

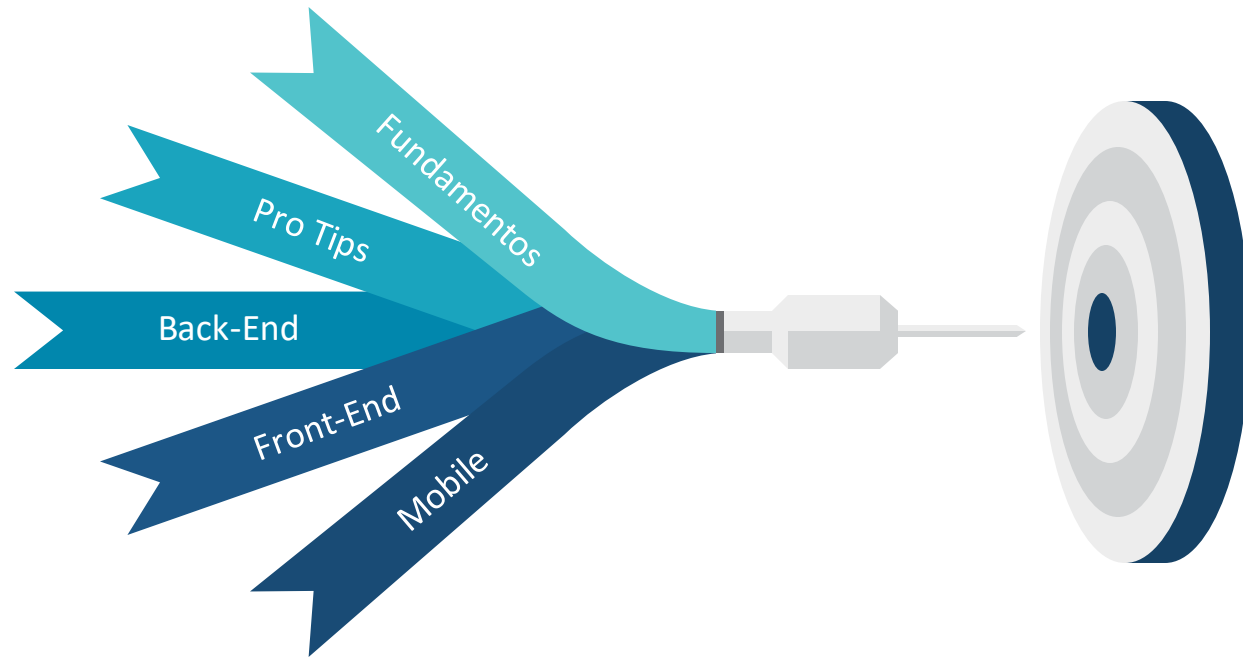
BOOTCAMP

FullStack



Apresentação

O **Bootcamp FullStack da Sysmap** busca preparar os profissionais para atuarem com aplicações distribuídas indo dos conceitos de arquitetura de software até o desenvolvimento de uma aplicação completa com Backend, Frontend Web e Mobile!



O foco desse **Bootcamp** é apresentar o novo mundo que todos os desenvolvedores terão de fazer parte. Aonde eles deixam de ser puramente criadores de código e tornam-se parte essencial do ciclo completo de entrega das soluções.



Carga Horária

2 Meses



Módulos

Conceitos, Backend, Frontend,
Mobile



Aulas

40 aulas online



Tecnologias

+7 ferramentas



Material

+50 materiais de apoio



Bônus

Flexibilidade nas
tecnologias utilizadas!





Instrutores



Renan Alves Medina

Arquiteto de Soluções




Danilo Rocha

COE Manager



Gabriel Mancini

Arquiteto de Soluções



Roteiro da Apresentação

1. Apresentação
2. Fundamentos de Engenharia de Software
 - a. Alan Turing x Alonzo Church;
3. Estrutura de Dados
4. Programação Orientada a Objetos
5. Programação Funcional
 - a. Higher Order Functions e Composição de Funções;
 - b. Imutabilidade, Pure Functions e Side-effects;
 - c. Listas/Mapas e Map, Filter & Reduce
6. Modelagem de Dados
7. Bancos Relacionais
 - a. ACID
 - b. 5 Formas Normais
8. NoSQL
 - a. CAP Theorem
 - b. BASE
9. DDD
10. Mensageria e Comunicação Síncrona vs Assíncrona
11. Brokers de Mensageria
12. Quebrando Aplicações Monolíticas
13. Concorrência e Complexidade ciclomática
14. Principais Patterns e Modelos Arquiteturais
15. Professional Tips
 1. Versionamento – Git/GitHub
 2. CI/CD – Botando em Produção
 3. DRY - Don't Repeat Yourself
 4. KISS - Keep It Simple, Stupid
 5. YAGNI - You Ain't Gonna Need It
16. Material de Apoio

Fundamentos de Engenharia de Software



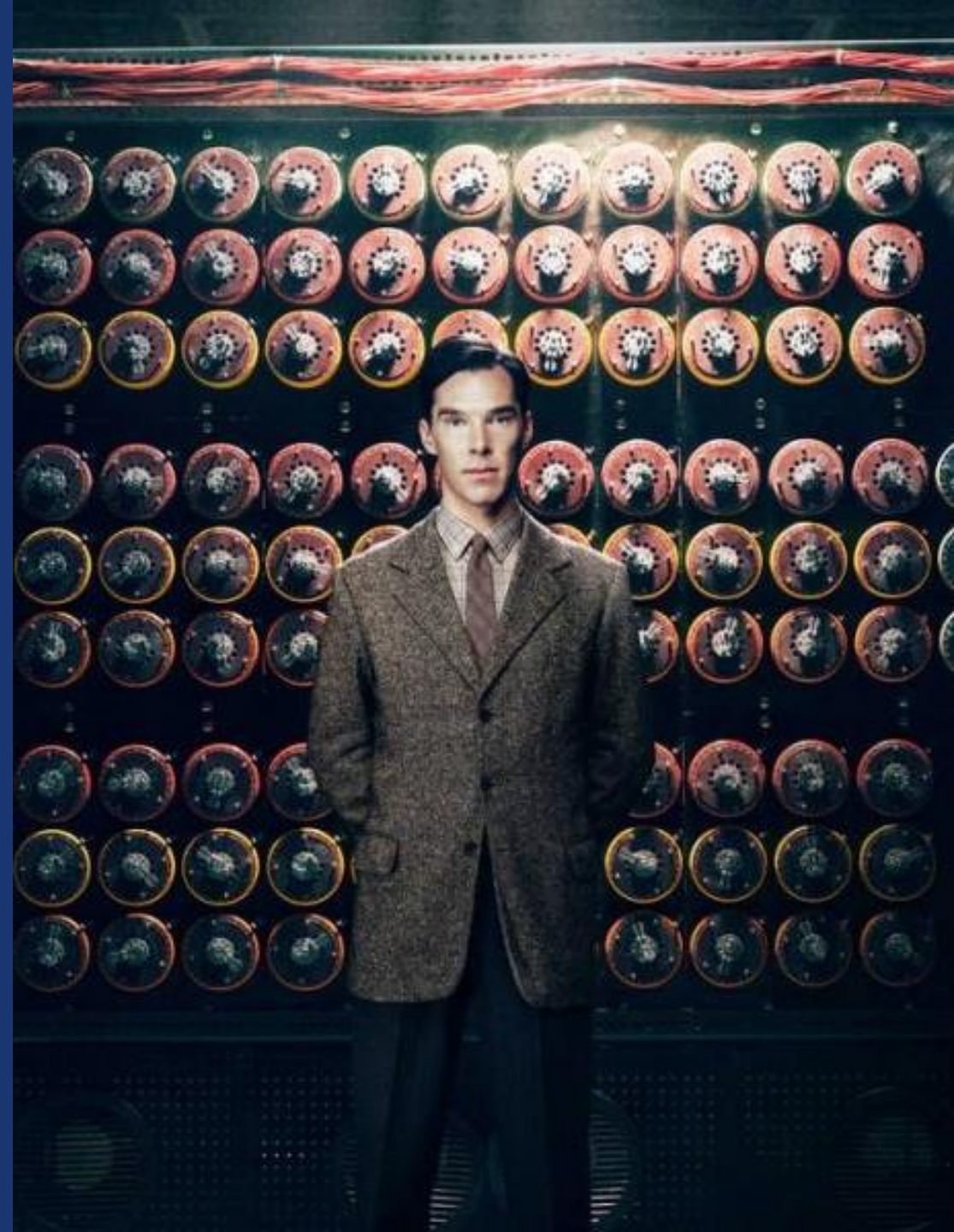
Alan Turing - Imperative vs Alonzo Church - Lambda Calculus

“

O pai da computação foi **Alan Turing**, foi ele quem desenvolveu a **maquina de Turing**, capaz de ler, interpretar e escrever dados em um dispositivo de armazenamento.

Posteriormente ele implementou essa ideia para construir uma maquina real na qual foi possível programar um *brute-force* para hackear o **enigma**.

Porém outro personagem tão importante como Alan foi seu professor **Alonzo Church**, matemático que fundamentou a teoria da ciência da computação.





**LAMBDA CALCULUS,
CHURCH ENCODING**

Alan Turing - Imperative VS Alonzo Church - Lambda Calculus

Ambos estavam resolvendo os mesmos problemas, porém cada um a sua maneira, enquanto Alan lia seus resultados e armazenava as respostas em seus registradores alterando assim o estado original, Alonzo tentava reaproveitar as regras/propriedades matemáticas afim de armazenar seus resultados dentro de suas proprias resolução sem alterar o estado original.

Nasce com isso os conceitos de **linguagem imperativa** (Alan) e o **lambda calculus** (Alonzo) que futuramente levaria a linguagens funcionais.

Para saber mais pesquise sobre "Church-Turing thesis"...



Estrutura de Dados

A **Estrutura de Dados** estuda os mais variados **tipos de dados**, quando iniciamos o estudo de uma nova linguagem de programação, entre os primeiros capítulos sempre encontraremos um reservado ao assunto, normalmente apresentando os tipos primitivos de dados.

Ex: *int, boolean, *string, etc...*

Algumas linguagens possuem tipos customizados, onde o programador consegue agrupar alguns tipos primitivos, assim organizando os dados de tal forma que reflitam as regras de negocio.

Ex: *em c/erlang struct, java/ts class, js prototype etc...*

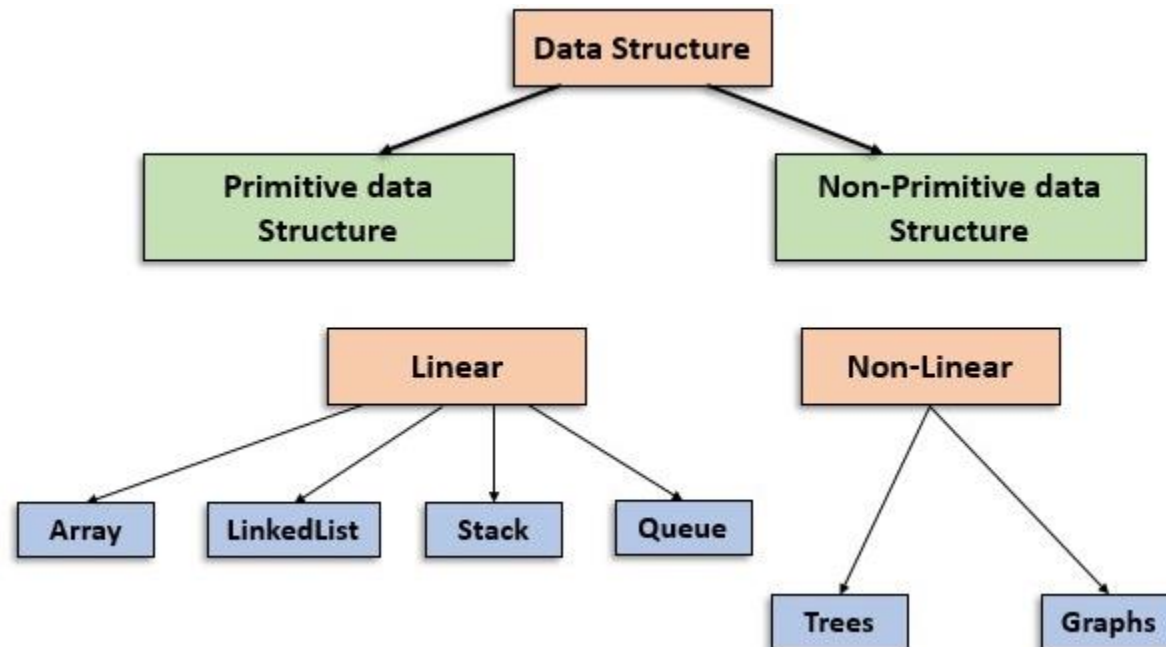
Uma outra opção é utilizar o pattern de Value-Objects.



Estrutura de Dados

As **Listas**, **Arrays**, **HashTables**... são outros tipos de estrutura de dados, porém com algumas propriedades especiais, como tamanho e uma ordem de itens indexados, usualmente são listas ligadas.

Por causa destas propriedades podemos iterar essas listas de formas diferentes e transforma-las outras estruturas como pilhas ou grafos e ainda com a capacidade de mapear para outros valores ou reduzir a ultimo valor.



Programação Orientada a Objetos

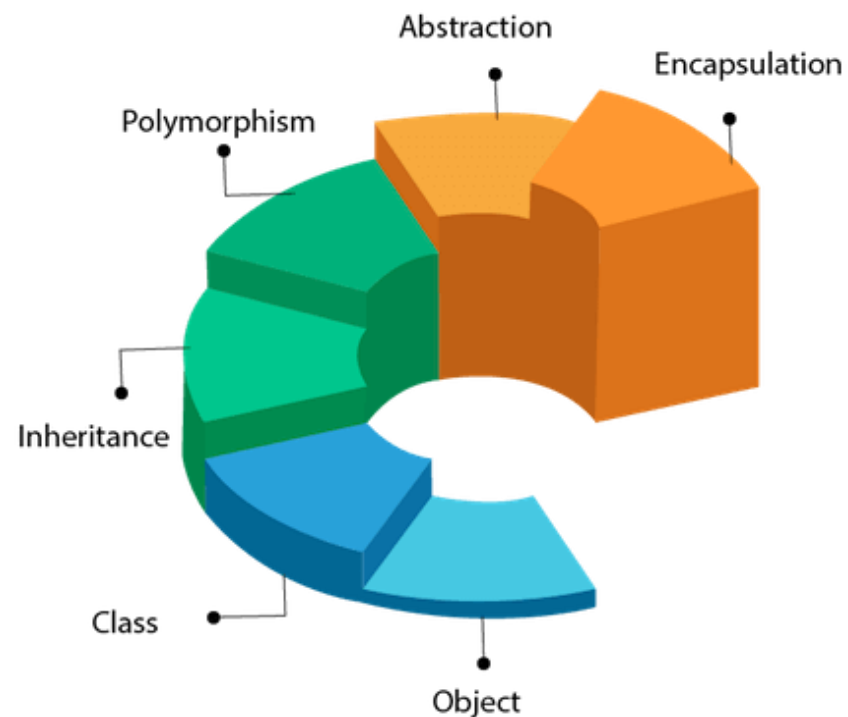
“

Tem como objetivo modelar os dados em atributos de classes e instanciar seus respectivos objetos, alocando um estado em memória para cada um e com a capacidade de alterar este estado quando necessário.

Esse tipo de representação de dados em objetos/classes, procura aproximar o sistema que está sendo criado ao que é observado no mundo real.

O principal objetivo da OOP é facilitar a reutilização de código e a segurança.

OOPs (Object-Oriented Programming System)

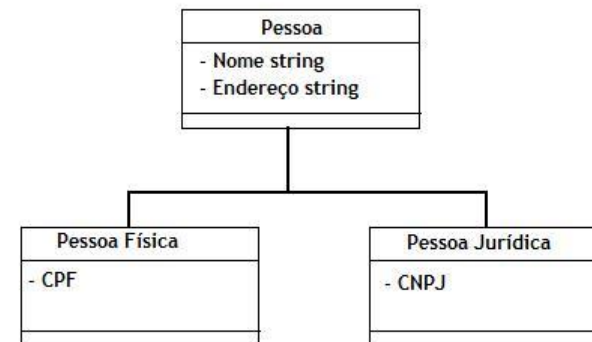




Programação Orientada a Objetos

Herança

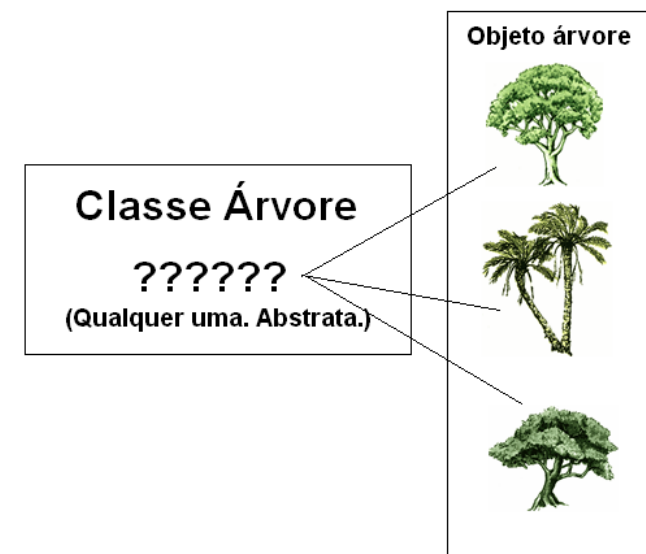
Permite criar novas classes a partir de classes já existentes, sendo estas filhas da classe base. com isso a reutilização de código e o compartilhamento de características que pode ser especificadas nas classes que as herdam.



Abstração

São aquelas classes que não criam instâncias e possuem métodos sem implementação. São como um molde para serem manipulados por subclasses.

Estas subclasses sobrecarregam os métodos abstratos.



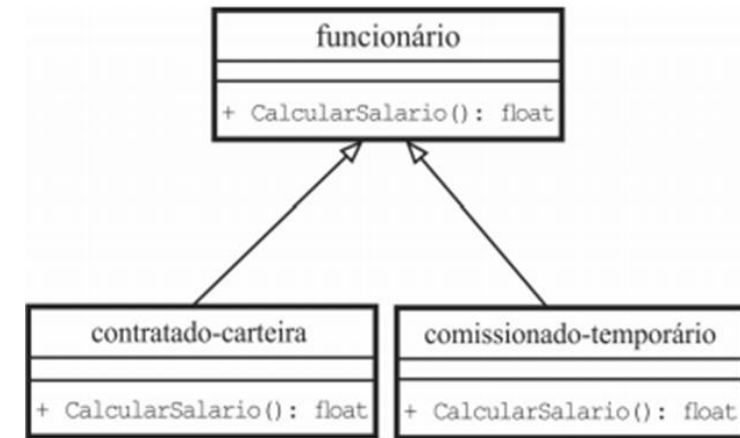


Programação Orientada a Objetos

Polimorfismo

Situação na qual um objeto pode adquirir maneiras diferentes dependendo de como foi criado. As superclasses (classes base) tem seus métodos sobrecarregados por subclasses (classes filhas) podendo elas implementá-los da forma que for conveniente

Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

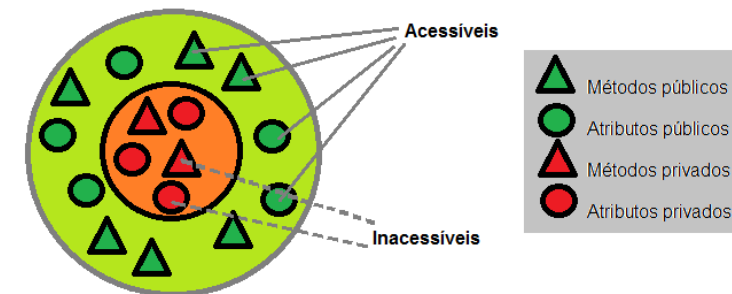


Encapsulamento

É a habilidade de um objeto em esconder dados da visão exterior, permitindo acesso somente por meio de métodos públicos.

Ex: (getters, setters).

Consiste em usar modificadores de acesso privados e fornecer métodos que possam acessá-los de forma segura. Ajuda a prevenir problemas de acesso indevido a dados.

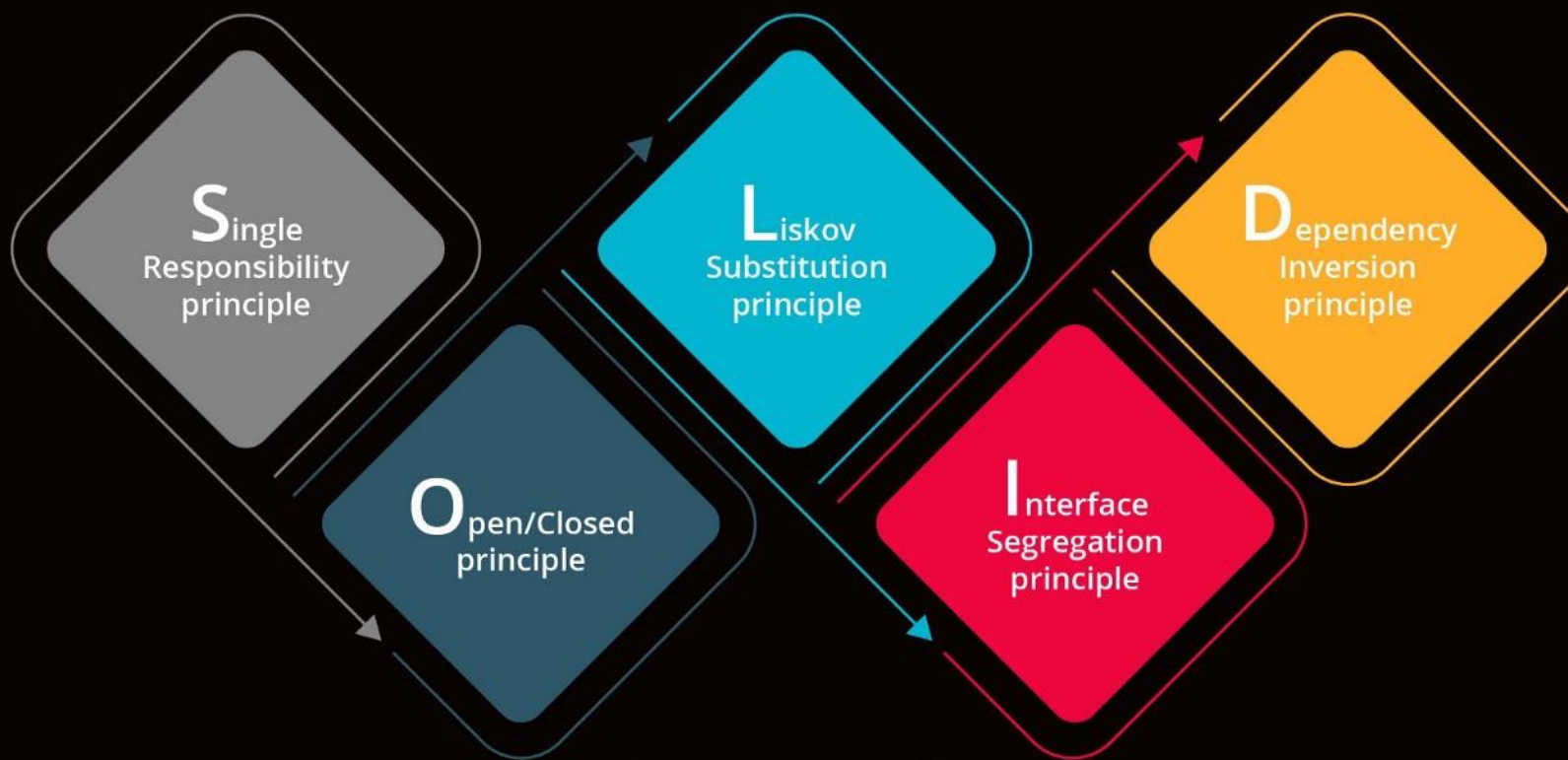




Programação Orientada a Objetos

S.O.L.I.D

PRINCIPLES OF OBJECT-ORIENTED DESIGN





Programação Funcional

Tem como objetivo, estabelecer um estado inicial em uma estrutura de dados e de forma declarativa, recriar o estado de entrada em uma nova versão obedecendo as regras de negocio e fazendo uso de algumas propriedades matemáticas, não alterando o estado original.

$$f(t) = a_0 + \sum_{n=1}^{\infty} \left[a_n \cdot \cos\left(\frac{n\pi t}{L}\right) + b_n \cdot \sin\left(\frac{n\pi t}{L}\right) \right]$$
$$= a_0 + a_1 \cdot \cos\left(\frac{\pi t}{L}\right) + b_1 \cdot \sin\left(\frac{\pi t}{L}\right) + a_2 \cdot \cos\left(\frac{2\pi t}{L}\right) + b_2 \cdot \sin\left(\frac{2\pi t}{L}\right)$$
$$a_0 = \frac{1}{2L} \int_{-L}^L f(t) dt = \frac{1}{2} \int_{-1}^1 f(t) dt$$
$$= \frac{1}{2} \int_{-1}^0 f(t) dt + \frac{1}{2} \int_0^1 f(t) dt$$
$$= \frac{1}{2} \int_{-1}^0 -1 dt + \frac{1}{2} \int_0^1 1 dt$$
$$= \frac{1}{2} \left[-t \right]_{-1}^0 + \frac{1}{2} \left[t \right]_0^1$$
$$= -\frac{1}{2} + \frac{1}{2} = \emptyset$$

$L=2 \Rightarrow L=1$
 $a_n \approx \frac{1}{n^2}$
 $b_n = \emptyset$

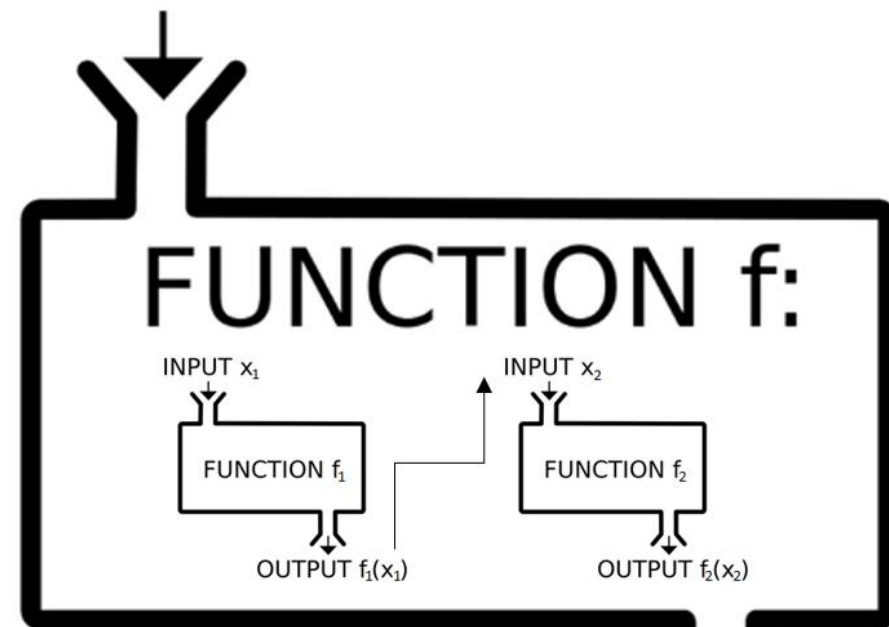
Higher Order Functions e Composição de Funções (curry)

“

Tem a capacidade de uma função receber como parâmetro uma ou mais funções para ser executada posteriormente, também chamada de callback, normalmente retornando uma função para quem o chamou, esta habilidade também pode ser composta em uma sequência de execução ou controle de fluxo, ou ainda ser aplicada parcialmente (curry).

Desta maneira podendo então tratar o estado passado e parte da regra na chamada da função.

INPUT x



OUTPUT $f(x)$



Imutabilidade, Pure Functions e Side-effects

O Conceito de imutabilidade, em ingles immutability, em grego... A imutabilidade evita a “reatribuição” de uma variável, e com isso torna as expressões matematicamente corretas, em um típico loop imperativo nos reatribuímos o valor de i , sendo $i = i + 1$ ou simplesmente $i++$, observando esta expressão podemos inferir que esta expressão não é matematicamente valida e por isso perde-se as propriedades que Church utilizava na resolução de seus problemas no lambda calculus, a versão imutável seria $j = i + 1$.



```
/**
 * To consider a function as pure, it must meet
 * two criteria:
 * • Given the same input, it should always return
 *   the same output
 * • The function should not cause any side effects
 *   outside of its scope
 */

// This function is pure
// Given input is (2, 2), the output will always be 4
const sum = (a, b) => a + b;

// This function is impure
// It will give different results even if the inputs
// are the same
const rand = (a, b) => Math.random() * (a - b);
```

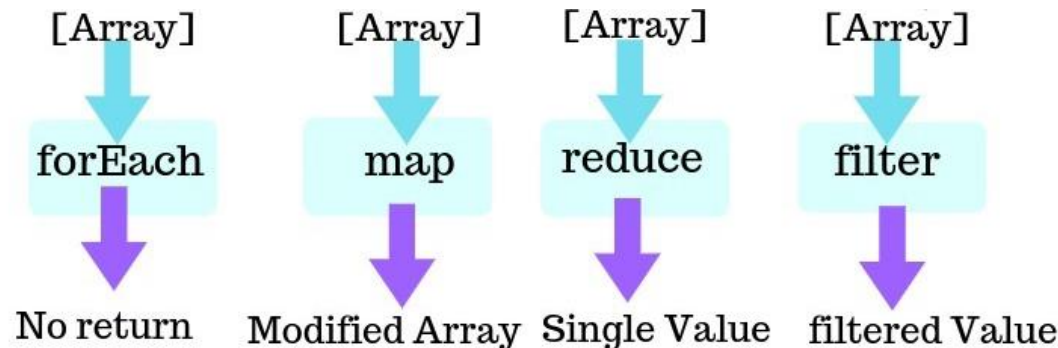
Imutabilidade, Pure Functions e Side-effects

Funções Puras são funções que não possuem efeitos-colateral, ou seja, são idempotentes, todas as vezes que forem chamadas com os mesmos parâmetros irão retornar sempre os mesmos resultados, logo, variáveis globais ou usos de I/O não podem estar presentes nestas funções.



Listas/Mapas e Map, Filter & Reduce

São Funções iterativas que trabalham com listas ou mapas e que retornam um novo estado. O uso concomitante de imutabilidade e funções puras e funções de iteração recursivas como **map**, **filter** e **reduce**, apresenta uma propriedade muito desejada por desenvolvedores que é a transparência referencial e por consequência lazy load, memoization, idempotência referencial, common subexpression elimination e paralelismo são desbloqueados para serem utilizados a fim de compor as regras de negocio de um novo estado.

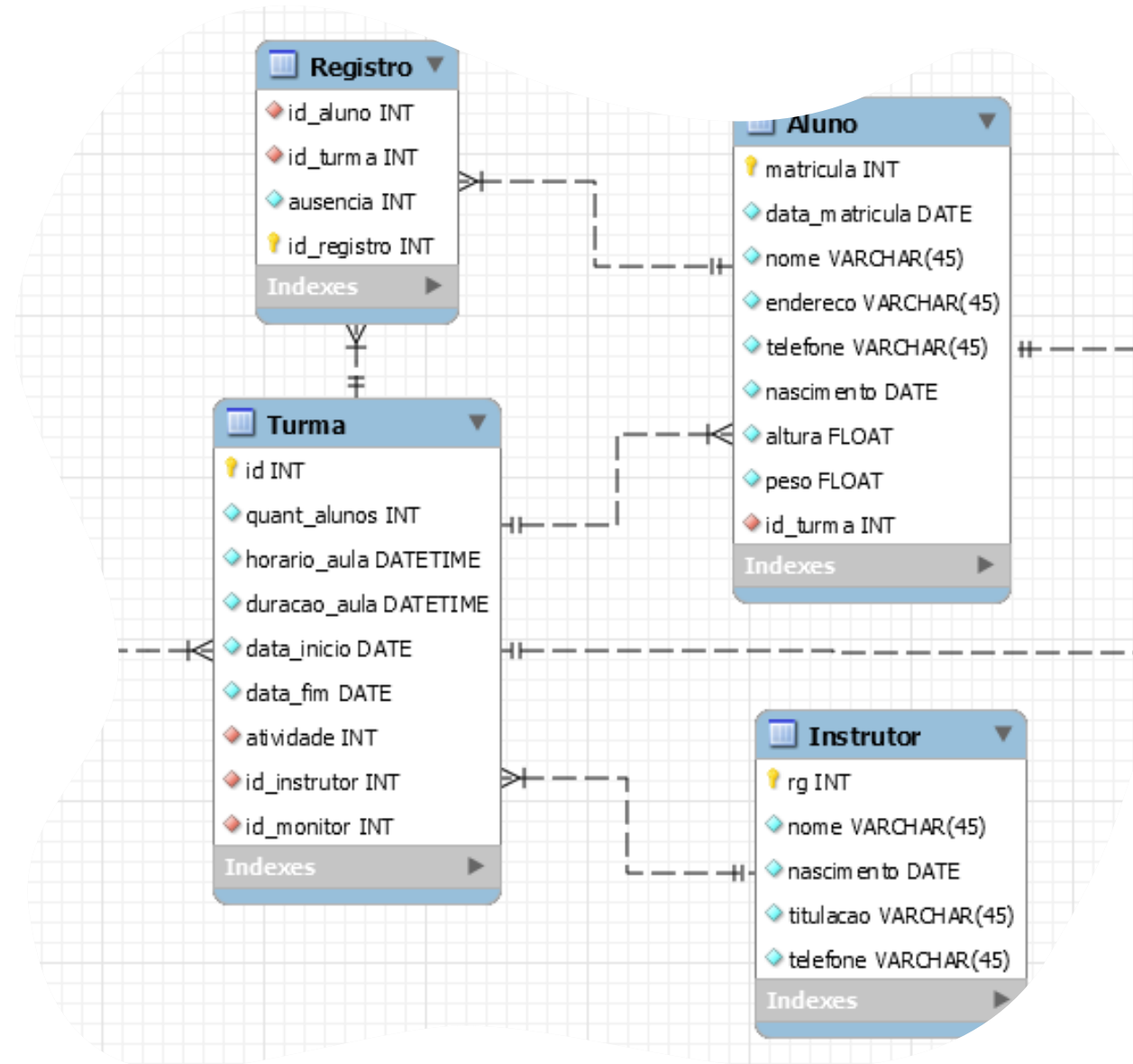




Modelagem de dados

O Tema de modelagem de dados tem por objetivo, descrever de maneira formal a relação dos dados em meios físicos, envolve alguns meios mais tradicionais e outros menos ortodoxos. Vamos nos aproximar deste assunto por 2 vertentes, a parte de Bancos de dados relacionais e via DDD com modelagem em código.

Apesar de parecer temas concorrentes não são, muitos frameworks trabalham em conjunto com o banco de dados afim de manter coeso os princípios de modelagem.





Bancos Relacionais

ORACLE®



O Banco de dados relacional é uma tecnologia que esta disponível desde a década de 70, tem como objetivo armazenar fisicamente os dados em estruturas de bancos, tabelas, registros e campos.

Ainda dentro de seu objetivo ser capaz de relacionar estas entidades de forma declarativa com chaves primarias (PK) e estrangeiras (FK).

E por ultimo, mas não menos importante, a capacidade de efetuar escritas e buscas em dados relacionados a partir de uma linguagem padrão (SQL) baseada em Álgebra Relacional.



ACID

Outra característica relevante dos Bancos de Relacionais eh o ACID Compliance:

Atomicidade

Em uma transação envolvendo duas ou mais partes de informações discretas, ou a transação será executada totalmente ou não será executada, garantindo assim que as transações sejam atômicas.

Consistência

Obedece a Integridade Referencial, isto e, a relação entre duas tabelas não pode ser desconsiderada ao realizar qualquer escrita no banco.

Isolamento

A concorrência entre transações não poderá interferir em outras transações em andamento, mas ainda, transações não validadas devem permanecer isoladas de qualquer outra operação inclusive leituras.

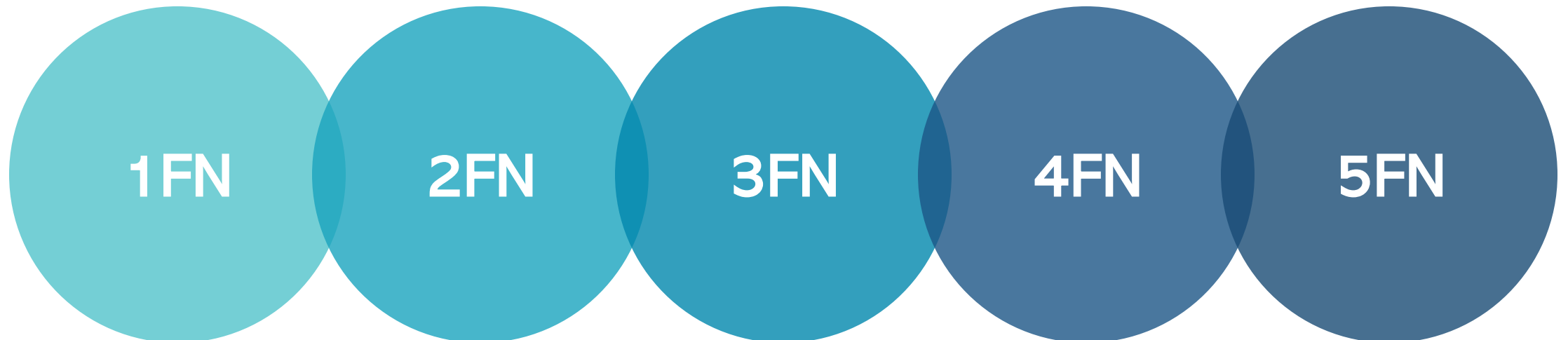
Durabilidade

Manter os dados seguros e escritos nos meios mais oportuno para o tal, sendo usualmente volumes de disco ou ssd's.



5 Formas Normais

Afim de modelar os dados em bancos, tabelas, colunas e linhas de uma maneira a evitar redundância dos dados, foi cunhado uma técnica de 5 passos para identificar problemas de modelagem.





5 Formas Normais

1FN - Cada campo deve conter apenas uma informação.

Ex.:

[PK]id	Name	Telephone
1	John Doe	451111-2222, 5522221111
2	Grace Hoper	453333-2222
3	Ada Lovelace	NULL
4	Andy Tanenbaum	451111-3333,553333-4444

Não compliance: (telephone)



5 Formas Normais

2FN - Cada campo não chave de uma tabela devera fazer referencia à chave da mesma como um todo.

Ex.:

<i>[PK] product_id</i>	<i>[PK] store_id</i>	<i>price</i>	<i>store_street</i>
1	1	12.50	Boulevard Bart
2	1	10.50	Boulevard Bart
3	2	9.25	Misiones
2	3	9.85	Brazil av.
5	3	2.30	Brazil av.

Não compliance: (*store_street*)



5 Formas Normais

3FN - Cada campo não chave de uma tabela não deverá fazer referência à uma chave estrangeira da mesma.

Ex.:

<i>[PK] project_id</i>	<i>[FK] university_id</i>	<i>name</i>	<i>university_phone</i>
1	1	Imunology	453333-2222
2	1	Citology	453333-2222
3	2	Robotics	559999-7777
4	3	Embbded Systems	331212-1212
5	3	Fluid Mechanics	331212-1212

Não compliance: (*university_phone*)



5 Formas Normais

4FN - Cada tabela de relacionamento deve possuir chaves estrangeiras porem deve se evitar ternários

Não compliance:

<i>[PK][FK] employee</i>	<i>[PK][FK] skill</i>	<i>[PK][FK] language</i>
--------------------------	-----------------------	--------------------------

Compliance:

<i>[PK][FK] employee</i>	<i>[PK][FK] skill</i>
--------------------------	-----------------------

<i>[PK][FK] employee</i>	<i>[PK][FK] language</i>
--------------------------	--------------------------



5 Formas Normais

5FN - Trata os casos onde uma determinada informação pode ser reconstruída de informações menores combinadas. Ela em nada difere da 4FN se não houver uma constante simétrica que atue como uma regra de negocio entre as tabelas em questão. Na ausência desta constante, se o esquema estiver na 4FN, automaticamente estará, também, na 5FN.

Não compliance:

<i>[PK][FK] seller</i>	<i>[PK][FK] company</i>	<i>[PK][FK] product</i>
<i>1</i>	<i>Samsung</i>	<i>Hard drives</i>
<i>1</i>	<i>Samsung</i>	<i>Phones</i>
<i>1</i>	<i>Apple</i>	<i>Phones</i>



5 Formas Normais

Compliance:

<i>[PK]/[FK] seller</i>	<i>[PK]/[FK] company</i>
<i>1</i>	<i>Samsung</i>
<i>1</i>	<i>Apple</i>

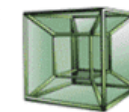
<i>[PK]/[FK] seller</i>	<i>[PK]/[FK] product</i>
<i>1</i>	<i>Hard drives</i>
<i>1</i>	<i>Phones</i>

<i>[PK]/[FK] company</i>	<i>[PK]/[FK] product</i>
<i>Samsung</i>	<i>Hard drives</i>
<i>Samsung</i>	<i>Phones</i>
<i>Apple</i>	<i>Hard drives</i>
<i>Apple</i>	<i>Phones</i>



NoSQL

São considerados Banco de Dados porém não obedecem aos requisitos de Bancos de Dados Relacionais, possuem um sistema de persistência, formas de acesso aos dados, com a capacidade de realizar escrita e leitura porém com as habilidades de clusterização e Tolerância a Partições.

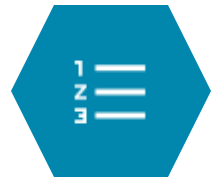
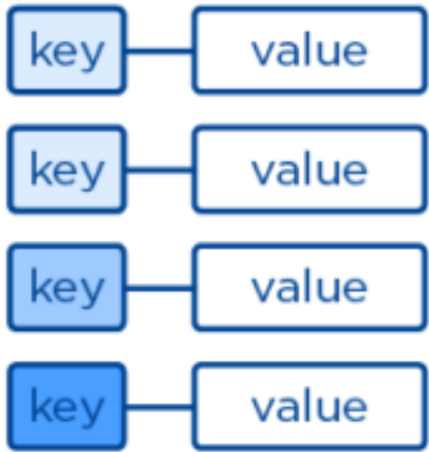


mongoDB

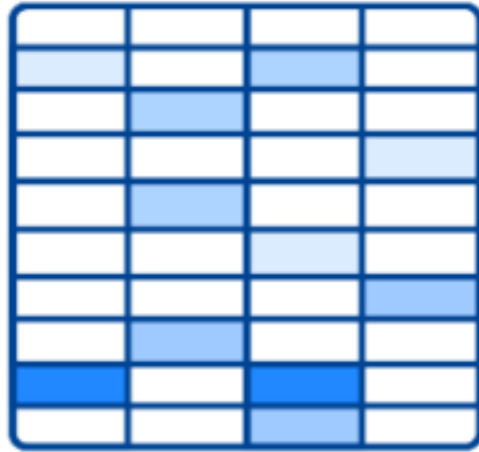
HYPERTABLE^{INC}



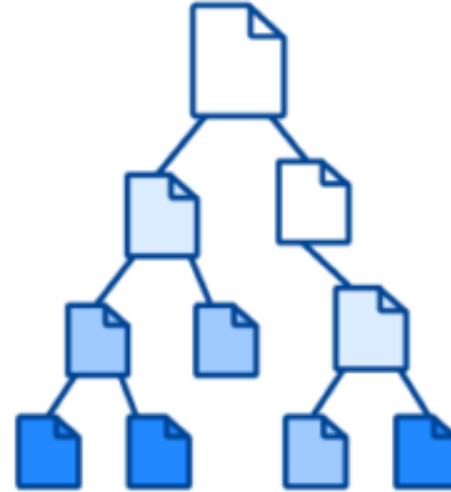
NoSQL



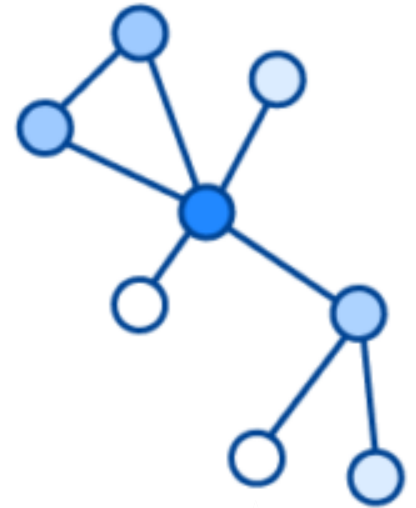
Chave-Valor



Colunar



Documental



Grafo

Casualmente distribuídos em 4 tipos



CAP Theorem

A ideia por trás do teorema CAP é escolher a sua base de dados baseado em princípios de tolerância a partição. Bancos de dados relacionais (CA) conseguem apenas escalar verticalmente, ou seja, só é possível aumentar a capacidade de resposta da base aprimorando o hardware, enquanto que, as bases NoSQL (AP/CP) possuem a habilidade de escalar horizontalmente, isto é, adicionar novos nós no cluster e por consequência aumentar a capacidade de resposta das bases de maneira virtualmente ilimitada, porém para isso precisamos escolher entre disponibilidade ou consistência.

AP: Nos Bancos NoSQL considerados eventualmente consistente, a base passa por momentos de inconsistência e isso significa que as versões dos dados podem estar desatualizadas, porém sempre está disponível e isso significa que mesmo com uma partição de rede ainda consegue receber leituras e escritas.

CP: Bancos NoSQL considerados consistente, a base sempre está consistente e isso significa que todos os clientes possuem as mesmas versões dos dados e mesmo com uma partição a base consegue receber as requisições de leitura e escrita, porém em caso de uma condição de corrida em escritas algumas requisições podem retornar erros de escrita.

Visual Guide to NoSQL Systems



“

[C] Consistency

[A] Availability

[P] Partition Tolerance



BASE

Em contrapartida do compliance ACID para os RDBMS's foi cunhado outro acrônimo para determinar uma base NoSQL.

Basic Availability

Soft State

Eventual consistency



Domain Driven Design

O termo foi cunhado por Eric Evans em seu livro de mesmo título publicado em 2003.

“É um conjunto de princípios com foco em domínio, exploração de modelos de formas criativas e definir e falar a linguagem Ubíqua, baseado no contexto delimitado.”

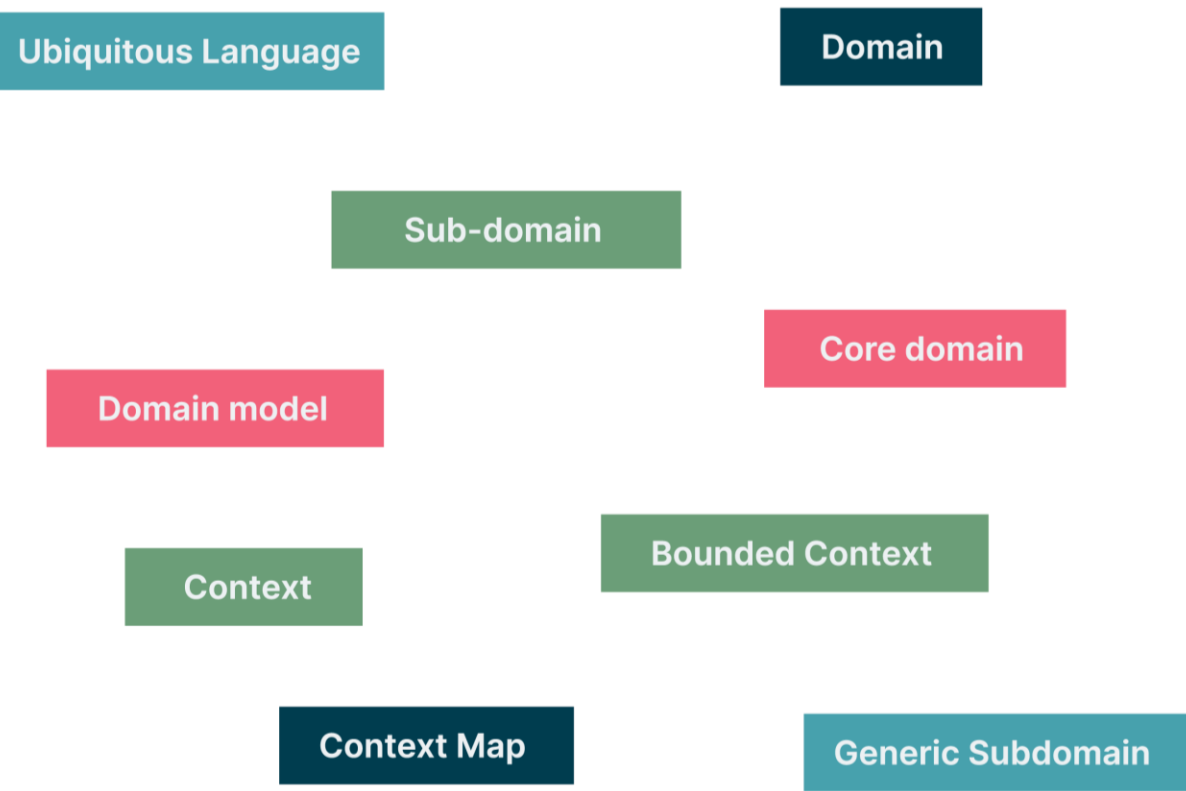
Evans, Eric – 2003

Tratando a questão do desenvolvimento de software sob uma ótica de semântica, apesar do core da ideia ser o **domínio**, antes de mais nada **DDD** se trata de um framework de ideias, não um framework propriamente dito como um Spring ou algo do gênero, mas sim um arcabouço que força o time a trabalhar dentro de um mesmo vocabulário, modelando o domínio de forma discreta e em conjunto com os especialistas dos mesmos além de cirandar fronteiras entre as intersecções do modelo de negocio.

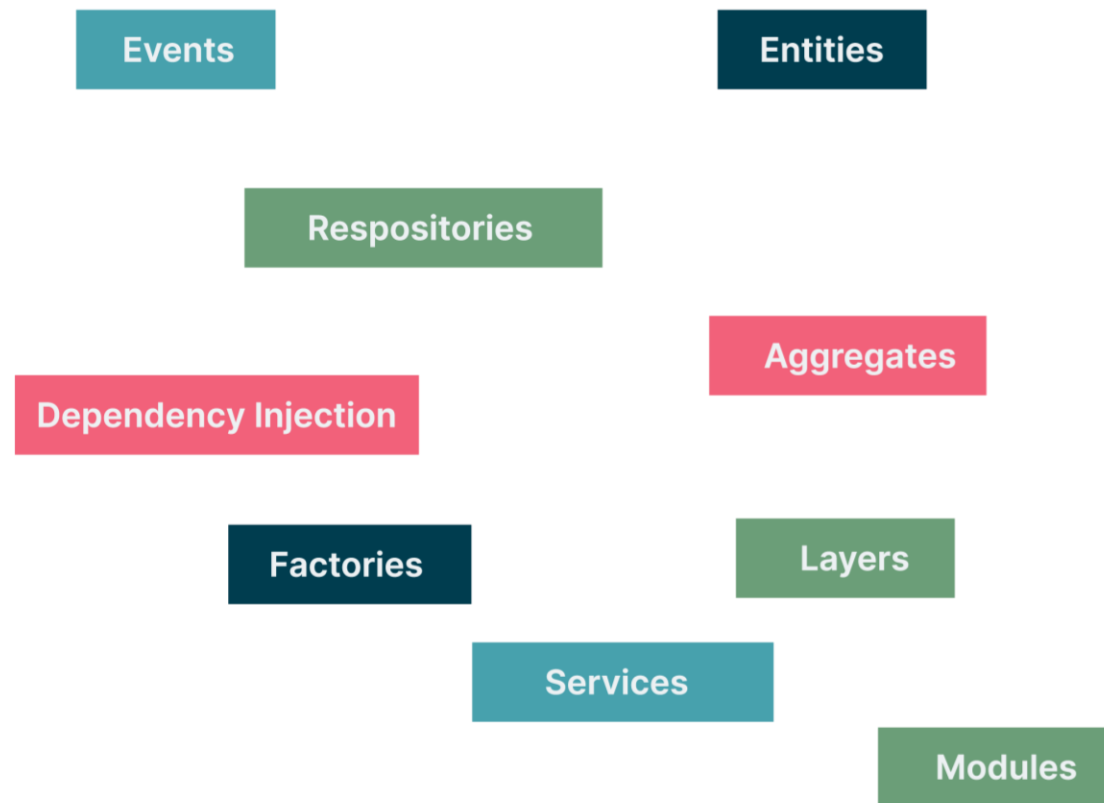


Domain Driven Design

Strategic design



Tactical patterns





Domain Driven Design

Domínio

Uma esfera de conhecimento, influência ou atividade. A área temática para a qual o usuário aplica um programa é o domínio do software.

Modelo

Um sistema de abstrações que descreve aspectos selecionados de um domínio e pode ser usado para resolver problemas relacionados a esse domínio.

Linguagem Ubíqua

Uma linguagem estruturada em torno do modelo de domínio e usada por toda a equipe membros dentro de um contexto limitado para conectar todas as atividades da equipe com o software.

Contexto

A configuração em que uma palavra ou declaração aparece que determina seu significado. As declarações sobre um modelo só podem ser entendidas em um contexto.

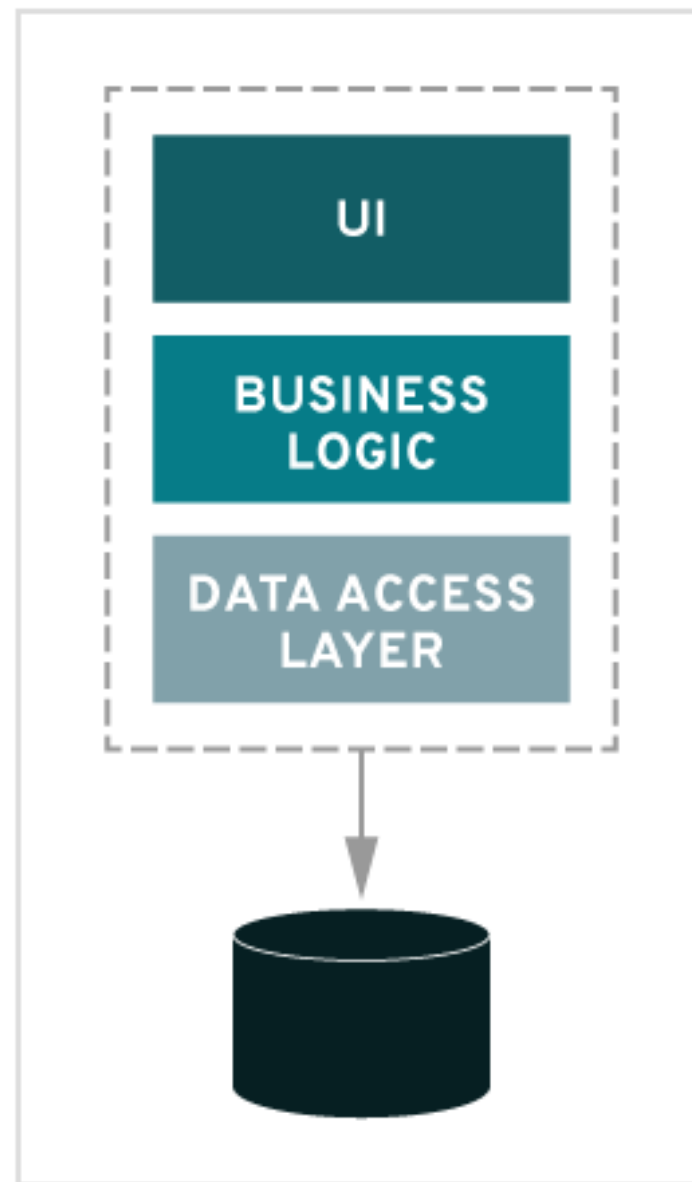
Contexto Limitado

Uma descrição de um limite (normalmente um subsistema ou o trabalho de um determinado equipe) dentro do qual um determinado modelo é definido e aplicável.

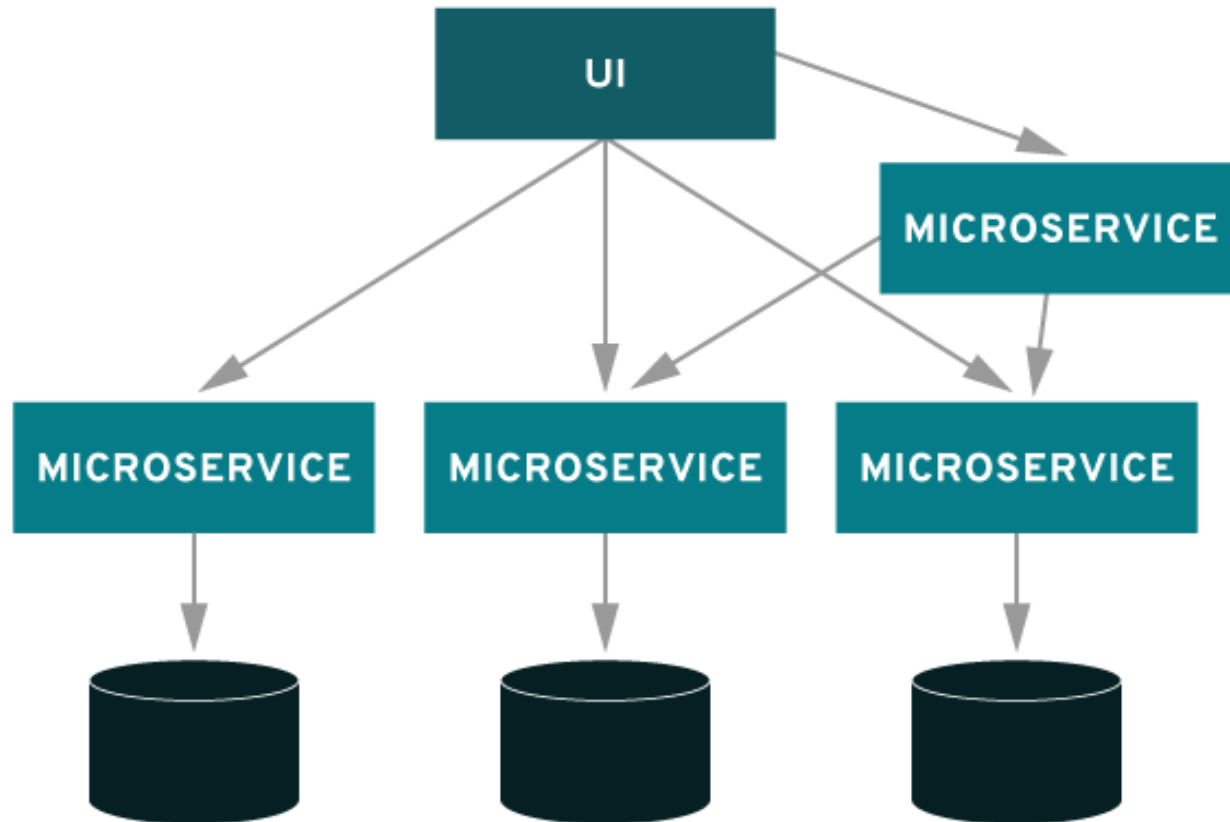
“

Modelo de software onde todos as features estão juntas. Possuindo alto acoplamento!

MONOLITHIC



MICROSERVICES



“

Conceito de modelagem de software onde os contextos do negócio são distribuídos em serviços independentes!



Mensageria e Comunicação Síncrona vs Assíncrona

Quando o cliente de nossa aplicação faz uma requisição para o servidor ele normalmente espera uma resposta, o padrão de request/response emerge a partir deste ponto, muito comum em serviços web, este é um fluxo de informação geralmente síncrono, onde o cliente que fez a requisição fica esperando o servidor retornar com a resposta para renderizar no browser.

Nada de novo ate aqui, agora imaginem que o serviço que esta sendo requisitado dependa de 3 outros serviços, uma consulta na base, uma persistência em outra base e um envio de e-mail, algo muito comum em um checkout de um e-commerce, então o serviço inicial segura o request, e faz 3 requisições sequenciais para estes serviços, a consulta, a persistência e envio de email e ao final retorna uma pagina com o resultado.

Criamos aqui uma interdependência acoplada e por consequência um problema!

Digamos que algum destes serviços esteja sofrendo com algum pico de acesso, todo o checkout do e-commerce ficaria lento e a quantidade de clientes pendurados no serviço web principal poderia derrubar o serviço como um todo. Uma maneira muito eficiente de resolver este tipo de problema é incorporar um Broker e transformar este fluxo sequencial e síncrono em um fluxo assíncrono, desta forma liberando o servidor web para suportar as requisições evitando que este seja derrubado.

Comunicação Síncrona vs Assíncrona

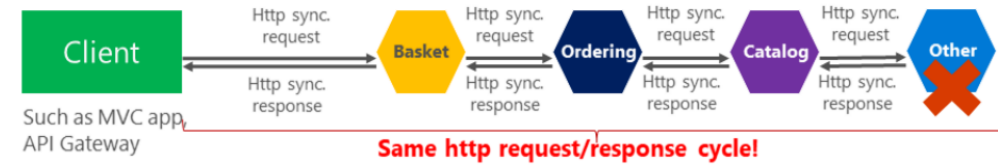
“

Em um ecossistema de software distribuído é recomendado o uso de comunicação assíncrona entre os microserviços!

Synchronous vs. async communication across microservices

Anti-pattern

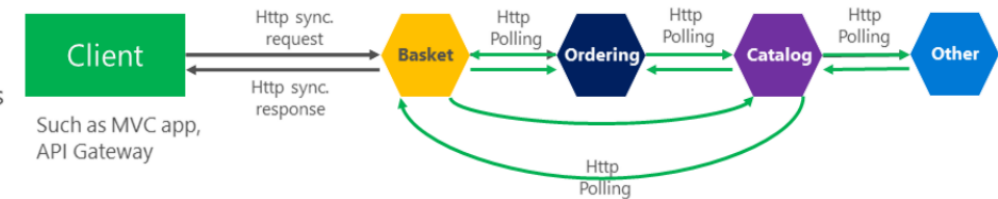
Synchronous
all request/response cycle



Asynchronous
Comm. across internal microservices
(EventBus: like **AMQP**)



“Asynchronous”
Comm. across internal microservices
(Polling: **Http**)





Brokers de Mensageria

Brokers, são tipicamente servidores de fila que implementam as técnicas de pub/sub e ou RPC entre outras que tem como proposito fazer a comunicação entre os serviços ou microsserviços.

Antes de definirmos a utilização de um broker devemos nos atentar para algumas características do mesmo, por exemplo temos

Protocolo: AMQP, MQTT, XMPP, (G)RPC...



Tipo da Mensagem: Protobuf, Json, Texto, Binário, Avro...



Arquitetura do broker:

- **RabbitMQ (AMQP):** Como um servidor de filas, a regra fica armazenada no próprio servidor baseada em rotas e dispatchers, onde a tendencia do servidor eh a fila zerar;
- **Kafka (RPC):** Como um log storage, a regra fica a cargo de quem produz ou consome as filas, onde a tendencia do servidor eh a fila crescer eternamente;
- **Mosquito (MQTT):** Como um servidor de filas, a regra fica a cargo de quem produz ou consome as filas, a tendencia do servidor é manter um buffer de mensagens fazendo um reload a fim de não estourar o espaço;
- **EJabberD (XMPP):** Como um servidor de filas, a regra fica armazenada no próprio servidor baseada em rotas e dispatchers, onde a tendencia do servidor é a fila zerar;

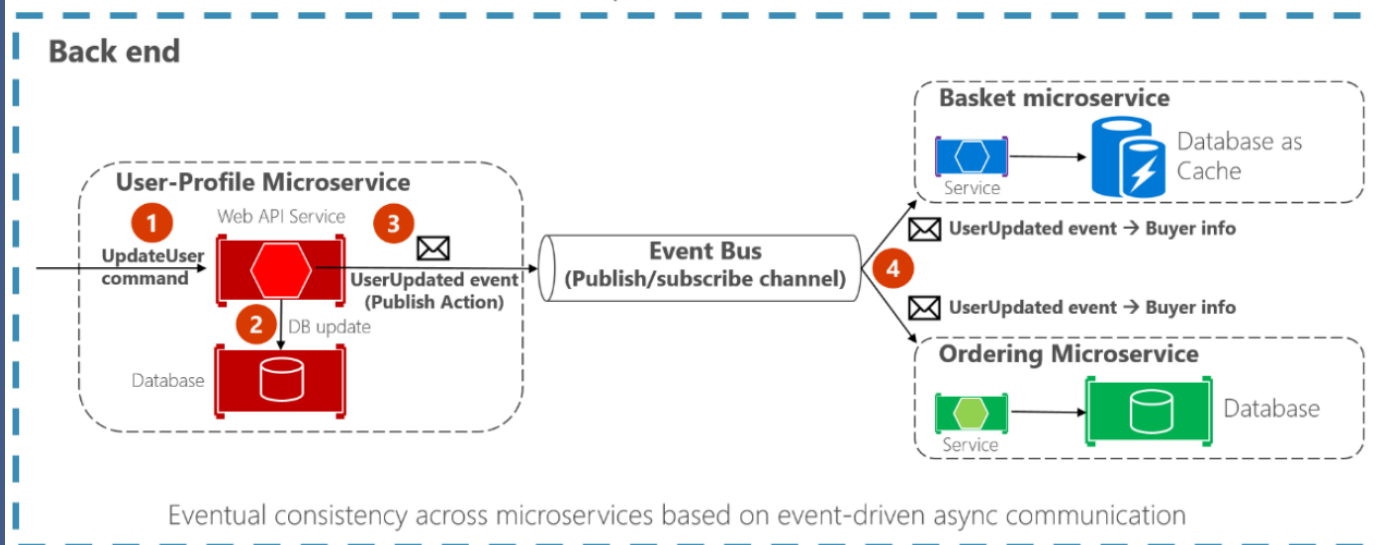
Comunicação Baseada em Eventos

“

Na comunicação assíncrona com eventos, um microserviço publica eventos em uma fila e muitos microserviços podem se inscrever nesta fila, para serem notificados e processarem os eventos recebidos!

Asynchronous event-driven communication

Multiple receivers

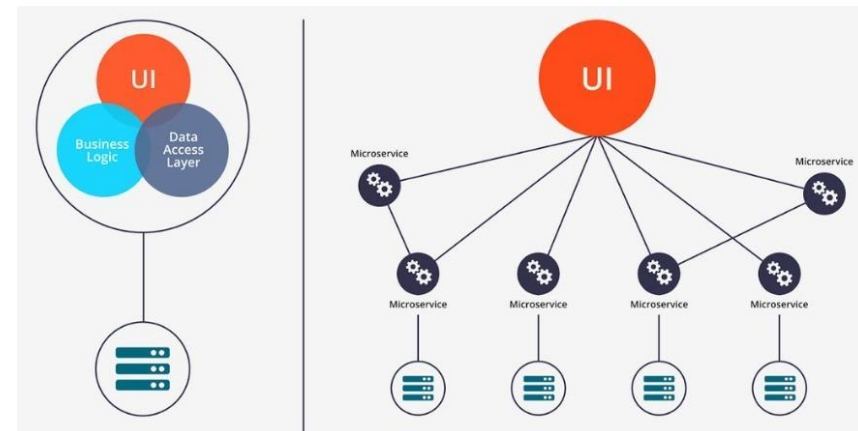




Quebrando Aplicações Monolíticas

Mesmo aplicando o DDD não necessariamente sairemos do outro lado com todas as melhores praticas de desenvolvimento implementadas. E ainda, levando em conta a regra de evitar otimização prematura de software, as vezes terminamos com um monolito nas mãos.

Um software considerado monolito eh aquele em que todos os contexto boundaries estão acoplados no mesmo pacote de release, isso significa que em caso de deploy toda a aplicação ficara fora do ar, e caso de algum erro catastrófico ocorra, nenhuma parte da aplicação permanecera em execução, além de ser um problema na hora de escalar.



A solução para este caso vem da ideia de microsserviços, divide-se o monolito em vários microsserviços onde estes por sua vez são responsáveis por fazer apenas uma tarefa, criando então uma rede de microsserviços consumindo uns aos outros afim de ao final ter resiliência na aplicação como um todo.

A comunicação dos microsserviços normalmente é feita com a ajuda de um broker de mensageria!



Concorrência e Complexidade ciclomática

Apesar de estes 2 assuntos estejam intimamente conectados a primeira vista parecem temas absolutamente separadas. Quando tratamos a respeito de concorrência, estamos lidando com a quantidade de usuários simultâneos sendo recepcionados pelos serviços.

Existem muitos modelos de concorrência, o multi-thread da jvm, o event-loop do javascript, o modelo de atores do beam entre outros como os híbridos que é o caso do nginx, todos com o mesmo objetivo, tentar ser o mais resiliente em relação a atender o máximo de requisições simultâneas.

A complexidade ciclomática se refere a quantidade de forks ou condicionais de uma rotina, existem maneiras de medir estes condicionais normalmente utilizando ferramentas de introspecção ou instrumentalizando o código para trazer esta informação para os desenvolvedores.

Ok, mas onde está a "cola" destas ideias? Bom levando em conta que a concorrência trata a quantidade de requisições e a complexidade ciclomática a quantidade de condicionais de uma rotina, podemos então entender que, quanto menor a complexidade ciclomática menor o tempo em que o processo vai estar ocupado e por consequência mais requisições serão atendidas.



Concorrência e Complexidade ciclomática

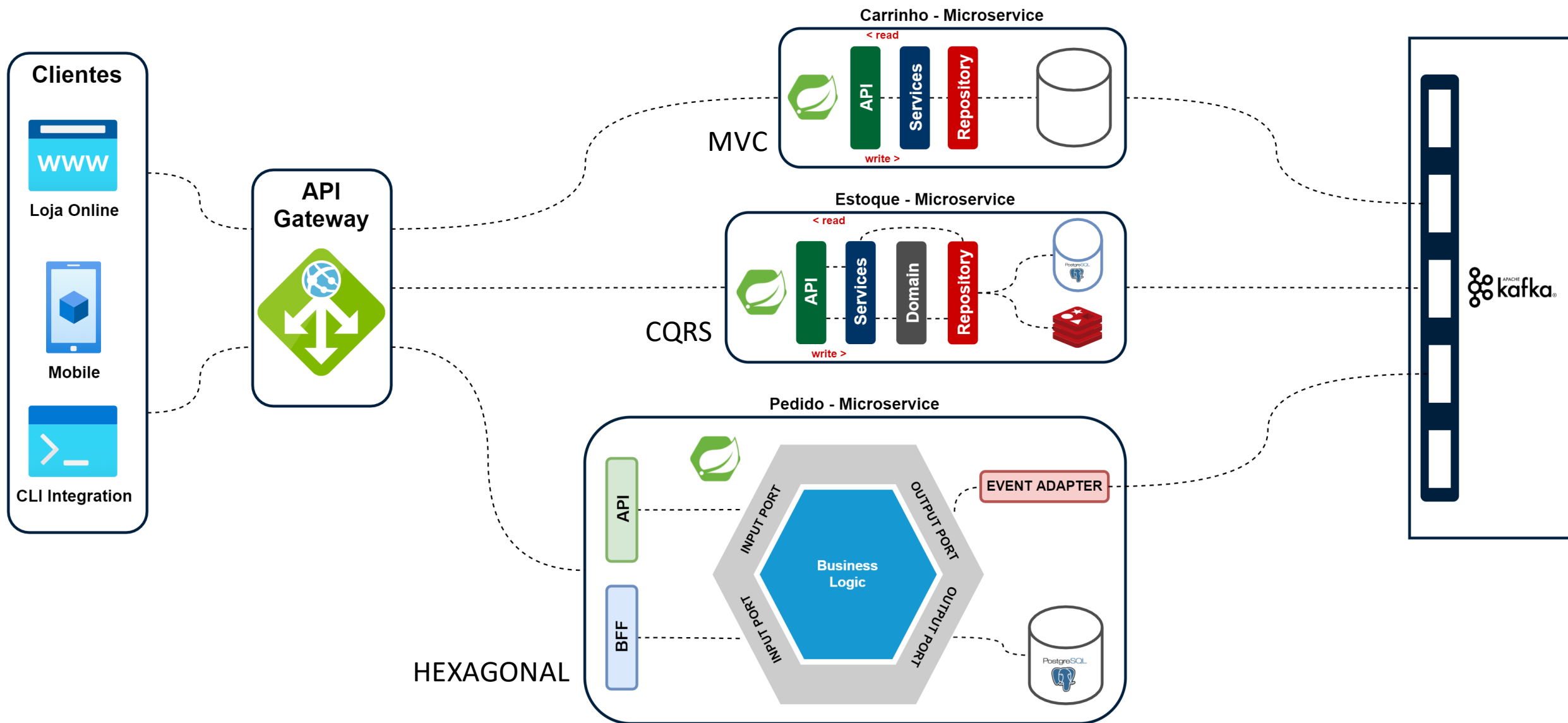
Porém não é apenas isso, no caso do nodejs que veio com a ideia de quebrar a barreira de "10k simultaneous users" isso pode ser mais perigoso do que parece, o modelo de concorrência do nodejs como já citado acima, é o event-loop e a sua natureza single thread, faz com que qualquer adição a complexidade ciclomática cause um block no processamento, fazendo com que todas as requisições fiquem estacionadas em um estado de espera para liberar o processamento. Sendo assim para atingirmos os famosos 10k users precisaremos utilizar de algumas técnicas.

Vamos a algumas:

- *Evite I/O Blocks, trate as chamadas de I/O como rede e disco como assíncronas dentro do modelo do event-loop;*
- *Evite Blocos de for ou while, troque pelas funções iterativas de Map, Filter e Reduce dentro de promessas, assim sempre que possível o processo irá ser capaz de continuar a atender as requisições mesmo realizando forks e condicionais;*
- *Utilize sempre que possível Stream & Pipes, afinal tudo nesta natureza é "não bloqueante" e é enviado para a libuv que irá processar isso em um C compilado;*



Principais Patterns e Modelos Arquiteturais





Professional Tips!!!





Versionamento – Git/GitHub

O git é um sistema de controle de versão distribuído gratuito e de código aberto projetado para lidar com tudo, desde projetos pequenos a muito grandes com velocidade e eficiência.

O **Github** é um "repositório" de **Gits** online dividido por usuários.

Onde o mundo constrói software, milhões de desenvolvedores e empresas criam, enviam e mantêm seus softwares.

Alguns repositórios **git** são criados no modelo **Monorepo** e precisam de ferramentas especiais para manipula-los



git



GitHub



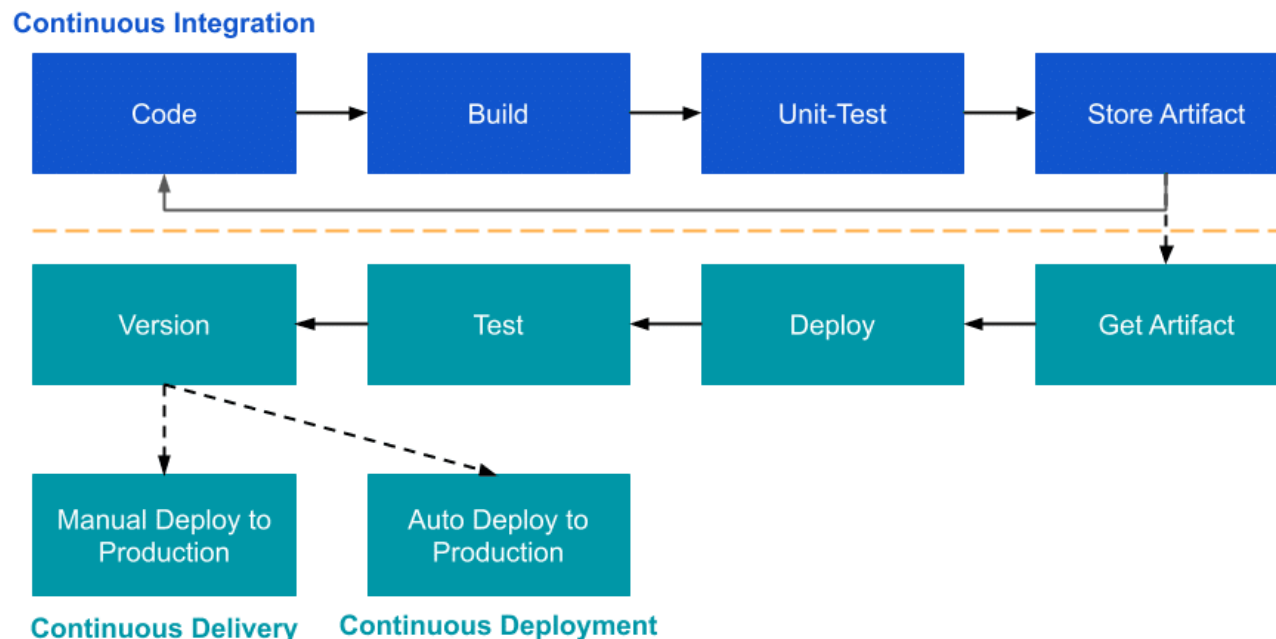
CI/CD – Botando em Produção

CI/CD preenche as lacunas entre as atividades e equipes de desenvolvimento e operação, reforçando a automação na compilação, teste e implantação de aplicativos.

O processo contrasta com os métodos tradicionais, onde todas as atualizações eram integradas em um grande lote antes de lançar a versão mais recente.

As práticas modernas de DevOps envolvem desenvolvimento contínuo, teste contínuo, integração contínua, implantação contínua e monitoramento contínuo de aplicativos de software ao longo de seu ciclo de vida de desenvolvimento.

CI/CD Workflow




```
class Repeat
  def print_message
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
  end
end
```

DRY - Don't Repeat Yourself

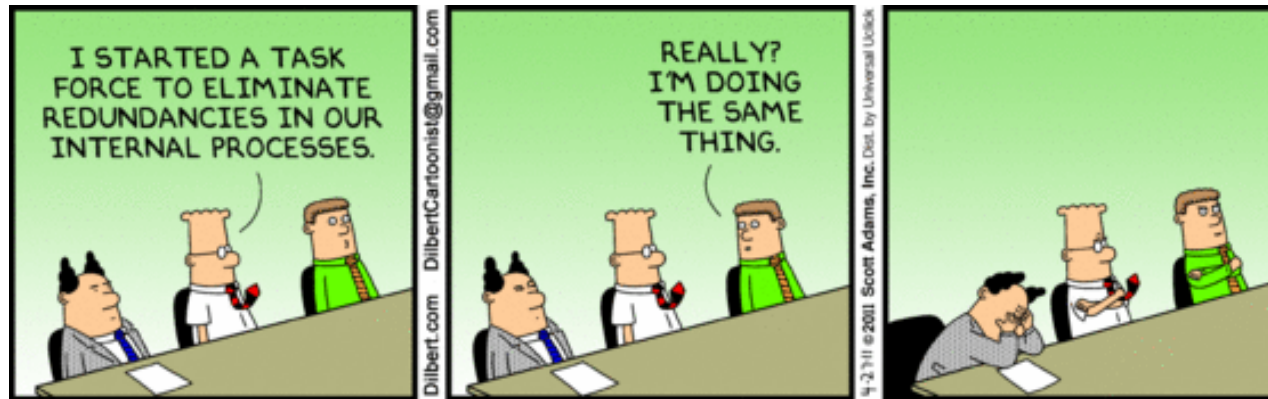
DRY é sobre a duplicação do conhecimento, da intenção. Trata-se de expressar a mesma ideia em dois lugares diferentes, possivelmente de duas maneiras totalmente diferentes.

O Argumento do Dry é que cada peça de software deve ter uma representação única, inequívoca e dentro de um sistema.



KISS - Keep It Simple, Stupid

KISS é sobre manter poucas funções em cada modulo, e cada função conter o mínimo possível de linhas, este número pode variar, mas não passa de 50/60 linhas por função.



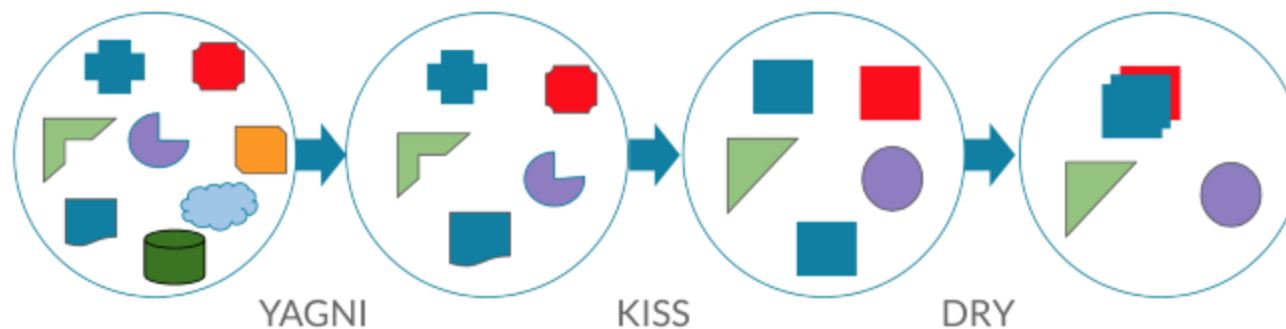
O Argumento do Kiss é que o código é escrito por devs para ser lido por devs, manter-se simples agiliza inclusive o onboard de novos membros e cria um ambiente saudável para manutenção.



YAGNI - You Ain't Gonna Need It

YAGNI é sobre se manter no core da aplicação, evitando otimizações prematuras ou ainda criação de boilerplates tentando *"facilitar tal feature no futuro"*.

O Argumento do Yagni é sobre gastar tempo e esforço de desenvolvimento com features que estão planejadas para o momento, desta forma evitando o crescimento de complexidade e focando no que realmente importa.





[Microservices // Dicionário do Programador - YouTube](#)

[O que são Microserviços? \(Microservices\) #HipstersPontoTube - YouTube](#)

[Microserviços ou Monolíticos? Qual você deve escolher? | por André Baltieri #balta - YouTube](#)

[Microserviços Java na prática com DDD, CQRS e Event Sourcing - Parte 1 - YouTube](#)

[SOLID fica FÁCIL com Essas Ilustrações - YouTube](#)

[DDD, SOLID, Clean Code, Clean Architecture... Isto é para você? - YouTube](#)

[A Solid Guide to SOLID Principles | Baeldung](#)

[Clean Code // Dicionário do Programador - YouTube](#)

[Descomplicando "Event Sourcing" - YouTube](#)

A blue-tinted background image showing a business meeting. A man in a grey blazer stands pointing at a whiteboard, while others sit at a table with laptops and tablets displaying charts.

Obrigado!