Inversão de recipientes de controle e o padrão de injeção de dependência

Na comunidade Java, tem havido uma onda de contêineres leves que ajudam a montar componentes de diferentes projetos em um aplicativo coeso. Subjacente a esses contêineres está um padrão comum de como eles realizam a fiação, um conceito que eles se referem sob o nome genérico de "Inversão de controle". Neste artigo, aprofundo como esse padrão funciona, sob o nome mais específico de "Injeção de dependência", e comparo-o com a alternativa do Service Locator. A escolha entre eles é menos importante do que o princípio de separar a configuração do uso.

23 de janeiro de 2004

Martin Fowler

CONTEÚDO

POPULAR

PROJETO

PROJETO DE COLABORAÇÃO DE OBJETOS

ARQUITETURA DE APLICAÇÃO

TRADUÇÕES: Chinês · Português · Francês · Italiano · Romeno · Espanhol · Polonês · Vietnamita

---

Uma das coisas interessantes sobre o mundo Java corporativo é a enorme quantidade de atividades na construção de alternativas para as tecnologias J2EE convencionais, muitas delas acontecendo em código aberto. Muito disso é uma reação à complexidade de peso no mundo J2EE mainstream, mas muito disso também está explorando alternativas e surgindo com ideias criativas. Um problema comum a ser enfrentado é como conectar diferentes elementos: como combinar essa arquitetura de controlador da web com o suporte da interface de banco de dados quando eles foram criados por equipes diferentes com pouco conhecimento umas das outras. Vários frameworks tentaram resolver esse problema e vários estão se ramificando

para fornecer uma capacidade geral de montar componentes de diferentes camadas. Estes são frequentemente referidos como contêineres leves, os exemplos incluemPicoContainer e Spring .

Subjacente a esses contêineres estão vários princípios de design interessantes, coisas que vão além desses contêineres específicos e, de fato, da plataforma Java. Aqui, quero começar a explorar alguns desses princípios. Os exemplos que uso estão em Java, mas, como a maioria dos meus escritos, os princípios são igualmente aplicáveis a outros ambientes OO, particularmente .NET.

Componentes e Serviços

O tópico de conectar elementos juntos me arrasta quase imediatamente para os complicados problemas de terminologia que envolvem os termos serviço e componente. Você encontra artigos longos e contraditórios sobre a definição dessas coisas com facilidade. Para meus propósitos, aqui estão meus usos atuais desses termos sobrecarregados.

Eu uso componente para significar um conjunto de software que se destina a ser usado, sem alterações, por um aplicativo que está fora do controle dos criadores do componente. Por 'sem alteração', quero dizer que o uso do aplicativo não altera o código-fonte dos componentes, embora possam alterar o comportamento do componente, estendendo-o de maneiras permitidas pelos criadores do componente.

Um serviço é semelhante a um componente no sentido de que é usado por aplicativos estrangeiros. A principal diferença é que espero que um componente seja usado localmente (pense em um arquivo jar, assembly, dll ou uma importação de código-fonte). Um serviço será usado remotamente por meio de alguma interface remota, síncrona ou assíncrona (por exemplo, serviço da web, sistema de mensagens, RPC ou soquete).

Eu uso principalmente serviços neste artigo, mas muito da mesma lógica pode ser aplicada a componentes locais também. Na verdade, muitas vezes você precisa de algum tipo de estrutura de componente local para acessar facilmente um serviço remoto. Mas escrever "componente ou serviço" é cansativo de ler e escrever, e os serviços estão muito mais na moda no momento.

Um exemplo ingênuo

Para ajudar a tornar tudo isso mais concreto, usarei um exemplo em execução para falar sobre tudo isso. Como todos os meus exemplos, é um daqueles exemplos super simples; pequeno o suficiente para ser irreal, mas com sorte o suficiente para você visualizar o que está acontecendo sem cair no pântano de um exemplo real.

Neste exemplo, estou escrevendo um componente que fornece uma lista de filmes dirigidos por um determinado diretor. Esta função incrivelmente útil é implementada por um único método.

classe MovieLister ...

```
  public Movie [] moviesDirectedBy (String arg) {
      List allMovies = finder.findAll ();
      para (Iterator it = allMovies.iterator (); it.hasNext ();) {
          Filme filme = (Filme) it.next ();
          if (! movie.getDirector (). equals (arg)) it.remove ();
      }
      return (Filme []) allMovies.toArray (novo Filme [allMovies.size ()]);
  }
```

A implementação dessa função é ingênua ao extremo, ela pede a um objeto localizador (que veremos em um momento) para retornar todos os filmes que conhece. Em seguida, ele apenas vasculha essa lista para retornar aqueles dirigidos por um determinado diretor. Não vou consertar essa ingenuidade em particular, pois é apenas o andaime do verdadeiro ponto deste artigo.

O verdadeiro ponto deste artigo é este objeto localizador, ou particularmente como conectamos o objeto Lister a um objeto localizador específico. A razão pela qual isso é interessante é que eu quero que meu `moviesDirectedBy`método maravilhoso seja completamente independente de como todos os filmes estão sendo armazenados. Portanto, tudo o que o método faz é se referir a um localizador, e tudo o que o localizador faz é saber como responder ao `findAll`método. Posso revelar isso definindo uma interface para o localizador.

```
interface pública MovieFinder {
    Lista findAll ();
}
```

Agora, tudo isso está muito bem dissociado, mas em algum momento eu tenho que criar uma classe concreta para realmente criar os filmes. Neste caso, coloquei o código para isso no construtor da minha classe de registro.

classe MovieLister ...

```
  localizador de MovieFinder privado;
  public MovieLister () {
    localizador = novo ColonDelimitedMovieFinder ("movies1.txt");
  }
```

O nome da classe de implementação vem do fato de que estou obtendo minha lista de um arquivo de texto delimitado por dois pontos. Vou poupar vocês dos detalhes, afinal a questão é que há alguma implementação.

Agora, se estou usando essa classe apenas para mim, tudo isso é ótimo e elegante. Mas o que acontece quando meus amigos são dominados pelo desejo por essa funcionalidade maravilhosa e desejam uma cópia do meu programa? Se eles também armazenam suas listas de filmes em um arquivo de texto delimitado por dois pontos denominado "movies1.txt", então tudo é maravilhoso. Se eles tiverem um nome diferente para seus arquivos de filmes, será fácil colocar o nome do arquivo em um arquivo de propriedades. Mas e se eles tivessem uma forma completamente diferente de armazenar sua lista de filmes: um banco de dados SQL, um arquivo XML, um serviço da web ou apenas outro formato de arquivo de texto? Nesse caso, precisamos de uma classe diferente para obter esses dados. Agora, como eu defini uma `MovieFinder`interface, isso não alterará meu`moviesDirectedBy`método. Mas ainda preciso encontrar uma maneira de colocar em prática uma instância da implementação do localizador certo.

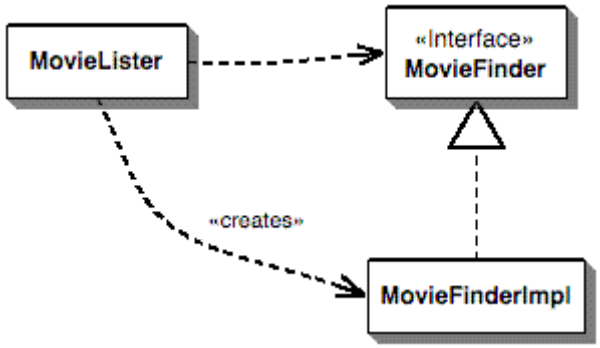

Figura 1: As dependências usando uma criação simples na classe Lister

A Figura 1 mostra as dependências para esta situação. A `MovieLister`classe depende da `MovieFinder`interface e da implementação. Nós preferiríamos que fosse dependente apenas da interface, mas então como fazemos uma instância para trabalhar?

Em meu livro P da EAA , descrevemos essa situação como um plug - in . A classe de implementação do localizador não está vinculada ao programa no momento da compilação, pois não sei o que meus amigos usarão. Em vez disso, queremos que meu professor trabalhe com qualquer implementação e que essa implementação seja conectada em algum momento posterior, fora de minhas mãos. O problema é como posso fazer esse link de modo que minha classe Lister ignore a classe de implementação, mas ainda possa falar com uma instância para fazer seu trabalho.

Expandindo isso em um sistema real, podemos ter dezenas desses serviços e componentes. Em cada caso, podemos abstrair nosso uso desses componentes conversando com eles por meio de uma interface (e usando um adaptador se o componente não for projetado com uma interface em mente). Mas se quisermos implantar esse sistema de maneiras diferentes, precisamos usar plug-ins para lidar com a interação com esses serviços para que possamos usar diferentes implementações em diferentes implantações.

Portanto, o problema central é como montamos esses plug-ins em um aplicativo? Esse é um dos principais problemas que essa nova geração de contêineres leves enfrenta e, universalmente, todos fazem isso usando Inversão de Controle.

Inversão de controle

Quando esses contêineres falam sobre como eles são tão úteis porque implementam "Inversão de Controle", fico muito confuso. A inversão de controle é uma característica comum das estruturas, então dizer que esses contêineres leves são especiais porque usam a inversão de controle é como dizer que meu carro é especial porque tem rodas.

A questão é: "que aspecto do controle eles estão invertendo?" Quando encontrei a inversão de controle pela primeira vez, ela estava no controle principal de uma interface de usuário. As primeiras interfaces de usuário eram controladas pelo programa aplicativo. Você teria uma sequência de comandos como "Digite o nome", "digite o endereço"; seu programa conduziria os prompts e obteria uma resposta para cada um. Com interfaces de usuário gráficas (ou mesmo baseadas em tela), a estrutura da interface de usuário conteria esse loop principal e, em vez disso, seu programa fornecia manipuladores de eventos para os vários campos da tela. O controle principal do programa foi invertido, afastado de você para o framework.

Para este novo tipo de contêiner, a inversão é sobre como eles procuram uma implementação de plug-in. Em meu exemplo ingênuo, o professor pesquisou a implementação do localizador instanciando-a diretamente. Isso impede que o localizador seja um plugin. A abordagem que esses contêineres usam é garantir que qualquer usuário de um plug-in siga alguma convenção que permite que um módulo montador separado injete a implementação na lista.

Como resultado, acho que precisamos de um nome mais específico para esse padrão. Inversão de controle é um termo muito genérico e, portanto, as pessoas o consideram confuso. Como resultado, com muita discussão com vários defensores de IoC, decidimos pelo nome *Injeção de Dependência* .

Vou começar falando sobre as várias formas de injeção de dependência, mas vou apontar agora que essa não é a única maneira de remover a dependência da classe do aplicativo para a implementação do plugin. O outro padrão que você pode usar para fazer isso é Service Locator, e discutirei isso depois de terminar de explicar a injeção de dependência.

Formas de injeção de dependência

A ideia básica da injeção de dependência é ter um objeto separado, um assembler, que preenche um campo na classe lister com uma implementação apropriada para a

interface do localizador, resultando em um diagrama de dependência ao longo das linhas da Figura 2

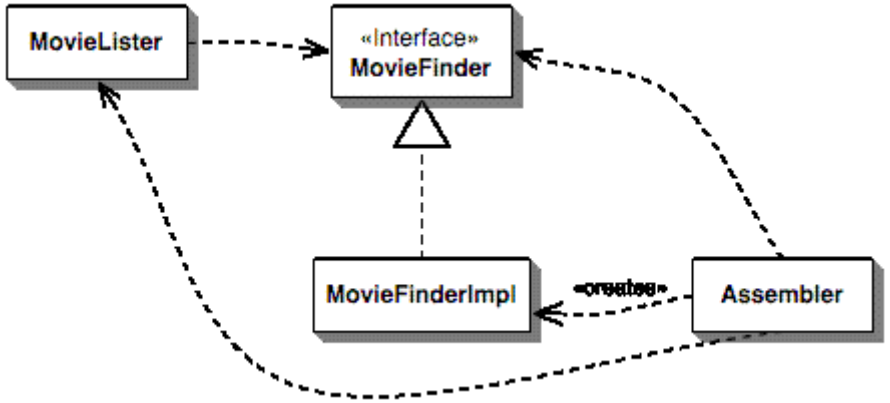

Figura 2: as dependências de um injetor de dependência

Existem três estilos principais de injeção de dependência. Os nomes que estou usando para eles são Injeção de Construtor, Injeção de Setter e Injeção de Interface. Se você leu sobre essas coisas nas discussões atuais sobre Inversão de Controle, você vai ouvir isso referido como IoC tipo 1 (injeção de interface), IoC tipo 2 (injeção setter) e IoC tipo 3 (injeção de construtor). Acho os nomes numéricos bastante difíceis de lembrar, por isso usei os nomes que tenho aqui.

Injeção de construtor com PicoContainer

Vou começar mostrando como essa injeção é feita usando um contêiner leve chamado PicoContainer . Estou começando aqui principalmente porque vários dos meus colegas na ThoughtWorks são muito ativos no desenvolvimento do PicoContainer (sim, é uma espécie de nepotismo corporativo).

O PicoContainer usa um construtor para decidir como injetar uma implementação de localizador na classe de registro. Para que isso funcione, a classe movie lister precisa declarar um construtor que inclui tudo o que precisa injetar.

classe MovieLister ...

```
  public MovieLister (MovieFinder finder) {
      this.finder = localizador;
  }
```

O próprio localizador também será gerenciado pelo contêiner pico e, como tal, terá o nome do arquivo de texto injetado nele pelo contêiner.

classe ColonMovieFinder ...

```
  public ColonMovieFinder (String nome do arquivo) {
      this.filename = filename;
  }
```

O contêiner pico precisa ser informado sobre qual classe de implementação deve ser associada a cada interface e qual string injetar no localizador.

```
private MutablePicoContainer configureContainer () {
    MutablePicoContainer pico = novo DefaultPicoContainer ();
    Parâmetro [] finderParams = {new ConstantParameter ("movies1.txt")};
    pico.registerComponentImplementation (MovieFinder.class, ColonMovieFinder.class, finderParams);
    pico.registerComponentImplementation (MovieLister.class);
    pico de retorno;
}
```

Esse código de configuração normalmente é definido em uma classe diferente. Para nosso exemplo, cada amigo que usa meu lister pode escrever o código de configuração apropriado em alguma classe de configuração própria. Claro, é comum manter esse tipo de informação de configuração em arquivos de configuração separados. Você pode escrever uma classe para ler um arquivo de configuração e

configurar o contêiner apropriadamente. Embora o PicoContainer não contenha essa funcionalidade em si, há um projeto intimamente relacionado chamado NanoContainer que fornece os wrappers apropriados para permitir que você tenha arquivos de configuração XML. Esse nano container irá analisar o XML e então configurar um pico container subjacente. A filosofia do projeto é separar o formato do arquivo de configuração do mecanismo subjacente.

Para usar o contêiner, você escreve um código semelhante a este.

```
public void testWithPico () {
    MutablePicoContainer pico = configureContainer ();
    MovieLister lister = (MovieLister) pico.getComponentInstance (MovieLister.class);
    Filme [] movies = lister.moviesDirectedBy ("Sergio Leone");
    assertEquals ("Era uma vez no oeste", movies [0] .getTitle ());
}
```

Embora neste exemplo eu tenha usado injeção de construtor, o PicoContainer também suporta injeção de setter, embora seus desenvolvedores prefiram injeção de construtor.

Injeção de Setter com Mola

A estrutura Spring é uma estrutura abrangente para desenvolvimento Java empresarial. Inclui camadas de abstração para transações, estruturas de persistência, desenvolvimento de aplicativos da web e JDBC. Como o PicoContainer, ele suporta injeção de construtor e setter, mas seus desenvolvedores tendem a preferir injeção de setter - o que o torna uma escolha apropriada para este exemplo.

Para fazer minha lista de filmes aceitar a injeção, defino um método de configuração para esse serviço

classe MovieLister ...

```
  localizador de MovieFinder privado;
public void setFinder (MovieFinder finder) {
  this.finder = localizador;
}
```

Da mesma forma, defino um setter para o nome do arquivo.

classe ColonMovieFinder ...

```
  public void setFilename (String filename) {
      this.filename = filename;
  }
```

A terceira etapa é definir a configuração dos arquivos. Spring suporta configuração por meio de arquivos XML e também por meio de código, mas XML é a maneira esperada de fazer isso.

```
<beans>
    <bean id = "MovieLister" class = "spring.MovieLister">
        <nome da propriedade = "localizador">
            <ref local = "MovieFinder" />
        </property>
    </bean>
    <bean id = "MovieFinder" class = "spring.ColonMovieFinder">
        <nome da propriedade = "nome do arquivo">
            <value> movies1.txt </value>
        </property>
    </bean>
</beans>
```

O teste fica assim.

```
public void testWithSpring () throws Exception {
    ApplicationContext ctx = new FileSystemXmlApplicationContext ("spring.xml");
    MovieLister lister = (MovieLister) ctx.getBean ("MovieLister");
```

```
    Filme [] movies = lister.moviesDirectedBy ("Sergio Leone");
    assertEquals ("Era uma vez no oeste", movies [0] .getTitle ());
}
```

Injeção de interface

A terceira técnica de injeção é definir e usar interfaces para a injeção. Avalon é um exemplo de estrutura que usa essa técnica em alguns lugares. Falarei um pouco mais sobre isso mais tarde, mas, neste caso, vou usá-lo com alguns códigos de amostra simples.

Com essa técnica, começo definindo uma interface que usarei para realizar a injeção. Esta é a interface para injetar um localizador de filme em um objeto.

```
interface pública InjectFinder {
    void injectFinder (localizador do MovieFinder);
}
```

Essa interface seria definida por quem fornece a interface do MovieFinder. Ele precisa ser implementado por qualquer classe que deseja usar um localizador, como o lister.

classe MovieLister implementa InjectFinder

```
  public void injectFinder (MovieFinder finder) {
      this.finder = localizador;
  }
```

Eu uso uma abordagem semelhante para injetar o nome do arquivo na implementação do localizador.

```
public interface InjectFinderFilename {
    void injectFilename (String nome do arquivo);
}
```

a classe ColonMovieFinder implementa MovieFinder, InjectFinderFilename ...

```
public void injectFilename (String filename) {
    this.filename = filename;
}
```

Então, como de costume, preciso de algum código de configuração para conectar as implementações. Para simplificar, vou fazer isso em código.

testador de classe ...

```
Container container privado;

 private void configureContainer () {
   container = novo Container ();
   registerComponents ();
   registerInjectors ();
   container.start ();
}
```

Esta configuração possui dois estágios, o registro de componentes por meio de chaves de pesquisa é bastante semelhante aos outros exemplos.

testador de classe ...

```
private void registerComponents () {
  container.registerComponent ("MovieLister", MovieLister.class);
  container.registerComponent ("MovieFinder", ColonMovieFinder.class);
}
```

Uma nova etapa é registrar os injetores que injetarão os componentes dependentes. Cada interface de injeção precisa de algum código para injetar o objeto dependente. Aqui eu faço isso registrando objetos injetores com o contêiner. Cada objeto injetor implementa a interface do injetor.

testador de classe ...

```
  private void registerInjectors () {
    container.registerInjector (InjectFinder.class, container.lookup ("MovieFinder"));
    container.registerInjector (InjectFinderFilename.class, new FinderFilenameInjector ());
  }
interface pública Injector {
  public void inject (Object target);

}
```

Quando o dependente é uma classe escrita para este contêiner, faz sentido que o componente implemente a própria interface do injetor, como faço aqui com o localizador de filme. Para classes genéricas, como string, uso uma classe interna dentro do código de configuração.

classe ColonMovieFinder implementa Injector ...

```
  public void inject (Object target) {
    ((InjectFinder) target) .injectFinder (this);
  }
```

testador de classe ...

```
  public static class FinderFilenameInjector implementa Injector {
    public void inject (Object target) {
      ((InjectFinderFilename) target) .injectFilename ("movies1.txt");
    }
    }
```

Os testes então usam o contêiner.

testador de classe ...

```
  public void testIface () {
    configureContainer ();
    MovieLister lister = (MovieLister) container.lookup ("MovieLister");
    Filme [] movies = lister.moviesDirectedBy ("Sergio Leone");
    assertEquals ("Era uma vez no oeste", movies [0] .getTitle ());
  }
```

O contêiner usa as interfaces de injeção declaradas para descobrir as dependências e os injetores para injetar os dependentes corretos. (A implementação específica do contêiner que fiz aqui não é importante para a técnica e não vou mostrá-la porque você só vai rir.)

Usando um localizador de serviço

O principal benefício de um Injetor de Dependência é que ele remove a dependência que a `MovieLister`classe tem da `MovieFinder`implementação concreta . Isso me permite fornecer listas para amigos e para eles conectarem uma implementação adequada para seu próprio ambiente. A injeção não é a única maneira de quebrar essa dependência, outra é usar um localizador de serviço .

A ideia básica por trás de um localizador de serviço é ter um objeto que saiba como obter todos os serviços de que um aplicativo pode precisar. Portanto, um localizador de serviço para este aplicativo teria um método que retorna um localizador de filme quando necessário. Claro que isso apenas desloca um pouco a carga, ainda temos que colocar o localizador no lister, resultando nas dependências da Figura 3

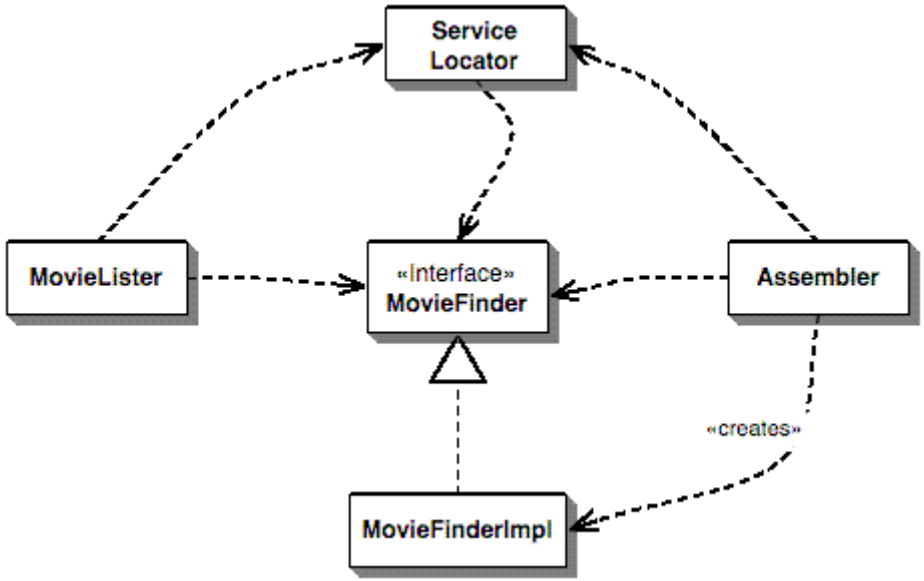

Figura 3: as dependências de um Service Locator

Nesse caso, usarei o ServiceLocator como um registro singleton . O lister pode então usar isso para obter o localizador quando for instanciado.

classe MovieLister ...

```
MovieFinder finder = ServiceLocator.movieFinder ();
```

classe ServiceLocator ...

```
public static MovieFinder movieFinder () {
    return soleInstance.movieFinder;
}
ServiceLocator estático privado soleInstance;
MovieFinder movieFinder privado;
```

Assim como na abordagem de injeção, temos que configurar o localizador de serviço. Aqui estou fazendo isso em código, mas não é difícil usar um mecanismo que lê os dados apropriados de um arquivo de configuração.

testador de classe ...

```
  private void configure () {
      ServiceLocator.load (novo ServiceLocator (novo ColonMovieFinder ("movies1.txt")));
  }
```

classe ServiceLocator ...

```
  public static void load (ServiceLocator arg) {
      soleInstance = arg;
  }

  public ServiceLocator (MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
```

Aqui está o código de teste.

testador de classe ...

```
  public void testSimple () {
      configure ();
      MovieLister lista = novo MovieLister ();
      Filme [] movies = lister.moviesDirectedBy ("Sergio Leone");
      assertEquals ("Era uma vez no oeste", movies [0] .getTitle ());
  }
```

Já ouvi muitas vezes a reclamação de que esses tipos de localizadores de serviço são ruins porque não podem ser testados porque não é possível substituí-los por implementações. Certamente você pode projetá-los mal para ter esse tipo de problema, mas não é necessário. Nesse caso, a instância do localizador de serviço é apenas um simples portador de dados. Posso criar facilmente o localizador com implementações de teste de meus serviços.

Para um localizador mais sofisticado, posso criar uma subclasse de localizador de serviço e passar essa subclasse para a variável de classe do registro. Posso alterar os

métodos estáticos para chamar um método na instância em vez de acessar variáveis de instância diretamente. Posso fornecer localizadores específicos de thread usando armazenamento específico de thread. Tudo isso pode ser feito sem alterar os clientes do localizador de serviço.

Uma maneira de pensar nisso é que o localizador de serviço é um registro, não um singleton. Um singleton fornece uma maneira simples de implementar um registro, mas essa decisão de implementação é facilmente alterada.

Usando uma interface segregada para o localizador

Um dos problemas com a abordagem simples acima é que o MovieLister depende da classe do localizador de serviço completo, embora use apenas um serviço. Podemos reduzir isso usando uma interface de função . Dessa forma, em vez de usar a interface do localizador de serviço completo, o lister pode declarar apenas o pedaço de interface de que precisa.

Nessa situação, o provedor do listador também forneceria uma interface de localizador de que ele precisa para obter o localizador.

```
public interface MovieFinderLocator {
    public MovieFinder movieFinder ();
```

O localizador então precisa implementar essa interface para fornecer acesso a um localizador.

```
Localizador MovieFinderLocator = ServiceLocator.locator ();
Localizador de MovieFinder = locator.movieFinder ();
public static ServiceLocator locator () {
     return soleInstance;
 }
```

```
public MovieFinder movieFinder () {
    return movieFinder;
}
ServiceLocator estático privado soleInstance;
MovieFinder movieFinder privado;
```

Você notará que, como queremos usar uma interface, não podemos mais acessar os serviços por meio de métodos estáticos. Temos que usar a classe para obter uma instância do localizador e, em seguida, usá-la para obter o que precisamos.

Um localizador de serviço dinâmico

O exemplo acima era estático, pois a classe do localizador de serviço possui métodos para cada um dos serviços de que você precisa. Esta não é a única maneira de fazer isso, você também pode criar um localizador de serviço dinâmico que permite armazenar nele qualquer serviço de que você precisa e fazer suas escolhas em tempo de execução.

Nesse caso, o localizador de serviço usa um mapa em vez de campos para cada um dos serviços e fornece métodos genéricos para obter e carregar serviços.

classe ServiceLocator ...

```
ServiceLocator estático privado soleInstance;
public static void load (ServiceLocator arg) {
    soleInstance = arg;
}
serviços de mapa privados = new HashMap ();
public static Object getService (String key) {
    retornar soleInstance.services.get (chave);
}
public void loadService (String key, Object service) {
```

```
        services.put (chave, serviço);
    }
```

A configuração envolve o carregamento de um serviço com uma chave apropriada.

testador de classe ...

```
    private void configure () {
        Localizador ServiceLocator = novo ServiceLocator ();
        locator.loadService ("MovieFinder", novo ColonMovieFinder ("movies1.txt"));
        ServiceLocator.load (localizador);
    }
```

Eu uso o serviço usando a mesma string de chave.

classe MovieLister ...

```
    MovieFinder finder = (MovieFinder) ServiceLocator.getService ("MovieFinder");
```

De modo geral, não gosto dessa abordagem. Embora seja certamente flexível, não é muito explícito. A única maneira de descobrir como acessar um serviço é por meio de chaves textuais. Eu prefiro métodos explícitos porque é mais fácil descobrir onde eles estão examinando as definições de interface.

Usando um localizador e injeção com Avalon

A injeção de dependência e um localizador de serviço não são conceitos necessariamente mutuamente exclusivos. Um bom exemplo de uso de ambos juntos é a estrutura Avalon. Avalon usa um localizador de serviço, mas usa injeção para dizer aos componentes onde encontrar o localizador.

Berin Loritsch me enviou esta versão simples do meu exemplo de execução usando Avalon.

```
public class MyMovieLister implementa MovieLister, Serviceable {
    localizador de MovieFinder privado;

    serviço void público (gerenciador ServiceManager) aciona ServiceException {
        localizador = (MovieFinder) manager.lookup ("localizador");
    }
```

O método de serviço é um exemplo de injeção de interface, permitindo que o contêiner injete um gerenciador de serviço em MyMovieLister. O gerenciador de serviço é um exemplo de localizador de serviço. Neste exemplo, o lister não armazena o gerenciador em um campo, em vez disso, ele o usa imediatamente para pesquisar o localizador, que ele armazena.

Decidir qual opção usar

Até agora, concentrei-me em explicar como vejo esses padrões e suas variações. Agora posso começar a falar sobre seus prós e contras para ajudar a descobrir quais usar e quando.

Localizador de serviço vs injeção de dependência

A escolha fundamental é entre Service Locator e Dependency Injection. O primeiro ponto é que ambas as implementações fornecem o desacoplamento fundamental que

está faltando no exemplo ingênuo - em ambos os casos, o código do aplicativo é independente da implementação concreta da interface de serviço. A diferença importante entre os dois padrões é sobre como essa implementação é fornecida para a classe do aplicativo. Com o localizador de serviço, a classe de aplicativo o solicita explicitamente por meio de uma mensagem ao localizador. Com injeção não há solicitação explícita, o serviço aparece na classe do aplicativo - daí a inversão de controle.

A inversão de controle é uma característica comum das estruturas, mas é algo que tem um preço. Tende a ser difícil de entender e causa problemas quando você está tentando depurar. Portanto, no geral, prefiro evitá-lo, a menos que seja necessário. Isso não quer dizer que seja uma coisa ruim, apenas que acho que precisa se justificar em relação à alternativa mais direta.

A principal diferença é que, com um localizador de serviço, cada usuário de um serviço depende do localizador. O localizador pode ocultar dependências para outras implementações, mas você precisa ver o localizador. Portanto, a decisão entre o localizador e o injetor depende se essa dependência é um problema.

Usar injeção de dependência pode ajudar a tornar mais fácil ver quais são as dependências do componente. Com o injetor de dependência, você pode apenas olhar para o mecanismo de injeção, como o construtor, e ver as dependências. Com o localizador de serviço, você deve pesquisar o código-fonte das chamadas para o localizador. IDEs modernos com um recurso de localização de referências tornam isso mais fácil, mas ainda não é tão fácil quanto olhar para o construtor ou métodos de configuração.

Muito disso depende da natureza do usuário do serviço. Se você estiver criando um aplicativo com várias classes que usam um serviço, uma dependência das classes do aplicativo para o localizador não é um grande problema. No meu exemplo de dar um

Movie Lister para meus amigos, usar um localizador de serviço funciona muito bem. Tudo o que eles precisam fazer é configurar o localizador para conectar as implementações de serviço corretas, por meio de algum código de configuração ou de um arquivo de configuração. Nesse tipo de cenário, não vejo a inversão do injetor como fornecendo algo atraente.

A diferença vem se o lister for um componente que estou fornecendo para um aplicativo que outras pessoas estão escrevendo. Nesse caso, não sei muito sobre as APIs dos localizadores de serviço que meus clientes irão usar. Cada cliente pode ter seus próprios localizadores de serviço incompatíveis. Posso contornar isso usando a interface segregada. Cada cliente pode escrever um adaptador que corresponda à minha interface ao seu localizador, mas, em qualquer caso, ainda preciso ver o primeiro localizador para pesquisar minha interface específica. E uma vez que o adaptador aparece, a simplicidade da conexão direta com um localizador começa a falhar.

Visto que com um injetor você não tem uma dependência de um componente para o injetor, o componente não pode obter mais serviços do injetor depois de configurado.

Um motivo comum que as pessoas dão para preferir a injeção de dependência é que isso torna o teste mais fácil. O ponto aqui é que, para fazer o teste, você precisa substituir facilmente as implementações de serviço reais por stubs ou simulações. No entanto, não há realmente nenhuma diferença aqui entre injeção de dependência e localizador de serviço: ambos são muito receptivos a stub. Suspeito que essa observação venha de projetos em que as pessoas não se esforçam para garantir que seu localizador de serviço possa ser facilmente substituído. É aqui que o teste contínuo ajuda, se você não conseguir fazer o stub dos serviços para teste, isso implica em um problema sério com seu design.

É claro que o problema de teste é exacerbado por ambientes de componentes que são muito intrusivos, como a estrutura EJB do Java. Minha opinião é que esses tipos de estruturas devem minimizar seu impacto sobre o código do aplicativo e, particularmente, não devem fazer coisas que retardem o ciclo de edição-execução. Usar plug-ins para substituir componentes pesados ajuda muito nesse processo, que é vital para práticas como o Desenvolvimento Orientado a Testes.

Portanto, o problema principal é para pessoas que estão escrevendo um código que espera ser usado em aplicativos fora do controle do redator. Nesses casos, até mesmo uma suposição mínima sobre um Service Locator é um problema.

Injeção de construtor versus setter

Para combinação de serviço, você sempre precisa ter alguma convenção para conectar as coisas. A vantagem da injeção é principalmente que ela requer convenções muito simples - pelo menos para as injeções de construtor e setter. Você não precisa fazer nada estranho em seu componente e é bastante simples para um injetor configurar tudo.

A injeção de interface é mais invasiva, pois você precisa escrever muitas interfaces para resolver tudo. Para um pequeno conjunto de interfaces exigidas pelo contêiner, como na abordagem do Avalon, isso não é tão ruim. Mas é muito trabalhoso montar componentes e dependências, e é por isso que a safra atual de contêineres leves vem com injeção de setter e construtor.

A escolha entre injeção de setter e construtor é interessante, pois reflete um problema mais geral com a programação orientada a objetos - você deve preencher campos em um construtor ou com setters.

Meu padrão de longa duração com objetos é o máximo possível, para criar objetos válidos no momento da construção. Este conselho remonta aos Padrões de práticas recomendadas Smalltalk de Kent Beck : Método do construtor e Método de parâmetro do construtor. Construtores com parâmetros fornecem uma declaração clara do que significa criar um objeto válido em um lugar óbvio. Se houver mais de uma maneira de fazer isso, crie vários construtores que mostram as diferentes combinações.

Outra vantagem da inicialização do construtor é que ela permite ocultar claramente quaisquer campos que sejam imutáveis simplesmente não fornecendo um setter. Eu acho que isso é importante - se algo não deve mudar, então a falta de um levantador comunica isso muito bem. Se você usar setters para inicialização, isso pode se tornar uma dor. (De fato, nessas situações, prefiro evitar a convenção de configuração usual, prefiro um método como `initFoo`, para enfatizar que é algo que você só deve fazer no nascimento.)

Mas, em qualquer situação, há exceções. Se você tiver muitos parâmetros de construtor, as coisas podem parecer confusas, principalmente em linguagens sem parâmetros de palavra-chave. É verdade que um construtor longo geralmente é um sinal de um objeto muito ocupado que deve ser dividido, mas há casos em que é disso que você precisa.

Se você tem várias maneiras de construir um objeto válido, pode ser difícil mostrar isso por meio de construtores, uma vez que os construtores podem variar apenas no número e tipo de parâmetros. É quando os Factory Methods entram em ação, eles podem usar uma combinação de construtores e configuradores privados para implementar seu trabalho. O problema com os Métodos de Fábrica clássicos para montagem de componentes é que eles geralmente são vistos como métodos estáticos e você não pode tê-los em interfaces. Você pode criar uma classe de fábrica, mas isso

se torna apenas outra instância de serviço. Um serviço de fábrica geralmente é uma boa tática, mas você ainda precisa instanciar a fábrica usando uma das técnicas aqui.

Os construtores também sofrem se você tiver parâmetros simples, como strings. Com a injeção de setter, você pode dar a cada setter um nome para indicar o que a string deve fazer. Com os construtores, você está apenas contando com a posição, que é mais difícil de seguir.

Se você tiver vários construtores e herança, as coisas podem ficar particularmente complicadas. Para inicializar tudo, você deve fornecer construtores para encaminhar para cada construtor da superclasse, enquanto também adiciona seus próprios argumentos. Isso pode levar a uma explosão ainda maior de construtores.

Apesar das desvantagens, minha preferência é começar com a injeção de construtor, mas esteja pronto para mudar para a injeção de setter assim que os problemas que descrevi acima começarem a se tornar um problema.

Esse problema gerou muito debate entre as várias equipes que fornecem injetores de dependência como parte de seus frameworks. No entanto, parece que a maioria das pessoas que constroem essas estruturas percebeu que é importante oferecer suporte a ambos os mecanismos, mesmo que haja uma preferência por um deles.

Arquivos de código ou configuração

Uma questão separada, mas freqüentemente confundida, é se usar arquivos de configuração ou código em uma API para conectar serviços. Para a maioria dos aplicativos que provavelmente serão implantados em muitos lugares, um arquivo de configuração separado geralmente faz mais sentido. Quase sempre será um arquivo XML e isso faz sentido. No entanto, há casos em que é mais fácil usar o código do

programa para fazer a montagem. Um caso é quando você tem um aplicativo simples que não tem muitas variações de implantação. Nesse caso, um pouco de código pode ser mais claro do que um arquivo XML separado.

Um caso contrastante é quando a montagem é bastante complexa, envolvendo etapas condicionais. Depois que você começa a se aproximar da linguagem de programação, o XML começa a quebrar e é melhor usar uma linguagem real que tenha toda a sintaxe para escrever um programa claro. Em seguida, você escreve uma classe de construtor que faz a montagem. Se você tiver cenários de construtor distintos, poderá fornecer várias classes de construtor e usar um arquivo de configuração simples para selecionar entre elas.

Costumo pensar que as pessoas estão ansiosas demais para definir arquivos de configuração. Freqüentemente, uma linguagem de programação é um mecanismo de configuração simples e poderoso. Linguagens modernas podem facilmente compilar pequenos montadores que podem ser usados para montar plug-ins para sistemas maiores. Se a compilação for uma dor, então existem linguagens de script que também podem funcionar bem.

Costuma-se dizer que os arquivos de configuração não devem usar uma linguagem de programação porque precisam ser editados por não-programadores. Mas com que frequência é esse o caso? As pessoas realmente esperam que os não-programadores alterem os níveis de isolamento da transação de um aplicativo complexo do lado do servidor? Os arquivos de configuração fora do idioma funcionam bem apenas na medida em que são simples. Se eles se tornarem complexos, é hora de pensar em usar uma linguagem de programação adequada.

Uma coisa que estamos vendo no mundo Java no momento é uma cacofonia de arquivos de configuração, onde cada componente tem seus próprios arquivos de configuração, que são diferentes dos demais. Se você usar uma dúzia desses

componentes, pode facilmente acabar com uma dúzia de arquivos de configuração para manter em sincronia.

Meu conselho aqui é sempre fornecer uma maneira de fazer todas as configurações facilmente com uma interface programática e, em seguida, tratar um arquivo de configuração separado como um recurso opcional. Você pode construir facilmente o manuseio do arquivo de configuração para usar a interface programática. Se você estiver escrevendo um componente, você deixará que seu usuário use a interface programática, o formato do arquivo de configuração ou escreva seu próprio formato de arquivo de configuração personalizado e o vincule à interface programática

Separando a configuração do uso

A questão importante em tudo isso é garantir que a configuração dos serviços seja separada de seu uso. Na verdade, este é um princípio de design fundamental que acompanha a separação das interfaces da implementação. É algo que vemos em um programa orientado a objetos quando a lógica condicional decide qual classe instanciar e, em seguida, as avaliações futuras dessa condicional são feitas por meio de polimorfismo em vez de por código condicional duplicado.

Se essa separação for útil em uma única base de código, será especialmente vital quando você estiver usando elementos externos, como componentes e serviços. A primeira pergunta é se você deseja adiar a escolha da classe de implementação para implementações específicas. Se sim, você precisa usar alguma implementação de plugin. Depois de usar os plug-ins, é essencial que a montagem dos plug-ins seja feita separadamente do restante do aplicativo para que você possa substituir facilmente diferentes configurações por diferentes implantações. Como você consegue isso é

secundário. Este mecanismo de configuração pode configurar um localizador de serviço ou usar injeção para configurar objetos diretamente.

Alguns outros problemas

Neste artigo, concentrei-me nas questões básicas de configuração de serviço usando Dependency Injection e Service Locator. Existem mais alguns tópicos relacionados a isso que também merecem atenção, mas ainda não tive tempo para aprofundar. Em particular, há a questão do comportamento do ciclo de vida. Alguns componentes têm eventos de ciclo de vida distintos: parar e iniciar, por exemplo. Outro problema é o crescente interesse em usar ideias orientadas a aspectos com esses containers. Embora eu não tenha considerado esse material no artigo no momento, espero escrever mais sobre isso estendendo este artigo ou escrevendo outro.

Você pode descobrir muito mais sobre essas idéias nos sites dedicados aos contêineres leves. Navegar nos sites do picocontainer e do spring levará você a uma discussão muito mais aprofundada sobre essas questões e a um início de algumas das outras questões.

Pensamentos Finais

Todo o fluxo atual de contêineres leves tem um padrão subjacente comum de como eles fazem a montagem do serviço - o padrão injetor de dependência. A injeção de dependência é uma alternativa útil para o Service Locator. Ao construir classes de aplicativo, os dois são aproximadamente equivalentes, mas acho que o Service Locator tem uma pequena vantagem devido ao seu comportamento mais direto. No entanto, se você estiver criando classes para serem usadas em vários aplicativos, a injeção de dependência é a melhor escolha.

Se você usar injeção de dependência, há vários estilos para escolher. Eu sugeriria que você siga a injeção de construtor, a menos que encontre um dos problemas específicos dessa abordagem, nesse caso, mude para injeção de setter. Se você está escolhendo construir ou obter um contêiner, procure um que suporte a injeção de construtor e setter.

A escolha entre Service Locator e Dependency Injection é menos importante do que o princípio de separar a configuração de serviço do uso de serviços em um aplicativo.

Agradecimentos

Meus sinceros agradecimentos às muitas pessoas que me ajudaram com este artigo. Rod Johnson, Paul Hammant, Joe Walnes, Aslak Hellesøy, Jon Tirsén e Bill Caputo ajudaram-me a entender esses conceitos e comentaram os primeiros rascunhos deste artigo. Berin Loritsch e Hamilton Verissimo de Oliveira deram alguns conselhos muito úteis sobre como Avalon se encaixa. Dave W Smith persistiu em fazer perguntas sobre meu código de configuração de injeção de interface inicial e, portanto, me fez confrontar o fato de que era estúpido. Gerry Lowry me enviou várias correções de erros de digitação - o suficiente para ultrapassar o limite de agradecimento.

▶ Revisões Significativas