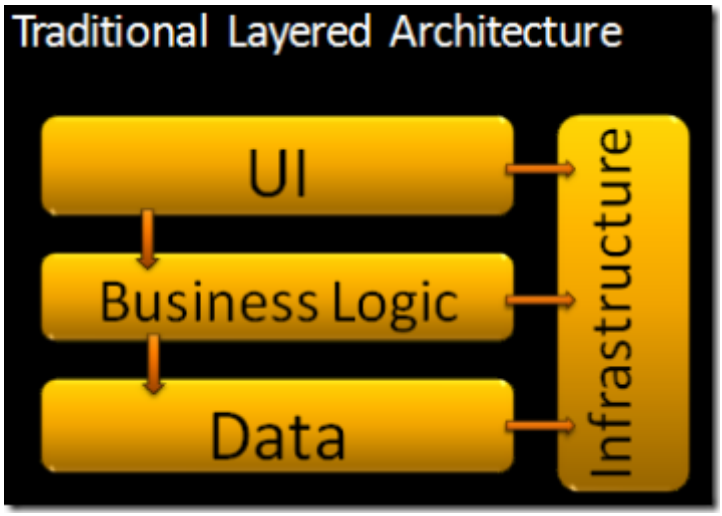
Jeffrey Palermo 8:08 am em 29 de julho de 2008 Tags: onion architecture (4)

A arquitetura da cebola: parte 1

Esta é a parte 1. parte 2 . parte 3 . parte 4 . Meu feed (rss) .

Já falei várias vezes sobre um tipo específico de arquitetura que chamo de “Onion Architecture”. Descobri que isso leva a aplicativos mais fáceis de manter, pois enfatiza a separação de interesses em todo o sistema. Devo definir o contexto para o uso dessa arquitetura antes de prosseguir. Essa arquitetura não é apropriada para sites pequenos. É apropriado para aplicativos de negócios de longa duração, bem como aplicativos com comportamento complexo. Enfatiza o uso de interfaces para contratos de comportamento e força a externalização da infraestrutura. O diagrama que você vê aqui é uma representação da arquitetura tradicional em camadas. Esta é a arquitetura básica que vejo mais frequentemente usada. Cada camada subsequente depende das camadas abaixo dela e, em seguida, cada camada normalmente dependerá de alguns serviços comuns de infraestrutura e utilidade. A grande desvantagem dessa arquitetura em camadas de cima para baixo é o acoplamento que ela cria. Cada camada é acoplada às camadas abaixo dela, e cada camada é frequentemente acoplada a várias questões de infraestrutura. No entanto, sem acoplamento, nossos sistemas não fariam nada útil, mas essa arquitetura cria acoplamento desnecessário.

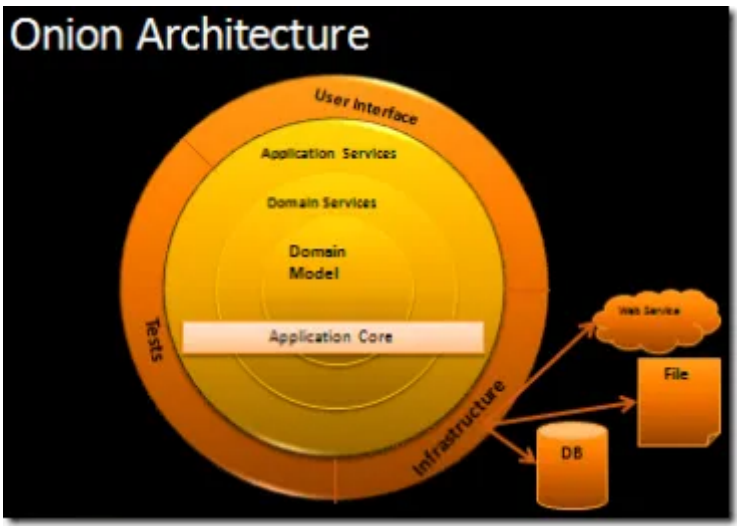


O maior ofensor (e mais comum) é o acoplamento da interface do usuário e da lógica de negócios ao acesso a dados. Sim, a interface do usuário é acoplada ao acesso a dados com essa abordagem. Dependências transitivas ainda são dependências. A interface do usuário não pode funcionar se a lógica de negócios não estiver lá. A lógica de negócios não pode funcionar se o acesso aos dados não estiver lá. Estou intencionalmente ignorando a infraestrutura aqui porque isso normalmente varia de sistema para sistema. O acesso aos dados muda com frequência. Historicamente, a indústria modificou as técnicas de acesso a dados pelo menos a cada três anos; portanto, podemos contar com a necessidade de modificar o acesso a dados daqui a três anos para qualquer sistema saudável e duradouro que seja de missão crítica para os negócios. Muitas vezes não mantemos os sistemas atualizados porque é impossível fazer isso. Se o acoplamento impedir a atualização fácil de peças do sistema, então a empresa não tem escolha a não ser deixar o sistema ficar para trás em um estado de abandono. É assim que os sistemas legados se tornam obsoletos e, eventualmente, são reescritos.

Proponho uma nova abordagem à arquitetura. Honestamente, não é completamente novo, mas estou propondo-o como um padrão de arquitetura nomeado. Os padrões são úteis porque dão aos profissionais de software um vocabulário comum com o qual se comunicar. Há muitos aspectos na Onion Architecture e, se tivermos um termo comum para descrever essa abordagem, podemos nos comunicar de forma mais eficaz.

O diagrama à esquerda mostra a Arquitetura Onion. A premissa principal é que ele controla o acoplamento. A regra fundamental é que todo código pode depender de camadas mais centrais, mas o código não pode depender de camadas mais distantes do núcleo. Em outras palavras, todo acoplamento é em direção ao centro. Essa arquitetura é descaradamente tendenciosa para a programação orientada a objetos e coloca os objetos antes de todos os outros.

Bem no centro vemos o Modelo de Domínio, que representa a combinação de estado e comportamento que modela a verdade para a organização. Ao redor do Modelo de Domínio existem outras camadas com mais comportamento. O número de camadas no núcleo do aplicativo irá variar, mas lembre-se de que o Modelo de Domínio é o próprio centro e, como todo acoplamento é



em direção ao centro, o Modelo de Domínio é acoplado apenas a si mesmo. A primeira camada em torno do Modelo de Domínio é normalmente onde encontraríamos interfaces que fornecem comportamento de salvamento e recuperação de objetos, chamadas de interfaces de repositório. No entanto, o comportamento de salvar objetos não está no núcleo do aplicativo, porque normalmente envolve um banco de dados. Apenas a interface está no núcleo do aplicativo. Nas bordas, vemos UI, Infraestrutura e Testes. A camada externa é reservada para coisas que mudam com frequência. Essas coisas devem ser intencionalmente isoladas do núcleo do aplicativo. No limite, encontraríamos uma classe que implementa uma interface de repositório. Essa classe é acoplada a um método específico de acesso a dados e é por isso que reside fora do núcleo do aplicativo. Essa classe implementa a interface do repositório e, portanto, é acoplada a ela.

A Onion Architecture depende muito do princípio de Inversão de Dependência . O núcleo do aplicativo precisa de implementação de interfaces principais e, se essas classes de implementação residem nas bordas do aplicativo, precisamos de algum mecanismo para injetar esse código em tempo de execução para que o aplicativo possa fazer algo útil.

O banco de dados não é o centro. É externo. A externalização do banco de dados pode ser uma grande mudança para algumas pessoas acostumadas a pensar em aplicativos como “aplicativos de banco de dados”. Com Onion Architecture, não há aplicativos de banco de dados. Existem aplicativos que podem usar um banco de dados como um serviço de armazenamento, mas apenas por meio de algum código de infraestrutura externa que implemente uma interface que faça sentido para o núcleo do aplicativo. Desacoplar o aplicativo do banco de dados, sistema de arquivos, etc., reduz o custo de manutenção durante a vida útil do aplicativo.

Alistair Cockburn escreveu um pouco sobre arquitetura hexagonal . A arquitetura hexagonal e a Onion Architecture compartilham a seguinte premissa: Externalize a infraestrutura e escreva o código do adaptador para que a infraestrutura não fique fortemente acoplada.

Estarei escrevendo mais sobre a Onion Architecture como uma abordagem padrão para a criação de aplicativos corporativos. Permanecerei no espaço do sistema corporativo e toda a discussão residirá nesse contexto. Isso fica ainda mais interessante quando há vários processos compondo um único sistema de software.