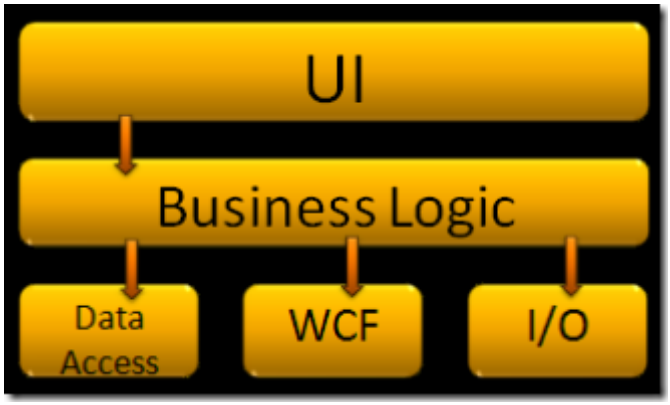
Jeffrey Palermo 9h34 em 4 de agosto de 2008 Tags: arquitetura de cebola (4)

A arquitetura da cebola: parte 3

Parte 1 – Parte 2 – Esta é a parte 3 . parte 4 . – Meu feed RSS

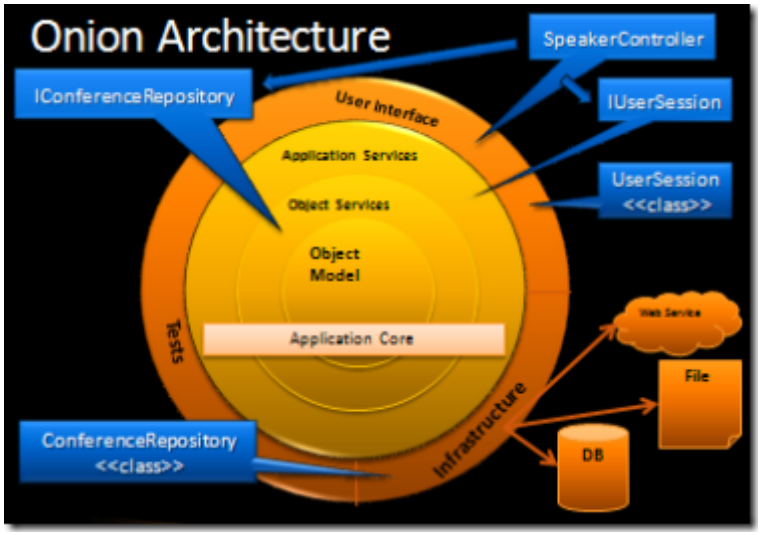
Em meus capítulos anteriores, descrevi o que se tornou minha abordagem para definir a arquitetura de um aplicativo. Com base no feedback, modifiquei um pouco meus diagramas para reduzir a ambiguidade e enfatizar os pontos-chave. O objetivo da parte 3 desta série é comparar e contrastar a Onion Architecture com a arquitetura tradicional em camadas. Vou achatar a Onion Architecture para ver como ela se parece em comparação com a arquitetura em camadas tradicional e forçarei a arquitetura em camadas em uma cebola. Considerando que a forma pode ser qualquer um, a estrutura da aplicação real é radicalmente diferente do que é comumente conhecido e aceito. Definirei quatro princípios da Onion Architecture no final.

Devo enfatizar novamente: não estou reivindicando nenhum avanço em tecnologia ou técnica. Aprendi com outros líderes da indústria como Martin Fowler , Ward Cunningham , Kent Beck , Michael Feathers e outros (especialmente aqueles com quem tive o privilégio de trabalhar aqui em Austin, TX). Estou apresentando a Onion Architecture como um padrão de arquitetura pelo qual podemos comunicar essa abordagem de arquitetura radicalmente diferente. Não "radicalmente diferente como no novo". Diferente como em não mainstream.



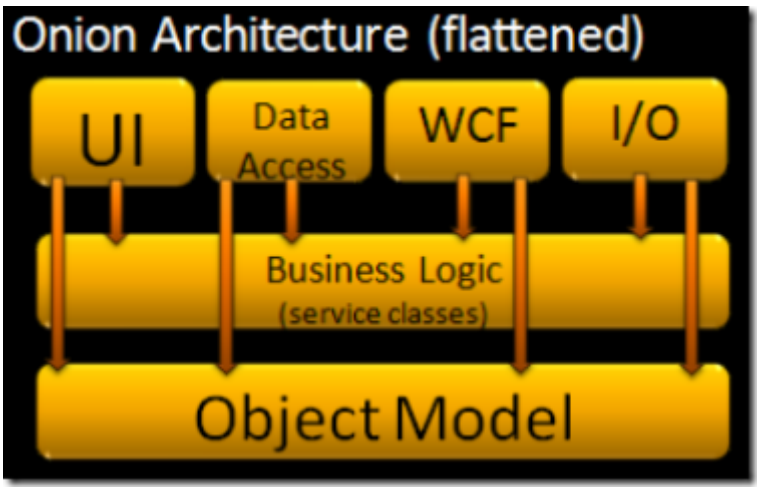
Vamos revisar. A arquitetura tradicional em camadas pode se parecer um pouco com o diagrama mostrado à direita. Cada camada se comunica com a camada abaixo dela. A interface do usuário fala com a lógica de negócios, mas não fala diretamente com o acesso a dados, WCF, etc. A abordagem de camadas chama a atenção para a necessidade de manter certas categorias de código fora da interface do usuário. A grande queda é que a lógica de negócios acaba acoplada às preocupações de infraestrutura. Acesso a dados, E/S e Web Services são todos infraestrutura. Infraestrutura é qualquer código que é uma **mercadoria e não dá ao seu aplicativo uma vantagem competitiva**. É mais provável que esse código seja alterado com frequência à medida que o aplicativo passa por anos de manutenção. Os serviços da Web ainda são relativamente novos, e a primeira versão em .Net, ASMX, já foi preterida em favor do WCF. Podemos ter certeza de que os dias do WCF também estão contados, portanto, é tolice acoplar firmemente a lógica de negócios ao WCF. O acesso aos dados muda a cada dois anos ou mais, então definitivamente não queremos estar fortemente ligados a ele. Para uma vida longa, gostaríamos que nossa lógica de negócios fosse independente dessas preocupações de infraestrutura para que, à medida que a infraestrutura mudasse, a lógica de negócios não precisasse.

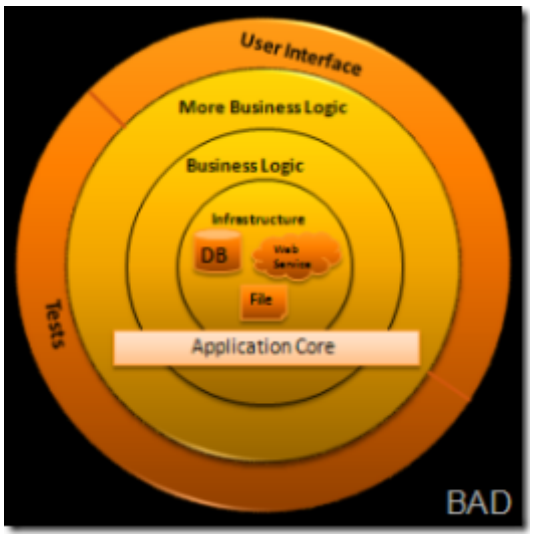
Vamos revisar Onion Architecture. O modelo de objeto está no centro com a lógica de negócios de suporte ao seu redor. A direção do acoplamento é para o centro. A grande diferença é que qualquer camada externa pode chamar diretamente qualquer camada interna. Com a arquitetura tradicional em camadas, uma camada só pode chamar a camada diretamente abaixo dela. **Este é um dos pontos-chave que torna a Onion Architecture diferente da arquitetura tradicional em camadas**. A infraestrutura é empurrada para as bordas, onde nenhum código de lógica de negócios se acopla a ela. O código que interage com o banco de dados implementará interfaces no núcleo do aplicativo. O núcleo do aplicativo é acoplado a essas interfaces, mas não o código de acesso a dados real. Dessa forma, podemos alterar o código em qualquer camada externa sem afetar o núcleo do aplicativo. Incluímos testes porque qualquer aplicativo de longa duração precisa de testes. Os testes ficam na periferia porque o núcleo do aplicativo não se une a eles, mas os testes são acoplados ao núcleo do aplicativo. Também poderíamos ter outra camada de testes em todo o exterior quando testamos a interface do usuário e o código de infraestrutura.



Essa abordagem à arquitetura do aplicativo garante que o núcleo do aplicativo não precise ser alterado como: as alterações da interface do usuário, alterações de acesso a dados, alterações de serviço da Web e infraestrutura de mensagens, alterações de técnicas de E/S.

À direita, criei um diagrama que tenta mostrar como seria a Onion Architecture quando representada como uma arquitetura tradicionalmente em camadas. A grande diferença é que o **Data Access é uma camada superior** junto com UI, I/O, etc. Outra diferença importante é que as camadas acima podem usar qualquer camada abaixo delas, não apenas a camada imediatamente abaixo. Além disso, a lógica de negócios é acoplada ao modelo de objeto, mas não à infraestrutura.





À esquerda, tentei representar a arquitetura tradicional em camadas usando círculos concêntricos. Eu usei linhas pretas ao redor das camadas para denotar que cada camada externa apenas fala com a camada imediatamente em direção ao centro. O grande destaque aqui é que vemos claramente que o aplicativo é construído em torno de acesso a dados e outras infraestruturas. Como o aplicativo possui esse acoplamento, quando o acesso a dados, serviços da Web etc. mudam, a camada de lógica de negócios terá que mudar. A diferença de visão de mundo é como lidar com a infraestrutura. A arquitetura tradicional em camadas se acopla diretamente a ela. A Onion Architecture deixa isso de lado e define abstrações (interfaces) das quais depender. Então o código de infraestrutura também depende dessas abstrações (interfaces). Depender de abstrações é um princípio antigo,

Princípios-chave da Onion Architecture:

- *O aplicativo é construído em torno de um modelo de objeto independente*
- *Camadas internas definem interfaces. Camadas externas implementam interfaces*
- *A direção do acoplamento é para o centro*
- *Todo o código principal do aplicativo pode ser compilado e executado separadamente da infraestrutura*

Eu encorajo você a usar o termo "Onion Architecture" ao falar sobre arquiteturas que aderem aos quatro princípios acima. Acredito que essa abordagem da arquitetura leva a sistemas de vida longa e fáceis de manter. Além disso, na minha experiência, essa arquitetura gera dividendos logo após o início de um projeto, pois facilita a alteração do código.

Embora eu não chame um contêiner IoC como um princípio fundamental, ao usar uma linguagem convencional como Java ou C#, um contêiner IoC faz com que o código se encaixe com muita facilidade. Algumas linguagens possuem recursos de IoC integrados, portanto, isso nem sempre é necessário. Se você estiver usando C#, eu recomendo usar Castle Windsor ou StructureMap .