplicar padrões não é exclusiva da Engenharia de Software. As áreas de Arquitetura e Engenharia tem um longo histórico de uso de padrões e uma das definições mais citadas é do arquiteto Christopher Alexander:

> Um padrão é uma entidade que descreve um problema que ocorre repetidamente em um ambiente e então descreve a essência da solução para este problema, de tal forma que você use esta solução milhões de vezes, sem nunca a utilizar do mesmo modo.

Na Arquitetura de Software, padrões são utilizados como *templates* para solucionar problemas recorrentes que se manifestam ao longo do ciclo de vida de um software. Por se tratar de propostas amplamente testadas e bem documentadas, os padrões contribuem para garantir a qualidade do produto gerado. Os padrões também colaboram para a redução dos riscos dos projetos e para o estabelecimento de um vocabulário comum entre os profissionais, sejam desenvolvedores, *testers*, arquitetos etc.

Padrões na Arquitetura de Software têm sido estudados por décadas. Dado o grande número de propostas de padrões, eles são frequentemente organizados e divulgados em catálogos de padrões. Alguns dos principais catálogos de padrões serão apresentados no Capítulo 2. Cada catálogo possui um contexto específico e documenta os padrões de forma diferente. Entretanto, a documentação dos padrões frequentemente possui os seguintes elementos:

- **Nome**: descreve de forma suscinta o padrão;
- **Problema**: define o problema que o padrão se propõe a resolver e quando o padrão deve ser utilizado.
- **Solução**: descreve como o padrão propõe resolver o problema.

  - A solução deve ser sempre apresentada de forma genérica para possibilitar que o padrão seja aplicado em diferentes domínios de problemas e de forma independente de tecnologia.

  - É comum a solução descrever os elementos, os relacionamentos entre os elementos e suas responsabilidades.

- **Resultados e consequências**: descrevem os benefícios da adoção do padrão e suas consequências de forma a fundamentar a possível utilização ou não do padrão. Podem conter, por exemplo, impactos na flexibilidade, extensibilidade, portabilidade e manutenibilidade.

Antes de iniciarmos o estudo dos catálogos de padrões, é importante conhecer alguns princípios importantes do projeto de um software. Nas próximas seções serão apresentados os princípios SOLID e Injeção de Dependência.

Os Princípios de Projetos SOLID

Na Orientação a Objetos, existem diversos conceitos importante para a qualidade do código como, por exemplo, encapsulamento, herança, interface e polimorfismo. Entretanto, esses conceitos por si só muitas vezes não garantem que o código seja estruturado e escrito de forma correta. Por exemplo, não é difícil encontrar classes com métodos que não deveriam fazer parte da classe ou hierarquias de classes equivocadas geradas por uso indevido de herança.

SOLID são princípios que auxiliam na organização de funções e dados em classes e em como interligar essas classes. Embora não se limite apenas a código fonte e a linguagens orientadas a objetos, os princípios propostos foram originalmente concebidos com esse foco. Além disso, esses princípios são importantes, uma vez que:

> bons sistemas de software começam com código limpo. Por outro lado, se os blocos não forem bem construídos, a arquitetura da construção importará pouco. Além disso, você pode fazer uma bagunça substancial com blocos bem construídos. É aqui que entram os princípios SOLID. (ROBERT, 2017).

SOLID é um acrônimo de:

- **S**: Single Responsibility Principle
- **O**: Open/Closed Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

Os cinco princípios são brevemente apresentados nas subseções a seguir e uma discussão completa pode ser conferida em JOSHI (2016) e MARTIN (2017).

Single Responsibility Principle (SRP)

O princípio SRP é um dos princípios menos compreendidos do SOLID. Ele dita que uma classe deve ter uma e apenas uma **responsabilidade**. Mas é importante salientar que uma responsabilidade não significa fazer apenas uma ação.

Open/Closed Principle (OCP)

Segundo o OCP, uma classe deve ser aberta para extensão, mas fechada para modificação. Isso imprime ao software a capacidade de estender o comportamento de uma classe sem que seja necessário alterá-la. O veto à alteração da classe se dá devido ao fato de que, provavelmente, a alteração irá quebrar outras partes do sistema que estão funcionando.

Liskov Substitution Principle (LSP)

Descreve uma coleção de diretrizes para criação de hierarquias de classes onde um consumidor pode usar, seguramente, qualquer classe ou subclasse da hierarquia sem que o comportamento esperado seja comprometido.

Interface Segregation Principle (ISP)

De acordo com o ISP, os consumidores de uma classe não devem ser forçados a dependerem de métodos que eles não usam. Para isso, os comportamentos da classe devem ser segregados em interfaces e as dependências nos consumidores substituídas pelas respectivas interfaces.

Dependency Inversion Principle (DIP)

Acoplamento é um ponto chave na flexibilidade de um software e para maximizar esse atributo de qualidade o princípio DIP recomenda que as dependências do código fonte sejam sempre apontamentos para abstrações.

Na próxima seção, serão apresentadas práticas importantes que estão diretamente relacionadas com o SOLID.

Inversion of Control (IoC) e Injeção de Dependência (DI)

No fluxo de controle padrão da execução de um código na programação procedural e na orientação a objetos, se um bloco de código *A* usa bloco de código *B*, então *A* tem o controle sobre o contexto da execução, ou seja, *A* é responsável por definir todas as instâncias das dependências utilizadas por *B*.

O princípio IoC é um princípio abstrato e propõe justamente a inversão desse fluxo de controle. Ele é classificado como abstrato porque não define como realizar essa versão do fluxo de controle. Uma abordagem que poderia ser utilizada para implementar o IoC é, quando o bloco de código *A* for utilizar o bloco *B*, o bloco B ser o responsável por definir suas dependências.

Entretanto, a implementação largamente utilizada é a injeção de dependências ou DI na sigla em inglês. De acordo com o princípio da DI, a obtenção das dependências deixa de ser realizada de forma *hard-coded* nos construtores das classes ou nas chamadas dos métodos. Isso significa que quando o bloco *A* for consumir o bloco *B*, ele receberá um objeto de *B* “pronto para uso” e as dependências de B serão resolvidas dinamicamente por um componente de terceiros denominado *IoC Container*.

Existem diferentes implementações de IoC Containers, como por exemplo:

- Unity
- Ninject
- Castle Windsor

- Autofac
- Spring
- StructureMap

Delegar a resolução das dependências para um container à parte contribui para a redução da complexidade do código e o uso efetivo de técnicas de testes automatizados de código, além de simplificar o gerenciamento de mudanças futuras.

Capítulo 2. Padrões de Projetos

Conforme introduzido no capítulo anterior, os resultados dos diversos estudos conduzidos ao longo dos anos com o intuito de identificar padrões são frequentemente organizados em catálogos, com o objetivo de auxiliar a descoberta, o aprendizado e o uso dos padrões. Os catálogos centralizam padrões com algum uso em comum, seja por tipo de problema ao qual se propõem a resolver ou por características das aplicações ao qual podem ser aplicados. Existem diversos catálogos de padrões na literatura e alguns são enumerados na lista a seguir:

- **Pattern Oriented Software Architecture (POSA)**: focado em padrões para desenvolvimento de sistemas de missão-crítica, muito utilizados no desenvolvimento de sistemas operacionais, servidores web, *middlewares* e softwares de plataforma. Publicado no livro BUSCHMANN e SOMERLAD (1996).

- **Patterms of Enterprise Application Architecture (POEAA)**: apresenta diversos padrões para “aplicações corporativas”, com foco nas linguagens .Net e Java. Publicado no livro FOWLER (2002).
- **DDD Patterns**: discute padrões relevantes para equipes que utilizam a metodologia *Domain Driven Design*. Os detalhes de alguns padrões de DDD podem ser consultados no livro MILLET e TUNE (2014).

O livro de PAI e XAVIER (2017) enumera outros catálogos de padrões. As próximas seções apresentam discussões sobre outros dois catálogos de padrões: (1) AntiPatterns BROWN et al. (2008) e MALVEAU et al. (1998) e (2) Design Patterns GAMMA et al. (1994), FREEMAN (2004) e SHALLOWAY e TROTT (2014).

AntiPatterns

AntiPatterns é descrito em BROWN et al. (2008) e MALVEAU et al. (1998) como um catálogo de padrões que usa uma abordagem diferente. Ele apresenta soluções **ruins**, que não devem ser utilizadas, para problemas de projeto e implementação de software. Tais soluções podem comprometer a compreensão, a

evolução e a manutenção do código fonte. Conhecê-los é igualmente importante visto que as soluções ruins descritas como AntiPatterns podem ocorrer quando um profissional não conhece a melhor solução para um problema, não tem conhecimento ou experiência suficiente para resolver um determinado tipo de problema ou ainda quando se aplica um padrão corretamente, porém no contexto errado.

Um dos AntiPatterns mais conhecidos é o Spaghetti Code. Ele se manifesta em (1) códigos com estruturas fracas ou com falta de clareza, (2) códigos comprometidos onde nem o desenvolvedor original consegue entendê-lo, (3) códigos com violações de diversos outros AntiPatterns ou Code Smells ou ainda (4) códigos difíceis de manter, sem uso racional de reuso e sem possibilidades de extensão.

Abaixo segue a lista completa de AntiPatterns:

- The Blob
- Continuous Obsolescence
- Lava Flow
- Ambiguous Viewpoint
- Functional Decomposition
- Poltergeists
- Boat Anchor
- Golden Hammer
- Dead End
- Long parameter list
- Spaghetti Code
- Input Kludge
- Walking through a Minefield
- Cut−and−Paste Programming
- Mushroom Management
- Error-hiding

- Reinventing the wheel
- Programming by permutation

Design Patterns

Design Patterns, originalmente publicado por GAMMA et al. (1994), é o catálogo de padrões mais conhecido. Também é referenciado pelo nome Gang of Four (GoF) devido aos quatro autores do livro de sua publicação original. Esse catálogo descreve um conjunto de vinte e três soluções para problemas que podem ocorrer no projeto de um software orientado a objetos. Os vinte e três padrões foram distribuídos em três categorias: (1) Creational Patterns, (2) Strucutral Patterns e (3) Behavioral Patterns. A Figura 2 enumera os padrões do GoF.

Figura 2 – Design Patterns do Catálogo GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Fonte: Fonte: GAMMA et al. (1994).

GoF – Creational Patterns

Os Creational Patterns ou padrões de criação são padrões voltados para soluções de problemas no processo de criação de instâncias de objetos. Conforme destacam GAMMA et al. (1994), por meio dos padrões de criação é possível projetar sistemas que não dependem de como os objetos são criados, compostos ou representados. Em muitos sistemas, o maior nível de complexidade está presente na composição dos objetos e como eles obtêm suas dependências. As soluções propostas por esses padrões são particularmente relevantes para esses casos.

No Capítulo 1, foram discutidos os conceitos de IoC e Injeção de Dependências. Esses princípios propõem abordagens para eliminar a necessidade de se codificar a obtenção das dependências de um objeto de forma fixa nos próprios objetos. Os padrões de criações fomentam outras abordagens com propósitos semelhantes.

GAMMA et al. (1994) destacam dois temas frequentes abordados:

1. Os padrões de criação concentram as regras sobre quais classes concretas o sistema deve usar.

2. Para isso é vital que as referências sempre apontem para tipos abstratos.

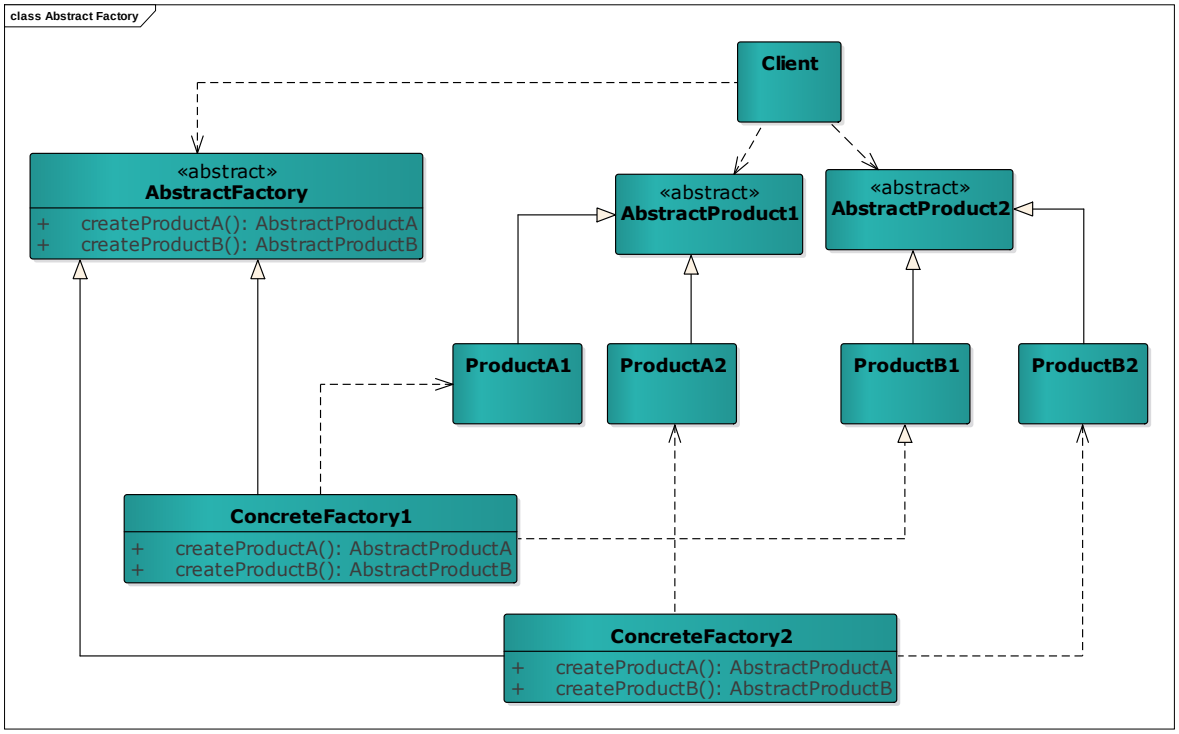
Uma das principais vantagens dos padrões de criação é a flexibilidade na definição das classes concretas que serão materializadas durante a execução do sistema. Tal flexibilidade torna o software capaz de determinar quais tipos devem ser materializados, como cria-los, quem pode cria-los e quando.

Nas próximas subseções estudamos alguns dos padrões de criação.

Abstract Factory

A documentação do Abstract Factory presente em GAMMA et al. (1994) o descreve como um padrão que permite disponibilizar uma interface para criação de famílias de objetos ou objetos dependentes sem que seja necessário especificar as classes concretas. A estrutura proposta para o padrão é ilustrada na Figura 3. O *client* representa um objeto que depende de dois tipos de “produtos”, porém existem diferentes implementações para cada um dos tipos de produtos. Para abstrair a obtenção dos diferentes tipos de produtos a classe *client* delega a criação para uma nova abstração denominada AbstractFactory e passa a depender de abstrações criadas para cada um dos tipos de produtos (AbstractProduct1 e AbstractProduct2). Como a *factory* é abstrata, se torna possível criar uma subclasse concreta para ela que instancia ProductA1 e ProductB1 (ConcreteFactory1) e outra *factory* que instancia ProductA2 e ProductB2 (ConcreteFactory2).

Figura 3 – Abstract Factory *Structure*



Fonte: GAMMA et al. (1994).

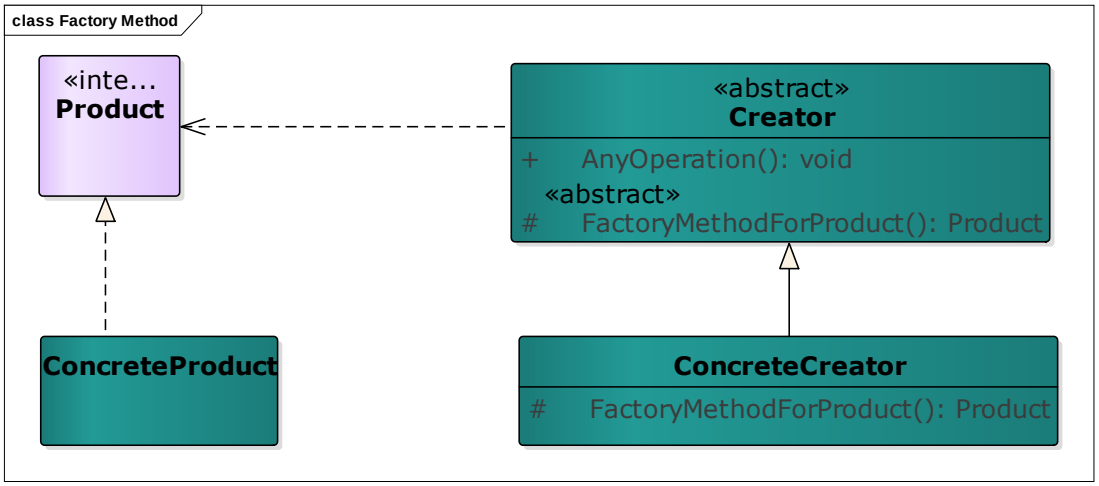
Factory Method

Conforme documentam GAMMA et al. (1994), o Factory Method tem por objetivo permitir que uma classe abstrata defina referências para outras classes ou interfaces, porém, delegando para as subclasses a definição de qual classe concreta deve realmente ser criada para essas referências. Seu uso é recomendado para cenários onde uma classe não é capaz de definir ou prefere não definir qual objeto deve ser criado (ex.: um *framework* que precisa criar uma conexão com banco de dados, mas qual tipo de banco de dados será utilizado é definido por outras classes criadas nas aplicações que utilizam o *framework*).

Em alguns casos pode ser interessante combinar o Factory Method com os padrões Abstract Factory, Template Methods e Prototype. A estrutura proposta para o padrão é ilustrada na

Figura 4.

Figura 4 – Factory Method *Structure*



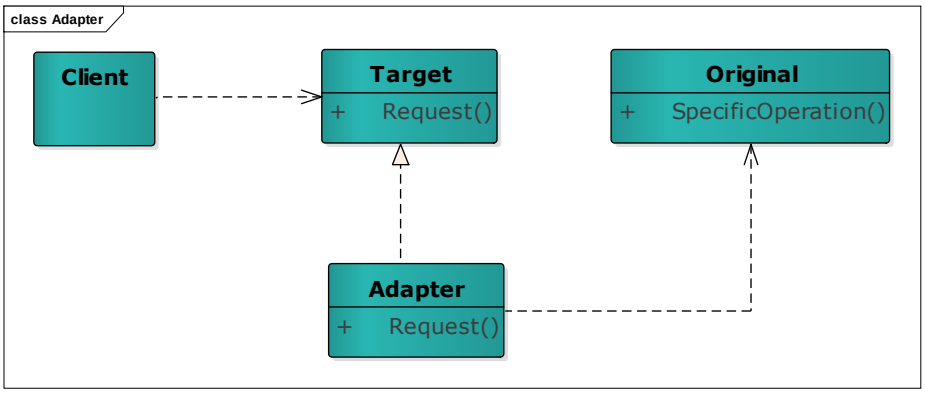
Fonte: GAMMA et al. (1994).

GoF – Structural Patterns

A definição adequada da estrutura das classes e dos objetos em sistemas grandes e complexos é uma atividade não trivial e suscetíveis a armadilhas que podem engessar o código do software, reduzindo sua capacidade de evolução. Os Structural Patterns ou Padrões Estruturais representam soluções para problemas relacionados à forma como classes e objetos são compostos.

De acordo com GAMMA et al., no escopo de classes, os padrões estruturais utilizam herança para compor interfaces ou implementações. Um exemplo de padrão estrutural com escopo de classes é o Adapter, cuja estrutura é ilustrada na Figura 5.

Figura 5 – Adapter *Structure*

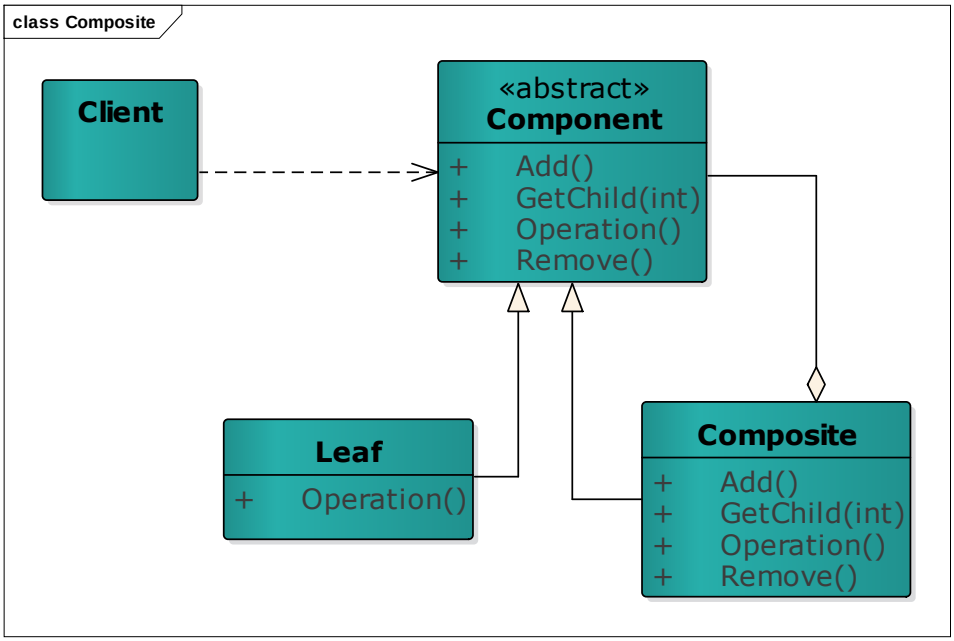


Fonte: GAMMA et al. (1994).

Com o padrão Adapter é possível fazer com que dois objetos “incompatíveis” interajam entre si. Por incompatíveis, entende-se que uma classe *Target* espera uma interface *X* e a classe *Original* fornece uma interface *Y*. A proposta consiste em criar uma classe intermediária *Adapter* que implementa a interface *X* e é responsável por traduzir as chamadas de e para a classe *Original*.

Já no escopo de objetos, conforme é ressaltado por GAMMA et al. (1994), os padrões dessa categoria visam descrevem abordagens para composição de objetos de tal forma que novos comportamentos possam ser adicionados, incluindo a possibilidade de adição de novas funcionalidades em *runtime*, contornando assim a limitação da composição estática das classes na orientação a objetos. Um exemplo de padrão estrutural de objetos é o Composite, descrito na Figura 6.

Figura 6 – Composite *Structure*



Fonte: GAMMA et al. (1994).

Com base no Composite, é possível definir árvores de objetos que formam estruturas do tipo todo-parte, porém com uma característica importante. Segundo o padrão Composite, os objetos *client* que desejam interagir com a árvore podem manipular qualquer nível dela de forma uniforme (padronizada), ou seja, não importa se uma determinada operação será realizada no objeto *root* ou em objeto do nível folha da árvore, para o *client* a forma de demandar a operação é a mesma.

GoF – Behavioral Patterns

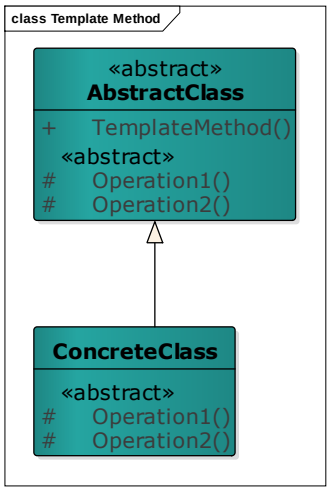
A Terceira categoria de *design patterns* é denominada Behavioral Patterns ou padrões comportamentais. GAMMA et al. (1994) definem essa categoria de padrões como padrões focados em algoritmos e na atribuição de responsabilidades durante a integração entre objetos. As soluções visam caracterizar e resolver questões relacionadas à dificuldade de acompanhar alguns fluxos de execução, seja pela complexidade ou por envolverem múltiplos objetos em sua interação.

Template Method

O Template Method é um padrão comportamental para classes. GAMMA et al. destaca que, como os demais padrões comportamentais voltados para classes, ele utiliza herança para distribuir comportamento entre classes. No caso específico

do Template Method, seu propósito consiste em estruturar o esqueleto de um algoritmo em uma classe pai, definindo cada um dos passos desse algoritmo. Entretanto, uma ou várias das etapas do algoritmo são delegadas para as classes filhas. A Figura 7 descreve a estrutura do padrão Template Method.

Figura 7 – Template Method *Structure*

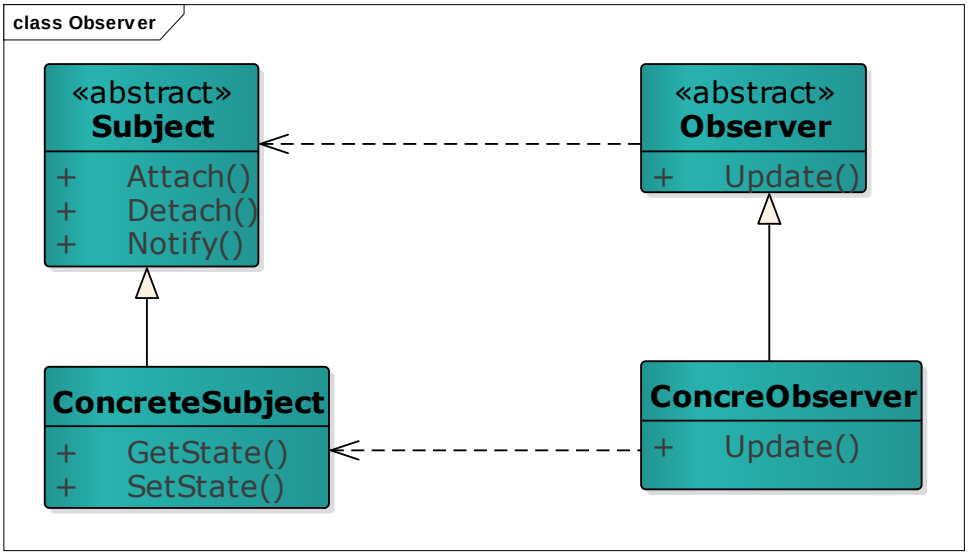


Fonte: GAMMA et al. (1994).

Observer

A interação entre elementos de um software pode ocorrer de diferentes maneiras, ou seguindo diferentes primitivas de comunicação, como por exemplo, comunicação síncrona, comunicação assíncrona, *call and forget*, *publish subscriber* etc. O padrão Observer permite a um objeto se relacionar com um ou vários objetos com base na primitiva de comunicação *publish subscriber*. Em sínteses, diferentes objetos (*observers*) podem manifestar o interesse de saber quando o estado de um objeto (*subject*) mudou. O objeto observado (*subject*) se torna responsável por notificar seus observadores sempre que seu estado mudou, porém, para ele não importa o que será feito com após as notificações.

Figura 8 – Observer *Structure*



Fonte: GAMMA et al. (1994).

Padrões de Acesso a Dados

Na arquitetura de um software existem requisitos transversais que impactam todo o código como, por exemplo, segurança e persistência de dados. As decisões em relação a esses requisitos têm impacto considerável e são fatores determinantes para diversos atributos de qualidade e sucesso dos projetos. Se tratando de persistência de dados, o arquiteto deve conhecer as principais abordagens para acesso às diferentes fontes de armazenamento de dados como bancos de dados relacionais, bancos NoSQL, arquivos, externos, *streams* etc. Dada a importância do acesso aos dados, era de se esperar que diversos estudos tenham sido dedicados a descobrir e documentar padrões de acesso a dados. No universo das aplicações corporativas e sistemas de informação, um dos principais estudos foi publicado no livro de FOWLER (2002) no qual está seção se baseia. A Tabela 1 enumera os principais padrões de acesso a dados documentados por FOWLER (2002).

Tabela 1 – Padrões de Acesso a Dados

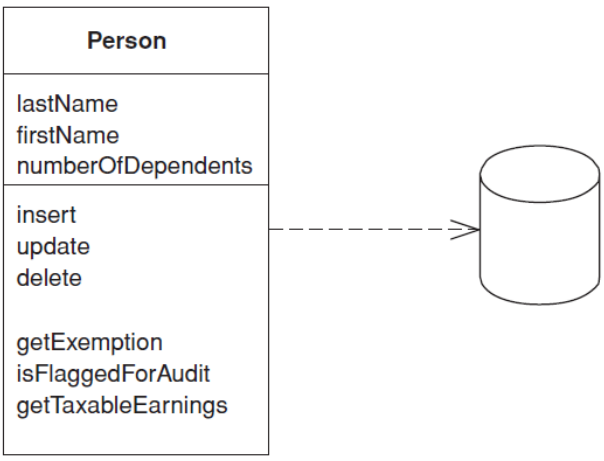
Table Data Gateway	Identity Field
Row Data Gateway	Foreign Key Mapping

Active Record	Association Table Mapping
Data Mapper	Dependent Mapping
Unit of Work	Embedded Value
Lazy Load	Serialized LOB
Query Object	Single Table Inheritance
Record Set	Class Table Inheritance
Repository	Concrete Table Inheritance
Identity Map	Inheritance Mappers

A seguir, vamos discutir brevemente alguns desses padrões.

- **Active Record**: padrão de acesso a dados recomendado para cenários onde as regras de persistências são simples. Consiste no encapsulamento, no mesmo objeto, de dados e comportamentos do negócio assim com as regras de persistência. Sua estrutura é ilustrada Figura 9.

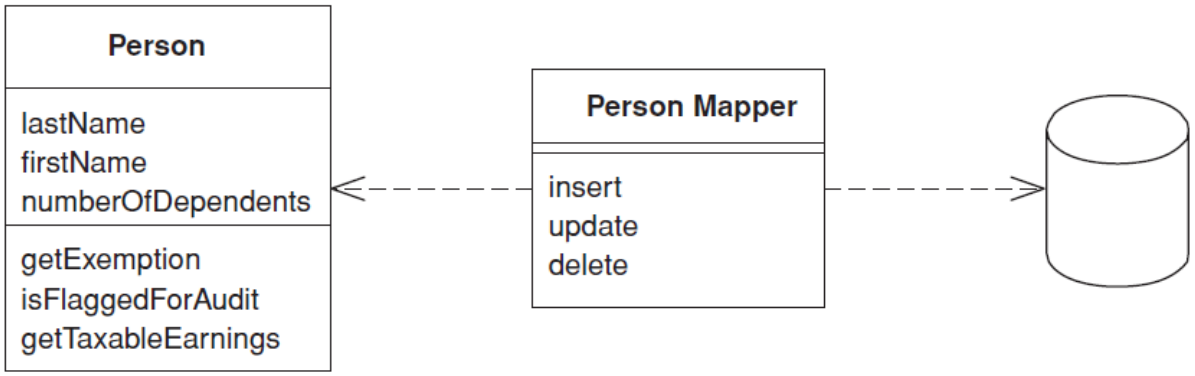
Figura 9 – Active Record *Structure*



Fonte: FOWLER (2002).

- **Data Mapper**: visa criar uma camada de tradução (baseada em mapeamento) capaz de traduzir os conceitos, princípios e comandos da Orientação a Objetos em conceitos e comandos da fonte de persistência. Sua estrutura é ilustrada na Figura 10.

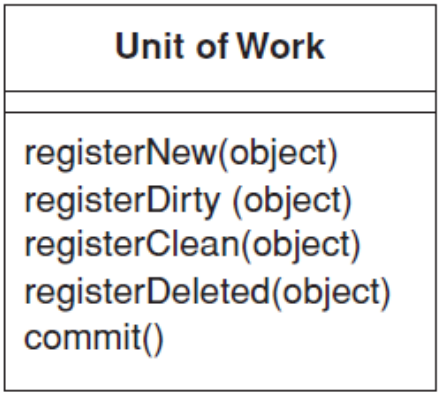
Figura 10 – Data Mapper *Structure*



Fonte: FOWLER (2002).

- **Unit of Work**: disponibiliza uma estrutura capaz de armazenar, em memória, todos os objetos afetados por uma transação (de negócio e não transação de banco de dados) e, posteriormente, coordena a sensibilização da fonte de persistência com as mudanças ocorridas nos objetos. Esse padrão é muito utilizado em aplicações web, por exemplo. Nesse cenário, se cria uma *unit of work* para cada requisição HTTP, de forma a centralizar nela as inclusões, alterações e exclusões necessárias para satisfazer o processo de negócio requisitado e ao final da execução da requisição, caso não ocorram erros, a *unit of work* transfere o resultado para a fonte de persistência. Sua estrutura é ilustrada na Figura 11.

Figura 11 – Unit of *Structure*



Fonte: FOWLER (2002).

- **Repository**: um repositório que funciona como uma coleção de objetos em memória que tem como objetivo atuar como uma camada intermediária de apoio. Nele se concentram as construções de *queries* e demais regras de acesso à fonte de persistência. Ao contrário do Data Mapper, nos repositórios os métodos específicos de acesso à fonte de persistência são públicos.

O padrão repositório é recomendado para aplicações com múltiplos domínios ou domínios com *queries* complexas. A Figura 12 ilustra um código de exemplo de montagem de *queries* no padrão Repository.

Figura 12 – Exemplo de construção de *queries* no Repository

```
public class Person {
   public List dependents() {
      Repository repository = Registry.personRepository();
      Criteria criteria = new Criteria();
      criteria.equal(Person.BENEFACTOR, this);
      return repository.matching(criteria);
   }
}
```

Fonte: FOWLER (2002).

Capítulo 3. Estilos Arquiteturais

Nas seções anteriores, foram discutidos diversos princípios e boas práticas de Arquitetura de Software e grande parte delas são voltadas para o *design time* ou implementação de algoritmos. Neste capítulo, serão apresentados outros princípios e práticas de arquitetura, porém o foco passa a ser questões que envolvem decisões de mais “alto nível” ou de maior impacto. Entretanto, isso não torna os tópicos discutidos anteriormente menos importantes. As comparações sobre detalhes de alto e de baixo nível em um projeto de uma casa apresentadas por ROBERT (2017) são um bom exemplo da importância de se dominar ambos os conceitos:

- Ao projetar uma casa, um arquiteto esboça diversos aspectos de alto nível como, por exemplo, o formato da casa, a sua localização no terreno, as curvas de níveis do terreno e o *layout* dos cômodos.
- Porém também são esboçados diversos aspectos de baixo nível como localização de tomadas e lâmpadas, orientação do sol, localização da caixa d’água etc.

Na Arquitetura de Software, muitos dos aspectos de alto nível estão relacionados com a distribuição do software e das responsabilidades de seus componentes no *runtime*, ou seja, no seu ambiente de execução e isso engloba plataformas de execução, regras de relacionamento entre os componentes e também as interfaces de integração com outras aplicações da organização ou de parceiros. E assim como nos detalhes de baixo nível, existem diversos padrões para as questões de alto nível, denominados estilos ou padrões arquiteturais. Os detalhes de alguns dos principais padrões arquiteturais são discutidos nas próximas seções.

Arquitetura Multicamadas

A abordagem de distribuição dos artefatos de um software em camadas é uma das mais utilizadas. A ideia central consiste em separar as responsabilidades dos componentes em camadas, na sua maioria de forma horizontal, compondo uma estrutura hierárquica. Além disso, o fluxo de comunicação ocorre de forma *top-down*.

Quando mencionamos o termo camadas podemos nos referir a dois conceitos distintos. O primeiro são as *layers*. Elas representam um agrupamento lógico referente à organização dos artefatos, o que está diretamente relacionado com *design time*. O outro conceito são as *tiers* que representam o *runtime*, ou a localização da execução de um componente.

Um exemplo clássico de arquitetura multicamadas é Arquitetura 3-Camadas onde os componentes do software são separados entre as camadas (1) apresentação, (2) negócio e (3) acesso a dados. Nas arquiteturas multicamadas uma cada pode ser:

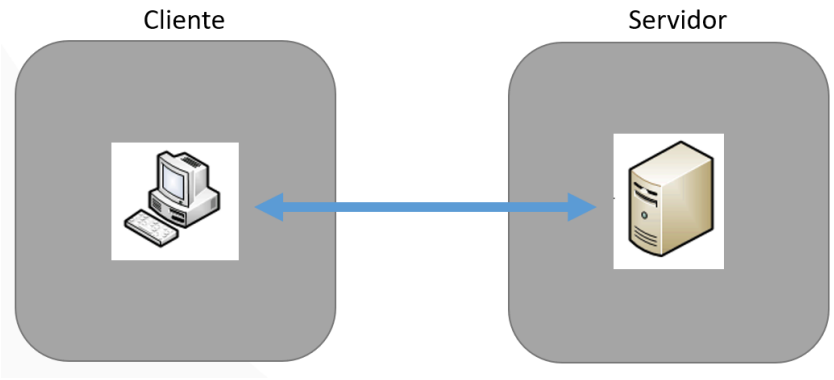
- **Fechada**: uma camada fechada deve sempre direcionar o fluxo de execução para a primeira camada logo abaixo.
- **Aberta**: uma camada aberta pode fazer o *by-pass* da camada imediatamente abaixo.

Arquitetura Cliente Servidor

A arquitetura Cliente Servidor se baseia nos princípios de arquiteturas multicamadas e têm foco na distribuição física do software (*tiers*). Conforme ilustra a Figura 13, na arquitetura Cliente Servidor o *runtime* do software é distribuído entre dois tipos de nós e ambos se integram por meio de um canal de comunicação. Os componentes presentes nos clientes podem ainda ser estruturados com base em outra separação de camadas (*layers*), assim como no servidor. Além disso, um cliente pode se comunicar com um ou vários servidores diferentes utilizando vários protocolos.

Um cliente pode requisitar dados e algoritmos para um servidor. Além disso, os clientes podem ser classificados como Thin Clients e Fat Clients. O critério para essa definição se baseia na capacidade de processamento e quantidade de responsabilidades atribuídas para os componentes de software que rodam no cliente.

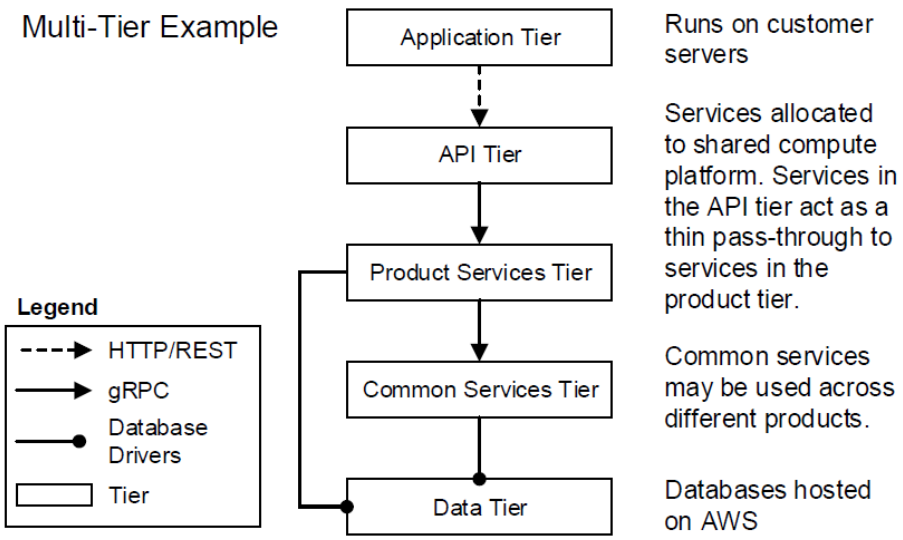
Figura 13 – Arquitetura Cliente Servidor



Arquitetura N-Tier

Nas seções anteriores, foi abordado o conceito de arquitetura multicamadas. Esse padrão pode aplicado na distribuição física do software e tal prática é denominada Arquitetura N-Tier. Seu objetivo principal é distribuir o software em três ou mais camadas, dependendo da complexidade e dos requisitos arquiteturais do software. Embora a implementação mais tradicional seja a separação em três camadas, nas aplicações modernas é comum a utilização de várias camadas como pode ser observado em um exemplo de arquitetura ilustrado na Figura 14.

Figura 14 – Arquitetura N-Tier

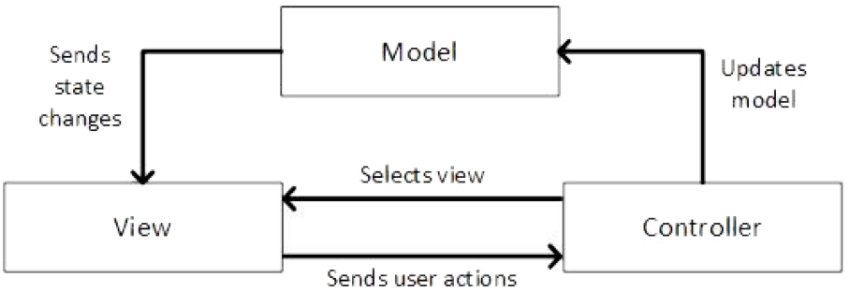


Fonte: KEELING (2017).

Model View Controller (MVC)

O padrão arquitetural MVC tem sido amplamente utilizado em aplicações modernas e ganhou destaque com a massificação da Web como plataforma de execução de aplicações. Sua estrutura auxilia, por exemplo, na separação de interesses nos projetos de interfaces de usuário (seja interface gráfica, de linha de comando ou *endpoints* para integração de sistemas). A Figura 15 apresenta a proposta arquitetural do MVC.

Figura 15 – Padrão Arquitetural MVC



Fonte: INEGNO (2018).

As responsabilidades de cada camada são descritas a seguir:

- **Model**: responsáveis pelo gerenciamento dos dados e do estado da aplicação. Além disso, acessa às fontes de persistência de dados. São independentes dos Controllers e Views.
- **Views**: camada de apresentação da aplicação, responsável coordenar a interação do usuário com o sistema, acionando o Controller sempre que necessário.
- **Controller**: atuam como pontes entre Views e Models, sendo responsáveis por atualizar os modelos e acionar a *view* adequada para apresentar o resultado de um processamento.

Dentre as vantagens do padrão MVC, se destacam:

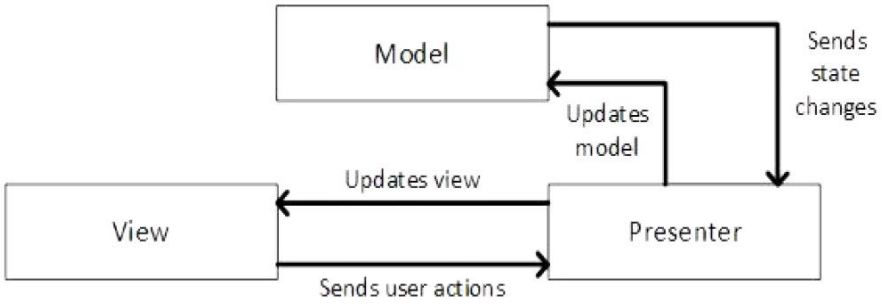
- Facilita os testes da aplicação.

- Contribui para que uma alteração em uma camada não afete as demais.
- Contribui para o reuso dos objetos.
- Auxilia na especificação das atividades de *frontend* e *backend*.

Model View Presenter (MVP)

O padrão MVP é uma variação do MVC que propõe a separação entre regras de UI e regras de negócio. Neste padrão cada interface de usuário possui uma View específica e os Controllers são substituídos pelos Presenters. As Views estão diretamente acopladas ao seu Presenter e não interagem com o Model.

Figura 16 – Padrão Arquitetural MVP



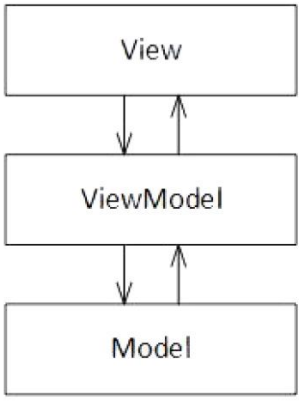
Fonte: INEGNO (2018).

Model View ViewModel (MVVM)

O padrão MVVM tem como principal diferença a separação da interface do usuário do restante da aplicação. Ele tem sido utilizado em aplicações para diferentes plataformas como Web, Desktop e Mobile e pode utilizar o conceito de *data binding* para interação entre as camadas. No MVVM o Model pode atuar de forma ativa, disparando eventos para sinalizar as mudanças em seu estado. Já as Views, além de capturar a entrada de dados pelo usuário e renderizar os resultados dos processos, passam a ter consciência da existência dos Models e dos ViewModels. Por fim, os ViewModels têm o papel de disponibilizar os dados para as Views, encapsular as regras de interação entre Views e Models e diferentemente dos outros padrões,

concentram as regras de navegação na aplicação. A Figura 17 apresenta a estrutura do padrão MVVM.

Figura 17 – Padrão Arquitetural MVVM



Fonte: INEGNO (2018).

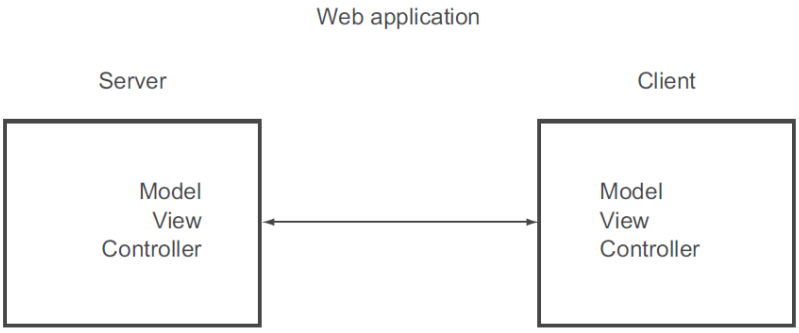
Single Page Applications (SPA)

Uma Single Page Application é uma aplicação web **completa** que contém uma página. Essa página funciona como uma casca (*shell* em inglês) para todas as funcionalidades, que em outras aplicações, representam outras páginas com URL específicas. Todo o processo de controle da interação do usuário e a navegação pelo conteúdo da aplicação é coordenado por Javascript, HTML e CSS.

As aplicações SPA fornecerem aos usuários uma experiência similar à encontrada em aplicações nativas (ex.: aplicações *desktops*), abstraindo por completo as caraterísticas inerentes dos protocolos de transmissão da Web e navegadores. Ou seja, uma vez carregada a *shell*, não existem mais recarregamentos completos da interface de usuário. Em seu desenvolvimento, são utilizados bibliotecas e *frameworks* que implementam vários dos padrões arquiteturais discutidos neste capítulo. Esses padrões podem ser utilizados, por exemplo, para simplificar o gerenciamento da complexidade da camada de apresentação que nas aplicações SPA, possuem uma concentração maior de código no *frontend* (navegador web).

Dessa forma, é interessante implementar o padrão MVC ou uma de suas variações no *frontend*, distribuindo as responsabilidades entre as unidades de código fonte Javascript, e posteriormente, implementar o padrão MVC no código fonte do servidor. Essa combinação repetida de implementações do padrão MVC, ilustrada na Figura 18, é denominada Fractal Model View Controller (FMVC). Também existem bibliotecas para SPA que disponibilizam outros padrões arquiteturais como MVVM.

Figura 18 – Padrão Arquitetural Factral MVC



Fonte: MIKOWSKI e POWELL (2013).

A arquitetura SPA é a arquitetura mais utilizada para *frontends* web e mobile modernos. Mais detalhes sobre seus conceitos e implementação podem ser consultados nos livros MIKOWSKI e POWELL (2013) e SHYAM e BRAD (2014).

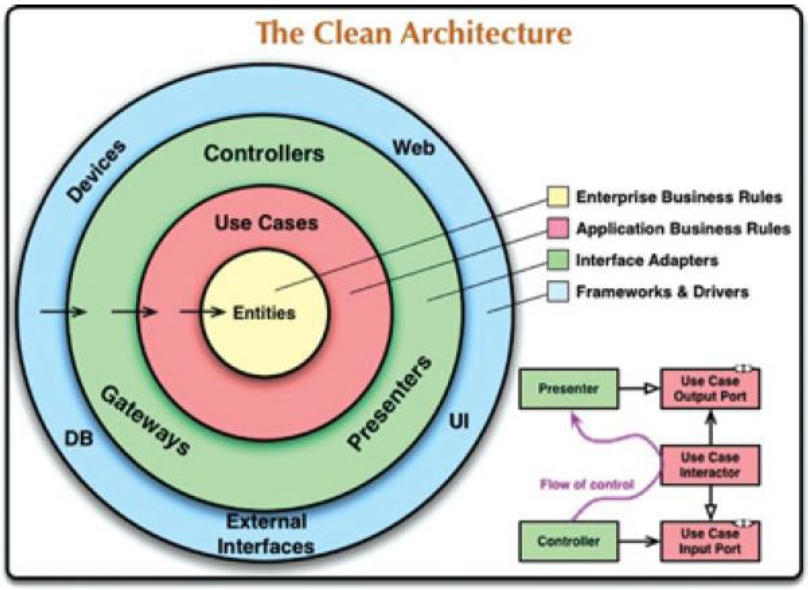
Clean Architecture

Os padrões arquiteturais discutidos até então neste capítulo se valem de uma organização do software em camadas horizontais, que compõem uma hierarquia. Entretanto, existem outras abordagens na literatura como a Hexagonal Architecture, Onion Architecture e Clean Architecture.

Na Clean Architecture as camadas do software formam uma estrutura concêntrica e a interação entre as camadas ocorre sempre em sentido ao núcleo. Isso implica que “as dependências do código-fonte devem apontar apenas para dentro, em direção a políticas de nível superior.” (MARTIN, 2017). A Clean Architecture propõe uma abordagem para organização de camadas com foco no cenário corporativo, permitindo a criação de uma arquitetura que possa ser utilizada por

múltiplos times e sistemas. A proposta original, ilustrada na Figura 19, e documentada em MARTIN (2017) propõe a organização da arquitetura nas camadas enumeradas a seguir:

Figura 19 – Clean Architecture



Fonte: MARTIN (2017).

- **Entities**: concentra as regras de negócio críticas ou estratégicas para a organização.
- **Use Cases**: encapsulam as regras de negócio específicas de um software da organização.
- **Interface Adapters**: responsável por adaptar os dados dos *use cases* para um formato mais adequado para os agentes externos (Web, Mobile, API etc.
- **Framework e Drivers**: códigos e demais artefatos específicos de plataformas, acesso a dados, segurança etc.

Capítulo 4. Arquiteturas para Sistemas Distribuídos (SD)

Após a adoção em massa das redes de computadores, houve uma mudança substancial na forma como os sistemas de software passaram a serem projetados e distribuídos em seus ambientes de execução. A possibilidade de interligar máquinas e sistemas por um meio de comunicação criou inúmeras possibilidades até então inviáveis, mas também trouxe vários desafios. Atualmente, a maioria significativa dos softwares relevantes não existiriam se fossem executados de forma isolada, sem a presença de componentes fisicamente distribuídos. Diante disso, um arquiteto precisa dominar diferentes princípios e tecnologias de desenvolvimento de sistemas distribuídos.

COULOURIS (2007) define sistemas distribuídos como uma coleção de computadores autônomos, interligados através de uma rede de computadores e equipados com software que permite o compartilhamento de recursos do sistema: hardware, software e dados. TANENBAUM (2007) os define como uma coleção de computadores independentes que se apresenta ao usuário como um sistema único e coerente. Para citar alguns exemplos temos a Internet, um DNS, um sistema de e-commerce, um Internet Banking etc.

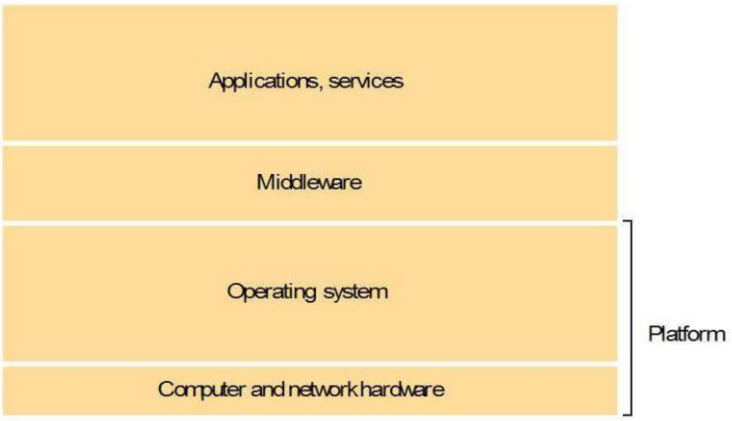
Nas próximas subseções, serão abordadas algumas das principais tecnologias para desenvolvimento de sistemas distribuídos, além dos conceitos de arquiteturas para sistemas distribuídos como Enterprise Application Integration (EAI) e Service Oriented Architecure (SOA).

Middlewares

Um *middleware* é uma camada de software que provê serviços para interação entre múltiplos processos em execução em um ou vários computadores. Por meio deles é possível invocar métodos remotos, enviar notificações de eventos, replicar dados etc. Além disso, um *middleware* fornece um modelo de programação homogêneo e simplificado, permitindo com que os desenvolvedores não tenham que se preocupar com detalhes da implementação necessários para a troca das mensagens entre os processos. Fazendo uma analogia, uma linguagem de

programação abstrai para os desenvolvedores questões relacionadas à hardware como registradores, instruções de máquina etc. Já os *middlewares* abstraem detalhes de rede como endereçamento de mensagens, protocolos de transmissão, formatação de mensagens, tratamento de falhas etc. A Figura 20 ilustra o fluxo de transmissão das mensagens da aplicação até o hardware do computador.

Figura 20 – Middleware



Existem diferentes tipos de *middlewares* cada um com características e aplicações distintas. Os principais são:

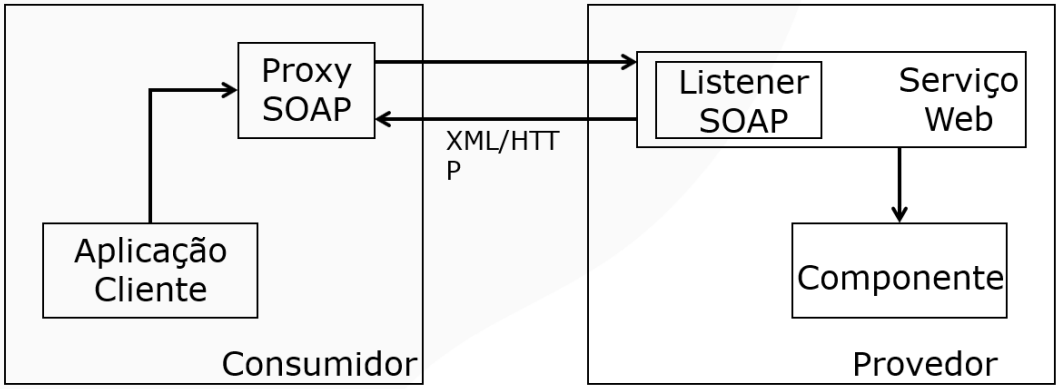
- Middleware procedural/Remote Procedure Call;
- Middleware orientado a transação/transacional;
- Middleware orientado a objetos;
- Middleware orientado a mensagem;
- Middleware baseados em componentes.

Web Services

Web Services é um *middleware* procedural para desenvolvimento de aplicações distribuídas que utiliza o protocolo SOAP para formatação de mensagens e o protocolo HTTP para transmissão das mensagens. Foi um dos principais *middlewares* para integração de aplicações na Internet, sendo adotado como tecnologia recomendada pelo W3C.

Por utilizar o protocolo SOAP, as mensagens (*requests* e *responses*) são formatadas em XML e encapsuladas em envelopes SOAP. Além disso, as interfaces dos serviços são descritas por meio da linguagem WSDL. A Figura 21 ilustra o fluxo de comunicação entre dois processos utilizando Web Service.

Figura 21 – Fluxo de comunicação com Web Service



- Aplicação cliente realiza a chamada.
- Proxy intercepta a chamada e constrói a mensagem XML de requisição.
- *Listener* SOAP recebe, analisa e valida a mensagem.
- *Listener* encaminha a requisição para o componente.
- *Listener* recebe o resultado, constrói e transmite o retorno.
- Proxy recebe e analisa a resposta e encaminhada ao cliente.

Enterprise Application Integration (EAI)

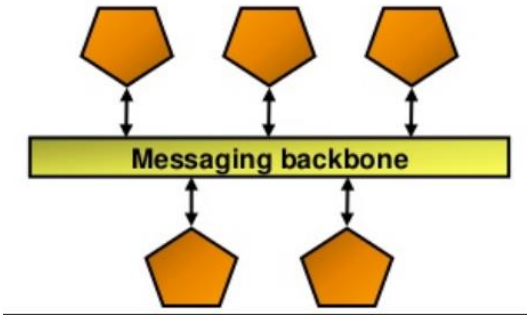
Nas seções anteriores, foram discutidas algumas tecnologias para desenvolvimento de aplicações distribuídas, porém a concepção de um sistema distribuído não deve ser realizada de forma *ad hoc* (desestruturada), ou seja, apenas implementando o consumo dos recursos remotos na medida em que se torna necessário. Existem propostas de arquiteturas para sistemas distribuídos e uma delas é a EAI. EAI é uma arquitetura abstrata para integração de corporativas que combina processos, softwares, padrões e hardware. Dentre seus objetivos, de destacam:

- Facilitar a integração, dados e processos de negócio entre aplicações e entre empresas.
- Permitir a integração sem mudanças significativas nas aplicações.

Uma arquitetura EAI emprega diferentes *middlewares* de forma a permitir que aplicações se integrem de diferentes maneiras, o que inclui transferência de arquivos, bancos de dados compartilhados, chamadas remotas de procedimentos e mensageria. Para isso uma arquitetura EAI possui dois princípios: (1) padronização das integrações e (2) centralização das comunicações.

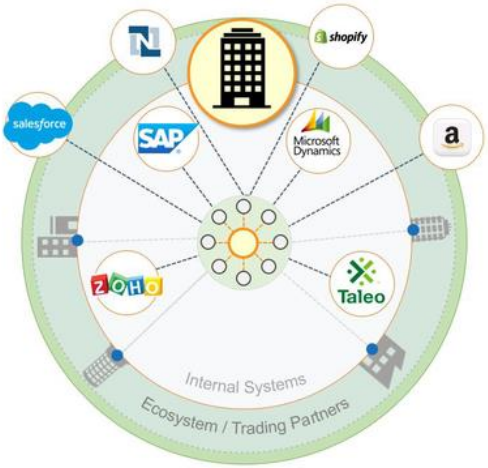
Conforme ilustram a Figura 22 e a Figura 23, em uma arquitetura EAI existe um elemento centralizador denominado *backbone*. Qualquer software que quiser se comunicar com outro deve utilizar o *backbone* para chegar até o software de destino. O *backbone* é responsável por rotear, mediar e transformar as mensagens e sua presença reduz o acoplamento entre os sistemas.

Figura 22 – Arquitetura EAI



Fonte: indigoo.com

Figura 23 – Arquitetura EAI



Fonte: cleo.com

Service Oriented Architecture (SOA)

SOA consiste em uma implementação da arquitetura EAI. Suas diretrizes descrevem regras para conceber, implantar e gerenciar tanto aplicações como infraestruturas de software. Além dos princípios de uma arquitetura EAI, uma arquitetura SOA determina que:

- Processos comuns do negócio devem ser encapsulados em serviços;

- Serviços devem ser disponibilizados em *endpoints*;
- As interfaces dos serviços devem ser baseadas em padrões abertos e interoperáveis.

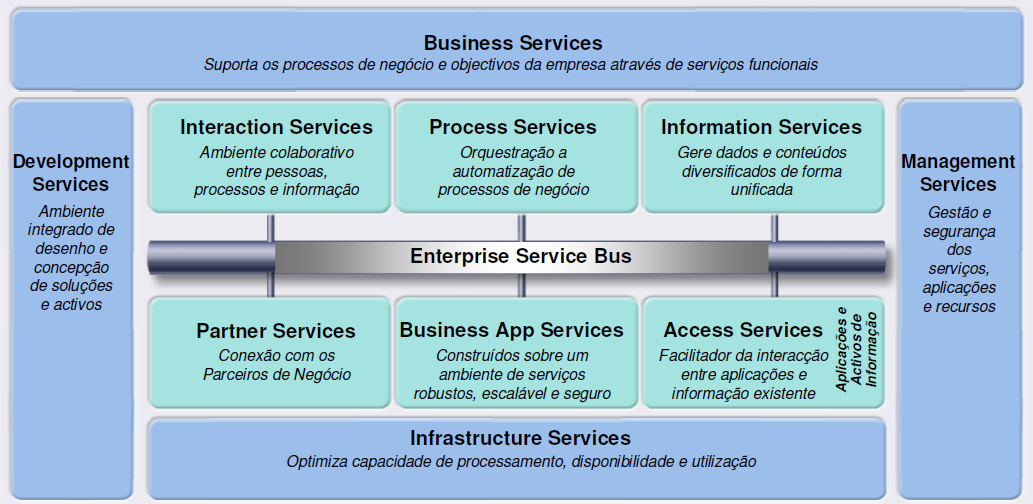
Os serviços que compõem uma arquitetura SOA são responsáveis por tarefas que se repetem no negócio ou dados corporativos que precisam ser compartilhados.

Ao longo dos anos 2000 diversas organizações iniciaram projetos de implantação de arquiteturas SOA com o intuito de dar agilidade ao negócio. O sucesso das empresas está cada vez mais relacionado à capacidade da TI de suportar a operação e se adaptar rapidamente às mudanças com agilidade e baixo custo. As empresas enxergavam na arquitetura SOA um meio fundamental para atingir esse objetivo.

Por ser uma arquitetura EAI, em SOA não temos comunicação ponto a ponto entre aplicações. As aplicações passam a ser compostas por serviços e esses, acessados por meio de um barramento central. Vale ressaltar que SOA tem como pilar fundamental a governança dos serviços. Isso torna importante as práticas de gerenciamento de ciclo de vida dos serviços (Service Lifecycle Management), composto por quatro fases: (1) modelagem, (2) montagem, (3) implantação e (4) gerenciamento.

Outro aspecto importante de uma arquitetura SOA é a classificação dos serviços em categorias, conforme ilustrado na Figura 24. Essa classificação é útil por servir de *guidance* para as organizações na implantação da arquitetura.

Figura 24 – Categoria de serviços em arquiteturas SOA



Enterprise Service Bus (ESB)

O *backbone* de uma arquitetura SOA é denominado Enterprise Service Bus (ESB). Esse componente de software atua como centralizador de todas as integrações entre serviços. Uma vez que todos os serviços devem ser catalogados, o ESB prove recursos para facilitar a descoberta de serviços, além de permitir a transformação de dados e o gerenciamento da governança da arquitetura. As principais funcionalidades de um ESB são:

- Fornecer transparência de localização para os serviços.
- Disponibilizar regras para conversão das mensagens nos mais diferentes formatos (HTML, XML, JSON, binário etc.).
- Suportar transmissão de mensagens nos mais diversos protocolos (SOAP 1.1, SOAP 1.2, WS-*, etc.).
- Disponibilizar fachas para dados e aplicações.
- Roteamento de mensagens.
- Mediação de mensagens.
- Atuar como gateway de mensagens, serviços, API e segurança.

Existem diferentes softwares de ESB no mercado como:

- IBM Web Sphere.
- Oracle Enterprise Service Bus.
- TBICO.
- Mule ESB.
- Microsoft BizTalk.
- Microsoft Azure ESB.
- Apache Camel.
- JBoss ESB.
- Petals ESB.

Referências

BROWN, William J. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. 1. ed. Wiley, 2008. 336 p.

BUSCHMANN, Frank; SOMERLAD, Peter. **Pattern-Oriented Software Architecture, a System of Patterns: Volume 1**. 1. ed. Wiley, 1996. 476 p.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas Distribuídos: conceitos e projeto**. 4. ed. Porto Alegre: Bookman, 2007. ISBN: 9788560031498.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. 1. ed. Addison-Wesley Professional, 2002. 560 p.

FREEMAN, Eric. **Head First - Design Patterns**. 1. ed. O'Reilly, 2004. 638 p.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Addison-Wesley Professional, 1994. 416 p.

HOHPE, Gregor. **The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise**. 1. ed. O′Reilly, 2020. 350 p.

INGENO, Joseph. **Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts**. 1. ed. Packt Publishing, 2018. 594 p.

JOSHI, Bipin. **Beginning SOLID Principles and Design Patterns for ASP.NET Developers**. 1. ed. Apress, 2016. 420 p.

KEELING, Michael. **Design It!: From Programmer to Software Architect**. 1. ed. O'Reilly, 2017. 354 p.

MALVEAU, Raphael C.; MOWBRAY Thomas J. **Software Architect Bootcamp**. 1. ed. Prentice Hall, 2000. 352 p.

MALVEAU, Raphael C. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. 1. ed. Wiley, 1998. 336 p.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design.** 1. ed. Pearson, 2017. 432 p.

MIKOWSKI, Michael and POWELL, Josh. **Single Page Web Applications: JavaScript end-to-end**. 1. ed. Greenwich:Manning Publications, 2013. 432 p.

MILLET, Scott; TUNE, Nick. **Patterns, Principles and Practices of Domain-Driven Design**. 1. ed. Wrox, 2014. 800 p.

PAI, Praseed; XAVIER, Shine. **.NET Design Patterns.** 1. ed. Packt Publishing, 2017. 314 p.

SHALLOWAY, Alan; TROTT, James R. **Design Patterns Explained: A New Perspective on Object Oriented Design.** 2. ed. Addison-Wesley Professional, 2014. 480 p.

SHYAM, Seshadri; BRAD, Green. **AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps Paperback**. 1. ed. Sebastopol:O'Reilly Media, 2014. 275 p.

TANENBAUM, Andrew S.; STEEN, Maarten Van. **Sistemas Distribuídos: princípios e paradigmas.** 2. ed. Pearson/Prentice-Hall, 2007. ISBN: 9788576051428.