# Clean Architecture

Meu entendimento...

Jeann Andrade
Desenvolvedor de Software
Jeann.andrade@outlook.com

# Design ou Arquitetura?

- A palavra Arquitetura geralmente é usada no contexto de mais alto nível de algo, desassociado dos detalhes de baixo nível.
- Design geralmente aparece relacionado a estruturas e decisões de baixo nível.
- No entanto os dois conceitos aparecem juntos na construção de um sistema. Detalhes de baixo nível e estruturas de alto nível fazem parte de um todo.

# Objetivo da Arquitetura

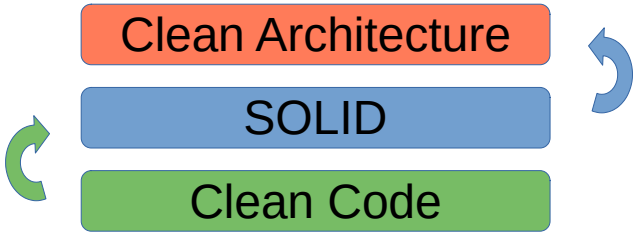
- O objetivo da Arquitetura de Software é minimizar os recursos humanos necessários para construir e manter um sistema.
- Para construir um sistema com um design e uma arquitetura que minimize esforços e maximize produtividade, você precisa saber quais atributos da arquitetura de sistemas levam a este fim.

# Valores do software

- Todo sistema de software fornece dois valores diferentes aos seus stakeholder:
  - Comportamento: programadores são contratados para fazer máquinas se comportarem de forma a fazer ou economizar dinheiro para os stakeholders.
  - Estrutura: o software precisa ser fácil de ser alterado. A dificuldade para se fazer uma mudança deve ser proporcional apenas ao escopo da mudança, não ao “formato” (shape) da mudança.

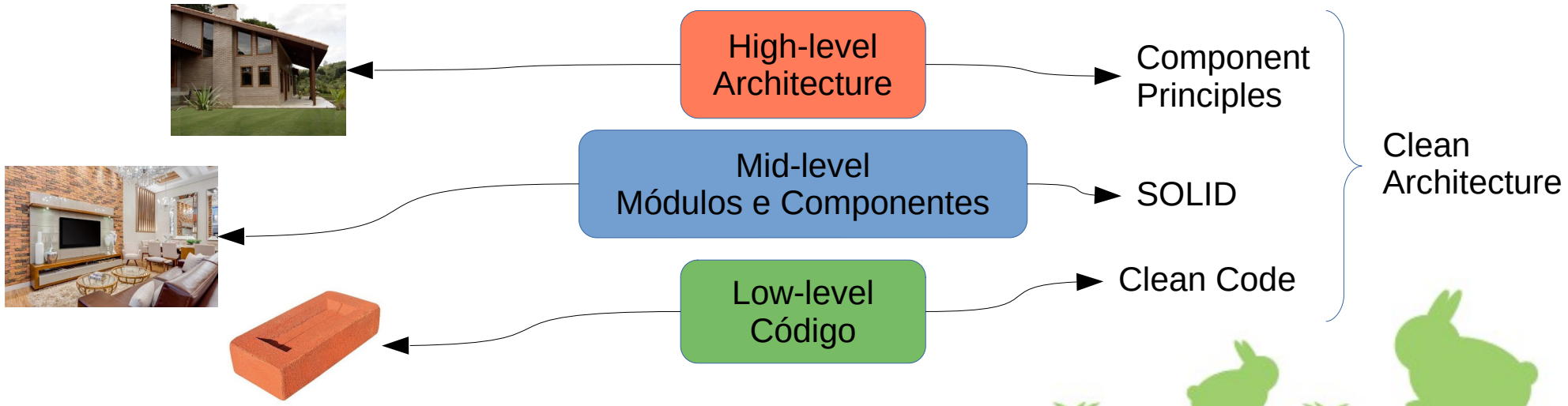
# SOLID

- Bom software começa com Clean Code.
- Entenda o Clean Code como os tijolos de construção que temos as mãos para fazer software. Se os tijolos não estão bem feitos, a arquitetura da construção não importa muito. Por outro lado, ainda podemos fazer um grande bagunça com bons tijolos. É neste ponto que entram os princípios do SOLID.
- Os princípios do SOLID nos dizem como organizar nossas funções e dados em classes e como essas classes deveriam estar interconectadas.



# Objetivo do SOLID

- O objetivo dos princípios é a criação de estrutura de software a nível de módulos e componentes que:
  - Toleram mudanças
  - São fáceis de entender
  - São a base de componentes que podem ser usados em muitos sistemas de software.



# Single Responsability Principle (SRP)

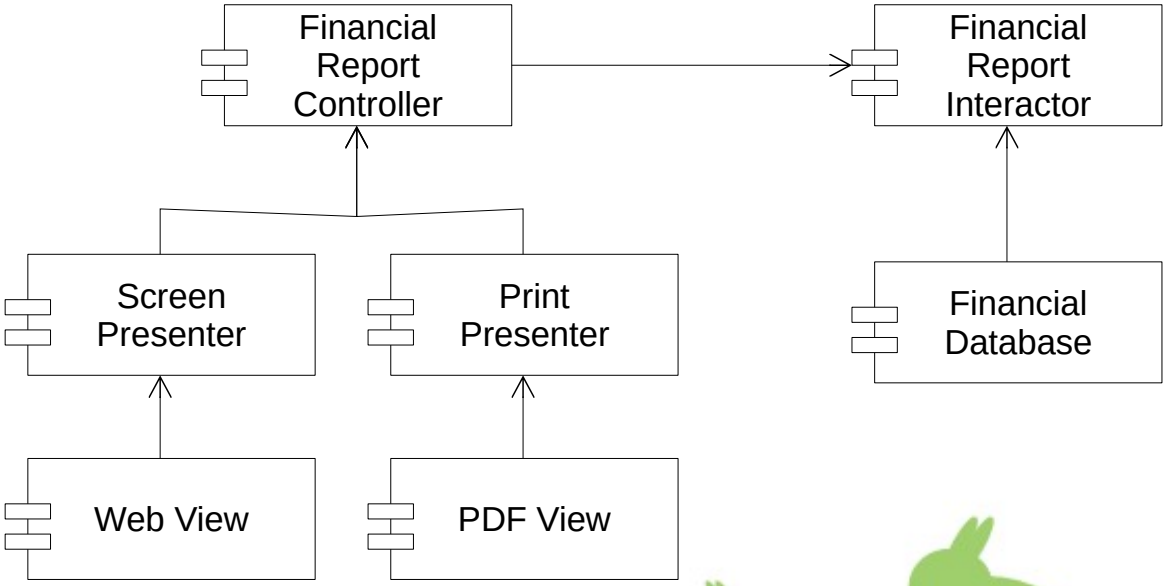
- Lembre-se que o SOLID se aplica a nível de módulos e componentes.
- O módulo deve ter uma, e apenas uma, razão para mudar.
- Sistemas de software são alterados para satisfazer usuários e stakeholders. Estes são a razão para mudar a que o princípio se refere. Podemos reescrever o princípios assim:
  - Um módulo deve ser responsável por um, e apenas um, ator.
- Módulo aqui é entendido como arquivo-fonte (source file)

# Open-Closed Principle

- Um artefato de software deve ser aberto para extensão mas fechado para modificação. Ou seja, o comportamento de um artefato de software deve ser extensível, sem que seja necessário modificar o artefato.

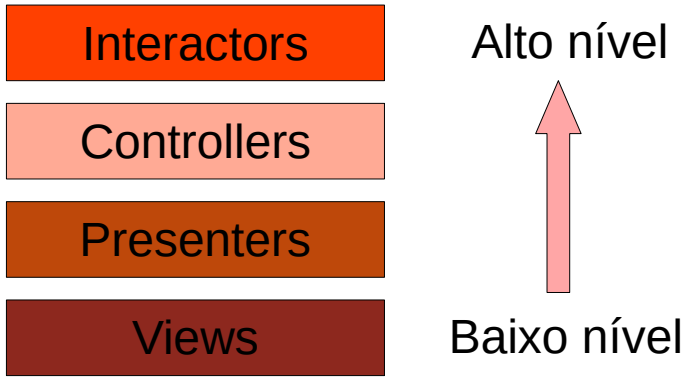
Se o componente A deve estar protegido de mudanças feitas no componente B, então o componente B deve depender do componente A.

Queremos proteger o Controller de mudanças nos Presenters. Queremos proteger os Presenters de mudanças nas Views. Queremos proteger o Interactor de mudanças em qualquer lugar.



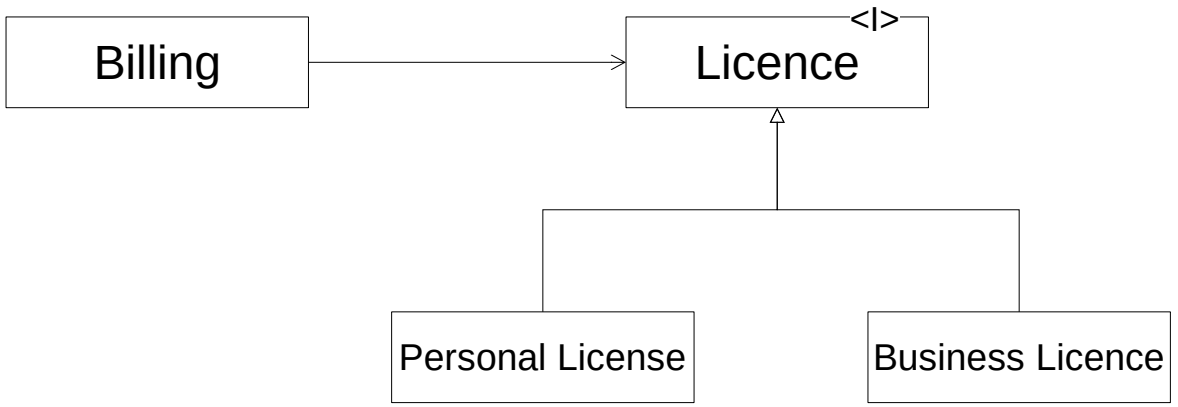
# Open-Closed Principle

- O OCP é uma das forças de direcionamento por traz da arquitetura de sistemas. O Objetivo é tornar o sistema fácil de estender sem que a mudança incorra em um alto impacto. Este objetivo é alcançado particionando o sistema em componentes e os organizando em hierarquias de dependências que protegem os componentes de alto nível de mudanças em componente de baixo nível.



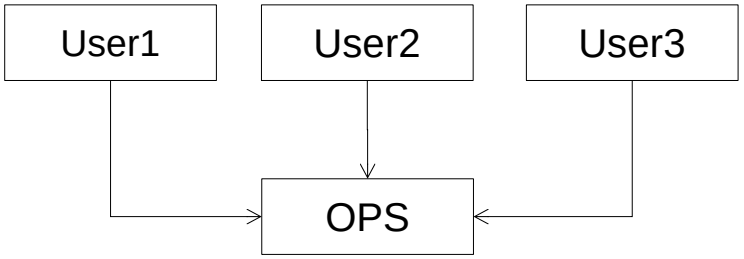
# Liskov Substitution Principle

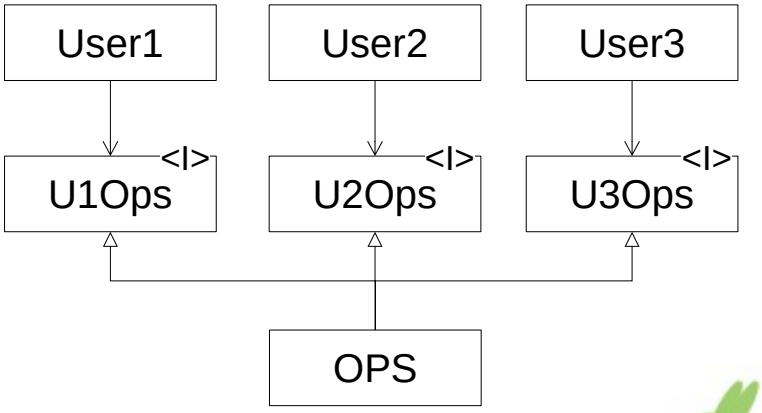
- Os usuários de um módulo/componente deve depender de interfaces bem definidas e na capacidade de substituição da implementação destas interfaces.



# Interface Segregation Principle

- Na figura 1, mesmo que User1 não use todos os métodos de OPS, ele será afetado caso OPS seja alterado. Uma nova compilação e um novo deploy precisa ser feito.
- Na figura 2, uma mudança em OPS não afeta o User1, pois este depende apenas dos métodos que necessita explicitados no contrato U1Ops. Neste caso você pode trocar o componente OPS que o User1 continua valendo.
- A lição aqui é depender de alguma coisa que carrega agregados que você não precisa pode causar problemas que você não espera.



# Dependency Inversion Principle

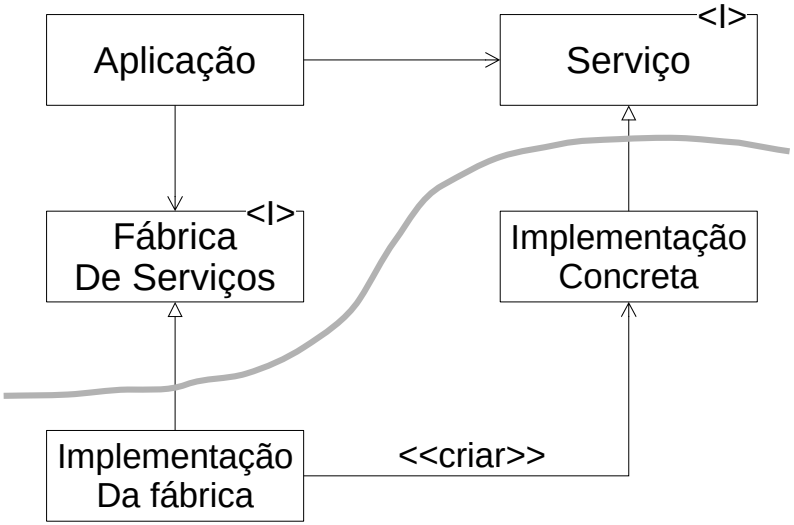
- O princípio nos diz que os sistemas mais flexíveis são aqueles em que as dependências do código fonte referem-se apenas a abstrações, não a implementações.
- Em linguagens estaticamente tipadas, isso significa que as declarações “using” deve ser referir apenas a módulos contendo interfaces e classes abstratas. Não deve haver dependência para nada concreto.
- Interfaces são menos voláteis que implementações. Bons designs de software e arquitetos trabalham duro para reduzir a volatilidade das interfaces.

# Dependency Inversion Principle

- Arquiteturas de software estáveis são aquelas que evitam depender de artefatos concretos voláteis em favor de usar interfaces abstratas estáveis.
  - Não referencie classes voláteis concretas, referencie interfaces abstratas.
  - Não derive de classes voláteis concretas. Em linguagens estaticamente tipadas a herança é a mais forte e mais rígida de todos os relacionamentos, use com bastante cuidado.
  - Não sobrescreva funções concretas. Funções concretas geralmente requerem dependência de código fonte. Quando são sobrescritas essas dependências não desaparecem, são herdadas. Para gerenciar essas dependências você deve fazer a dependência abstrata e criar múltiplas implementações.

# Factories

- A criação de um objeto em qualquer linguagem requer uma dependência de código fonte na definição concreta deste objeto.
- Abstract Factory são usadas para gerenciar essas dependências indesejadas.




A linha curva divide o sistema em dois componentes: um abstrato e outro concreto. Os componentes abstratos contém todas as regra de negócio de alto nível da aplicação. Os componentes concretos contém todas os detalhes de implementação que as regras de negócio manipulam.

# Princípios por trás dos componentes

- Se o SOLID nos diz como organizar os tijolos em paredes e salas, os princípios relacionados aos componentes nos dizem como organizar as salas em prédios.
- Os componentes são as unidades implantáveis (deployment). São as menores entidades que podem ser implantadas como parte do sistema. Em C# estamos falando de DLLs.
- Com os princípios queremos responder quais classes pertencem a quais componentes. São 3 princípios que tratam da coesão dos componentes:
  - REP: The Reuse/Release Equivalence Principle
  - CCP: The Common Closure Principle
  - CRP: The Common Reuse Principle

# The Reuse/Release Equivalence Principle

- Pessoas que querem reutilizar componentes de software não podem, e não o farão, a menos que esses componentes sejam rastreados através de um processo de release e recebam números de release.
- Do ponto de vista de design e arquitetura de software, esse princípio significa que as classes e módulos que são formados em um componente devem pertencer a um grupo coeso. O componente não pode simplesmente consistir em uma mistura aleatória de classes e módulos; em vez disso, deve haver algum tema ou finalidade abrangente que todos esses módulos compartilhem.
- Classes e módulos agrupados em um componente devem ser liberados juntos. O fato de compartilharem o mesmo número de versão e o mesmo rastreamento de versão e serem incluídos na mesma documentação de versão deve fazer sentido tanto para o autor quanto para os usuários.

# The Common Closure Principle

- Reúna em componentes as classes que mudam pelos mesmos motivos e no mesmo momento. Separe em componentes diferentes as classes que mudam em momentos diferentes e por motivos diferentes.
- Este é o princípio de responsabilidade única, atualizado para os componentes. Assim como o SRP diz que uma classe não deve conter múltiplos motivos para mudar, o CCP diz que um componente não deve ter múltiplos motivos para mudar.
- O CCP pede que reunamos em um só lugar todas as classes que provavelmente mudarão pelos mesmos motivos. Se duas classes são tão fortemente ligadas, fisicamente ou conceitualmente, que sempre mudam juntas, elas pertencem ao mesmo componente. Isso minimiza a carga de trabalho relacionada à liberação, revalidação e reimplantação do software.
- O CCP é o SRP para componente. O SRP nos diz para separar métodos em classes diferentes, se eles mudarem por razões diferentes. O CCP nos diz para separar as classes em diferentes componentes, se elas mudarem por diferentes motivos. Ambos os princípios podem ser resumidos pela seguinte frase:
  - Reúna as coisas que mudam ao mesmo tempo e pelas mesmas razões. Separe as coisas que mudam em momentos diferentes ou por motivos diferentes.

# The Common Reuse Principle

- Não force os usuários de um componente a dependerem de coisas que eles não precisam.
- O CRP afirma que classes e módulos que tendem a ser reutilizados juntos pertencem ao mesmo componente.
- As classes raramente são reutilizadas isoladamente. Mais tipicamente, as classes reutilizáveis colaboram com outras classes que fazem parte da abstração reutilizável. O CRP afirma que essas classes pertencem juntas ao mesmo componente. Nesse componente, esperaríamos ver classes com muitas dependências uma da outra.
- O CRP nos diz mais do que apenas quais classes devem ser reunidas em um componente: Ele também nos diz quais classes não devem ser mantidas juntas em um componente. Quando um componente A usa outro componente B, uma dependência é criada entre A e B. Talvez o componente A use apenas uma classe dentro do componente B - mas isso ainda não enfraquece a dependência. O componente A ainda depende do componente B.
- Devido a essa dependência, toda vez que o componente B é alterado, o componente A provavelmente precisará de alterações correspondentes. Mesmo que não sejam necessárias alterações no componente A, ele provavelmente ainda precisará ser recompilado, revalidado e reimplantado.
- Assim, quando dependemos de um componente, queremos ter certeza de que dependemos de todas as classes desse componente. Em outras palavras, queremos garantir que as classes que colocamos em um componente sejam inseparáveis - que é impossível depender de alguns e não dos outros. Caso contrário, reimplantaremos mais componentes do que o necessário e desperdiçaremos um esforço significativo.
- Portanto, o CRP nos diz mais sobre quais classes não devem estar juntas do que sobre quais classes devem estar juntas. O CRP diz que as classes que não estão fortemente ligadas entre si não devem estar no mesmo componente.

# Acoplamento entre componentes

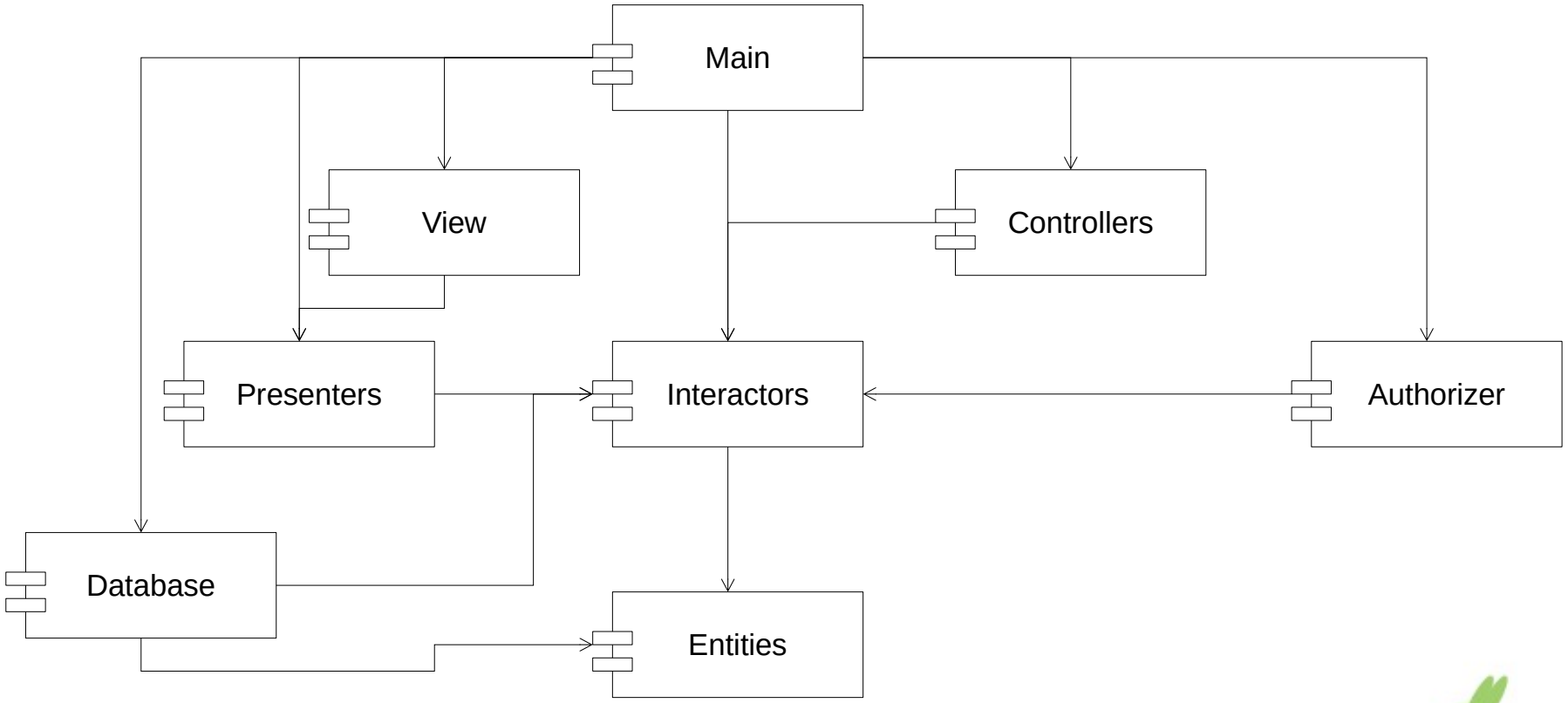
- 3 princípios tratam sobre o relacionamento entre componentes:
  - Acyclic Dependencies Principle
    - Não é permitido dependências cíclicas no grafo de dependência.
  - Stable Dependencies Principle
    - Dependa na direção da estabilidade.
  - Stable Abstractions Principle
    - Um componente deve ser tão abstrato como é estável.

# Acyclic Dependencies Principle

- A dependência entre classes faz com que um software que estava funcionando num dia, passe a não funcionar no outro dia (the morning after syndrome) simplesmente porque alguém alterou algo em que a sua classe dependia.
- Uma solução para o problema acima é particionar o ambiente de desenvolvimento em componentes versionáveis. O componente torna-se uma unidade de trabalho que passa a ser responsabilidade de um desenvolvedor ou de um time. Quando um componente torna-se funcional ele é versionado e entregue à equipe.
- Quando uma nova versão é entregue, os times clientes do componente podem decidir se adotam ou não a nova versão. Assim, nenhum time está a merce do outro. Mudanças em um componente não precisam ter um efeito imediato no outro.
- No entanto, para fazer isso funcionar, é preciso gerenciar a estrutura de dependência entre os componentes. Não pode haver ciclos, se houver a síndrome da manhã seguinte não pode ser evitada.

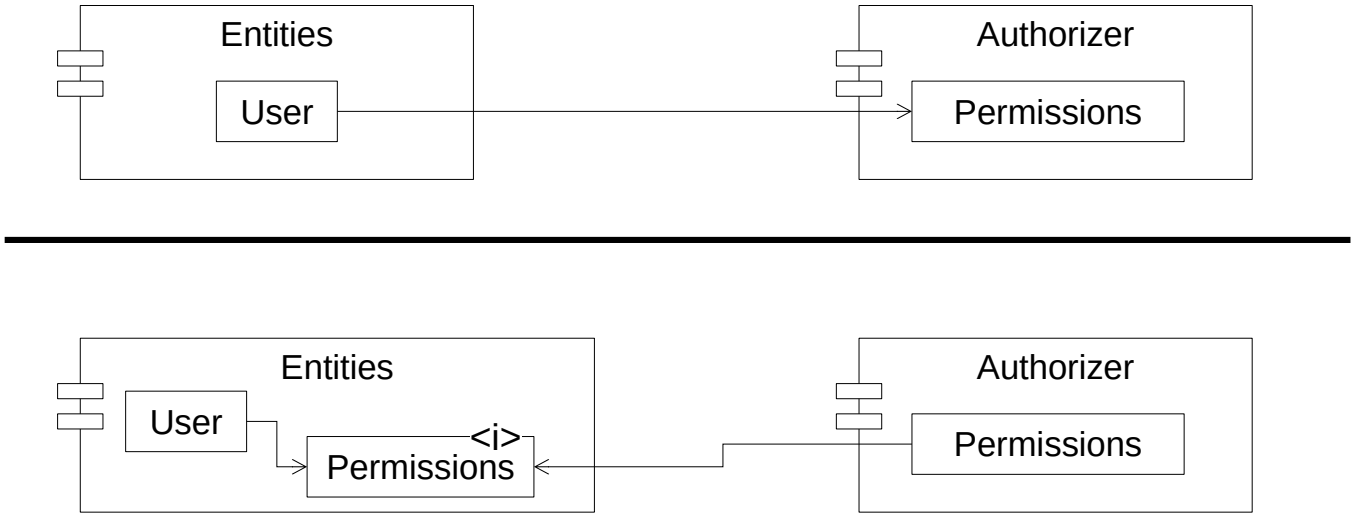
# Acyclic Dependencies Principle

- Um grafo de componentes sem ciclos.



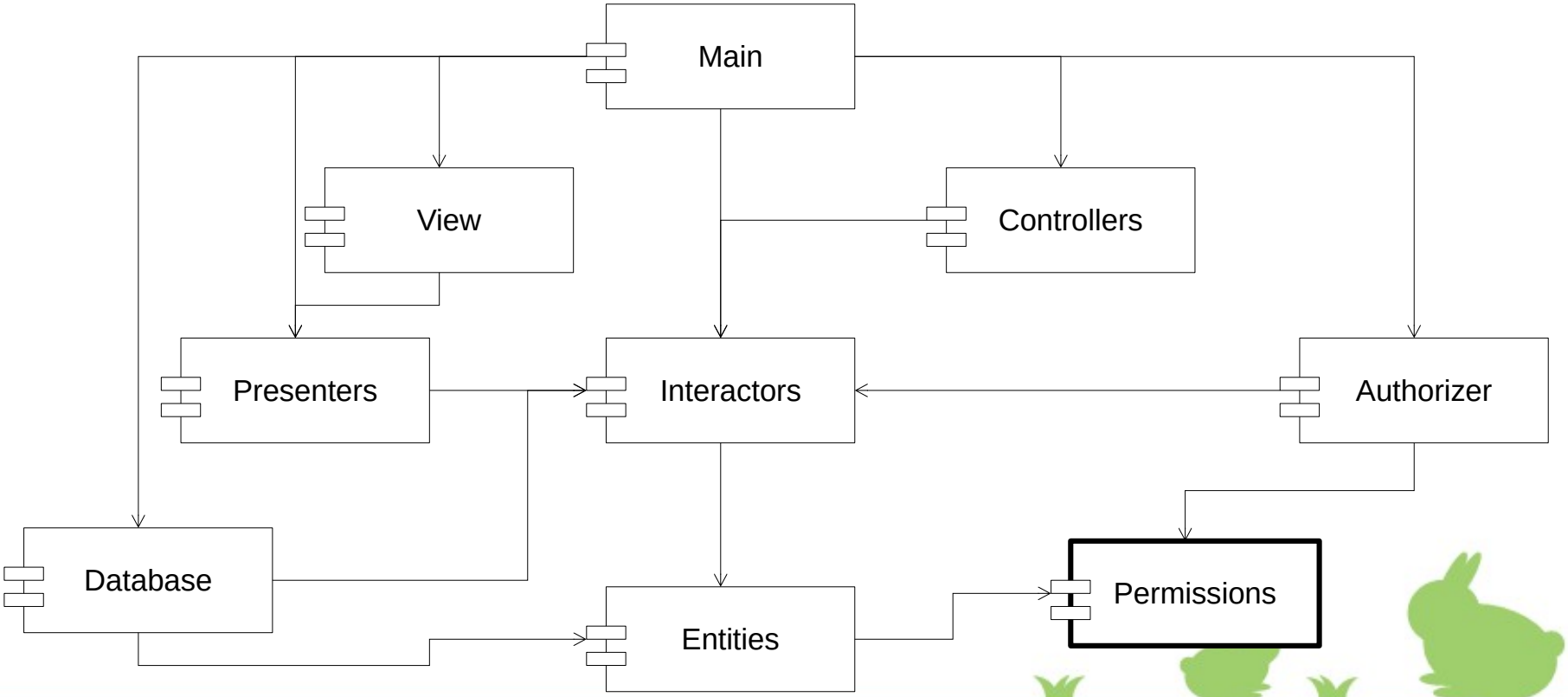
# Acyclic Dependencies Principle

- Quebrando os ciclos, primeiro mecanismo:
  - Aplique DIP. Crie uma interface que tem os métodos que você usa do componente que você dependia e faça este componente implementar esta interface. Isso inverte a dependência entre os dois componentes, quebrando o ciclo.



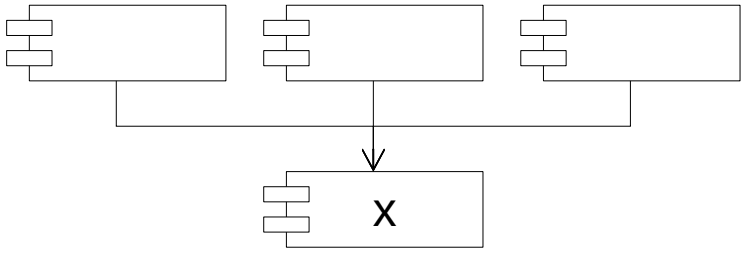
# Acyclic Dependencies Principle

- Quebrando os ciclos, segundo mecanismo:
  - Crie um novo componente que ambas os componentes irão depender. Mova as classes que estes dois componentes dependem para este novo componente.



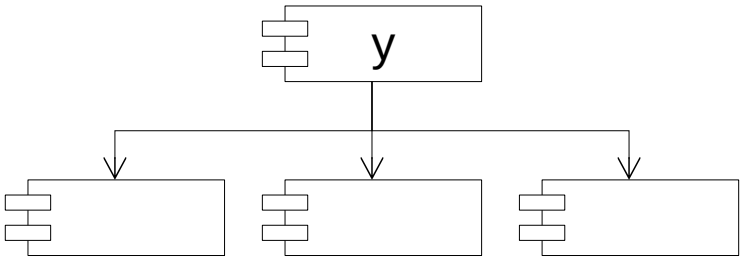
# Stable Dependencies Principle

- Conforme o CCP, criamos componentes que são sensíveis a certos tipos de mudança mas imunes a outros. Alguns componentes são projetados para serem voláteis, esperamos que eles sofram mudanças.
- Qualquer componentes esperado que sofra mudança não deve depender de um componente que seja difícil de mudar.
- Uma forma de tornar um componente de software difícil de mudar é fazer outros componentes de software depender dele.




X é um componente estável, 3 componentes dependem dele, então ele tem 3 bons motivos para não mudar. Dizemos que x é responsável por estes 3 componentes.

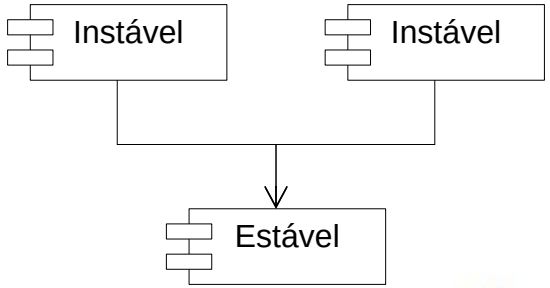
# Stable Dependencies Principle



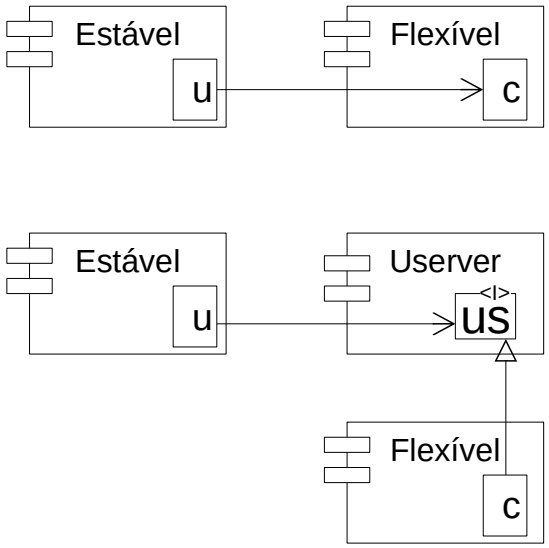

y é um componente instável, nenhum outro depende dele, porem ele depende de outros 3 componentes, mudanças podem vir de 3 fontes externas. Dizemos que Y é dependente.

- Se todos os componentes em um sistema fosse estáveis ao extremo o sistema seria inalterável. Essa não é uma situação desejável. Queremos projetar nossa estrutura de componentes em que alguns componentes sejam instáveis e outros sejam estáveis.

Numa situação ideia os componentes que esperamos que mudem devem estar no topo, dependendo de componentes estáveis na base. Colocar os componentes instáveis no topo do diagrama é uma convenção útil porque qualquer seta direcionada para cima viola o SDP.



# Stable Dependencies Principle




Precisamos de alguma forma quebra a dependência da classe estável para a classe flexível. Aplicando DIP, criamos uma interface chamada Userver e colocamos nela todos métodos que U necessita. Fazemos C implementar este interface. Isso quebra a dependência e faz com que os dois componentes dependam de Userver, que é uma classe estável. Todas as dependência agora seguem em direção da estabilidade.

# Stable Abstractions Principle

- Algum software no sistema não deve mudar frequentemente. Esses software representam arquitetura de alto nível e decisões políticas. Não queremos que esses decisões arquiteturais de negócio sejam voláteis. Esses software devem ser colocados em componentes estáveis.
- Componentes instáveis devem conter apenas software volátil – software que queiramos que sejam rápida e facilmente alteráveis.
- No entanto, se políticas de alto nível são colocadas em componentes estáveis, então o código que representa estas políticas será difícil de mudar. Isso pode fazer a arquitetura como um todo inflexível.
- Como fazer um componente maximamente estável ser flexível bastante para permitir mudanças? A resposta está no OCP. No caso do OCP, que está a nível de classes, classes abstratas se alinham ao OCP.
- O SAP estabelece uma relação entre estabilidade e abstração. Por um lado, diz que um componente estável também deve ser abstrato para que sua estabilidade não impeça que ele seja estendido. Por outro lado, diz que um componente instável deve ser concreto, pois sua instabilidade permite que o código concreto dentro dele seja facilmente alterado.
- Portanto, se um componente deve ser estável, ele deve consistir em interfaces e classes abstratas para que possa ser estendido. Componentes estáveis que são extensíveis são flexíveis e não restringem demais a arquitetura.

# Stable Abstractions Principle

- O SAP e o SDP combinados equivalem ao DIP dos componentes. Isso é verdade porque o SDP diz que as dependências devem correr na direção da estabilidade e o SAP diz que a estabilidade implica abstração. Assim, as dependências correm na direção da abstração.
- O DIP, no entanto, é um princípio que lida com classes - e com classes não há tons de cinza. Uma classe é abstrata ou não é. A combinação do SDP e do SAP lida com componentes e permite que um componente possa ser parcialmente abstrato e parcialmente estável.

# O que é arquitetura

- A arquitetura de um sistema de software é a forma dada a esse sistema por quem o constrói. A relevância nesta forma está na divisão desse sistema em componentes, na organização desses componentes e nas maneiras como esses componentes se comunicam.
- O objetivo desse formato é facilitar o desenvolvimento, implantação, operação e manutenção do sistema de software contido nele.
- A estratégia por trás dessa facilitação é deixar tantas opções em aberto quanto possível, pelo maior tempo possível.
- O principal objetivo da arquitetura é oferecer suporte ao ciclo de vida do sistema. A boa arquitetura torna o sistema fácil de entender, fácil de desenvolver, fácil de manter e fácil de implantar. O objetivo final é minimizar o custo da vida útil do sistema e maximizar a produtividade do programador.

# Mantenha as opções abertas

- O software foi inventado porque precisávamos de uma maneira rápida e fácil de alterar o comportamento das máquinas. Mas essa flexibilidade depende criticamente da forma do sistema, do arranjo de seus componentes e da maneira como esses componentes estão interconectados.
- Todos os sistemas de software podem ser decompostos em dois elementos principais: política e detalhes. O elemento da política incorpora todas as regras e procedimentos de negócios.
- A política é onde reside o verdadeiro valor do sistema. Os detalhes são as coisas necessárias para permitir que humanos, outros sistemas e programadores se comuniquem com a política, mas que não afetam o comportamento dela. Eles incluem dispositivos de I/O, bancos de dados, sistemas web, servidores, frameworks, protocolos de comunicação e assim por diante.
- O objetivo do arquiteto é criar uma forma para o sistema que reconheça a política como o elemento mais essencial do sistema e, ao mesmo tempo, torne os detalhes irrelevantes para essa política. Isso permite que as decisões sobre esses detalhes sejam adiadas e adiadas.

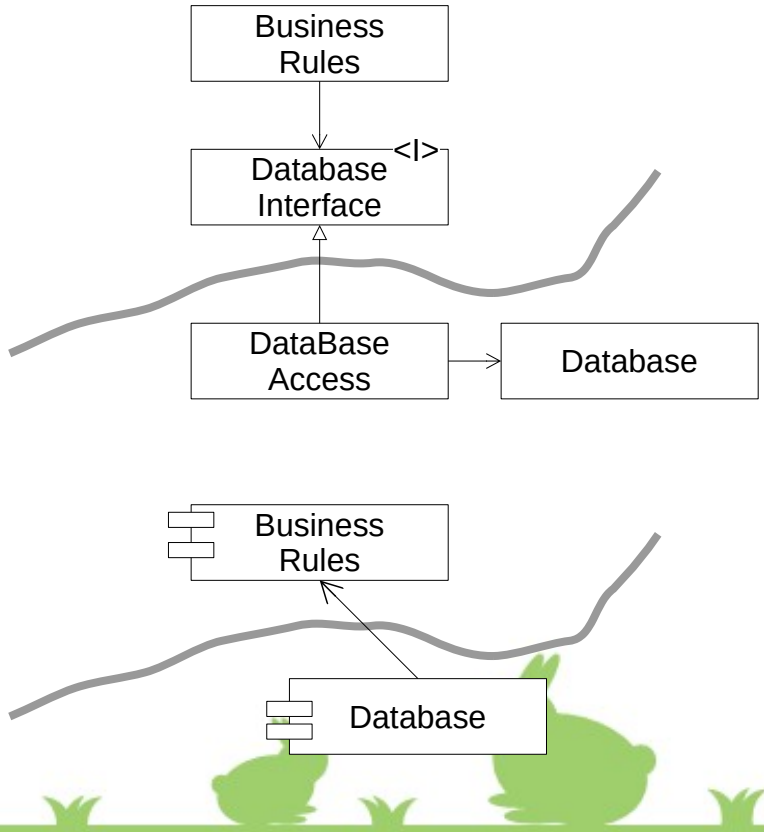
# Fronteiras (boundaries)

- Arquitetura de software é a arte de desenhar linhas que eu chamo de limites. Esses limites separam os elementos de software um do outro e restringem aqueles de um lado a saber sobre os do outro. Algumas dessas linhas são traçadas muito cedo na vida de um projeto - mesmo antes de qualquer código ser escrito. Outros são traçados muito mais tarde. Aqueles que são traçados com antecedência tem o objetivo de adiar decisões pelo maior tempo possível e impedir que essas decisões poluam a lógica de negócios principal.
- Que tipos de decisões são prematuras? Decisões que nada têm a ver com os requisitos de negócios - os casos de uso - do sistema. Isso inclui decisões sobre frameworks, bancos de dados, servidores da web, bibliotecas de utilitários, injeção de dependência e similares. Uma boa arquitetura do sistema não depende dessas decisões. Uma boa arquitetura do sistema permite que essas decisões sejam tomadas no momento mais tardio possível, sem impacto significativo.
- Você desenha linhas entre coisas que importam e coisas que não importam. A GUI não importa para as regras de negócios; portanto, deve haver uma linha entre elas. O banco de dados não importa para a GUI; portanto, deve haver uma linha entre eles. O banco de dados não importa para as regras de negócios; portanto, deve haver uma linha entre elas.

# Fronteiras (boundaries)

- As classes e interfaces neste diagrama são simbólicas. Em um aplicativo real, haveria muitas classes de regras de negócios, muitas classes de interface de banco de dados e muitas implementações de acesso ao banco de dados. Todos eles, porém, seguiriam aproximadamente o mesmo padrão.

  - Observe as duas setas saindo da classe DatabaseAccess. Essas duas setas apontam para longe da classe DatabaseAccess. Isso significa que nenhuma dessas classes sabe que a classe DatabaseAccess existe.
  - Agora vamos recuar um pouco. Veremos o componente que contém muitas regras de negócios e o componente que contém o banco de dados e toda a sua classe de acesso.
  - Observe a direção da seta. Database conhece BusinessRules. BusinessRules não conhecem Database. Isso implica que as classes DatabaseInterface residem no componente BusinessRules, enquanto as classes DatabaseAccess residem no componente Database.



# Anatomia da fronteiras

- A arquitetura de um sistema é definida por um conjunto de componentes de software e os limites que os separam. Esses limites vêm de muitas formas diferentes. Em tempo de execução, um cruzamento de limites nada mais é do que uma função de um lado do limite, chamando uma função do outro lado e transmitindo alguns dados.
- Monólito
  - O mais simples e o mais comum dos limites arquitetônicos não tem representação física estrita. É simplesmente uma segregação disciplinada de funções e dados em um único processador e um único espaço de endereço.
- Componentes implantáveis
  - A representação física mais simples de um limite de arquitetura é uma biblioteca vinculada dinamicamente como uma DLL em .Net. A implantação não envolve compilação. Em vez disso, os componentes são entregues em formato binário ou equivalente implementável. Este é o modo de desacoplamento no nível de implantação. O ato de implantação é simplesmente a reunião dessas unidades implementáveis de alguma forma conveniente, como um arquivo WAR ou apenas um diretório.
- Processos locais
  - Um limite arquitetural físico muito mais forte é o processo local. Um processo local é normalmente criado a partir da linha de comandos ou de uma chamada de sistema equivalente. Os processos locais são executados no mesmo processador ou no mesmo conjunto de processadores em um multicore, mas em espaços de endereço separados. A proteção de memória geralmente impede que esses processos compartilhem memória, embora as partições de memória compartilhada sejam frequentemente usadas.
  - Na maioria das vezes, os processos locais se comunicam usando soquetes ou algum outro tipo de recurso de comunicação do sistema operacional, como caixas de correio ou filas de mensagens.

# Anatomia da fronteiras

- Serviços
  - O limite mais forte é um serviço. Um serviço é um processo, geralmente iniciado na linha de comando ou através de uma chamada de sistema equivalente. Os serviços não dependem de sua localização física. Dois serviços de comunicação podem, ou não, operar no mesmo processador físico ou multicore. Os serviços pressupõem que todas as comunicações ocorram na rede.
  - As comunicações através dos limites do serviço são muito lentas se comparadas às chamadas de função. O tempo de resposta pode variar de dezenas de milissegundos a segundos. Deve-se tomar cuidado para evitar conversar sempre que possível. As comunicações nesse nível devem lidar com altos níveis de latência.
  - Caso contrário, as mesmas regras se aplicam aos serviços e aos processos locais. Serviços de nível inferior devem "conectar-se" a serviços de nível superior. O código-fonte dos serviços de nível superior não deve conter nenhum conhecimento físico específico (por exemplo, um URI) de qualquer serviço de nível inferior.
- A maioria dos sistemas, exceto monólitos, usa mais de uma estratégia de fronteira. Um sistema que utiliza limites de serviço também pode ter alguns limites de processo locais. De fato, um serviço geralmente é apenas uma fachada para um conjunto de processos locais em interação. Um serviço ou um processo local quase certamente será um monólito composto por componentes de código-fonte ou um conjunto de componentes de implantação vinculados dinamicamente. Isso significa que os limites de um sistema geralmente são uma mistura de limites de conversas locais e limites mais preocupados com a latência.

# Política a níveis

- Os sistemas de software são declarações de política. No fundo, isso é tudo que um programa de computador realmente é. Um programa de computador é uma descrição detalhada da política pela qual as entradas são transformadas em saídas.
- Na maioria dos sistemas não triviais, essa política pode ser dividida em várias declarações menores de política. Algumas dessas declarações descreverão como regras de negócios específicas devem ser calculadas. Outros descreverão como determinados relatórios devem ser formatados. Outros ainda descreverão como os dados de entrada devem ser validados.
- Parte da arte de desenvolver uma arquitetura de software é separar cuidadosamente essas políticas umas das outras e reagrupá-las com base nas maneiras que elas mudam. Políticas que são alteradas pelos mesmos motivos e, ao mesmo tempo, estão no mesmo nível e pertencem ao mesmo componente. Políticas que mudam por diferentes motivos ou em momentos diferentes, estão em níveis diferentes e devem ser separadas em componentes diferentes.
- A arte da arquitetura geralmente envolve a transformação dos componentes reagrupados em um gráfico acíclico direcionado. Os nós do gráfico são os componentes que contêm políticas no mesmo nível. As arestas direcionadas são as dependências entre esses componentes. Eles conectam componentes que estão em diferentes níveis.
- Em uma boa arquitetura, a direção dessas dependências é baseada no nível dos componentes que eles se conectam. Em todos os casos, os componentes de baixo nível são projetados para que dependam dos componentes de alto nível.

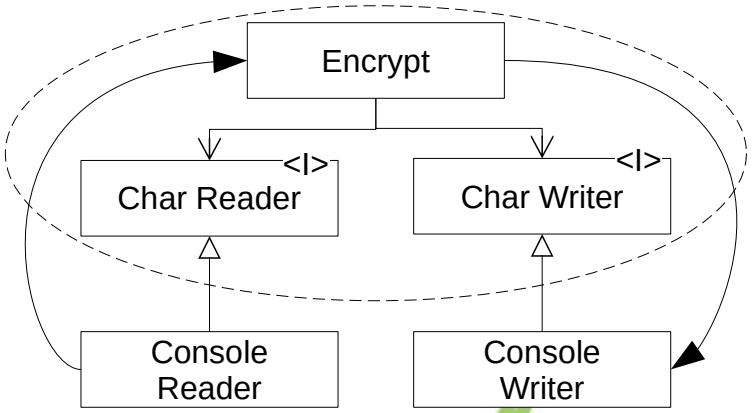
# Política a níveis

- Uma definição direta de "nível" é "a distância das entradas e saídas". Quanto mais longe a política estiver das entradas e saídas do sistema, maior será o seu nível. As políticas que gerenciam entrada e saída são as políticas de nível mais baixo do sistema.

- O diagrama de fluxo de dados mostra um programa de criptografia simples que lê caracteres de um dispositivo de entrada, converte os caracteres usando uma tabela e depois grava os caracteres traduzidos em um dispositivo de saída. Os fluxos de dados são mostrados como setas sólidas curvas. As dependências do código fonte projetadas corretamente são mostradas como linhas tracejadas retas.

- Observe que os fluxos de dados e as dependências do código-fonte nem sempre apontam na mesma direção. Isso, novamente, faz parte da arte da arquitetura de software. Queremos que as dependências do código-fonte sejam dissociadas do fluxo de dados e acopladas ao nível.

- Borda tracejada ao redor da classe Encrypt e as interfaces CharWriter e CharReader indicam que esta unidade é o elemento de nível mais alto do sistema. Todas as dependências cruzam essa fronteira para dentro.
- Observe como essa estrutura desacopla a política de criptografia de alto nível das políticas de entrada / saída de nível inferior. Isso torna a política de criptografia utilizável em uma ampla variedade de contextos. Quando são feitas alterações nas políticas de entrada e saída, elas provavelmente não afetam a política de criptografia.
- Lembre-se de que as políticas são agrupadas em componentes com base na maneira como elas mudam. Políticas que mudam pelos mesmos motivos e ao mesmo tempo são agrupadas pelo SRP e pelo CCP. Políticas de nível superior - aquelas que estão mais distantes das entradas e saídas - tendem a mudar com menos frequência, e por razões mais importantes, do que as políticas de nível inferior. As políticas de nível inferior - as mais próximas das entradas e saídas - tendem a mudar com frequência e com mais urgência, mas por razões menos importantes.



# O tema de uma arquitetura

- Assim como as plantas de uma casa ou biblioteca gritam sobre os casos de uso desses edifícios, a arquitetura de um aplicativo de software deve gritar sobre os casos de uso do aplicativo.
- As arquiteturas não são (ou não deveriam ser) sobre frameworks. As arquiteturas não devem ser fornecidas por frameworks. Frameworks são ferramentas a serem usadas, não arquiteturas a serem conformes. Se sua arquitetura é baseada em frameworks, não pode ser baseada em seus casos de uso.
- Boas arquiteturas são centradas em casos de uso, para que os arquitetos possam descrever com segurança as estruturas que suportam esses casos de uso sem se comprometer com frameworks, ferramentas e ambientes. Mais uma vez, considere os planos para uma casa. A primeira preocupação do arquiteto é garantir que a casa seja utilizável - não garantir que a casa seja feita de tijolos. De fato, o arquiteto se esforça para garantir que o proprietário possa tomar decisões sobre o material externo (tijolos, pedra ou cedro) posteriormente, depois que os planos garantirem que os casos de uso sejam atendidos.

# Clean Architecture

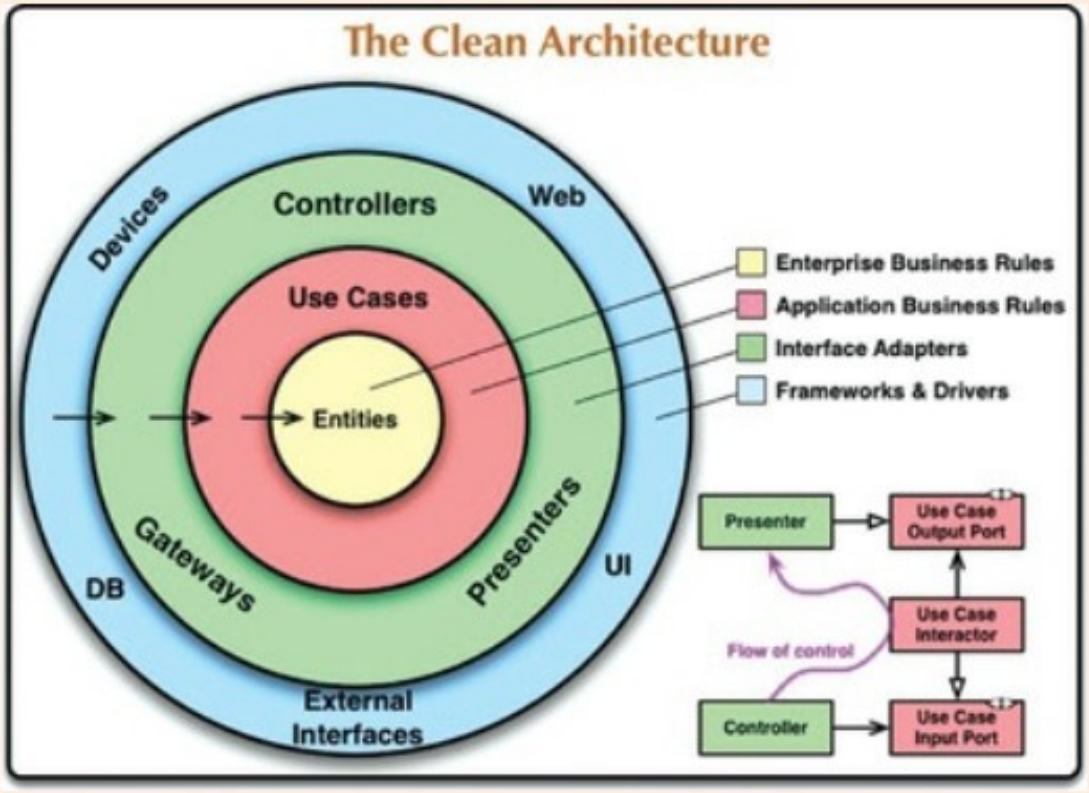



**Figure 22.1** The clean architecture

# Clean Architecture

- Os círculos concêntricos representam diferentes áreas do software. Em geral, quanto mais você avança, mais alto o nível do software. Os círculos externos são mecanismos. Os círculos internos são políticas.
- A regra de substituição que faz essa arquitetura funcionar é a Regra de Dependência:
  - As dependências do código-fonte devem apontar apenas para dentro, em direção a políticas de nível superior.
- Nada em um círculo interno pode saber algo sobre algo em um círculo externo. Em particular, o nome de algo declarado em um círculo externo não deve ser mencionado pelo código em um círculo interno. Isso inclui funções, classes, variáveis ou qualquer outra entidade de software nomeada.

# Entidades

- As entidades encapsulam as regras gerais críticas da empresa. Uma entidade pode ser um objeto com métodos ou um conjunto de estruturas e funções de dados. Não importa, desde que as entidades possam ser usadas por muitos aplicativos diferentes na empresa.
- Se você não possui uma empresa e está escrevendo apenas um único aplicativo, essas entidades são os objetos de negócios do aplicativo. Eles encapsulam as regras mais gerais e de alto nível. Eles são os menos propensos a mudar quando algo externo muda. Por exemplo, você não espera que esses objetos sejam afetados por uma alteração na navegação ou segurança da página. Nenhuma mudança operacional em qualquer aplicativo específico deve afetar a camada da entidade.

# Casos de uso

- O software na camada de casos de uso contém regras de negócios específicas do aplicativo. Ele encapsula e implementa todos os casos de uso do sistema. Esses casos de uso orquestram o fluxo de dados de e para as entidades e direcionam essas entidades a usarem suas Regras críticas de negócios para atingir os objetivos do caso de uso.
- Não esperamos que mudanças nesta camada afetem as entidades. Também não esperamos que essa camada seja afetada por alterações nas externalidades, como banco de dados, interface do usuário ou qualquer uma das estruturas comuns. A camada de casos de uso é isolada de tais preocupações.
- Entretanto, esperamos que alterações na operação do aplicativo afetem os casos de uso e, portanto, o software nessa camada. Se os detalhes de um caso de uso mudarem, algum código nesta camada certamente será afetado.

# Interface Adapters

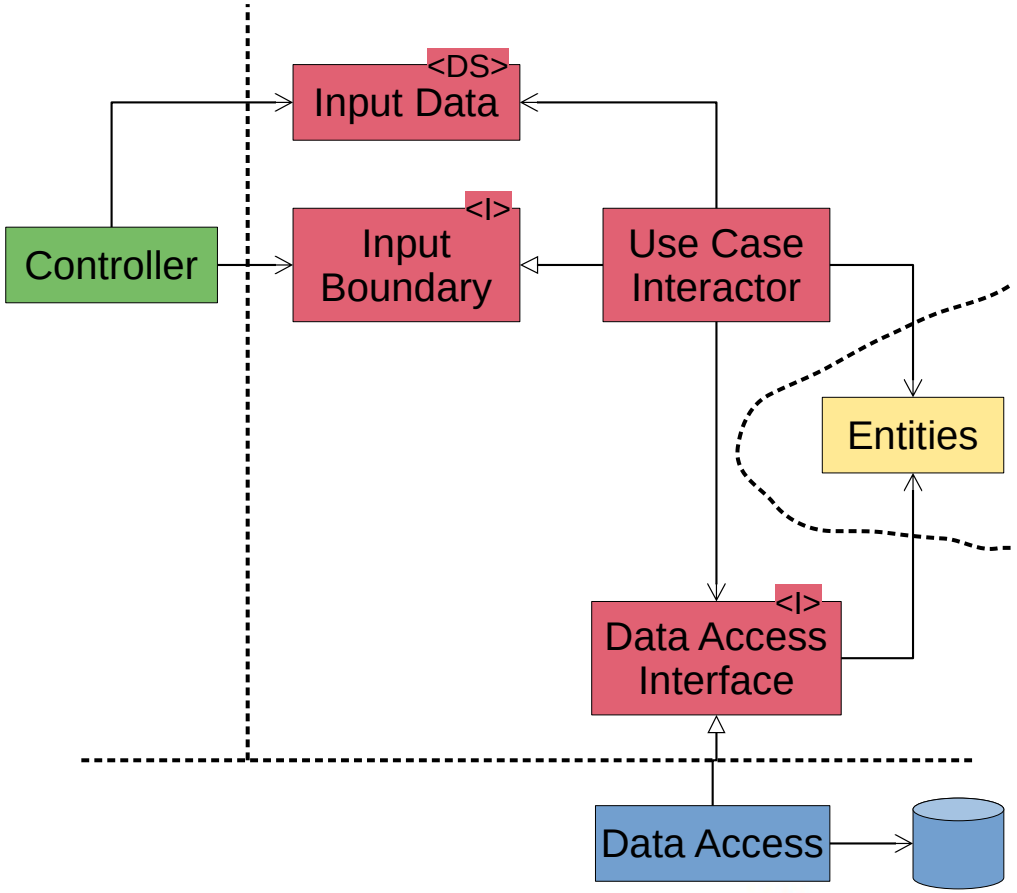
- O software na camada de adaptadores de interface é um conjunto de adaptadores que convertem dados do formato mais conveniente para os casos de uso e entidades, para o formato mais conveniente para alguma agência externa, como o banco de dados ou a Web. É essa camada, por exemplo, que conterá totalmente a arquitetura MVC de uma GUI. Todos os presenters, views e controllers pertencem à camada de adaptadores de interface. Os modelos provavelmente são apenas estruturas de dados que são passadas dos controllers para os casos de uso e, em seguida, voltam dos casos de uso para os presenters e views.
- Da mesma forma, os dados são convertidos, nesta camada, do formato mais conveniente para entidades e casos de uso, para o formato mais conveniente para qualquer estrutura de persistência que esteja sendo usada (ou seja, o banco de dados). Nenhum código dentro deste círculo deve saber nada sobre o banco de dados. Se o banco de dados for um banco de dados SQL, todo o SQL deverá ser restrito a essa camada - e, em particular, às partes dessa camada relacionadas ao banco de dados.
- Também nesta camada está qualquer outro adaptador necessário para converter dados de algum formato externo, como um serviço externo, para o formulário interno usado pelos casos de uso e entidades.

# Frameworks e Drivers

- A camada mais externa do modelo é geralmente composta de frameworks e ferramentas como o banco de dados e a frameworks web. Geralmente, você não escreve muito código nessa camada, exceto o código de cola que se comunica com o próximo círculo para dentro.
- A camada de frameworks e drivers é para onde vão todos os detalhes. A web é um detalhe. O banco de dados é um detalhe. Mantemos essas coisas do lado de fora, onde elas podem causar pouco dano.

# Um cenário típico

The diagram shows a typical scenario for a web-based Java system using a database. The web server gathers input data from the user and hands it to the Controller on the upper left. The Controller packages that data into a POJO and passes this object through the InputBoundary to the UseCaseInteractor. The UseCaseInteractor interprets that data and uses it to control the dance of the Entities. It also uses the DataAccessInterface to bring the data used by those Entities into memory from the Database.



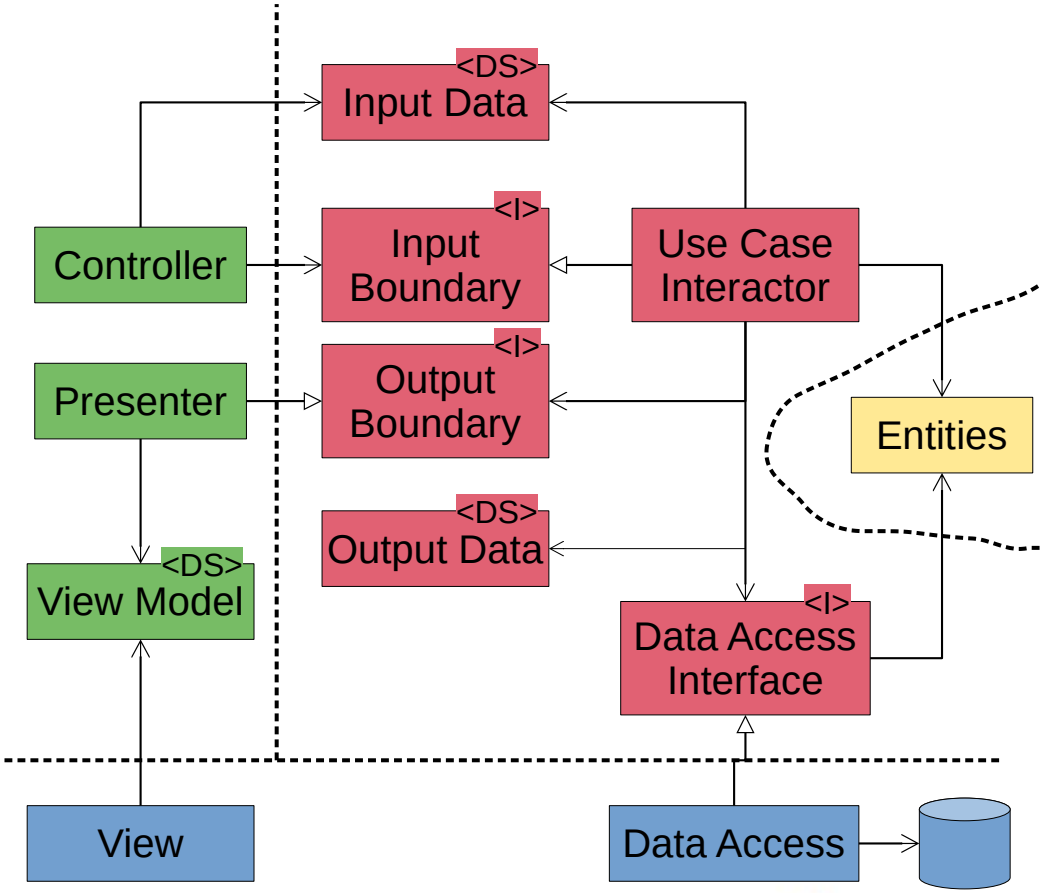
# Um cenário típico

Upon completion, the UseCaseInteractor gathers data from the Entities and constructs the OutputData as another POJO. The OutputData is then passed through the OutputBoundary interface to the Presenter. The job of the Presenter is to repackage the OutputData into viewable form as the ViewModel, which is yet another POJO. The ViewModel contains mostly Strings and flags that the View uses to display the data.

Whereas the OutputData may contain Date objects, the Presenter will load the ViewModel with corresponding Strings already formatted properly for the user. The same is true of Currency objects or any other business-related data. Button and MenuItem names are placed in the ViewModel, as are flags that tell the View whether those Buttons and MenuItems should be gray.

This leaves the View with almost nothing to do other than to move the data from the ViewModel into the HTML page.

Note the directions of the dependencies. All dependencies cross the boundary lines pointing inward, following the Dependency Rule.



# Presenters e Humble Objects

- O padrão Humble Object é um padrão de design que foi originalmente identificado como uma maneira de ajudar os testadores de unidade a separar comportamentos difíceis de testar e comportamentos fáceis de testar. A ideia é muito simples: divida os comportamentos em dois módulos ou classes. Um desses módulos é humilde (humble); contém todos os comportamentos difíceis de testar, despojados de sua essência mais simples. O outro módulo contém todos os comportamentos testáveis que foram retirados do objeto humilde.
- A View é o objeto humilde que é difícil de testar. O código neste objeto é mantido o mais simples possível. Ele move os dados para a GUI, mas não os processa.
- O Presenter é o objeto testável. Seu trabalho é aceitar dados do aplicativo e formatá-los para apresentação, para que o View possa simplesmente movê-los para a tela. Por exemplo, se o aplicativo quiser que uma data seja exibida em um campo, ele entregará ao Presenter um objeto Data. O Presenter formatará esses dados em uma sequência apropriada e os colocará em uma estrutura de dados simples chamada View Model, onde a View pode encontrá-los.
- Tudo o que aparece na tela e o aplicativo tem algum tipo de controle é representado no modelo de exibição como uma string, um booleano ou uma enumeração. Nada resta para o View fazer outra coisa senão carregar os dados do View Model na tela. Assim, a visão é humilde.

# Database Gateways

- Entre os Use Case interactors e o banco de dados, estão os gateways do banco de dados. Esses gateways são interfaces polimórficas que contêm métodos para todas as operações de criação, leitura, atualização ou exclusão que podem ser executadas pelo aplicativo no banco de dados. Por exemplo, se o aplicativo precisar conhecer os sobrenomes de todos os usuários que efetuaram login ontem, a interface UserGateway terá um método chamado getLastNamesOfUsersWhoLoggedInAfter que usa Date como argumento e retorna uma lista de sobrenomes.
- Lembre-se de que não permitimos SQL na camada de casos de uso; em vez disso, usamos interfaces de gateway que possuem métodos apropriados. Esses gateways são implementados por classes na camada de banco de dados. Essa implementação é o objeto humilde. Ele simplesmente usa SQL, ou qualquer que seja a interface do banco de dados, para acessar os dados exigidos por cada um dos métodos. Os interactors, por outro lado, não são humildes porque encapsulam regras de negócios específicas de aplicativos. Embora não sejam humildes, esses interactors são testáveis, porque os gateways podem ser substituídos por stubs e test-doubles.

# Data Mappers

- Voltando ao tópico dos bancos de dados, em qual camada você acha que os ORMs como o Hibernate pertencem?
- Primeiro, vamos esclarecer uma coisa: não existe um objeto mapeador relacional (ORM). O motivo é simples: os objetos não são estruturas de dados. Pelo menos, eles não são estruturas de dados do ponto de vista de seus usuários. Os usuários de um objeto não podem ver os dados, pois são todos privados. Esses usuários veem apenas os métodos públicos desse objeto. Portanto, do ponto de vista do usuário, um objeto é simplesmente um conjunto de operações.
- Uma estrutura de dados, por outro lado, é um conjunto de variáveis de dados públicas que não têm comportamento implícito. Os ORMs seriam melhor denominados "mapeadores de dados", porque carregam dados nas estruturas de dados a partir de tabelas de bancos de dados relacionais.
- Onde esses sistemas ORM devem residir? Na camada de banco de dados, é claro. De fato, os ORMs formam outro tipo de limite do Humble Object entre as interfaces do gateway e o banco de dados.

# Service Listeners

- E os serviços? Se seu aplicativo precisar se comunicar com outros serviços, ou se ele fornecer um conjunto de serviços, encontraremos o padrão Humble Object criando um service boundary?
- Claro! O aplicativo carregará dados em estruturas de dados simples e depois passará essas estruturas através do limite para módulos que formatam corretamente os dados e os enviam para serviços externos. No lado da entrada, os ouvintes do serviço receberão dados da interface do serviço e os formatarão em uma estrutura de dados simples que pode ser usada pelo aplicativo. Essa estrutura de dados é então passada através do limite do serviço.

# The Main Component

- O componente Principal é o detalhe final - a política de nível mais baixo. É o ponto de entrada inicial do sistema. Nada além do sistema operacional depende disso. Seu trabalho é criar todas as fábricas, estratégias e outros facilitadores globais e, em seguida, entregar o controle para as partes abstratas de alto nível do sistema.
- É neste componente Principal que as dependências devem ser injetadas por um framework de Injeção de Dependências. Uma vez injetadas no Main, o Main deve distribuir essas dependências normalmente, sem usar a estrutura.
- Pense em Main como o mais sujo de todos os componentes sujos.

# Services

- Primeiro, vamos considerar a noção de que o uso de serviços, por natureza, é uma arquitetura. Isso é evidentemente falso. A arquitetura de um sistema é definida por limites que separam a política de alto nível dos detalhes de baixo nível e seguem a Regra de Dependência. Os serviços que simplesmente separam os comportamentos dos aplicativos são pouco mais do que chamadas de função caras e não são necessariamente importantes em termos de arquitetura.
- Isso não quer dizer que todos os serviços sejam arquitetonicamente significativos. Muitas vezes, existem benefícios substanciais na criação de serviços que separam a funcionalidade entre processos e plataformas - obedecendo ou não à regra de dependência. Os serviços, por si só, não definem uma arquitetura.
- Uma analogia útil é a organização das funções. A arquitetura de um sistema monolítico ou baseado em componente é definida por determinadas chamadas de função que cruzam os limites da arquitetura e seguem a Regra de Dependência. Muitas outras funções nesses sistemas, no entanto, simplesmente separam um comportamento do outro e não são arquitetonicamente significativas.
- O mesmo acontece com os serviços. Afinal, os serviços são apenas chamadas de função através dos limites do processo e / ou da plataforma. Alguns desses serviços são arquitetonicamente significativos e outros não.