# Arquitetura hexagonal

*Crie seu aplicativo para funcionar sem uma interface do usuário ou um banco de dados para que você possa executar testes de regressão automatizados no aplicativo, trabalhar quando o banco de dados ficar indisponível e vincular aplicativos sem qualquer envolvimento do usuário.*

---

*Tradução em japonês deste artigo em* http://blog.tai2.net/hexagonal_architexture.html
*Tradução em espanhol deste artigo em* http://academyfor.us/posts/arquitectura-hexagonal *cortesia de Arthur Mauricio Delgadillo*
*explicação original com atualizações em* http: //wiki.c2.com/?HexagonalArchitecture *e* http://wiki.c2.com/?PortsAndAdaptersArchitecture

Veja também Perguntas frequentes sobre arquitetura hexagonal (discussão: Re: Perguntas frequentes sobre arquitetura hexagonal)

---

Arquitetura hexagonal pic 1 a 4 socket.jpg

# O Padrão: Portas e Adaptadores ("Objeto Estrutural")

## Nome alternativo: "Portas e adaptadores"

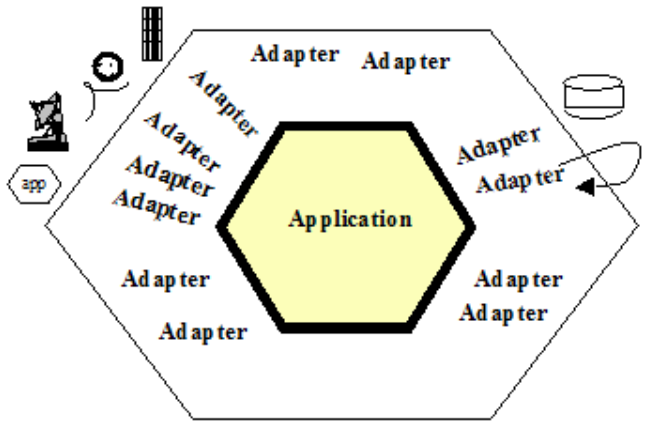
## Nome alternativo: "Arquitetura Hexagonal"

## Intenção

Permitir que um aplicativo seja conduzido igualmente por usuários, programas, testes automatizados ou scripts em lote e seja desenvolvido e testado isoladamente de seus eventuais dispositivos e bancos de dados de tempo de execução.

Quando qualquer driver deseja usar o aplicativo em uma porta, ele envia uma solicitação que é convertida por um adaptador para a tecnologia específica do driver em uma chamada ou mensagem de procedimento utilizável, que passa para a porta do aplicativo. O aplicativo é felizmente ignorante da tecnologia do driver. Quando o aplicativo tem algo para enviar, ele o envia através de uma porta para um adaptador, que cria os sinais apropriados necessários para a tecnologia receptora (humana ou automatizada). O aplicativo tem uma interação semanticamente sólida com os adaptadores em todos os lados, sem realmente conhecer a natureza das coisas do outro lado dos adaptadores.

Figura 1: Arquitetura hexagonal basic.gif




# Motivação

Um dos grandes bugs dos aplicativos de software ao longo dos anos tem sido a infiltração da lógica de negócios no código da interface do usuário. O problema que isso causa é triplo:

- Em primeiro lugar, o sistema não pode ser testado com perfeição com conjuntos de testes automatizados porque parte da lógica que precisa ser testada depende de detalhes visuais que mudam com frequência, como tamanho do campo e posicionamento dos botões;
- Exatamente pela mesma razão, torna-se impossível mudar de um uso do sistema dirigido por humanos para um sistema de execução em lote;
- Ainda pela mesma razão, torna-se difícil ou impossível permitir que o programa seja conduzido por outro programa quando isso se torna atraente.

A solução tentada, repetida em muitas organizações, é criar uma nova camada na arquitetura, com a promessa de que desta vez, real e verdadeiramente, nenhuma lógica de negócios será colocada na nova camada. No entanto, não tendo nenhum mecanismo para detectar quando ocorre uma violação dessa promessa, a organização descobre alguns anos depois que a nova camada está cheia de lógica de negócios e o antigo problema reapareceu.

Imagine agora que “toda” funcionalidade que o aplicativo oferece estivesse disponível por meio de uma API (interface programada do aplicativo) ou chamada de função. Nessa situação, o departamento de teste ou controle de qualidade pode executar scripts de teste automatizados no aplicativo para detectar quando qualquer nova codificação interrompe uma função que estava funcionando anteriormente. Os especialistas em negócios podem criar casos de teste automatizados, antes que os detalhes da GUI sejam finalizados, informando aos programadores quando eles fizeram seu trabalho corretamente (e esses testes se tornam os executados pelo departamento de teste). O aplicativo pode ser implantado no modo "headless", portanto, apenas a API está disponível, e outros programas podem fazer uso de sua funcionalidade — isso simplifica o design geral de conjuntos de aplicativos complexos e também permite que aplicativos de serviço business-to-business usem uns aos outros sem intervenção humana na web. Finalmente, os testes de regressão de função automatizada detectam qualquer violação da promessa de manter a lógica de negócios fora da camada de apresentação. A organização pode detectar e corrigir o vazamento de lógica.

Um problema semelhante interessante existe no que normalmente é considerado “o outro lado” do aplicativo, onde a lógica do aplicativo fica vinculada a um banco de dados externo ou outro serviço. Quando o servidor de banco de dados fica inativo ou sofre retrabalho ou substituição significativa, os programadores não podem trabalhar porque seu trabalho está vinculado à presença do banco de dados. Isso causa custos de atraso e muitas vezes sentimentos ruins entre as pessoas.

Não é óbvio que os dois problemas estejam relacionados, mas há uma simetria entre eles que aparece na natureza da solução.

# Natureza da Solução

Os problemas do lado do usuário e do lado do servidor, na verdade, são causados pelo mesmo erro de design e programação — o emaranhamento entre a lógica de negócios e a interação com entidades externas. A assimetria a explorar não é aquela entre os lados "esquerdo" e "direito" da aplicação, mas entre "dentro" e

"fora" da aplicação. A regra a obedecer é que o código referente à parte "interna" não deve vazar para a parte "externa".

Removendo por um momento qualquer assimetria esquerda-direita ou de cima para baixo, vemos que o aplicativo se comunica por "portas" para agências externas. A palavra “porta” supostamente evoca pensamentos de “portas” em um sistema operacional, onde qualquer dispositivo que adere aos protocolos de uma porta pode ser conectado a ela; e "portas" em aparelhos eletrônicos, onde, novamente, qualquer dispositivo que se encaixe nos protocolos mecânicos e elétricos pode ser conectado.

- O protocolo para uma porta é dado pela *finalidade da conversa* entre os dois dispositivos.

O protocolo assume a forma de uma interface de programa de aplicativo (API).

Para cada dispositivo externo existe um "adaptador" que converte a definição da API para os sinais necessários para esse dispositivo e vice-versa. Uma interface gráfica de usuário ou GUI é um exemplo de adaptador que mapeia os movimentos de uma pessoa para a API da porta. Outros adaptadores que se encaixam na mesma porta são equipamentos de teste automatizados, como FIT ou Fitnesse, drivers de lote e qualquer código necessário para comunicação entre aplicativos na empresa ou na rede.

Em outro lado do aplicativo, o aplicativo se comunica com uma entidade externa para obter dados. O protocolo é normalmente um protocolo de banco de dados. Do ponto de vista do aplicativo, se o banco de dados for movido de um banco de dados SQL para um arquivo simples ou qualquer outro tipo de banco de dados, a conversação na API não deverá ser alterada. Adaptadores adicionais para a mesma porta incluem um adaptador SQL, um adaptador de arquivo simples e, mais importante, um adaptador para um banco de dados “simulado”, um que fica na memória e não depende da presença do banco de dados real.

Muitos aplicativos têm apenas duas portas: a caixa de diálogo do lado do usuário e a caixa de diálogo do lado do banco de dados. Isso lhes dá uma aparência assimétrica, o que faz parecer natural construir o aplicativo em uma arquitetura empilhada unidimensional, de três, quatro ou cinco camadas.

Há dois problemas com esses desenhos. Primeiro e pior, as pessoas tendem a não levar a sério as “linhas” no desenho em camadas. Eles permitem que a lógica do aplicativo vaze pelos limites da camada, causando os problemas mencionados acima. Em segundo lugar, pode haver mais de duas portas para o aplicativo, para que a arquitetura não se encaixe no desenho da camada unidimensional.

A arquitetura hexagonal, ou portas e adaptadores, resolve esses problemas observando a simetria da situação: há um aplicativo no interior se comunicando por um número de portas com coisas do lado de fora. Os itens fora do aplicativo podem ser tratados simetricamente.

O hexágono destina-se a destacar visualmente

(a) a assimetria de dentro para fora e a natureza semelhante dos portos, para fugir da imagem em camadas unidimensional e tudo o que evoca, e

(b) a presença de um número definido de portas diferentes – duas, três ou quatro (quatro é o máximo que encontrei até hoje).

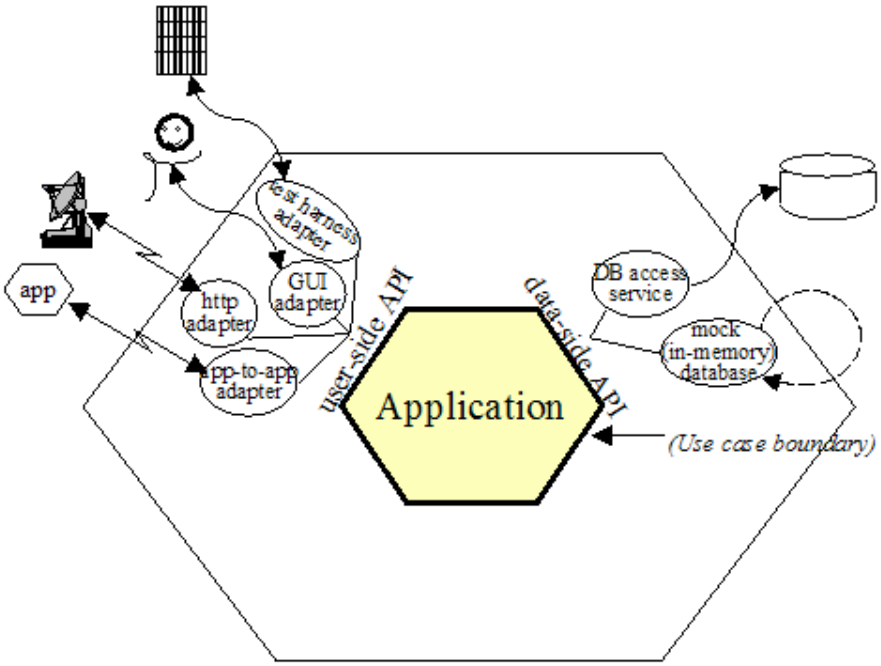
O hexágono não é um hexágono porque o número seis é importante, mas sim para permitir que as pessoas que desenham tenham espaço para inserir portas e adaptadores conforme necessário, não sendo restringidas por um desenho em camadas unidimensional. O termo "arquitetura hexagonal" vem desse efeito visual.

O termo “porta e adaptadores” capta os “propósitos” das partes do desenho. Uma porta identifica uma conversa intencional. Normalmente, haverá vários adaptadores para qualquer porta, para várias tecnologias que podem ser conectadas a essa porta. Normalmente, eles podem incluir uma secretária eletrônica, uma voz humana, um telefone de tom, uma interface gráfica humana, um equipamento de teste, um driver de lote, uma interface http, uma interface direta de programa para programa, uma simulação (em -memory), um banco de dados real (talvez bancos de dados diferentes para desenvolvimento, teste e uso real).

Nas Notas de Aplicação, a assimetria esquerda-direita será exibida novamente. No entanto, o objetivo principal desse padrão é focar na assimetria de dentro para fora, fingindo brevemente que todos os itens externos são idênticos do ponto de vista do aplicativo.

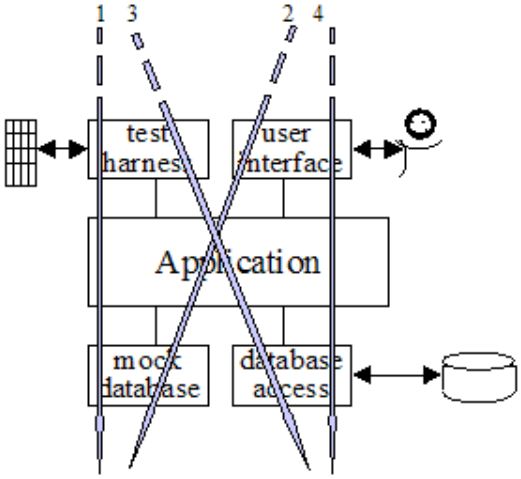
# Estrutura

Figura 2: Arquitetura hexagonal com adapters.gif




A Figura 2 mostra um aplicativo com duas portas ativas e vários adaptadores para cada porta. As duas portas são o lado de controle de aplicativos e o lado de recuperação de dados. Este desenho mostra que o aplicativo pode ser conduzido igualmente por um conjunto de testes de regressão automatizado em nível de sistema, por um usuário humano, por um aplicativo http remoto ou por outro aplicativo local. No lado dos dados, o aplicativo pode ser configurado para ser executado desacoplado de bancos de dados externos usando um oráculo na memória, ou substituição de banco de dados "simulado"; ou pode ser executado no banco de dados de tempo de teste ou de execução. A especificação funcional do aplicativo, talvez em casos de uso, é feita em relação à interface do hexágono interno e não em relação a qualquer uma das tecnologias externas que possam ser usadas.

Figura 3: Porta de celeiro de arquitetura hexagonal image.gif




A Figura 3 mostra o mesmo aplicativo mapeado para um desenho arquitetônico de três camadas. Para simplificar o desenho, apenas dois adaptadores são mostrados para cada porta. Este desenho destina-se a mostrar como vários adaptadores se encaixam nas camadas superior e inferior e a sequência na qual os vários adaptadores são usados durante o desenvolvimento do sistema. As setas numeradas mostram a ordem em que uma equipe pode desenvolver e usar o aplicativo:

1. Com um equipamento de teste FIT conduzindo o aplicativo e usando o banco de dados simulado (em memória) substituindo o banco de dados real;
2. Adicionando uma GUI ao aplicativo, ainda executando o banco de dados simulado;
3. Em testes de integração, com scripts de teste automatizados (por exemplo, do Cruise Control) conduzindo o aplicativo contra um banco de dados real contendo dados de teste;
4. Em uso real, com uma pessoa usando o aplicativo para acessar um banco de dados ao vivo.

# Código de amostra

O aplicativo mais simples que demonstra as portas e adaptadores felizmente vem com a documentação do FIT. É um aplicativo de computação de desconto simples:

```
desconto(valor) = valor * taxa(valor);
```

Na nossa adaptação, o valor virá do usuário e a taxa virá de um banco de dados, então serão duas portas. Nós os implementamos em etapas:

- Com testes, mas com uma taxa constante em vez de um banco de dados simulado,
- então com a GUI,
- em seguida, com um banco de dados simulado que pode ser trocado por um banco de dados real.

*Obrigado a Gyan Sharma da IHC por fornecer o código para este exemplo.*

### Estágio 1: FITAplicativobanco de dados constante como simulado

Primeiro, criamos os casos de teste como uma tabela HTML (consulte a documentação do FIT para isso):

| TestDiscounter | |
|---|---|
| montante | desconto() |
| 100 | 5 |
| 200 | 10 |

Observe que os nomes das colunas se tornarão nomes de classes e funções em nosso programa. O FIT contém maneiras de se livrar desse “programador”, mas para este artigo é mais fácil apenas deixá-los.

Sabendo quais serão os dados de teste, criamos o adaptador do lado do usuário, o ColumnFixture que vem com o FIT conforme enviado:

```
importação fit.ColumnFixture;
classe pública TestDiscounter estende ColumnFixture
{
   app de desconto privado = new Desconto();
   valor em dobro público;
   desconto duplo público()
   { return app.discount(quantia); }
}
```

Isso é realmente tudo o que existe para o adaptador. Até agora, os testes são executados a partir da linha de comando (consulte o livro FIT para obter o caminho necessário). Usamos este:

```
set FIT_HOME=/FIT/FitLibraryForFit15Feb2005
java -cp %FIT_HOME%/lib/javaFit1.1b.jar;%FIT_HOME%/dist/fitLibraryForFit.jar;src;bin
fit.FileRunner test/Discounter.html TestDiscount_Output.html
```

O FIT produz um arquivo de saída com cores mostrando o que passou (ou falhou, caso tenhamos cometido um erro de digitação em algum lugar ao longo do caminho).

Neste ponto, o código está pronto para fazer check-in, conectar-se ao Cruise Control ou à sua máquina de compilação automatizada e incluir no pacote de compilação e teste.

### Estágio 2: IUAplicativobanco de dados constante como simulado

Vou deixar você criar sua própria interface do usuário e fazer com que ela dirija o aplicativo Discounter, já que o código é um pouco longo para incluir aqui. Algumas das principais linhas do código são estas:

```
...
 App de desconto = new Desconto();
public void actionPerformed(evento ActionEvent)
{
    ...
   String quantidadeStr = text1.getText();
   double quantidade = Double.parseDouble(amountStr);
   desconto = app.desconto(valor));
   text3.setText( "" + desconto );
   ...
```

Neste ponto, o aplicativo pode ser demonstrado e testado por regressão. Os adaptadores do lado do usuário estão em execução.

## Estágio 3: (FIT ou UI)Aplicativobanco de dados simulado

Para criar um adaptador substituível para o lado do banco de dados, criamos uma "interface" para um repositório, um "RepositoryFactory" que produzirá o banco de dados simulado ou o objeto de serviço real e o simulado na memória para o banco de dados .

```
interface pública RateRepository
{
   double getRate(double montante);
 }
classe pública RepositoryFactory
{
   public RepositoryFactory() { super(); }
   público estático RateRepository getMockRateRepository()
   {
      retornar novo MockRateRepository();
   }
}
classe pública MockRateRepository implementa RateRepository
{
   public double getRate(quantidade dupla)
   {
      if(quantidade <= 100) return 0,01;
      if(quantidade <= 1000) return 0,02;
      retorno 0,05;
    }
 }
```

Para conectar esse adaptador ao aplicativo Discounter, precisamos atualizar o próprio aplicativo para aceitar um adaptador de repositório a ser usado e fazer com que o adaptador do lado do usuário (FIT ou UI) passe o repositório a ser usado (real ou simulado) para o construtor do próprio aplicativo. Aqui está o aplicativo atualizado e um adaptador FIT que passa em um repositório simulado (o código do adaptador FIT para escolher se passar no adaptador do repositório simulado ou real é mais longo sem adicionar muitas informações novas, então omito essa versão aqui).

```
importar repositório.RepositoryFactory;
importar repositório.RateRepository;
desconto de classe pública
{
   rateRepository privado rateRepository;
   Public Discounter(RateRepository r)
   {
      super();
      taxaRepositório = r;
    }
   desconto duplo público (valor duplo)
```

```
    {
       taxa dupla = rateRepository.getRate(quantidade);
       valor de retorno * taxa;
     }
}
importar app.Discounter;
importação fit.ColumnFixture;
classe pública TestDiscounter estende ColumnFixture
{
   aplicativo de desconto privado =
       new Discounter(RepositoryFactory.getMockRateRepository());
   valor em dobro público;
   desconto duplo público()
   {
      return app.discount(quantidade);
   }
}
```

Isso conclui a implementação da versão mais simples da arquitetura hexagonal.

Para uma implementação diferente, usando Ruby e Rack para uso do navegador, consulte https://github.com/totheralistair/SmallerWebHexagon

# Notas de aplicação

### A assimetria esquerda-direita

O padrão de portas e adaptadores é escrito deliberadamente fingindo que todas as portas são fundamentalmente semelhantes. Essa pretensão é útil no nível arquitetural. Na implementação, portas e adaptadores aparecem em dois tipos, que chamarei de "primário" e "secundário", por razões que logo serão óbvias. Eles também podem ser chamados de adaptadores "condutores" e adaptadores "conduzidos".

O leitor de alerta deve ter notado que em todos os exemplos dados, os acessórios FIT são usados nas portas do lado esquerdo e simulações no lado direito. Na arquitetura de três camadas, o FIT fica na camada superior e o mock fica na camada inferior.

Isso está relacionado à ideia de casos de uso de “atores primários” e “atores secundários”. Um "ator primário" é um ator que dirige o aplicativo (o tira do estado quiescente para executar uma de suas funções anunciadas). Um "ator secundário" é aquele que o aplicativo direciona, seja para obter respostas ou apenas para notificar. A distinção entre "primário" e "secundário" está em quem desencadeia ou está no comando da conversa.

O adaptador de teste natural para substituir um ator "primário" é o FIT, uma vez que essa estrutura é projetada para ler um script e conduzir o aplicativo. O adaptador de teste natural para substituir um ator "secundário", como um banco de dados, é uma simulação, uma vez que é projetado para responder a consultas ou registrar eventos do aplicativo.

Essas observações nos levam a seguir o diagrama de contexto de caso de uso do sistema e desenhar as "portas primárias" e"adaptadores primários" no lado esquerdo (ou superior) do hexágono, e as "portas secundárias" e ' 'adaptadores secundários" no lado direito (ou inferior) do hexágono.

A relação entre portas/adaptadores primários e secundários e sua respectiva implementação em FIT e mocks é útil ter em mente, mas deve ser usado como consequência do uso da arquitetura de portas e adaptadores, não para curto-circuito. O benefício final de uma implementação de portas e adaptadores é a capacidade de executar o aplicativo em um modo totalmente isolado.

### Casos de uso e o limite do aplicativo

É útil usar o padrão de arquitetura hexagonal para reforçar a maneira preferida de escrever casos de uso.

Um erro comum é escrever casos de uso para conter um conhecimento íntimo da tecnologia localizada fora de cada porta. Esses casos de uso ganharam um nome justificadamente ruim no setor por serem longos, difíceis de ler, chatos, frágeis e caros de manter.

Entendendo a arquitetura de portas e adaptadores, podemos ver que os casos de uso geralmente devem ser escritos no limite da aplicação (o hexágono interno), para especificar as funções e eventos suportados pela aplicação, independente da tecnologia externa. Esses casos de uso são mais curtos, mais fáceis de ler, mais baratos de manter e mais estáveis ao longo do tempo.

### Quantas Portas?

O que exatamente um porto é e não é, em grande parte, é uma questão de gosto. Em um extremo, cada caso de uso poderia receber sua própria porta, produzindo centenas de portas para muitos aplicativos. Alternativamente, pode-se imaginar mesclando todas as portas primárias e todas as portas secundárias para que haja apenas duas portas, um lado esquerdo e um lado direito.

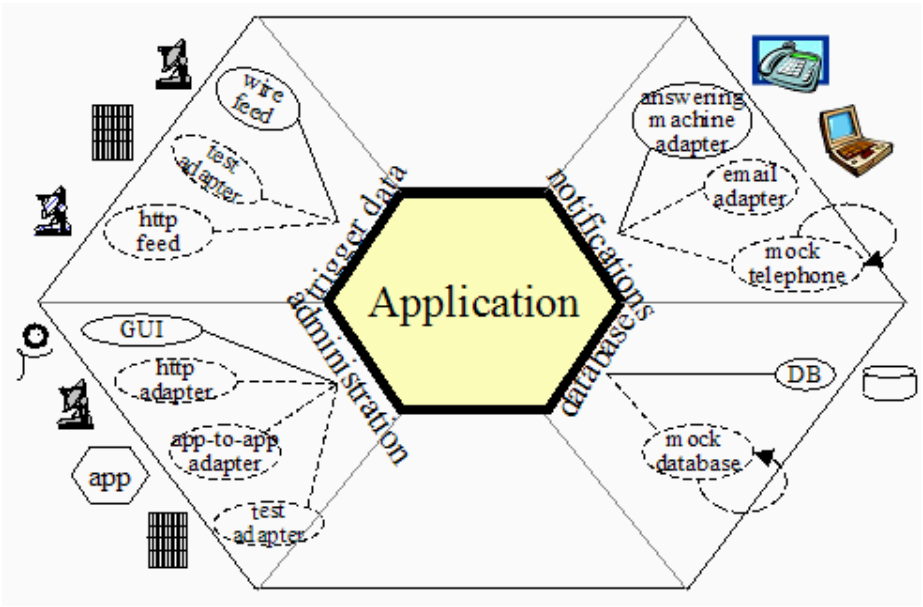
Nenhum dos extremos parece ótimo.

O sistema meteorológico descrito nos Usos Conhecidos tem quatro portas naturais: o feed meteorológico, o administrador, os assinantes notificados, o banco de dados de assinantes. Um controlador de máquina de café possui quatro portas naturais: o usuário, o banco de dados contendo as receitas e preços, os dispensadores e a caixa de moedas. Um sistema de medicação hospitalar pode ter três: um para a enfermeira, um para o banco de dados de prescrição e um para os dispensadores de medicamentos controlados por computador.

Não parece haver nenhum dano específico em escolher o número “errado” de portas, então isso continua sendo uma questão de intuição. Minha seleção tende a favorecer um número pequeno, duas, três ou quatro portas, conforme descrito acima e nos Usos Conhecidos.

# Usos conhecidos

Figura 4: Exemplo complexo de arquitetura hexagonal.gif




A Figura 4 mostra um aplicativo com quatro portas e vários adaptadores em cada porta. Isso foi derivado de um aplicativo que ouvia alertas do serviço meteorológico nacional sobre terremotos, tornados, incêndios e inundações e notificava as pessoas em seus telefones ou secretárias eletrônicas. Na época em que discutimos esse sistema, as interfaces do sistema foram identificadas e discutidas por "tecnologia, vinculada ao propósito". Havia uma interface para dados de disparo que chegavam por uma alimentação de arame, uma para dados de notificação a serem enviados para secretárias eletrônicas, uma interface administrativa implementada em uma GUI e uma interface de banco de dados para obter seus dados de assinante.

As pessoas estavam lutando porque precisavam adicionar uma interface http do serviço meteorológico, uma interface de e-mail para seus assinantes e precisavam encontrar uma maneira de agrupar e desagrupar seu crescente conjunto de aplicativos para diferentes preferências de compra dos clientes. Eles temiam estar diante de um pesadelo de manutenção e testes, pois precisavam implementar, testar e manter versões separadas para todas as combinações e permutações.

Sua mudança no design foi arquitetar as interfaces do sistema "por propósito" em vez de por tecnologia, e fazer com que as tecnologias fossem substituíveis (em todos os lados) por adaptadores. Eles imediatamente adquiriram a capacidade de incluir o feed http e a notificação por e-mail (os novos adaptadores são mostrados no desenho com linhas tracejadas). Ao tornar cada aplicativo executável no modo headless por meio de APIs, eles podem adicionar um adaptador app-to-add e desagrupar o conjunto de aplicativos, conectando os subaplicativos sob demanda. Por fim, ao tornar cada aplicativo executável completamente isoladamente, com adaptadores de teste e simulação instalados, eles ganharam a capacidade de testar seus aplicativos com scripts de teste automatizados autônomos.

## Mac, Windows, Google, Flickr, Web 2.0

No início da década de 1990, os aplicativos Macintosh, como aplicativos de processador de texto, precisavam ter interfaces acionáveis por API, para que aplicativos e scripts escritos pelo usuário pudessem acessar todas as funções dos aplicativos. Os aplicativos de desktop do Windows desenvolveram a mesma capacidade (não tenho o conhecimento histórico para dizer o que veio primeiro, nem isso é relevante para o ponto).

A tendência atual (2005) em aplicativos da web é publicar uma API e permitir que outros aplicativos da web acessem essas APIs diretamente. Assim, é possível publicar dados de crimes locais em um mapa do Google ou criar aplicativos da web que incluem as habilidades de arquivamento e anotação de fotos do Flickr.

Todos esses exemplos são sobre tornar as APIs de "portas" primárias visíveis. Não vemos informações aqui sobre as portas secundárias.

## Saídas Armazenadas

Este exemplo escrito por Willem Bogaerts no wiki C2:

“Encontrei algo semelhante, mas principalmente porque minha camada de aplicação tinha uma forte tendência a se tornar uma central telefônica que gerenciava coisas que não deveria fazer. Meu aplicativo gerou saída, mostrou-a ao usuário e, em seguida, teve alguma possibilidade de armazená-la também. Meu principal problema era que você não precisava armazená-lo sempre. Então, meu aplicativo gerou saída, teve que armazenar em buffer e apresentá-lo ao usuário. Então, quando o usuário decidiu que queria armazenar a saída, o aplicativo recuperou o buffer e o armazenou de verdade.

Eu não gostei nada disso. Então eu encontrei uma solução: ter um controle de apresentação com instalações de armazenamento. Agora, o aplicativo não canaliza mais a saída em direções diferentes, mas simplesmente a envia para o controle de apresentação. É o controle de apresentação que armazena a resposta e dá ao usuário a possibilidade de armazená-la.

A arquitetura tradicional em camadas enfatiza que “UI” e “armazenamento” são diferentes. A arquitetura de portas e adaptadores pode reduzir a saída a ser simplesmente “saída” novamente. ”

## Exemplo anônimo do C2-wiki

“Em um projeto em que trabalhei, usamos o SystemMetaphor de um sistema estéreo componente. Cada componente possui interfaces definidas, cada uma com uma finalidade específica. Podemos então conectar os componentes de maneiras quase ilimitadas usando cabos e adaptadores simples.”

## Desenvolvimento Distribuído de Grandes Equipes

Este ainda está em uso experimental e, portanto, não conta corretamente como um uso do padrão. No entanto, é interessante considerar.

Todas as equipes em diferentes locais constroem a arquitetura Hexagonal, usando FIT e simulações para que os aplicativos ou componentes possam ser testados no modo autônomo. A compilação do CruiseControl é executada a cada meia hora e executa todos os aplicativos usando a combinação FIT+mock. À medida que o subsistema do aplicativo e os bancos de dados são concluídos, os mocks são substituídos por bancos de dados de teste.

## Separando o desenvolvimento da interface do usuário e da lógica do aplicativo

Este ainda está em uso experimental inicial e, portanto, não conta como uso do padrão. No entanto, é interessante considerar.

O design da interface do usuário é instável, pois eles ainda não decidiram sobre uma tecnologia de direção ou uma metáfora. A arquitetura de serviços de back-end não foi decidida e, de fato, provavelmente mudará várias vezes nos próximos seis meses. No entanto, o projeto começou oficialmente e o tempo está passando.

A equipe de aplicativos cria testes e simulações de FIT para isolar seu aplicativo e cria funcionalidades testáveis e demonstráveis para mostrar a seus usuários. Quando as decisões da interface do usuário e dos serviços de back-end finalmente forem atendidas, “deve ser simples” adicionar esses elementos ao aplicativo. Fique atento para saber como isso funciona (ou tente você mesmo e me escreva para me informar).

# Padrões relacionados

## Adaptador

O livro "Design Patterns" contém uma descrição do padrão genérico "Adaptador": “Converta a interface de uma classe em outra interface que os clientes esperam.” O padrão de portas e adaptadores é um uso específico do padrão "Adaptador".

## Model-View-Controller

O padrão MVC foi implementado já em 1974 no projeto Smalltalk. Ele recebeu, ao longo dos anos, muitas variações, como Model-Interactor e Model-View-Presenter. Cada um deles implementa a ideia de portas e adaptadores nas portas primárias, não nas portas secundárias.

## Objetos simulados e loopback

“Um objeto simulado é um “agente duplo” usado para testar o comportamento de outros objetos. Primeiro, um objeto simulado atua como uma implementação falsa de uma interface ou classe que imita o comportamento externo de uma implementação verdadeira. Em segundo lugar, um objeto simulado observa como outros objetos interagem com seus métodos e compara o comportamento real com as expectativas predefinidas. Quando ocorre uma discrepância, um objeto simulado pode interromper o teste e relatar a anomalia. Se a discrepância não puder ser notada durante o teste, um método de verificação chamado pelo testador garante que todas as expectativas foram atendidas ou as falhas relatadas.” — De http://MockObjects.com

Totalmente implementados de acordo com a agenda do mock-object, os mock objects são usados em toda a aplicação, não apenas na interface externa. Eu tomo emprestada a palavra “mock” como a melhor descrição curta de um substituto na memória para um ator secundário externo.

O padrão Loopback é um padrão explícito para criar um substituto interno para um dispositivo externo.

## Pedestais

Em “Patterns for Generating a Layered Architecture”, Barry Rubel descreve um padrão sobre a criação de um eixo de simetria em software de controle que é muito semelhante a portas e adaptadores. O padrão "Pedestal" exige a implementação de um objeto representando cada dispositivo de hardware dentro do sistema e a vinculação desses objetos em uma camada de controle. O padrão "Pedestal" pode ser usado para descrever ambos os lados da arquitetura hexagonal, mas ainda não enfatiza a semelhança entre os adaptadores. Além disso, sendo escrito para um ambiente de controle mecânico, não é tão fácil ver como aplicar o padrão em aplicativos de TI.

## Verificações

A linguagem padrão de Ward Cunningham para detectar e lidar com erros de entrada do usuário é boa para tratamento de erros nos limites do hexágono interno.

## Inversão de Dependência (Injeção de Dependência) e SPRING

O Princípio de Inversão de Dependência de Bob Martin (também chamado de Injeção de Dependência por Martin Fowler) afirma que “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.” O padrão "Dependency Injection" de Martin Fowler fornece algumas implementações. Eles mostram como criar adaptadores de ator secundário trocáveis. O código pode ser digitado diretamente, como feito no código de exemplo do artigo, ou usando arquivos de configuração e fazendo com que a estrutura SPRING gere o código equivalente.

# Reconhecimentos

Agradecemos a Gyan Sharma, da Intermountain Health Care, por fornecer o código de exemplo usado aqui. Obrigado a Rebecca Wirfs-Brock por seu livro "Object Design", que quando lido junto com o padrão "Adapter" do livro "Design Patterns", me ajudou a entender o que era o hexágono. Obrigado também às pessoas no wiki de Ward, que forneceram comentários sobre esse padrão ao longo dos anos (por exemplo, particularmente o http://silkandspinach.net/blog/2004/07/hexagonal_soup.html de Kevin Rutherford ).

# Referências e leituras relacionadas

FIT, A Framework for Integrating Testing: Cunningham, W., online em http://fit.c2.com , e Mugridge, R. e Cunningham, W., "Fit for Developing Software", Prentice-Hall PTR, 2005.

O padrão "Adaptador": em Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns", Addison-Wesley, 1995, pp. 139-150.

O padrão "Pedestal": em Rubel, B., "Patterns for Generating a Layered Architecture", em Coplien, J., Schmidt, D., "PatternLanguages of Program Design", Addison-Wesley, 1995, pp 119-150.

O padrão "Cheques": por Cunningham, W., online em http://c2.com/ppr/checks.html

O "Princípio da Inversão de Dependência": Martin, R., em"Padrões e Práticas de Princípios de Desenvolvimento de Software Ágil", Prentice Hall, 2003, Capítulo 11: “O Princípio de Inversão de Dependência”, e online em http://www.objectmentor.com/resources/articles/dip.pdf

O padrão "Dependency Injection": Fowler, M., online em http://www.martinfowler.com/articles/injection.html

O padrão "Mock Object": Freeman, S. online em http://MockObjects.com

O padrão "Loopback": Cockburn, A., online em http://c2.com/cgi/wiki?LoopBack

"Casos de uso:" Cockburn, A., "Writing Effective Use Cases", Addison-Wesley, 2001, e Cockburn, A., “Structuring Use Cases with Goals”, online em http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm

---

Comentários do site antigo:

---

O blog de André Boonzaaijer While True discute uma aplicação usando a arquitetura Hexagonal (discussão: Re: Arquitetura Hexagonal) e também tem uma imagem legal da arquitetura.

Kevin Rutherford iniciou várias notas e discussões em torno disso:

> *Gravidade e adaptabilidade de software*
>
> *Arquitetura hexagonal*
>
> *Bancos de dados como suporte de vida para objetos de domínio*
>
> *Sopa hexagonal*

Timo escreveu um artigo chamado Wrap it thinly sobre seu uso com TDD.

Gerard Meszaros em seu livro sobre padrões Xunit escreveu http://xunitpatterns.com/Hexagonal%20Architecture.html .

Brian Anderson passou várias entradas de blog pensando nisso:

> *Sucesso!*
>
> *Problemas com Smart Clients hoje*
>
> *em tempo de compilação vs visualização em tempo de execução*
>
> *o uso de simetria na abordagem hexagonal*
>
> *Voltar para Arquitetura Hexagonal*
>
> *algumas reflexões sobre o padrão "Design Pattern"*
>
> *http://www.brianmandersen.com/blog/page/2/*

A página original estava no wiki de Ward em http://c2.com/cgi/wiki?HexagonalArchitecture

---

# Utah Code Camp 19 de setembro de 2009: Atribuição de codificação

A aplicação mais simples vem com a documentação do FIT. É um aplicativo de computação de desconto simples:

> *desconto(valor) = valor * descontoRate(valor);*

- 'Amount' vem do usuário ou de um framework de teste (ou um arquivo)
- 'Rate' vem de um banco de dados ou uma simulação na memória de um banco de dados

Implemente o pp em etapas:

1. Entrada de uma estrutura de teste, usando uma constante para o discountRate,
2. Entrada de uma GUI, ainda usando uma constante para o discountRate.
3. Entrada de teste ou GUI, discountRate de um banco de dados simulado que pode ser trocado por um banco de dados real.

---

# Implementação de Ruby / Rack (sem Rails)

Eu fiz uma pequena versão para web reader disso, veja em

- https://github.com/totheralistair/SmallerWebHexagon

Esse leva um navegador ou driver de rack, ou apenas um driver de teste no lado esquerdo, e uma constante, ou em código, ou de um banco de dados de taxa de arquivo no lado direito. Eu escrevi isso para mostrar a implementação em uma configuração simples, mas real (2014).

---

# Outras discussões e implementações

Você pode encontrar mais online sobre essa arquitetura pesquisando no Google ou no Twitter (em particular). Veja também:

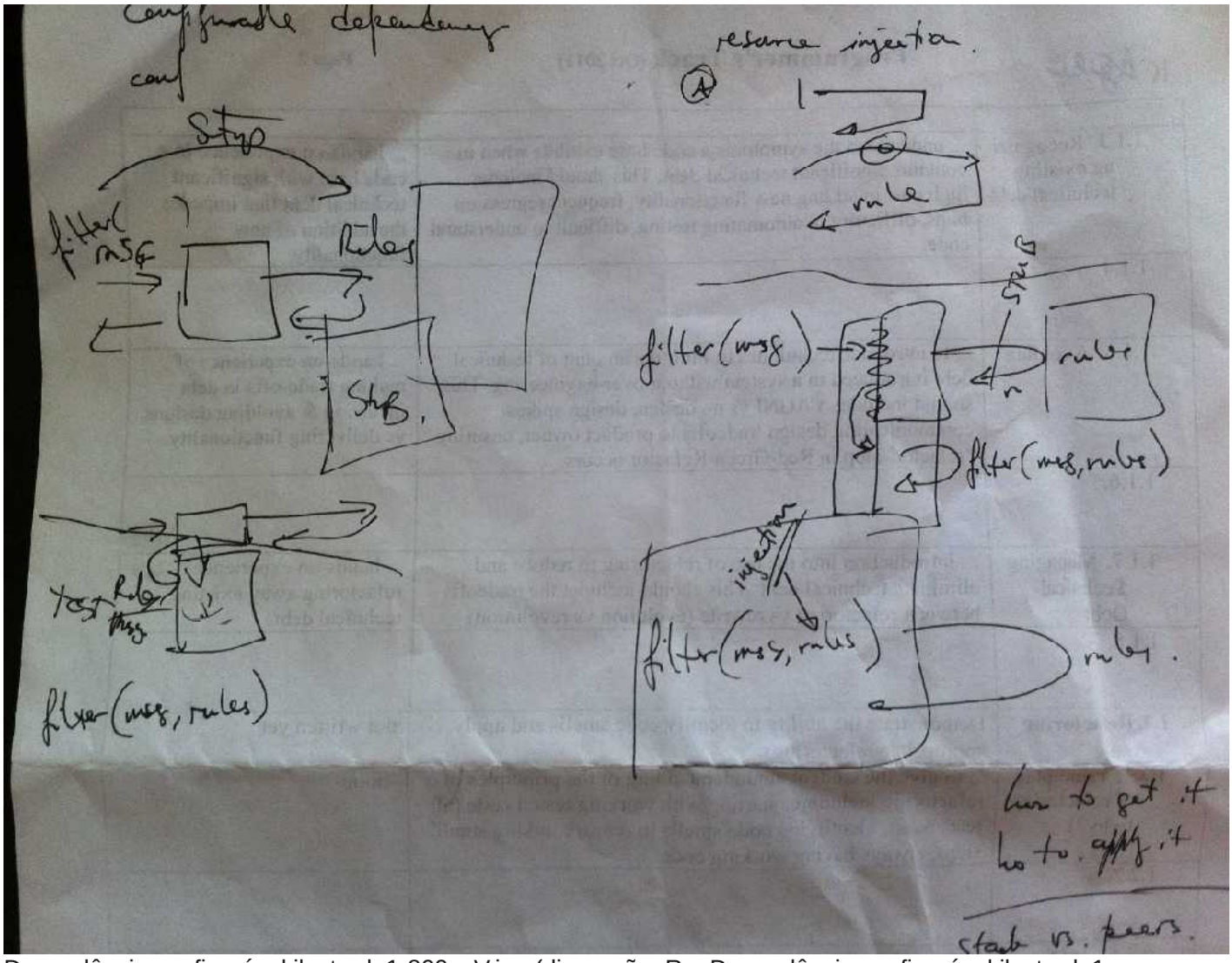
- http://tpierrain.blogspot.fr/2013/08/a-zoom-on-hexagonalcleanonion.html
- A apresentação de slides da palestra sobre isso que dei no Utah Code é http://alistair.cockburn.us/Hexagonal+Architecture+Keynote+at+Utah+Code+Camp+September+2009.pps

- também boa elaboração com notas para si mesmo por Duncan Nisbet em http://www.duncannisbet.co.uk/hexagonal-architecture-for-testers-part-1
- meu vídeo relâmpago talk na Mountain West Ruby Conference em 2010: Vídeo da arquitetura hexagonal de Alistairs CQRS Lightning Talk Mountain West Ruby Conference 2010 (discussão: Re: Vídeo da arquitetura hexagonal de Alistairs CQRS Lightning talk Mountain West Ruby Conference 2010)
- https://twitter.com/search?q=%22hexagonal%20architecture%22
- http://twitter.com/andrzejkrzywda/status/267420878487310336
- (somente porta de usuário) um pequeno CMS https://github.com/totheralistair/SmallWebHexagon
- https://github.com/patmaddox/hexarch2 Pat Maddox começa com uma arquitetura hexagonal e a transforma em baseada em eventos e depois em CQRS. Dê uma olhada.
- Adorável longa discussão detalhada sobre arquitetura hexagonal e Rails com BadrinathJanakiraman e Martin Fowler: http://thoughtworks.wistia.com/medias/uxjb0lwrcz
- evolução detalhada em https://github.com/Lunch-box/SimpleOrderRouting/wiki/Logbook-4#day-15-october-27th-2014

---

# Dependências configuráveis, atores primários e secundários

Tentei tornar este padrão verdadeiramente simétrico, daí o hexágono. No entanto, observando várias implementações, lentamente ficou claro que há uma assimetria (que é uma coisa que torna isso fundamentalmente diferente de padrões vizinhos, como a arquitetura onion). Como dito acima (veja A assimetria esquerda-direita), a assimetria corresponde ao conceito de **atores primários** e secundários de Ivar Jacobson e afeta como a Dependência Configurável (discussão: Re: Dependência Configurável) é implementada. (Isso é mostrado brevemente no esboço de dependência configurável:




Dependência configurável ilustrada1-800pxV.jpg (discussão: Re: Dependência configurável ilustrada1-800pxV.jpg)

A diferença entre um ator primário e um secundário está apenas em quem *inicia* a conversa. Um **ator principal** conhece e inicia a conversa com o sistema ou aplicativo; para um **ator secundário** , é o sistema ou aplicativo que conhece e inicia a discussão com o outro. Essa é, na verdade, a única diferença entre os dois, no caso de uso da terra.

Na implementação, essa diferença é importante: quem iniciar a conversa deve receber a alça do outro.

No caso de **portas de ator primário** , o construtor de macro passará para a interface do usuário, estrutura de teste ou driver o identificador do aplicativo e dirá "Vá falar com isso". O ator principal ligará para o aplicativo e o aplicativo provavelmente nunca saberá quem o chamou. (Isso é normal para destinatários de uma chamada).

Em contraste, para **portas de ator secundário** , o construtor de macro passará para a interface do usuário, estrutura de teste ou driver o identificador para o ator secundário a ser usado, que será passado como um parâmetro para o aplicativo, e o aplicativo agora saber quem/o que é o destinatário da chamada de saída. (Isso é novamente normal para enviar uma chamada).

Assim, o sistema ou aplicativo é construído de forma diferente para portas de atores primários e secundários: ignorante e inicialmente passivo para os atores primários, e tendo uma maneira de armazenar e chamar as portas de atores secundários.

Ambas as portas implementam dependência configurável (discussão: Re: Dependência configurável) , mas de forma diferente.

Exemplo simples para um sistema de 3 portas, como uma máquina de café ou uma unidade médica de hospital dispensando medicamentos por via intravenosa (que estranho que eles saiam iguais, arquitetonicamente!):

- O comprador, equipamento de teste ou administrador do hospital é um ator principal que conduz o sistema
- O banco de dados de receitas ou banco de dados médico é um ator secundário, oferecendo suas informações
- Os dispensadores de produtos químicos em ambos são atores secundários.

Resultado final: o aspecto primário/secundário de uma porta não pode ser ignorado.

---

Veja também Perguntas frequentes sobre arquitetura hexagonal (discussão: Re: Perguntas frequentes sobre arquitetura hexagonal)

4 de janeiro de 2005 | Tags: Artigos , Hexagonal , OO design , Padrões , Engenharia de software | 40 Comentários