COTI INFORMÁTICA
ESCOLA DE NERDS

Resumo – Arquitetura Limpa

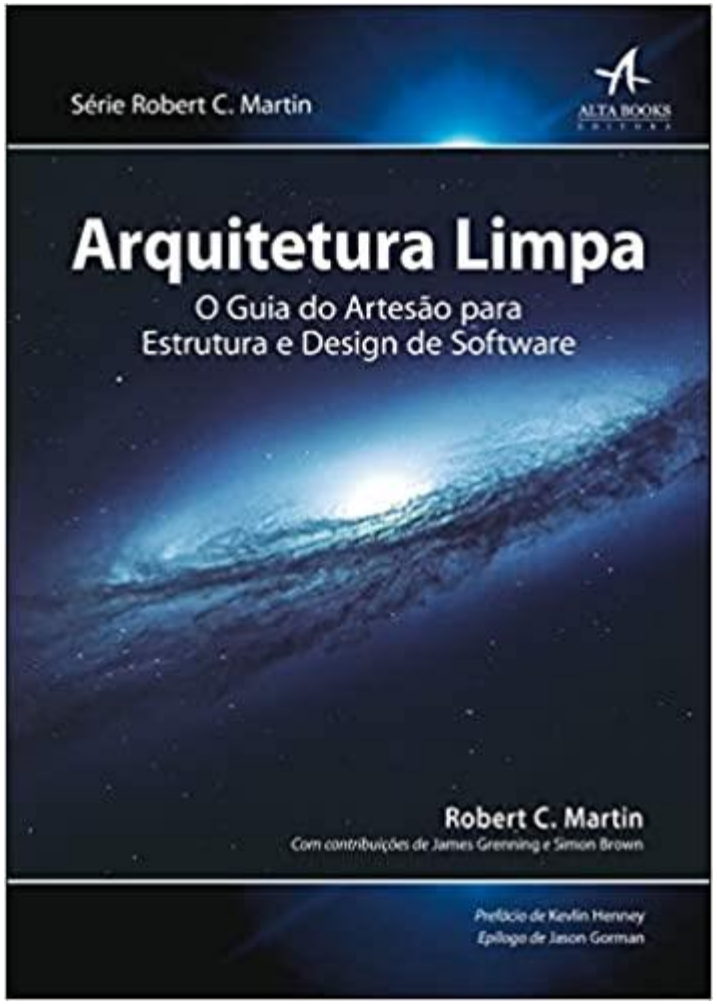
Java WebDeveloper – Formação FullStack

Professor Sergio Mendes

COTI INFORMÁTICA
ESCOLA DE NERDS

Resumindo os principais conceitos do livro **Arquitetura Limpa**

Arquitetura limpa: O guia do artesão para estrutura e design de software
Robert C. Martin



Arquitetura é o processo de maximizar o trabalho e a produtividade com o mínimo esforço necessário, através de técnicas auxiliares como, por exemplo, TDD. Desenvolvedores mais jovens devem ter em mente que começar um código sujo, somente com o objetivo de *"entregar e fazer funcionar"* é uma atitude irresponsável. Há uma falsa ilusão de que depois que o projeto for entregue, é possível a sua refatoração e reescrita.

O desenvolvedor, como arquiteto, é o responsável pela entrega do produto e, em muitos casos, pela sua evolução.

Uncle Bob reforça que o processo de se aplicar uma boa engenharia antes, durante e após o projeto é imprescindível. Isso porque, é mais fácil alterar um programa que não funciona, porém está bem dividido e *"arquitetado"*, ao invés de um sistema que funciona, contudo está engessado. É dever do desenvolvedor/arquiteto defender a boa engenharia, os bons princípios e os padrões de qualidade para o código.

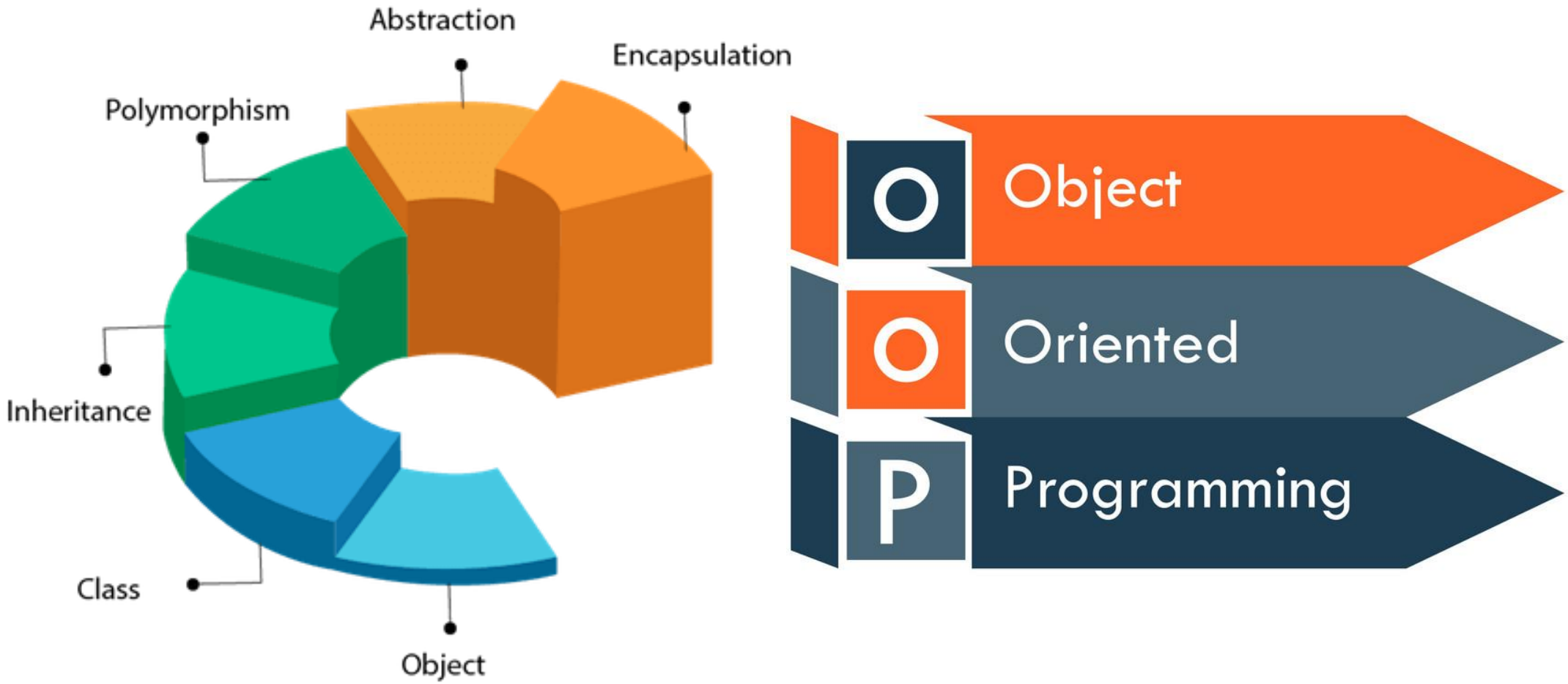
Programação orientada a objetos (OO)

Começo dizendo que OO é a base da arquitetura. Sem entender os seus principais conceitos, acaba sendo impossível construir uma boa arquitetura e saber o que de fato se está fazendo.

OO é a técnica de abstrair um objeto do mundo físico em uma representação virtual. Podemos utilizar técnicas como encapsulamento, herança e polimorfismo.

Com o encapsulamento definimos limites. Com a herança damos a um objeto o poder de herdar comportamentos de seu objeto pai. Já com o polimorfismo podemos trocar de lugar objetos por outros que tenham o pai em comum e sobrescrever seu funcionamento, sem alterar o funcionamento do sistema. Esse último em minha opinião é o mais importante dos conceitos.

OOPs (Object-Oriented Programming System)



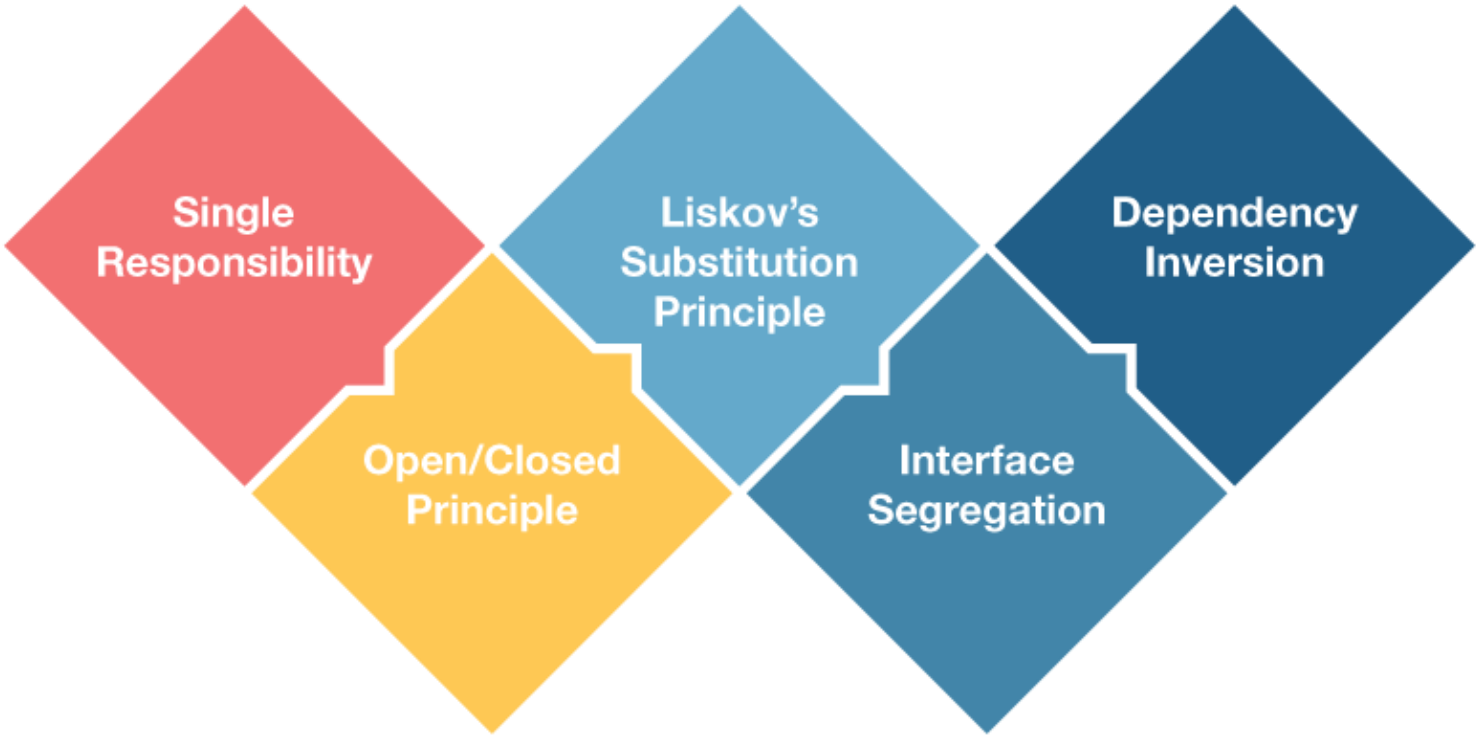
Princípios de Design (SOLID)

Resumidamente, o solid nos orienta a criar sistemas de fácil escrita, manutenção e portabilidade.
Seus pilares são:

S — Princípio da Responsabilidade Única (SRP): Uma classe deve ter um, e somente um, motivo para mudar.
O — Princípio do Aberto/Fechado (OCP): Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo.
L — Princípio da Substituição de Liskov (LSP): As classes base devem ser substituíveis por suas classes derivadas.
I — Princípio da Segregação de Interfaces (ISP): Muitas interfaces específicas são melhores do que uma interface única.
D — Princípio da Inversão de Dependências (DIP):
Dependa de uma abstração e não de uma implementação.

COTI INFORMÁTICA
ESCOLA DE NERDS

S.O.L.I.D.



Coesão de Componentes

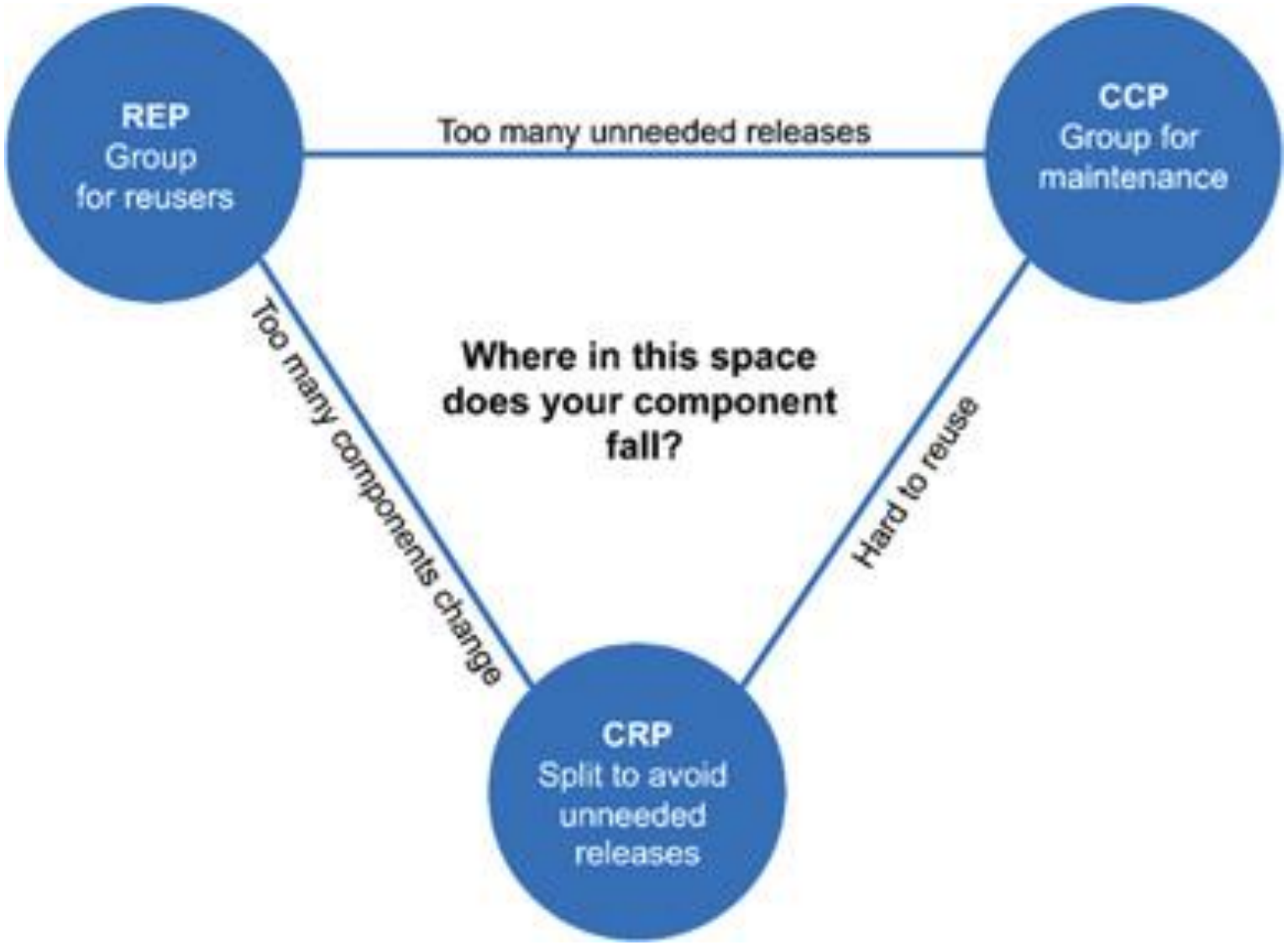
Aqui definimos alguns princípios de componentes (componente é um classe do seu sistema).

Os princípios são:

REP — Princípio da Equivalência do Reuso/Release: diz que os componentes devem ficar todos juntos para que seja fácil a sua reutilização.

CCP — Princípio do Fechamento Comum: diz que devemos deixar junto o que faz sentido, ou seja, todo o código que tem tendência a mudar junto.

CRP — Princípio do Reuso Comum: esse princípio muito importante diz que não devemos entregar para o usuário mais do que ele realmente precisa. Devemos manter juntas somente as classes que o usuário irá utilizar.


REP
Group
for reusers
Too many unneeded releases
CCP
Group for
maintenance
Too many components change
Where in this space
does your component
fall?
Hard to reuse
CRP
Split to avoid
unneeded
releases

Acoplamento de Componentes

Resumidamente, devemos organizar nosso código de uma forma que os módulos com as menores chances de mudança não dependam de módulos com maiores chances de mudança. Utilizamos uma abordagem topdown com DIP, invertendo o controle das dependências. Temos também alguns princípios a seguir.

SDP : Princípio da Dependência Estável: diz que devemos sempre buscar maximizar a dependência de objetos estáveis. Porém, devemos entender que alguma volatilidade faz parte do sistema.

SAP: Princípio da abstração Estável: esse princípio nos informa que os componentes mais estáveis são os abstratos, ou seja, nossas dependências devem sempre buscar utilizar abstrações.

O que é arquitetura?

Já podemos dizer que arquitetura é uma forma de criar um sistema de fácil entendimento, bem como de fácil desenvolvimento. Além disso, fácil de manter e também fácil para implementar.

Lembre-se: **os melhores arquitetos são ótimos programadores.**

Fica impossível criar soluções sem ter experiência e sofrer com os problemas. O arquiteto deve evoluir o sistema deixando opções abertas para uma rápida mudança.

Também, deve evitar ao máximo fazer escolhas que podem ser adiadas, os detalhes são só detalhes.

Independência

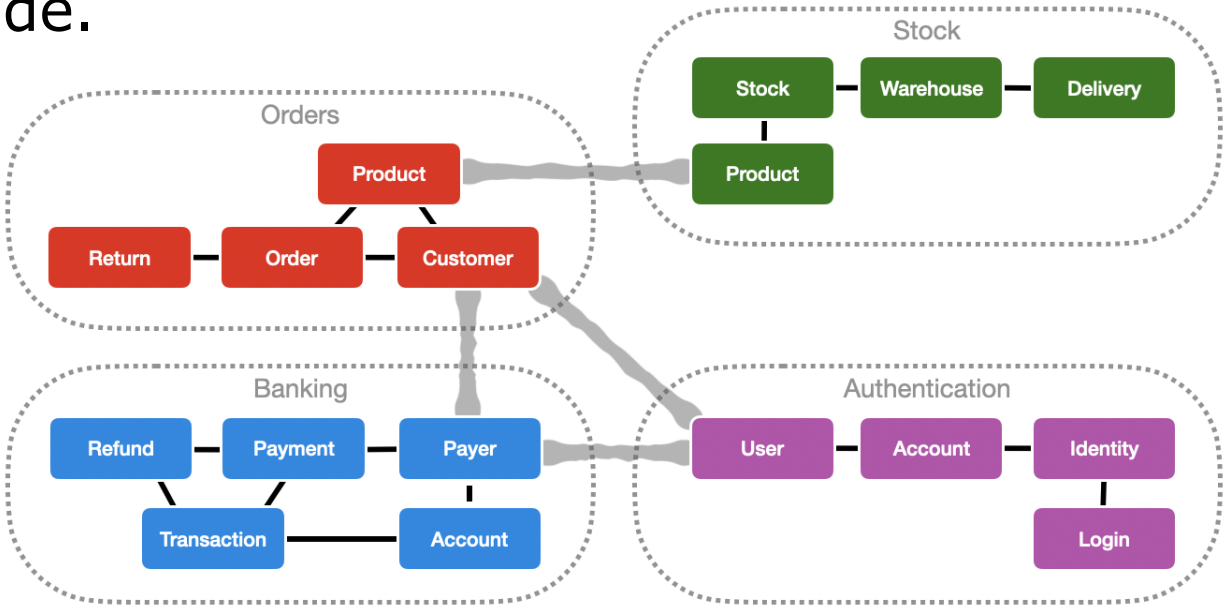
Neste capítulo podemos entender a importância de separar as responsabilidades corretamente. Não devemos ter pressa de criar paredes e separar sem que isso seja realmente necessário. Devemos evoluir o sistema deixando as opções em aberto.

O papel do arquiteto é visualizar pontos corretos de mudança e ter uma certa percepção de possíveis mudanças futuras. Uma dica é pensar nas partes que estão mais propensas a mudar em momentos diferentes, como a interface gráfica e a camada de regras de negócio.

Separar as camadas corretamente irá permitir, por exemplo, alterar a UI sem alterar as regras de negócio, alterar o tipo de banco de dados sem alterar as regra de negócio e alterar o tipo de comunicação sem alterar as regras de negócio. Sim, as regras de negócio devem ser o centro do seu sistema.

Fronteiras: Estabelecendo Limites

O papel mais importante do arquiteto é enxergar os pontos de mudança, os quais, normalmente, indicam que um limite/divisão deve ser ali criado. Aplicamos princípios já estudados, visando sempre deixar cada parte do sistema bem dividida e separando de forma coesa, tendo como objetivo atingir a estabilidade.



Regras de negócio

Entidade : Regras cruciais de negócio.

Casos de uso: Regras específicas da aplicação.

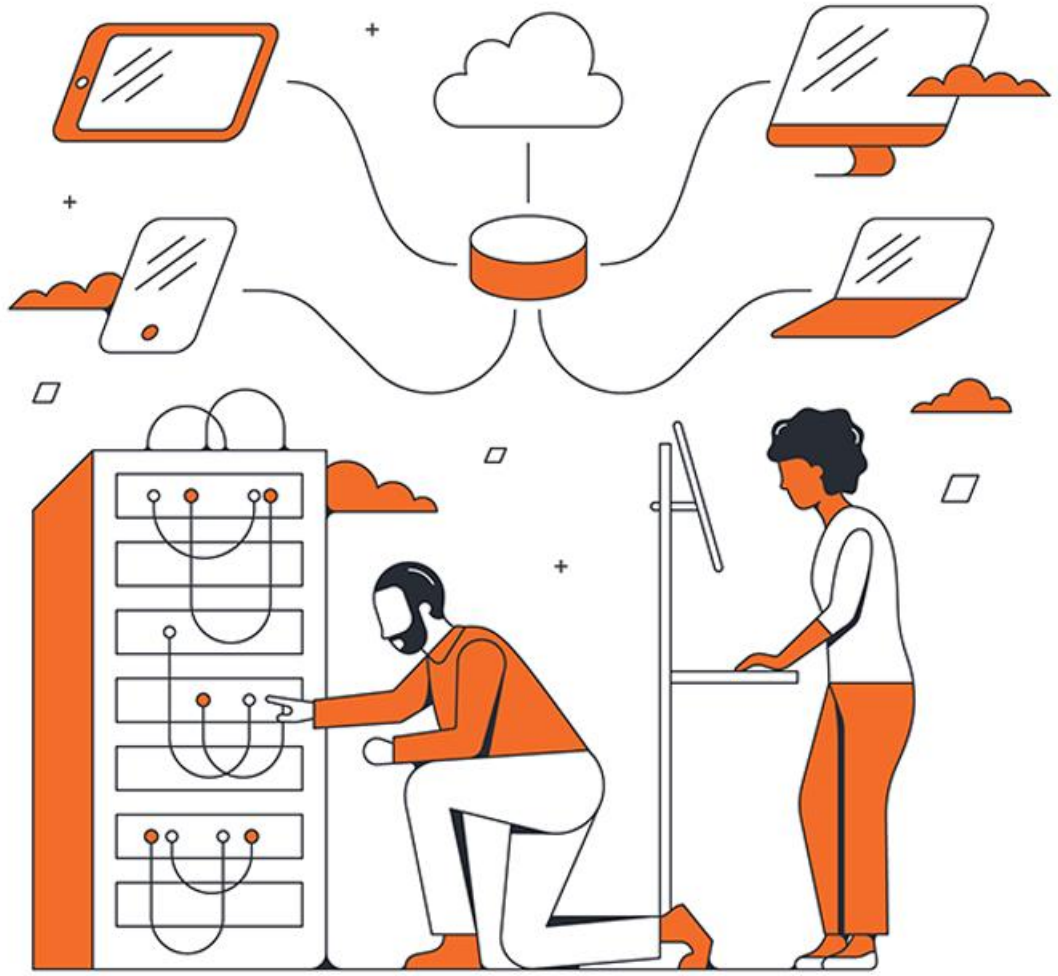
Começamos este capítulo entendendo que existem dois tipos de regras de negócios: as que ficam na **entidade** e as que são **caso de uso**.

A regra de negócio na entidade é uma regra que se aplica a entidade em si. Normalmente é idêntica em qualquer sistema que contenha essa entidade. Já a regra de **negócio de caso de uso** será onde iremos manter as regras da nossa aplicação. Ela pode mudar de aplicação para aplicação.
O caso de uso possui o tipo de entrada que o usuário deve oferecer, a saída que será gerada e os passos para chegar neste resultado esperado.

Arquitetura gritante

A primeira impressão do seu sistema é a que fica. Sua arquitetura deve ser capaz de descrever facilmente o que ela é e o que está fazendo.

Seu código deve deixar explícito para qualquer um que bater os olhos o que ele faz e como ele faz. Também devemos deixar claro que não dependemos de frameworks e que eles são só detalhes que podem ser modificados e alterados a qualquer momento.

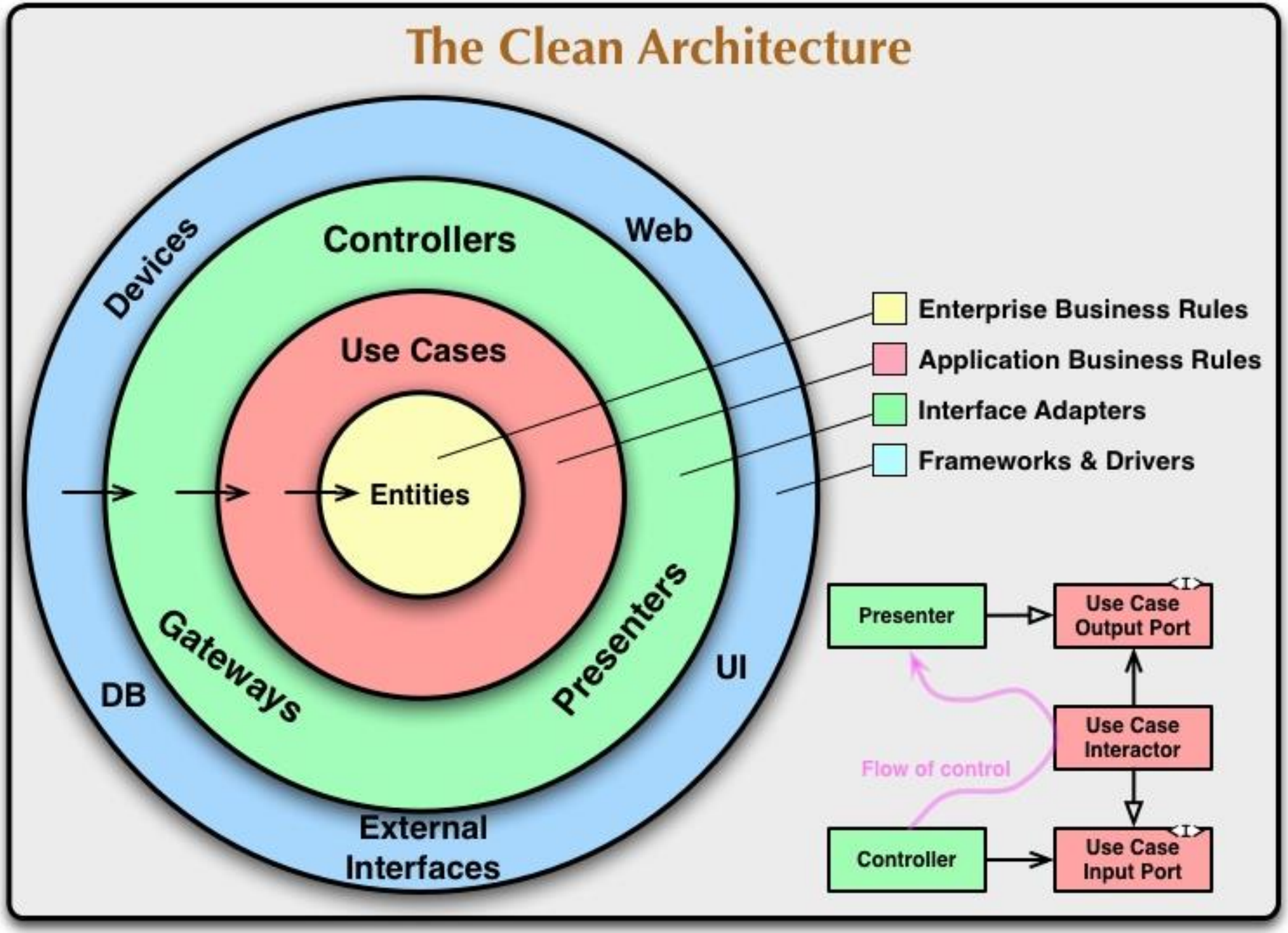


Arquitetura Limpa

Seu principal objetivo é a separação das preocupações.
Ela prega: Isolamento das regras de negocio, independência de frameworks, testabilidade, independência de UI, independência de banco de dados e independência de qualquer coisa externa.

Arquitetura Limpa propõe:

1 — Organização topdown level na qual no topo fica o mais importante e menos suscetível a mudanças.
2 — Elementos das camadas internas não devem ter conhecimento sobre as camadas externas.
3 — Frameworks e drives devem nas camadas mais afastadas do centro, onde não afetam a aplicação.

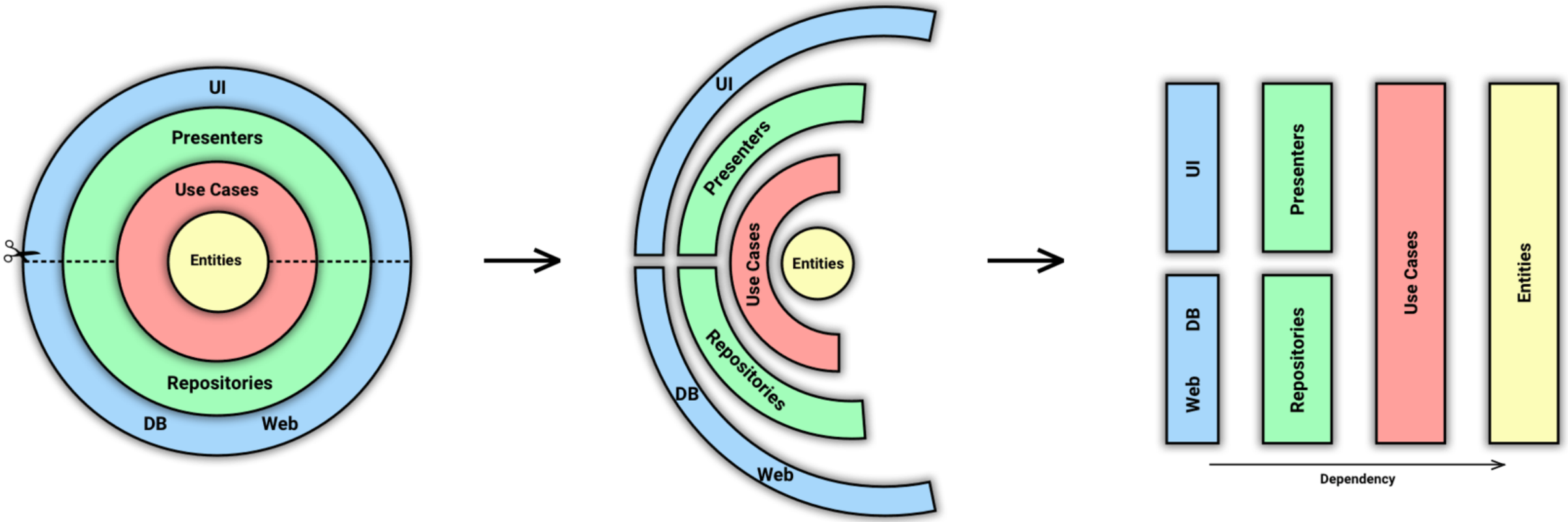

The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
<I>
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port
<I>

O Limite Teste

O teste é um ponto muito importante. Conseguimos orientar uma boa arquitetura com o decorrer dos testes. Ele nos dá dicas de como seguir. Os testes ficam na camada mais externa e eles tem acesso a tudo. Testes são todos iguais para a arquitetura. Devemos ter em mente não depender de coisas voláteis. Nossos testes não podem ser quebrados pela alteração de um framework, por exemplo.

Detalhes

Tudo que não faz parte da sua regra de negócio é detalhe. Como arquiteto, você deve sempre postergar os detalhes para mais tarde e definir bem os seus limites, para então, quando o momento chegar, a escolha por qual detalhe utilizar seja simples e de fácil alteração, quando necessária. Sobre os frameworks, vale ressaltar que eles não são arquiteturas. Se algum tenta ser, fuja e não amarre seu software nele. Não sabemos as direções que ele pode tomar e quando precisarmos sair dele para outro não será possível.


UI
Presenters
Use Cases
Entities
Repositories
DB
Web
UI
Presenters
Use Cases
Entities
Repositories
DB
Web
UI
Presenters
Use Cases
Entities
Web
DB
Repositories
Dependency

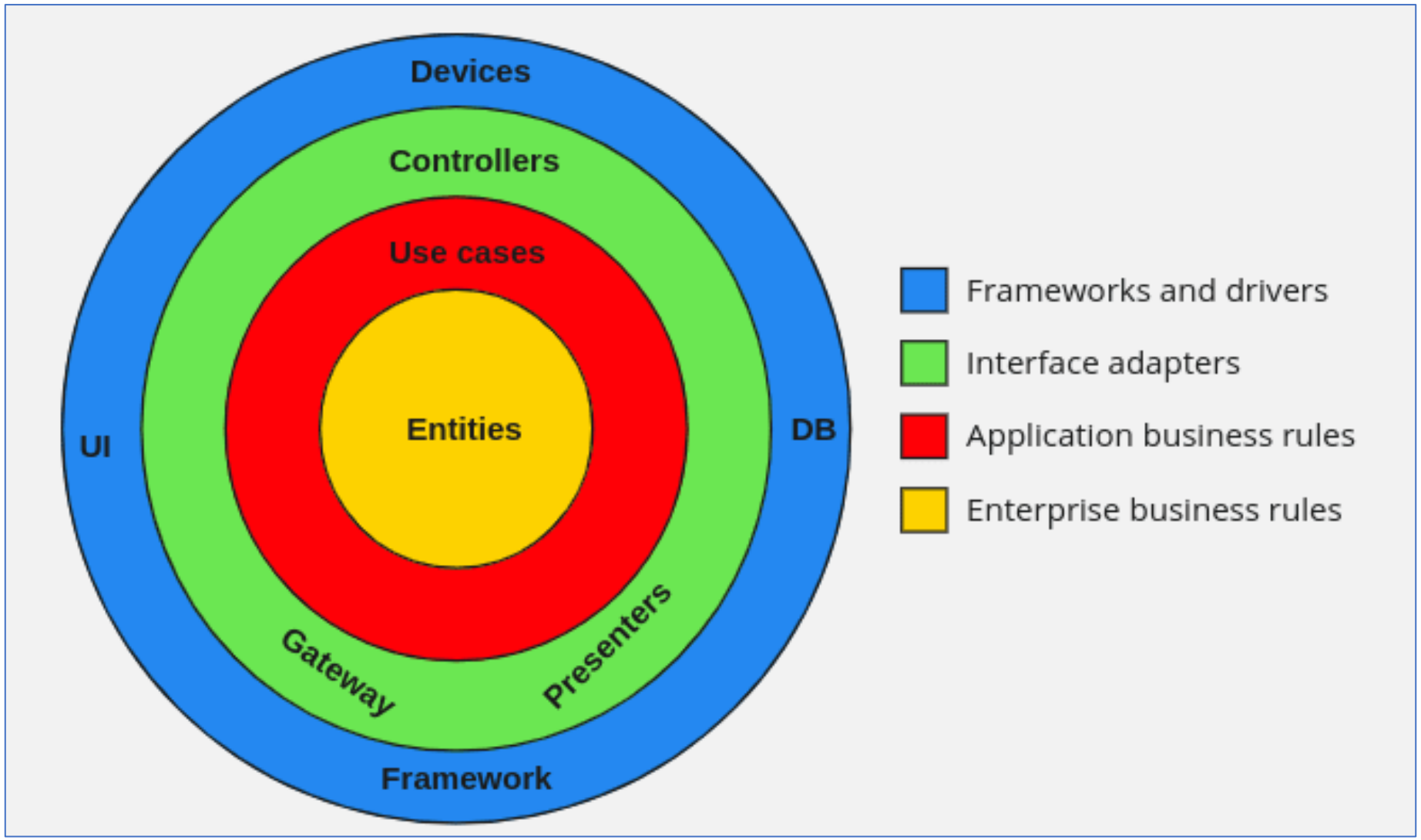
Apresentadores e objetos humble

Padrão utilizado em conjunto com práticas de teste, separando um objeto complexo em dois objetos, um para fácil teste e outro com atributos de alta complexibilidade de teste.

Também é útil para encontrar os limites da nossa aplicação, separando esses objetos.

O componente main

É a parte de nível mais baixa da sua aplicação, onde devemos fazer as injeções de dependência. A main conhece tudo.


Devices
Controllers
Use cases
Entities
UI
DB
Gateway
Presenters
Framework
Frameworks and drivers
Interface adapters
Application business rules
Enterprise business rules

{ COTI INFORMÁTICA
ESCOLA DE NERDS }