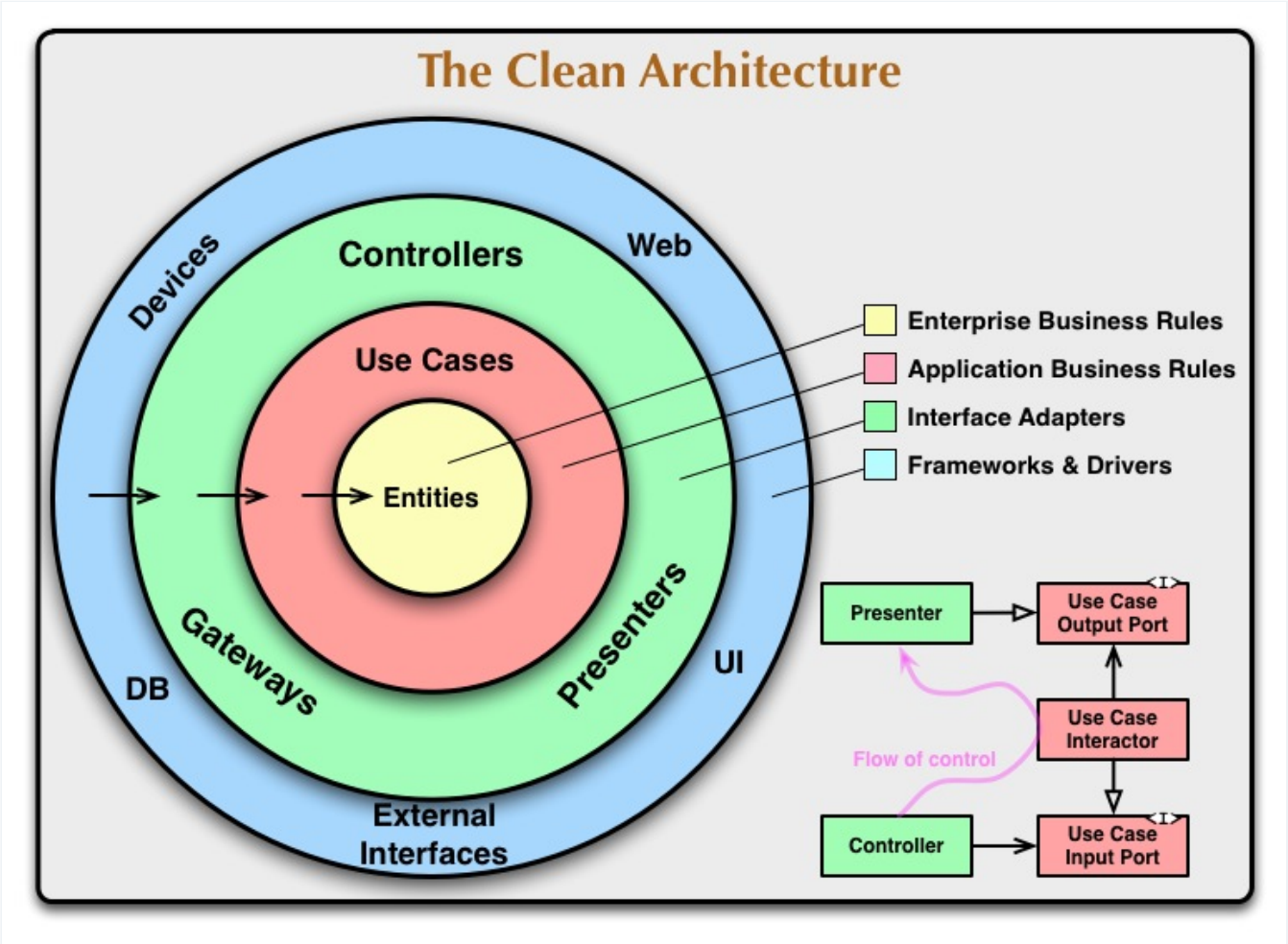
A Arquitetura Limpa

13 de agosto de 2012



Ao longo dos últimos anos, vimos toda uma gama de ideias sobre a arquitetura de sistemas. Esses incluem:

- Hexagonal Architecture (aka Ports and Adapters) por Alistair Cockburn e adotada por Steve Freeman, e Nat Pryce em seu maravilhoso livro Growing Object Oriented Software
- Onion Architecture por Jeffrey Palermo
- Arquitetura gritando de um blog meu ano passado
- DCI de James Coplien e Trygve Reenskaug.
- BCE por Ivar Jacobson de seu livro *Object Oriented Software Engineering: A Use-Case Driven Approach*

Embora todas essas arquiteturas variem um pouco em seus detalhes, elas são muito semelhantes. Todos eles têm o mesmo objetivo, que é a separação de interesses. Todos eles conseguem essa separação dividindo o software em camadas. Cada um tem pelo menos uma camada para regras de negócios e outra para interfaces.

Cada uma dessas arquiteturas produz sistemas que são:

1. Independente de Estruturas. A arquitetura não depende da existência de alguma biblioteca de software carregado de recursos. Isso permite que você use essas estruturas

como ferramentas, em vez de ter que enfiar seu sistema em suas restrições limitadas.
2. Testável. As regras de negócios podem ser testadas sem a interface do usuário, banco de dados, servidor Web ou qualquer outro elemento externo.
3. Independente da interface do usuário. A interface do usuário pode ser alterada facilmente, sem alterar o restante do sistema. Uma interface do usuário da Web pode ser substituída por uma interface do usuário do console, por exemplo, sem alterar as regras de negócios.
4. Independente de Banco de Dados. Você pode trocar Oracle ou SQL Server por Mongo, BigTable, CouchDB ou qualquer outra coisa. Suas regras de negócios não estão vinculadas ao banco de dados.
5. Independente de qualquer agência externa. Na verdade, suas regras de negócios simplesmente não sabem nada sobre o mundo exterior.

O diagrama no início deste artigo é uma tentativa de integrar todas essas arquiteturas em uma única ideia acionável.

A Regra da Dependência

Os círculos concêntricos representam diferentes áreas de software. Em geral, quanto mais você avança, mais alto o nível do software se torna. Os círculos externos são mecanismos. Os círculos internos são políticas.

A regra de substituição que faz essa arquitetura funcionar é *a Regra de Dependência* . Essa regra diz que *as dependências do código-fonte* só podem apontar *para dentro* . Nada em um círculo interno pode saber qualquer coisa 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.

Da mesma forma, os formatos de dados usados em um círculo externo não devem ser usados por um círculo interno, especialmente se esses formatos forem gerados por uma estrutura em um círculo externo. Não queremos que nada em um círculo externo afete os círculos internos.

Entidades

As entidades encapsulam as regras de negócios de *toda a empresa.* Uma entidade pode ser um objeto com métodos, ou pode ser um conjunto de estruturas de dados e funções. Não importa, desde que as entidades possam ser usadas por muitos aplicativos diferentes na empresa.

Se você não tiver uma empresa e estiver apenas escrevendo um único aplicativo, essas entidades serão 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 esperaria que esses objetos fossem afetados por uma alteração na navegação de página ou segurança. Nenhuma mudança operacional em qualquer aplicativo em particular deve afetar a camada de entidade.

Casos de uso

O software nesta camada 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 usar suas regras de negócios em *toda* a empresa para atingir as metas 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 o banco de dados, a interface do usuário ou qualquer uma das estruturas comuns. Esta camada está isolada de tais preocupações.

No entanto, esperamos que as 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 nessa camada certamente será afetado.

Adaptadores de interface

O software nesta camada 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. Os Apresentadores, Visualizações e Controladores pertencem aqui. Os modelos provavelmente são apenas estruturas de dados que são passadas dos controladores para os casos de uso e depois retornam dos casos de uso para os apresentadores e visualizações.

Da mesma forma, os dados são convertidos, nesta camada, da forma mais conveniente para entidades e casos de uso, para a forma 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 deve ser restrito a essa camada e, em particular, às partes dessa camada que têm a ver com o banco de dados.

Também nesta camada está qualquer outro adaptador necessário para converter dados de algum formulário externo, como um serviço externo, para o formulário interno usado pelos casos de uso e entidades.

Estruturas e Drivers.

A camada mais externa é geralmente composta de frameworks e ferramentas como o Banco de Dados, o Web Framework, etc. Geralmente você não escreve muito código nesta camada além do código de cola que se comunica com o próximo círculo interno.

Esta camada é onde todos os detalhes vão. A Web é um detalhe. O banco de dados é um detalhe. Nós mantemos essas coisas do lado de fora, onde elas podem causar pouco dano.

Apenas quatro círculos?

Não, os círculos são esquemáticos. Você pode achar que precisa de mais do que apenas esses quatro. Não há nenhuma regra que diga que você deve sempre ter apenas esses quatro. No entanto, *a Regra de Dependência* sempre se aplica. As dependências do código-fonte sempre apontam para dentro. À medida que você avança, o nível de abstração aumenta. O

círculo mais externo é o detalhe concreto de baixo nível. À medida que você avança, o software se torna mais abstrato e encapsula políticas de nível superior. O círculo mais interno é o mais geral.

Cruzando fronteiras.

No canto inferior direito do diagrama há um exemplo de como cruzamos os limites do círculo. Ele mostra os Controladores e Apresentadores se comunicando com os Casos de Uso na próxima camada. Observe o fluxo de controle. Ele começa no controlador, percorre o caso de uso e termina em execução no apresentador. Observe também as dependências do código-fonte. Cada um deles aponta para os casos de uso.

Geralmente resolvemos essa aparente contradição usando o Princípio de Inversão de Dependência . Em uma linguagem como Java, por exemplo, organizaríamos interfaces e relacionamentos de herança de modo que as dependências do código-fonte se opusessem ao fluxo de controle apenas nos pontos certos da fronteira.

Por exemplo, considere que o caso de uso precisa chamar o apresentador. No entanto, esta chamada não deve ser direta porque isso violaria *a Regra da Dependência* : Nenhum nome em um círculo externo pode ser mencionado por um círculo interno. Portanto, temos o caso de uso chamando uma interface (Mostrada aqui como Porta de saída do caso de uso) no círculo interno e o apresentador no círculo externo a implementa.

A mesma técnica é usada para cruzar todas as fronteiras nas arquiteturas. Aproveitamos o polimorfismo dinâmico para criar dependências de código-fonte que se opõem ao fluxo de controle para que possamos estar em conformidade com a *Regra da Dependência* , não importa em que direção o fluxo de controle esteja indo.

Quais dados ultrapassam os limites.

Normalmente, os dados que cruzam os limites são estruturas de dados simples. Você pode usar estruturas básicas ou objetos simples de Transferência de dados, se desejar. Ou os dados podem ser simplesmente argumentos em chamadas de função. Ou você pode empacotá-lo em um hashmap ou construí-lo em um objeto. O importante é que estruturas de dados simples e isoladas sejam passadas além dos limites. Não queremos trapacear e passar linhas de *Entidades* ou Banco de Dados. Não queremos que as estruturas de dados tenham qualquer tipo de dependência que viole *a Regra de Dependência* .

Por exemplo, muitas estruturas de banco de dados retornam um formato de dados conveniente em resposta a uma consulta. Podemos chamar isso de RowStructure. Não queremos passar essa estrutura de linha para dentro através de um limite. Isso violaria *a Regra da Dependência* porque forçaria um círculo interno a saber algo sobre um círculo externo.

Então, quando passamos dados através de um limite, é sempre na forma que é mais conveniente para o círculo interno.

Conclusão

Conformar-se com essas regras simples não é difícil e vai lhe poupar muitas dores de cabeça daqui para frente. Ao separar o software em camadas e em conformidade com a *Regra de Dependência* , você criará um sistema que é intrinsecamente testável, com todos os benefícios que isso implica. Quando qualquer uma das partes externas do sistema se tornar obsoleta, como o banco de dados ou a estrutura da Web, você poderá substituir esses elementos obsoletos com um mínimo de confusão.