dependency injection

in delphi

```
type
  TFileDisplayerRegistry = class(TInterfacedObject, IFileDisplayerRe
  private
    FDictionary: IDictionary<string, IDisplayFile>;
    FDefaultDisplayer: IDisplayFile;
  public
    constructor Create;
    procedure AddDisplayer(const aExt: string; aDisplayer: IDisplayF
    function GetDisplayer(const aExt: string): IDisplayFile;
    function GetExtensions: TArray<string>;
    property DefaultDisplayer: IDisplayFile read FDefaultDisplayer w
  end;
```

Nick Hodges

Technical Review by Stefan Glienke

Injeção de dependência em Delphi

Nick Hodges

Este livro está à venda em http://leanpub.com/dependencyinjectionindelphi

Esta versão foi publicada em 2017-02-25

ISBN 978-1-941266-19-9

Este é um Leanpub livro. Leanpub capacita autores e editores com o processo Lean Publishing. Publicação enxuta é o ato de publicar um e-book em andamento usando ferramentas leves e muitas iterações para obter feedback do leitor, girar até que você tenha o livro certo e criar tração assim que o fizer.

À minha maravilhosa e linda esposa Pamela. Eu amo Você.

Conteúdo

Prefácio

Por que um livro inteiro sobre Dependency Injection (DI) em Delphi, especialmente desde que escrevi sobre isso no meu primeiro livro Coding in Delphi?

Bem, existem algumas razões para isso.

Primeiro, vejo agora que a codificação em Delphi apenas arranhou a superfície do que é injeção de dependência e como fazê-lo corretamente. Isso não quer dizer que as informações nele contidas não sejam úteis, mas muita coisa mudou nos últimos anos em relação à DI, e eu também aprendi muito nos anos seguintes. Este livro tem como objetivo atualizá-lo com os últimos desenvolvimentos em injeção de dependência usando Delphi.

Em segundo lugar, houve uma série de novos recursos muito poderosos adicionados ao contêiner Spring for Delphi Dependency Injection que precisam ser discutidos. Stefan Gleinke, o mantenedor e desenvolvedor do Spring4D, não ficou ocioso e acrescentou muito ao Container nos últimos anos. Seu grande trabalho merece mais atenção.

E terceiro, quero escrever este livro porque ainda vejo pessoas fazendo tentativas incipientes de fazer a Injeção de Dependência e lutam para fazê-lo bem. Eu os vejo tentando usá-lo para gerenciar seus formulários – algo que não recomendo. Eu os vejo não aplicando DI básico e pulando direto para usar o DI Container no Spring4D antes de entender por que e como usá-lo. E quando eles usam um contêiner, eles tentam forçar coisas que não pertencem a ele. Tudo isso implora para ser explicado e corrigido.

Este livro existe para abordar todas essas questões. Meu primeiro objetivo é cobrir o tópico completa e completamente – de cima para baixo, da esquerda para a direita e da frente para trás. Pretendo fazer isso explicando tudo o que há para saber sobre injeção de dependência.
Meu segundo objetivo é mostrar a você o uso adequado de um contêiner DI, bem como mostrar como não usá-lo e demonstrar essas coisas usando o Spring4D Container. E, finalmente, gostaria que este livro se tornasse a palavra final para desenvolvedores Delphi sobre como fazer injeção de dependência.

A injeção de dependência é na verdade a coisa mais simples do mundo, mas às vezes dá muito trabalho entender coisas simples. Posso dizer sem vergonha que sei muito

mais sobre isso do que quando escrevi sobre isso em meu primeiro livro. Lá dediquei apenas dois capítulos ao tema. Aqui vou dedicar um livro inteiro a ele. Vou me esforçar para ajudar “a lâmpada a acender” sobre a Injeção de Dependência, para que seu uso se torne uma segunda natureza para você. Eu o ajudarei a ver os antipadrões que são muito usados e mostrarei os padrões apropriados para construir aplicativos fracamente acoplados.

A maior parte do material neste livro será novo. Não usarei os mesmos exemplos de Codificação em Delphi, mas você pode ver alguns dos mesmos princípios discutidos nesse livro também abordados aqui.

Eu acho que também é importante notar que este é realmente um livro sobre injeção de dependência que usa Delphi como a linguagem para as demos. Pelo menos, eu quero que seja isso. Os princípios discutidos aqui são verdadeiros para qualquer linguagem de desenvolvimento, então se você por acaso for um desenvolvedor C# ou um usuário de alguma linguagem diferente de Object Pascal que esteja lendo isso, não tenha medo. O código Delphi é fácil de ler e entender, e você deve ser capaz de aprender tudo o que tenho a dizer sem saber um pingo de Delphi.

Este livro está dividido em duas partes: uma que discute a injeção de dependência e outra que discute o uso do que é comumente chamado de DI Container. A DI não exige o uso de container, como vou enfatizar muitas vezes daqui para frente.

Este livro também fará uso extensivo do Spring for Delphi Framework. Usarei o DI Container do Spring4D, a classe Guard e talvez algumas das coleções.
Se você não adicionou Spring4D ao seu arsenal, deve fazê-lo imediatamente. Sua excelente implementação de coleções genéricas baseadas em interface já vale a pena.
O Spring4D Framework pode ser encontrado em http://www.spring4d.org.

O código para este livro está disponível neste link: http://bit.ly/diidcode

Todo o código deste livro foi escrito e compilado com Delphi 10.1 Berlin. Não posso prometer que funcionará em qualquer versão anterior, mas não vejo por que não funcionaria.

Ok, então vamos ao que interessa. Você não tem nada a perder além do seu código de espaguete.

Nick Hodges

Gilbertsville, PA

fevereiro de 2017

Reconhecimentos

Em primeiro lugar, gostaria de agradecer à minha esposa, Pamela, que apoiou este esforço do início ao fim.

Gostaria de agradecer a Baoquan Zuo que originalmente escreveu o framework Spring para Delphi, incluindo o Dependency Injection Container. Suas contribuições para a comunidade Delphi são muitas e muito apreciadas.

Eu também gostaria de agradecer Stefan Glienke por todo o seu grande trabalho na manutenção do Spring para Delphi depois de substituí-lo de Baoquan. Sem você, não teríamos o framework Spring for Delphi como é hoje.

Além disso, sou profundamente grato a Stefan por sua revisão técnica cuidadosa e completa deste livro. É muito melhor como resultado.

Bruce McGee gentilmente leu o rascunho e forneceu um feedback inestimável que melhorou muito o livro. Sou grato pela ajuda dele.

O que é injeção de dependência?

Introdução

Eu tenho três filhos. Eles são mais velhos agora, mas quando eram mais jovens, era divertido assar com eles. Nós assávamos um bolo ou biscoitos ou algum outro doce que eles gostavam de comer. Mas havia um problema – fizemos uma bagunça enorme quando assamos. Havia farinha e açúcar e todos os tipos de coisas por todo o lugar e em cima de nós. O processo de assar nos deixou com um grande trabalho de limpeza. Deu muito trabalho. Claro, nos divertimos, mas a bagunça estava lá, no entanto.

Como evitar a confusão? Bem, poderíamos ser mais cuidadosos, suponho, mas crianças serão crianças. Muitas vezes, estávamos assando um bolo de aniversário. Uma alternativa, é claro, para todo o incômodo e bagunça de fazer um bolo seria ir à loja e comprar um bolo pré-fabricado. Uma solução ainda melhor seria encontrar uma padaria que entregasse bolos de aniversário sob encomenda, deixando à sua porta exatamente o que você queria para o seu bolo de aniversário.

Agora, aqui está outra coisa a considerar. Digamos que o entregador apareça e esteja pisando na lama e suas botas estejam sujas, e ele está com uma tosse terrível. Você vai deixá-lo entrar na sua casa? Não, claro que não. Você provavelmente daria uma olhada nele pela fresta da porta e lhe diria para colocar a caixa de bolo na varanda e dar o fora do seu gramado. Você teria muito cuidado com o bolo, também, eu acho, mas por uma questão de argumento vamos supor que está bem embrulhado e protegido de qualquer catarro que o entregador estivesse emitindo. Em outras palavras, você deseja que sua interação com o entregador seja a mínima possível, mas ainda quer o bolo. Seu filho vai ficar muito chateado se ele ou ela não conseguir apagar as velas do bolo de aniversário. Você quer o bolo sem bagunça e com a menor interação com o entregador.

Pode-se até dizer que você tinha uma dependência de um bolo de aniversário, e a padaria estava injetando essa dependência entregando o bolo para você. Hmm.

Ou digamos que você está no supermercado e tem um carrinho cheio de mantimentos.
Você os leva para o caixa, e ele liga para todos. Ele diz "Isso vai ser $ 123,45". Então, o que você faz?
Você entrega sua carteira a ele e pede para ele procurar dinheiro ou cartão de crédito? Claro que não – você quer manter a interação o mínimo possível com sua carteira – você não quer que ele fuça nela. Quem sabe o que irá acontecer. Em vez disso, você dá dinheiro ao cara ou pega seu cartão de crédito e entrega a ele. Ou melhor ainda, você mesmo passa o cartão para que o caixa nem toque no seu cartão. Novamente, você quer que essa interação seja a mínima possível. Você pode dizer que deseja manter a interface entre você e o funcionário no mínimo.

Mais um. Digamos que você esteja pensando em comprar uma casa. Você procura por toda a cidade e finalmente encontra um lugar que realmente gosta. Boa localização, bom distrito escolar, boa vizinhança. Só há um problema - todos os dispositivos elétricos estão conectados à casa. Todas as lâmpadas, a torradeira, o secador de cabelo, tudo é cabeado. Não há tomadas na casa. Tudo é conectado diretamente ao sistema elétrico, então você não pode substituir nada facilmente. As interfaces normais – plugues elétricos – entre os dispositivos elétricos e o sistema elétrico simplesmente não existem. Você não pode substituir sua torradeira sem chamar um eletricista. Pode-se dizer que a falta de interfaces tornou as coisas muito difíceis de lidar e, assim, você decide não comprar a casa.

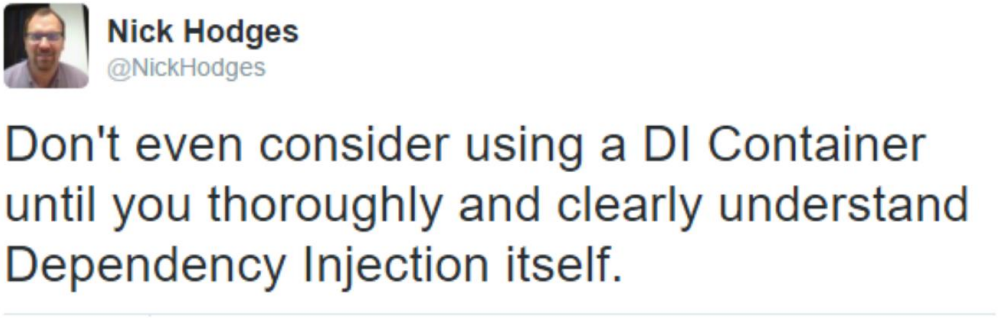
Você está vendo um padrão aqui?

Ok, chega de histórias. Vou apenas dizer. Em seu código, a interação entre os objetos deve ser o mais fina e limpa possível. Você deve pedir exatamente o que você precisa e nada mais. Se você precisa de algo, você deve pedir – faça com que seja “entregue” a você – em vez de tentar criá-lo você mesmo. Dessa forma, assim como com o bolo, o caixa e a casa, as coisas ficam arrumadas, limpas e flexíveis. Acho que o código que é limpo, limpo e flexível soa muito bem.

Essa é a essência da injeção de dependência. Realmente não é nada mais do que isso.
Injeção de Dependência é um termo de vinte e cinco dólares para um conceito de dez centavos: Se você precisar de algo, peça. Não crie coisas você mesmo – empurre isso para outra pessoa.
Cada classe de qualquer substância provavelmente precisará da ajuda de outras classes. Se sua classe precisar de ajuda, peça; não tente “faça você mesmo”. Lembre-se, toda vez que você tenta “faça você mesmo” você cria uma dependência codificada. E, novamente, esses devem ser evitados como variáveis globais.

É, como veremos, realmente tão simples.

Uma coisa a notar – e este é um ponto sobre o qual insisto ao longo da primeira parte do livro – é que ainda não usamos a palavra “Contêiner”. Na verdade, aqui está um tweet meu:



Um tweet muito sábio

Isso pode soar um pouco estranho, mas faz um ponto importante: “Fazendo injeção de dependência” e “Usando um contêiner de injeção de dependência” são realmente duas coisas muito diferentes. Eles estão relacionados – o último tornando o primeiro mais fácil de escalar – mas não são **a** mesma coisa. Na verdade, vou cobrir muito território antes de falar sobre o DI Container novamente.

Então, o que exatamente é a injeção de dependência?

Até agora você provavelmente está se perguntando o que é exatamente DI. Bem, de acordo com Mark Seemann, autor do maravilhoso livro “Dependency Injection in .Net”, Dependency Injection é “um conjunto de princípios e padrões de design de software que nos permitem desenvolver código fracamente acoplado”. Essa é uma boa definição, mas um pouco anti-séptica. Vamos explorar um pouco mais profundamente.

Se você vai injetar uma dependência, você precisa saber o que é uma dependência. Uma dependência é qualquer coisa que uma determinada classe precisa para fazer seu trabalho, como campos, outras classes, etc. Se TClassA precisa que TClassB esteja presente para compilar, então TClassA é dependente de TClassB. Ou em outras palavras, TClassB é uma dependência de TClassA.
As dependências são criadas quando, bem, você cria uma. Aqui está um exemplo:

```
tipo

  TClassB = classe
  fim;

  TClassA = classe
  privado
    FClassB: TClassB;
  público
    construtor Criar; fim;


construtor TClassA.Create; começar
FClassB : = TClassB.Create;

fim;
```

No código acima, criamos uma dependência codificada para TClassB em TClassA.
É como o secador de cabelo ou a torradeira discutidos anteriormente – está conectado à classe. TClassA é completamente dependente de TClassB. TClassB é “acoplado” a TClassA.
Fortemente acoplado. Tão fortemente acoplado quanto você pode obter, realmente. E o acoplamento apertado é ruim.

Uma palavra sobre acoplamento Como

vimos acima, acoplamento é a noção de uma coisa ser dependente de outra. Acoplamento forte é quando as coisas **realmente** dependem umas das outras e estão estritamente ligadas. Você não pode compilar a classe A sem que a definição completa da classe B esteja presente. B está permanentemente preso a A. O acoplamento apertado é ruim – ele cria um código inflexível. Pense em como seria difícil se mover se você estivesse algemado a outra pessoa.
É assim que uma classe se sente quando você cria dependências codificadas.

O que você quer é manter o acoplamento entre suas classes e módulos o mais “solto” possível. Ou seja, você quer que suas dependências sejam tão finas quanto possível. Se você tiver uma classe que precisa de um nome de cliente, passe apenas o nome do cliente. Não passe no cliente inteiro. E como veremos, se sua classe precisar de outra classe, passe uma abstração – uma interface, normalmente – dessa classe. Uma interface é como um fio de fumaça – há algo lá, mas você realmente não consegue segurá-lo.

> Essa noção é tão importante que a discutiremos mais detalhadamente no próximo capítulo.

Como você cria essas dependências codificadas e fortemente acopladas? Você os cria, bem, criando coisas dentro de outras classes. Veja o exemplo acima. O construtor de TClassA cria uma instância de TClassB e a armazena internamente. Essa chamada Create cria a dependência. Você precisa de TClassB para compilar TClassA.

O que fazer?

Bem, a primeira coisa a fazer é não criar TClassB dentro de TClassA. Em vez disso, “injete” a dependência por meio do construtor:

```
tipo

  TClassB = classe
  fim;

  TClassA = classe
  privado
    FClassB: TClassB;
  construtor público
    Create (aClassB: TClassB); fim;


construtor TClassA.Create (aClassB: TClassB); começar

  FClassB := aClassB;
fim;
```

Ao fazer isso, você afrouxa um pouco o acoplamento. Primeiro, observe que agora você pode passar em qualquer instância de TClassB que desejar. Você pode até passar em um descendente se isso fizer sentido. Você ainda está vinculado ao TClassB, e o TClassA ainda não pode ser compilado sem o TClassB, mas você adicionou alguma flexibilidade ao afrouxar o acoplamento apenas um toque. TClassA e TClassB ainda são acoplados, mas são mais fracamente acoplados.

> Então, neste ponto, temos dois tipos de acoplamento: um onde a primeira classe realmente cria uma instância da segunda classe e outro onde a primeira classe precisa da segunda classe, ou seja, onde ela é injetada. Este último é

> preferível ao primeiro porque envolve menos acoplamento (mais solto). No próximo capítulo, veremos a hierarquia do acoplamento – um conceito chamado "connascência".

E isso, amigos, é a essência do DI. Injete suas dependências em vez de criá-las. É isso. Isso é injeção de dependência. Eu poderia parar por aqui, e se isso fosse a única coisa que eu te ensinasse, seu código estaria melhor do que está hoje (porque, convenhamos, sua base de código está cheia de dependências codificadas, certo?). Eu poderia terminar o livro aqui e você teria uma nova ferramenta em sua caixa de ferramentas que realmente melhoraria as coisas.

Mas, é claro, não é tão simples assim. Há mais do que isso e, à medida que os projetos aumentam, as coisas ficam complicadas, mas, como primeiro exemplo, é um longo caminho para descrever o que é a DI e como ela funciona. Nos próximos capítulos, vamos explorá-la mais profundamente, mas se você entendeu e viu o poder dessa técnica simples, você está no caminho certo para escrever um código melhor, mais limpo, mais fácil de manter e mais flexível.

Princípios Básicos a Seguir

Existem alguns princípios subjacentes que permeiam este livro e têm relação com o tópico de injeção de dependência. Eles são os seguintes:

Código contra abstrações, não implementações

Erich Gamma da "Gang of Four" (os autores do livro "Design Patterns") é creditado por cunhar esta frase, e é uma ideia poderosa e essencial. Se você ensina aos novos desenvolvedores apenas uma coisa, deve ser esse aforismo. As abstrações – geralmente interfaces, mas nem sempre (veja abaixo) – são flexíveis. Interfaces (ou classes abstratas) podem ser implementadas de várias maneiras. As interfaces podem ser codificadas antes mesmo que a implementação seja concluída. Se você codificar para uma implementação, estará criando um sistema fortemente acoplado e inflexível. Não se prenda a uma única implementação. Em vez disso, use abstrações e permita que seu código seja flexível, reutilizável e flexível.

Nunca crie coisas que não deveriam ser criadas

Suas aulas devem seguir o Princípio da Responsabilidade Única – a ideia de que uma aula deve fazer apenas uma coisa. Se eles fazem isso, então eles não deveriam estar criando coisas, porque isso faz duas coisas que eles estão fazendo. Em vez disso, eles devem solicitar a funcionalidade de que precisam e deixar que outra coisa crie e forneça essa funcionalidade.

Criativos vs. Injetáveis

Então, o que deve ser criado? Bem, existem dois tipos diferentes de objetos com os quais devemos nos preocupar: “Criatables” e “Injectables”.

Criativos são classes que você deve seguir em frente e criar. São classes RTL ou utilitárias que são comuns e bem conhecidas. Para desenvolvedores Delphi, são coisas como TStringList e TList<T>. Geralmente, as classes no tempo de execução do Delphi devem ser consideradas Creatables. Classes como esta não devem ser injetadas, mas devem ser criadas por suas classes. Eles geralmente têm vidas curtas, frequentemente vivendo não mais do que o período de um único método. Se eles forem requeridos pela classe como um todo, eles podem ser criados no construtor. Deve-se passar apenas outros Creatables para o construtor de um Creatable.

Injetáveis, por outro lado, são classes que nunca queremos criar diretamente. Eles são os tipos de classes para os quais nunca queremos codificar uma dependência e que sempre devem ser passados via injeção de dependência. Eles normalmente serão solicitados como dependências em um construtor. Seguindo a regra acima, injetáveis devem ser referenciados por meio de interfaces e não referências diretas a uma instância. Os injetáveis geralmente serão classes que você escreve como parte de sua lógica de negócios. Eles devem estar sempre escondidos atrás de uma abstração, geralmente uma interface. Observe também que os injetáveis podem solicitar outros injetáveis em seu construtor.

Mantenha os construtores simples

Construtores devem ser mantidos simples. O construtor de uma classe não deve estar fazendo nenhum “trabalho” – ou seja, ele não deve estar fazendo nada além de verificar nil, criar Creatables e armazenar dependências para uso posterior. Não devem incluir

qualquer lógica de codificação. Uma cláusula 'if' no construtor de uma classe que não está verificando nil é um grito para que essa classe seja dividida em duas classes. (Existem maneiras de verificar parâmetros de valor nil que não envolvem uma instrução if . Abordaremos a noção de “Nunca aceite nil” em um capítulo posterior.) Um construtor complexo é um sinal claro de que sua classe está fazendo o mesmo. Muito de. Mantenha os construtores curtos, simples e livres de qualquer lógica.

Não assuma nada sobre a implementação

As interfaces são, obviamente, inúteis sem uma implementação. No entanto, você, como desenvolvedor, nunca deve fazer suposições sobre o que é essa implementação. Você só deve codificar de acordo com o contrato feito pela interface. Você pode ter escrito a implementação, mas não deve codificar na interface com essa implementação em mente. Dito de outra forma, codifique em sua interface como se uma implementação radicalmente nova e melhor dessa interface estivesse ao virar da esquina.
Uma interface bem projetada lhe dirá o que você precisa fazer e como ela deve ser usada. A implementação dessa interface deve ser irrelevante para o uso da interface.

Não assuma que uma interface é uma abstração

As interfaces são boas, e eu certamente canto seus elogios o tempo todo. No entanto, é importante perceber que nem toda interface é uma abstração. Por exemplo, se sua interface é uma representação exata da parte pública de sua classe, você realmente não está “abstraindo” nada, certo? (Tais interfaces são chamadas de “interfaces de cabeçalho” porque se assemelham a arquivos de cabeçalho C++). Interfaces extraídas de classes podem ser facilmente acopladas a essa classe sozinha, tornando a interface inútil como uma abstração.
Finalmente, as abstrações podem ser “vazadas”, ou seja, podem revelar detalhes específicos de implementação sobre sua implementação. As abstrações com vazamento também são normalmente vinculadas a uma implementação específica. (Você pode ler mais sobre essa noção em uma excelente postagem no blog de Mark Seemann em [http://bit.ly/2awOhmn]http://bit.ly/2awOhmn).)

Conclusão

Ok, então isso deve servir como uma introdução básica à ideia de injeção de dependência.
A injeção de dependência é um meio para um fim, e esse fim é um código fracamente acoplado.

Obviamente, há mais do que isso, daí o resto deste livro. Mas se você entender as noções de que deve codificar em relação a abstrações e que deve solicitar a funcionalidade de que precisa, está no caminho certo para entender a injeção de dependência e escrever um código melhor.

Benefícios da Dependência Injeção

Por que devemos fazer tudo isso? Por que se dar ao trabalho de organizar nosso código da maneira específica exigida pelos princípios da injeção de dependência? Bem, porque há benefícios. Vamos falar um pouco sobre esses benefícios da Injeção de Dependência, porque eles são muitos e atraentes.

Manutenibilidade – Provavelmente, o principal benefício da injeção de dependência é a manutenibilidade. Se suas classes são fracamente acopladas e seguem o princípio de responsabilidade única – o resultado natural do uso de DI – então seu código será mais fácil de manter.
Classes simples e independentes são mais fáceis de corrigir do que classes complicadas e fortemente acopladas. O código que pode ser mantido tem um custo total de propriedade menor. Os custos de manutenção geralmente excedem o custo de construção do código em primeiro lugar, portanto, qualquer coisa que melhore a capacidade de manutenção do seu código é uma coisa boa. Todos nós queremos economizar tempo e dinheiro, certo?

Testabilidade – Na mesma linha que a manutenibilidade é a testabilidade. O código que é fácil de testar é testado com mais frequência. Mais testes significa maior qualidade. Classes fracamente acopladas que fazem apenas uma coisa – novamente, o resultado natural do uso de DI – são muito fáceis de testar de unidade. Ao usar a injeção de dependência, você facilita muito a criação de duplas de teste (comumente chamadas de “mocks”). Se as dependências são passadas para as classes, é bem simples passar em uma implementação dupla de teste. Se as dependências forem codificadas, é impossível criar duplicatas de teste para essas dependências. O código testável que realmente é testado é o código de qualidade. Ou pelo menos é de maior qualidade do que o código não testado. Acho difícil aceitar o argumento de que os testes unitários são uma perda de tempo – eles sempre valem o tempo para mim. (Certamente não sou o único que acha estranho que isso esteja mesmo em disputa?)

Legibilidade – Código que usa DI é mais simples. Segue o Princípio da Responsabilidade Única e, portanto, resulta em aulas menores, mais compactas e objetivas. Construtores não são tão confusos e cheios de lógica. As classes são mais claramente definidas, abertamente

declarando o que eles precisam. Por tudo isso, o código baseado em DI é mais legível. E o código que é mais legível é mais sustentável.

Flexibilidade – Código de acoplamento flexível – mais uma vez, o resultado do uso de DI – é mais flexível e utilizável de diferentes maneiras. Classes pequenas que fazem uma coisa podem ser reagrupadas e reutilizadas mais facilmente em diferentes situações. Classes pequenas são como Legos(tm) – elas podem ser facilmente reunidas para fazer uma infinidade de coisas, ao contrário de blocos Duplo(tm), que são mais volumosos e menos flexíveis. Ser capaz de reutilizar o código economiza tempo e dinheiro. Todo software precisa ser capaz de mudar e se adaptar a novos requisitos. O código fracamente acoplado que usa injeção de dependência é flexível e capaz de se adaptar a essas mudanças.

Extensibilidade – Código que usa injeção de dependência resulta em uma estrutura de classe que é mais extensível. Ao confiar em abstrações em vez de implementações, o código pode variar facilmente uma determinada implementação. Quando você codifica contra abstrações, pode codificar com a noção de que uma implementação radicalmente melhor do que você está fazendo está chegando. Classes pequenas e flexíveis podem ser estendidas facilmente, seja por herança ou composição. A base de código de um aplicativo nunca permanece estática e você provavelmente precisará adicionar novos recursos à medida que sua base de código cresce e surgem novos requisitos. O código que é extensível está à altura desse desafio.

Desenvolvimento de equipe – Se você está em uma equipe e essa equipe precisa trabalhar em conjunto em um projeto (quando isso não é verdade?), a injeção de dependência facilitará o desenvolvimento da equipe. (Mesmo que você esteja trabalhando sozinho, é muito provável que seu trabalho seja passado para alguém no futuro). A injeção de dependência exige que você codifique com base em abstrações e não em implementações. Se você tem duas equipes trabalhando juntas, cada uma precisando do trabalho da outra, você pode definir as abstrações antes de fazer as implementações, e então cada equipe pode escrever seu código usando as abstrações, mesmo antes das implementações serem escritas. Além disso, como o código é fracamente acoplado, essas implementações não dependem umas das outras e, portanto, são facilmente divididas entre as equipes.

Então aí está. A injeção de dependência resulta em código sustentável, testável, legível, flexível e extensível que é facilmente distribuído entre os membros da equipe. Parece difícil imaginar que qualquer desenvolvedor não queira tudo isso.

Um olhar mais atento sobre o acoplamento: Conhecimento

Introdução

Se você está prestando atenção, você notou que eu não gosto de acasalamento. Código fortemente acoplado me deixa enjoado. Eu falo sobre isso em meus posts, em meus livros, em todos os lugares. O acoplamento apertado é ruim. Sabemos que queremos um acoplamento frouxo e devemos evitar um acoplamento apertado. Isso é meio que certo, mas tem havido muito pouca discussão sobre exatamente o que tudo isso significa. Tem sido notoriamente difícil de descrever. Tem sido um tipo de coisa do tipo “eu sei quando vejo”. Como a injeção de dependência tem tudo a ver com a redução do acoplamento, achei que seria uma boa ideia incluir este capítulo abordando um pouco mais sobre o que é o acoplamento.

O acoplamento é uma medida das relações e conexões entre dois módulos.
O código tem que ser acoplado de alguma forma ou então não pode fazer nada. Mas como são feitas essas medições? O que exatamente está sendo medido? Essas são perguntas difíceis. Dado que o acoplamento apertado é ruim, queremos limitá-lo o máximo possível. Mas como, exatamente, fazemos isso? O que diabos **é** acoplamento exatamente?

Conhecimento

Felizmente, existe uma maneira de medir o acoplamento. Chama-se “connascência”. (Ouvi dizer “Cuh-NAY-desde”) No domínio do desenvolvimento de software, dois módulos são considerados “connascentes” se a alteração de um requer uma alteração no outro para manter a correção geral do sistema . Isso é praticamente o que pensamos quando pensamos em acoplamento. O termo foi usado pela primeira vez dessa maneira por Meilir Page-Jones com o objetivo de quantificar e qualificar exatamente o que é acoplamento. Ele discutiu isso pela primeira vez em seu livro “O que todo programador deve saber sobre design orientado a objetos”. O livro foi publicado em 1995, então essa noção não é nenhuma novidade.

No entanto, como é frequentemente o caso, essa ideia de vinte anos só agora está entrando em cena. Connascence é uma maneira de medir o acoplamento em seu código.

A conascência é discutida em duas dimensões. Primeiro, existem nove níveis diferentes de conascência. Em segundo lugar, todos esses níveis têm certas qualidades. Discutirei essas qualidades primeiro, e depois passaremos a falar sobre os níveis de conascência. Os níveis de conascência nos permitem formar uma taxonomia de acoplamento e nos dão um vocabulário para falar sobre isso. Isso é realmente útil, pois as discussões anteriores sobre acoplamento geralmente caíam em uma discussão muito amorfa sobre “apertado versus solto” – dificilmente uma visão científica das coisas.

Qualidades de Consciência

A conascência tem três qualidades: Força, Grau e Localidade.

Força da Consciência

Diz-se que um nível de conascência é mais forte do que outro se corrigi-lo exigir mudanças mais profundas e mais difíceis. Por exemplo, se reduzir o acoplamento entre duas entidades requer uma mudança fácil e simples, então a conascência é considerada menos forte do que a conascência que requer uma mudança complexa. Se um nível de conascência é difícil de refatorar, diz-se que é um nível de conascência forte. Os Níveis de Consciência descritos abaixo são listados por força crescente. Essa ordenação dá uma ideia de como priorizar sua refatoração. Ou seja, você deve trabalhar para refatorar os acoplamentos mais fortes para níveis mais fracos de acoplamento.

Grau de Consciência

O grau de conascência é uma medida do nível em que a conascência ocorre. A conascência pode ocorrer em pequeno ou grande grau. Por exemplo, dois módulos podem ser conectados por várias referências em oposição a uma única referência.
Diz-se que uma conexão de múltiplas referências tem um alto grau de conascência. Um determinado método pode ter muitos parâmetros que o acoplam a muitas classes externas.
Tal método tem um alto grau de conascência.

Localidade de Connascência

Às vezes, o acoplamento ocorre próximo – você tem duas classes na mesma unidade que estão interconectadas. A conascência pode ocorrer dentro de um único método. Mas às vezes esse acoplamento ocorre em duas unidades que estão muito distantes uma da outra.
Todos nós já vimos isso – você faz uma alteração na parte “inferior esquerda” do seu aplicativo e isso tem um efeito distante no “superior direito” do programa. A conascência que está próxima é melhor do que aquela que está distante.

Níveis de Consciência

Algum acoplamento é necessário. Sem ele, nada pode acontecer em um aplicativo.
No entanto, queremos manter a força de acoplamento o mais fraca possível, o grau o menor possível e a localidade o mais próxima possível. Se pudéssemos medir o acoplamento, saberíamos que estamos fazendo isso, certo? Bem, Page-Jones descreveu nove níveis de conascência, cada um mais forte, de maior grau e/ou de localidade mais distante que o anterior. Uma vez que reconhecemos esses níveis de acoplamento, podemos fazer coisas para reduzi-los a um nível inferior. Vamos dar uma olhada e ver como tudo funciona.

Conhecimento Estático

Os primeiros cinco níveis de Connascenses são considerados estáticos porque podem ser encontrados examinando visualmente seu código.

Connascimento do nome

Connascence of Name ocorre quando duas coisas têm que concordar sobre o nome de algo. Esta é a forma mais fraca de conascência, e aquela a que devemos aspirar a nos limitar. É quase óbvio e não pode ser evitado. Se você declarar um procedimento:

```
procedimento TMyClass.DoSomething
```

você tem que chamá-lo usando o nome DoSomething. Mudar o nome de algo requer mudanças em outros lugares. Se você quiser alterar o nome do procedimento, precisará alterá-lo em todos os lugares em que o chamou. Isso parece óbvio, e é claro que esse nível de conascência é inevitável. Na verdade, é desejável. É o nível mais baixo de acoplamento que podemos ter e, portanto, devemos procurá-lo e usá-lo ao máximo. Se pudermos limitar nosso acoplamento ao Connascence of Name, estaríamos indo muito bem.

Connascença do Tipo

Connascence of Type ocorre quando duas entidades têm que concordar sobre o tipo de algo. O exemplo mais óbvio são os parâmetros de um método. Se você declarar uma função da seguinte forma:

```
function TSomeClass.ProcessWidget(aWidget: TWidget; aAction: TWidgetActionType): Boolean;
```

então qualquer código de chamada deve passar um TWidget e um TWidgetActionType como parâmetros da função ProcessWidget e deve aceitar um Boolean como um tipo de resultado. Delphi é fortemente tipado, então esse tipo de conascência é quase sempre capturado pelo compilador. Connascence of Type não é tão fraco quanto Connascence of Name, mas ainda é considerado um nível de connascência fraco e aceitável. Na verdade, você não poderia realmente passar sem ele, poderia – você tem que ser capaz de chamar um método de uma classe para fazer qualquer coisa, e não é muito trabalhoso garantir que seus tipos correspondam. Na verdade, no Delphi, você precisa acoplar o código via Connascence of Type para compilar seu código.

Consciência do Significado

A Connascência do Significado ocorre quando os componentes devem concordar com o significado de valores particulares. A Connascência de Significado ocorre mais frequentemente quando usamos “números mágicos”, ou seja, um valor específico que tem significado e que é usado em vários lugares. Considere o seguinte código:

```
função GetWidgetType(aWidget: TWidget): inteiro; começar

    if aWidget.Status = 'Working' então comece

        Resultado := 1;
    fim mais
    começar
        se aWidget.Status = 'Broken' então
        começar
            Resultado := 2;
        fim mais

        comece se aWidget.State = 'Missing' então
            comece Resultado := 3;

            fim mais
            começar
                Resultado := 0;
            fim;
        do que;
    do que; do
que;
```

Se você quiser usar o código acima, precisará saber o significado do código de resultado para a função GetWidgetType . Se você alterar um dos tipos de resultado ou adicionar um novo, precisará alterar o código que usa essa função onde quer que ela seja usada. E para fazer essa alteração, você precisa saber o significado de cada código de resultado.

A solução óbvia aqui é refatorar o código para usar nomes constantes para o resultado code, ou melhor, um tipo enumerado que define os códigos de resultado. Isso reduz sua conascência de Conascência de Significado para Conascência de Nome, um resultado desejável. Lembre-se, sempre que você puder refatorar de um nível de conascência mais alto para um mais baixo, você diminuiu o acoplamento e, portanto, melhorou seu código.

Outro exemplo de Connascence of Meaning é o uso de nil como sinal. Muitas vezes, os desenvolvedores usam nil para significar “sem valor” ou “Desculpe, não consegui fazer/encontrar/completar isso”, e seu código precisa lidar com isso. Como veremos em capítulos posteriores, esse uso de nil deve ser evitado, pois cria acoplamento via Connascence of Meaning.

Consciência da Posição

Connascence of Position ocorre quando o código em dois lugares diferentes deve concordar com a posição das coisas. Isso ocorre mais comumente em listas de parâmetros em que a ordem dos parâmetros na lista de parâmetros de um método é necessária para manter essa ordem. Se você

adicionar um parâmetro no meio de uma lista de parâmetros existente, todos os usos desse método devem adicionar o novo parâmetro na posição correta.

Algumas linguagens permitem que você nomeie seus parâmetros para que possam ser incluídos em qualquer ordem, mas o Delphi não permite isso. Assim, é necessário acoplar o código usando Connascence of Position ao escrever o código Delphi.

Agora, o grau de sua Connascência de Posição pode ser reduzido limitando o número de parâmetros em qualquer rotina. Para limitar Connascence of Position, você pode reduzir uma lista de parâmetros a um único tipo, passando de Connascence of Position para Connascence of Type. Connascence of Type é um acoplamento mais fraco e, portanto, isso é algo que você deve tentar fazer.

Aqui está um exemplo. Considere esta rotina:

```
procedimento TUserManager.AddUser(aFirstName: string; aLastName: string; aAge: integer; aBirthdate: TDate\ Time; aAddress: TAddress;
aPrivileges: TPrivileges);
```

Para usar AddUser , você precisa ter certeza de obter todos os parâmetros exatamente na posição correta. Se você adicionar um parâmetro no meio, precisará garantir que qualquer uso de AddUser coloque esse novo valor exatamente no lugar certo.

Podemos reduzir este exemplo de Connascence of Position refatorando-o para usar Connascence of Type. Por exemplo:

```
tipo
  TUserRecord = registro
    Nome: string;
    Sobrenome: string;
    Idade: inteiro;
    Aniversário: TDateTime;
    Endereço: Tendereço;
    Privilégios: TPrivilégios;
  fim;

procedimento TUserManager.AddUser(aUser: TUserRecord);
```

Agora reduzimos o acoplamento criando um tipo e fazendo com que o procedimento AddUser dependa do tipo do parâmetro em vez da posição de uma longa lista de parâmetros. Ao reduzir a força do acoplamento, melhoramos o código. Essa técnica também é chamada de "objeto de parâmetro". (Consulte http://refactoring.com/catalog/ introduzirParameterObject.html)

Connascença do Algoritmo

Connascence of Algorithm ocorre quando dois módulos devem concordar em um algoritmo específico para funcionarem juntos.

Imagine que você tivesse um sistema que tivesse uma API baseada em C# que deveria ser consumida por um cliente Delphi. As informações enviadas entre esses dois módulos são confidenciais e devem ser criptografadas. Assim, esses dois módulos são acoplados pelo Connascence of Algorithm, pois ambos devem concordar com o algoritmo de criptografia que será utilizado. Se o remetente alterar o algoritmo de criptografia, o destinatário deverá alterar para o mesmo al

Reduzir a Connascência do Algoritmo é difícil, porque na maioria das vezes tem um alto grau de localidade (ou seja, o acoplamento ocorre longe). Uma solução pode ser criar um único módulo que se torne o único local onde o algoritmo é encontrado e, em seguida, fazer com que ambos os módulos de consumo usem esse único módulo.

Conhecimento Dinâmico

Os próximos quatro níveis de conascência são considerados “dinâmicos” porque só podem ser descobertos executando seu código. Esses níveis são mais fortes do que os estáticos porque eles só se revelam em tempo de execução, tornando-os difíceis de detectar e muitas vezes mais difíceis de corrigir.

Consciência da Execução

Connascence of Execution ocorre quando a ordem de execução do código é necessária para que o sistema esteja correto. É muitas vezes referido como “Acoplamento Temporal”.

Aqui está um exemplo usando o código acima:

```
UserRecord.FirstName := 'Alicia';
UserRecord.LastName := 'Florrick';
UserRecord.Idade = 47;
UserManager.AddUser(UserRecord);
UserRecord.Birthday := EncodeDate(1968, 12, 3);
```

Esse código adiciona o valor de Aniversário **após** a adição do usuário. Isso claramente não vai funcionar. Obviamente, isso é perceptível ao examinar o código, mas você pode imaginar um cenário mais complexo e mais difícil de detectar. Considere este código:

```
SprocketProcesser.AddSprocket(SomeSprocket);
SprocketProcessor.ValidateSprocket;
```

A ordem dessas duas declarações importa? A roda dentada precisa ser adicionada e validada ou deve ser validada antes de ser adicionada? É difícil dizer, e alguém não bem versado no sistema pode cometer o erro de colocá-los na ordem errada. Isso é Connascense of Execução.

Aqui está outro exemplo. Imagine uma fila que contém mensagens. A primeira mensagem diz “Lista inicial”. Em seguida, as próximas duas mensagens têm itens de lista para adicionar itens à lista. Então, finalmente, a fila tem uma mensagem que diz “Finalizar lista”. Isso funciona muito bem se você tiver um único thread de trabalho puxando itens da fila. Todos os itens serão retirados em ordem. Mas e se você tivesse vários threads retirando itens da fila, e um dos threads funcionasse um pouco mais rápido que os outros e, em seguida, retirasse a mensagem "Finalizar lista" da fila antes da mensagem final "Aqui está outro item" foi processado? Isso seria ruim. Isso também seria um erro causado pelo Connascense of Execution.

Consciência do Tempo

Connascence of Timing ocorre quando o timing da execução faz diferença no resultado da aplicação. O exemplo mais óbvio disso é uma condição de corrida encadeada, na qual dois encadeamentos buscam o mesmo recurso e apenas um dos encadeamentos pode vencer a corrida. Connascence of Timing é notoriamente difícil de encontrar e diagnosticar, e pode revelar-se de maneiras imprevisíveis.

Connascença de valor

Connascence of Value ocorre quando vários valores devem ser devidamente coordenados entre os módulos. Por exemplo, imagine que você tem um teste de unidade que se parece com isso:

```
[Teste]
procedimento TestCheckoutValue;
Onde
    PriceScanner: IPPriceScanner;
começar
    PriceScanner := TPriceScanner.Create;
    PriceScanner.Scan(' Bombas de Açúcar Congelado');
    Assert.Equals(50, PriceScanner.CurrentBalance);
fim;
```

Então nós escrevemos o teste. Agora, no espírito do Test Driven Development, farei o teste passar da maneira mais fácil e simples possível.

```
procedimento TPriceScanner.Scan(altem: string); começar

    Saldo Corrente := 50;
fim;
```

Agora temos um acoplamento estreito entre o TPriceScanner e nosso teste. Obviamente, temos Connascense of Name, porque ambas as classes contam com o nome CurrentBalance. Mas isso é um nível relativamente baixo e perfeitamente aceitável. Temos Connascence of Type, porque ambos devem concordar com o tipo TPriceScanner, mas, novamente, isso é benigno. Temos Connascence of Meaning, porque ambas as rotinas têm uma dependência codificada do número 50. Isso deve ser refatorado. Mas o verdadeiro problema é a Conascência de Valor que ocorre porque ambas as classes conhecem o preço – ou seja, o “Valor” – do preço das Bombas de Açúcar Gelado. Se o preço mudar, mesmo nosso teste muito simples será interrompido.

A solução é refatorar para um nível mais baixo de conascência. A primeira coisa que você pode fazer é refatorar para que o conhecimento do preço (o valor) das Bombas de Açúcar Gelado seja mantido em apenas um lugar:

```
procedimento TPriceScanner.Scan(altem: string; aPrice: integer); começar

    Saldo Corrente : = aPreço ;
fim;
```

e agora nosso teste pode ser lido da seguinte forma:

```
[Teste]
procedimento TestCheckoutValue;
Onde
    PriceScanner: IPPriceScanner;
começar
    PriceScanner := TPriceScanner.Create;
    PriceScanner.Scan(' Bombas de Açúcar Gelado', 50);
    Assert.Equals(50, PriceScanner.CurrentBalance);
fim;
```

E agora não temos mais Connascence of Value entre os dois módulos e nosso teste ainda passa. Excelente.

Consciência da Identidade

Connascence of Identity ocorre quando dois componentes devem se referir ao mesmo objeto. Se os dois módulos se referem à mesma coisa, e então um muda essa referência, o outro objeto deve mudar para a mesma referência. Muitas vezes é uma forma sutil e difícil de detectar de conascência. Como resultado, esta é a forma mais complexa de conascência.

Considere o seguinte código:

```
programa Identidade;

{$APPTYPE CONSOLE}

{$R *.res}

usa
    System.SysUtils;

tipo
    TReportInfo = classe
    privada
        FReportCoisas: string;
        procedimento SetReportStuff(const Valor: string);
    público
        propriedade ReportStuff: string lida FReportStuff write SetReportStuff;
    fim;

procedimento TReportInfo.SetReportStuff(const Value: string); começar

    FReportStuff := Valor;
fim;

tipo
    TInventárioRelatório = classe
```

```
    privado
        FReportInfo: TReportInfo;
    público
        construtor Create(aReportInfo: TReportInfo); propriedade
        ReportInfo: TReportInfo ler FReportInfo escrever FReportInfo;
    fim;

    TSalesReport = classe
    privada
        FReportInfo: TReportInfo;
    construtor público Create(aReportInfo:
        TReportInfo); propriedade ReportInfo: TReportInfo ler
        FReportInfo escrever FReportInfo;
    fim;

construtor TInventoryReport.Create(aReportInfo: TReportInfo); começar

    FReportInfo := aReportInfo;
fim;

construtor TSalesReport.Create(aReportInfo: TReportInfo);
começar
    FReportInfo := aReportInfo;
fim;

Onde
    ReportInfo: TReportInfo;
    NewReportInfo: TReportInfo;
    Relatório de Inventário: Relatório de Inventário ;
    Relatório de Vendas: TSalesReport;



comece tente ReportInfo := TReportInfo.Create;
        RelatórioInventário := TInventoryReport.Create (ReportInfo); SalesReport :=
        TSalesReport.Create(ReportInfo); experimentar

            // Faça coisas com relatórios

            NewReportInfo := TReportInfo.Create; tente
            InventoryReport.ReportInfo := NewReportInfo; //
                Faça coisas com o relatório // Mas os relatórios agora
                apontam para diferentes ReportInfos.

                // Isso é Connascência de Identidade
            finalmente
                NewReportInfo.Free;
            fim;
        finalmente
            ReportInfo.Free;
            InventoryReport.Free;
            SalesReport.Free; fim;
        exceto
```

```
      em E: Exceção do
         Writeln(E.ClassName, ': ', E.Message);
   fim;
fim.
```

Aqui temos dois relatórios: um relatório de estoque e um relatório de vendas. O domínio requer que os dois relatórios sempre se refiram à mesma instância de TReportInfo.
No entanto, como você pode ver acima, no meio do processo de geração de relatórios, o Relatório de Inventário obtém uma nova instância ReportInfo. Isso é bom, mas o Relatório de vendas também precisa se referir a essa nova instância TReportInfo . Em outras palavras, se você alterar a referência em um relatório, o outro relatório deverá ser alterado para a mesma referência. Isso é chamado de Connascence of Identity, pois as duas classes devem alterar a identidade de sua referência para que o sistema continue funcionando corretamente.

O que fazer com a conascência?

Bem, agora que temos os nove níveis de conascência definidos, o que devemos fazer sobre o acoplamento em nosso código?

Embora algum acoplamento deva ocorrer, você deve se esforçar para manter sua conascência no nível mais baixo possível. Ou seja, você deve reduzir o Grau de Connascência em seu código. Um aplicativo muito limpo geralmente terá Connascence of Name e Type, e tentará limitar a Connascence of Meaning e Position tanto quanto possível. Todos os outros tipos de conascência devem realmente ser refatorados.

Você também deve aumentar o Locality of Connascence em seu código. Você deve trabalhar para reduzir o escopo de todos os identificadores em seu código. Você deve limitar o escopo de um tipo ao menor escopo possível. As coisas que pertencem umas às outras devem ser mantidas juntas e não expostas a lugares a que não pertencem. O princípio DRY – “Não se repita” – é um exemplo de localidade crescente. Assim é o Princípio da Responsabilidade Única.

Finalmente, você deve preferir a estabilidade. A conascência é realmente apenas a medida da necessidade de mudança, e quanto menos vezes você precisar mudar alguma coisa, menos erros ocorrerão como resultado de um acoplamento apertado. Coisas que são estáveis terão muito menos probabilidade de causar erros de acoplamento.

Conclusão

Todos concordamos que o acoplamento apertado é ruim (pelo menos espero que sim!). Mas a noção de acoplamento tem sido tipicamente mal definida. Espero que esta revisão de conascência – a ideia de que se algo muda em um lugar, outra coisa tem que mudar em outro lugar – permita que você fale um pouco mais especificamente sobre o que é acoplamento. Também espero que os Levels of Connascence permitam que você encontre código com alto grau de acoplamento e refatore essas áreas para diminuir seu acoplamento. Se você se esforçar para reduzir o nível geral de Connascence em seu código, terá uma base de código mais limpa e fácil de manter.

Injeção de Construtor

Ok, então no primeiro capítulo, você viu o código que injetava uma classe em outra. Isso reduziu o acoplamento e tornou a dependência mais fraca. A classe foi injetada através do construtor. Você provavelmente ficará surpreso ao saber que isso é chamado de “Injeção de Construtor”. Neste capítulo, falarei um pouco mais a fundo sobre injeção de construtor.

Injeção de Construtor

Injeção de Construtor é o processo de usar o construtor para passar as dependências de uma classe. As dependências são declaradas como parâmetros do construtor. Como resultado, você não pode criar uma nova instância da classe sem passar uma variável do tipo exigido pelo construtor.

Esse último ponto é fundamental – quando você declara uma dependência como parâmetro no construtor, você está dizendo “Desculpe, pessoal, mas se você quiser criar esta classe, você **deve** passar este parâmetro”. Assim, uma classe é capaz de especificar as dependências de que necessita e ter a garantia de que as obterá. Você não pode criar a classe sem eles. Se você tiver este código:

```
TPayrollSystem = classe
privado
   FBankService: TBankService;

construtor público Create (aBankingService: TBankingService);
fim;

construtor TPayrollSystem.Create (aBankingService: TBankingService); começar
FBankService : = aBankingService; fim;
```

você não pode criar um TPayrollSystem sem passar uma instância de TBankingSer vice. (Bem, infelizmente, você pode passar nil, mas vamos lidar com isso em um minuto.) TPay rollSystem declara muito claramente que requer um TBankingService, e os usuários da classe devem fornecer um.

Nunca aceite nada

Como mencionei, é lamentável que a classe acima possa e leve nil como parâmetro. Digo “take” porque enquanto um usuário da classe pode passar em nil, a classe em si não precisa aceitar nil. Na verdade, defendo que todos os métodos devem rejeitar explicitamente nil como valor para qualquer parâmetro de referência a qualquer momento, incluindo construtores e métodos regulares. Em nenhum momento um parâmetro deve ser nil sem que o método gere uma exceção. Se você passar nil para o TPayrollSystem acima e a classe tentar usá-lo, ocorrerá uma violação de acesso. E as violações de acesso são ruins. Eles devem – e neste caso podem – ser evitados.

O código acima realmente deve ser algo assim:

```
TPayrollSystem = classe
privada
   FBankService: TBankService; construtor
público Create (aBankingService:
   TBankingService); fim;


construtor TPayrollSystem.Create (aBankingService: TBankingService); começar

   if aBankingService = nil then begin
   raise Exception.Create('Que diabos
      você pensa que está fazendo? Como você ousa me passar um serviço bancário nil!?!');

   fim;

   FBankService : = aBankingService; fim;
```

Este código nunca permitirá que o campo interno seja nulo. Ele levantará uma exceção se alguém ousar passar nil como o valor do parâmetro do construtor. Assim é como deve ser. Você poderia aceitar nil, mas teria que verificar em todo o seu código, e quem quer isso? Nas palavras imortais de Barney Fife, você deve cortar o uso do nil pela raiz, recusando-se a aceitá-lo no ponto de entrada.

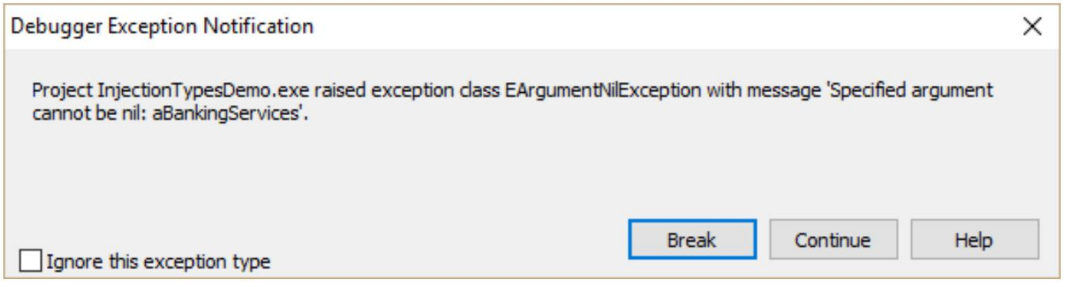
A verificação de nil é um código clichê, e a estrutura Spring4D fornece um meio para fazer facilmente uma verificação de nil . Proteger contra nil sendo passado como parâmetro é chamado de “Padrão de Guarda”, e você não ficará surpreso ao saber que Spring4D fornece um registro chamado Guarda que possui vários métodos estáticos que permitem verificar – ou “proteger contra” – certas situações ocorram.

> Na verdade, o padrão de guarda é definido como qualquer expressão booleana que deve ser avaliada como True antes que a execução do programa possa continuar. Geralmente é usado para garantir que certas pré-condições sejam atendidas antes que um método possa continuar, garantindo que o código a seguir possa ser executado corretamente. Verificar se uma referência não é nula é provavelmente o uso mais comum – mas não o único – para o Padrão de Guarda.

No caso em questão, estamos usando o padrão Guard para proteger contra um parâmetro ser nil, então podemos usar a classe Guard para simplificar nosso código:

```
TPayrollSystem = classe
privado
   FBankService: TBankService;

construtor público Create (aBankingService: TBankingService);
fim;

construtor TPayrollSystem.Create (aBankingService: TBankingService); começar
Guard.CheckNotNull (aBankingService, 'aBankingService'); FBankService : = aBankingService;
   fim;
```

Guard.CheckNotNull recebe dois parâmetros. O primeiro é o item a ser verificado e o segundo é o nome do item que foi verificado como uma string. Agora, se você passar nil para o construtor, receberá este erro:



Um erro de cláusula Guard

Ok, chega de passar zero. Você deve ter a mensagem agora.

Quando usar a injeção de construtor

Você deve usar a Injeção de Construtor quando sua classe tiver uma dependência que a classe requer para funcionar corretamente. Se sua classe não pode funcionar sem uma dependência, injete-a por meio do construtor. Se sua classe precisar de três dependências, exija todas as três no construtor. (No capítulo sobre Anti-Padrões, discutiremos a situação em que você acaba com muitas dependências no construtor.)

Além disso, você deve usar a Injeção de Construtor quando a dependência em questão tiver uma vida útil maior do que um único método. As dependências passadas para o construtor devem ser úteis para a classe de maneira geral, com seu uso abrangendo vários métodos na classe. Se uma dependência for usada em apenas um ponto, a injeção de método (abordada em um próximo capítulo) deve ser usada.

A injeção de construtor deve ser a principal maneira de fazer a injeção de dependência. É simples: uma classe precisa de algo e, portanto, pede por isso antes mesmo de poder ser construído.
Ao usar o Padrão Guard, você pode usar a classe com confiança, sabendo que a variável de campo que armazena essa dependência será uma instância válida. Além disso, é muito simples e claro de fazer. A injeção de construtor deve ser sua técnica para código claro e desacoplado. Mas não deve ser a única ferramenta na caixa de ferramentas. A seguir – outras formas e razões para injetar dependências.

Injeção de propriedade

Ok, então usamos Constructor Injection quando queremos declarar dependências obrigatórias. Mas o que fazer quando uma dependência não é necessária? Às vezes, uma classe tem uma dependência que não é estritamente necessária, mas é de fato usada pela classe. Um exemplo pode ser uma classe de documento que pode ou não ter um verificador gramatical instalado. Se houver um, ótimo; a classe pode usá-lo. Se não houver um, ótimo – a classe pode incluir uma implementação padrão como um espaço reservado.

A solução aqui é a injeção de propriedades. Você adiciona uma propriedade em sua classe que pode ser definida como uma instância válida da classe em questão. Como a dependência é uma propriedade, você pode defini-la como desejar. Se a dependência não for desejada ou necessária, você poderá deixar a propriedade como está. Seu código deve agir como se a dependência estivesse lá, então você deve fornecer uma implementação padrão do-nothing para que o código ainda possa ser executado com ou sem uma dependência real. (Lembre-se, nunca queremos que nada seja nulo, então a implementação padrão deve ser válida). Assim, se o usuário da classe quiser fornecer uma implementação de trabalho, ele pode, mas se não, há um padrão de trabalho que permitirá que a classe que a contém ainda funcione.

> Use Injeção de Propriedade quando uma dependência for opcional e/ou quando uma dependência puder ser alterada após a instanciação da classe. Use-o quando quiser que os usuários da classe que o contém sejam capazes de fornecer sua própria implementação da interface em questão. Você só deve usar a injeção de propriedade quando puder fornecer uma implementação padrão da interface em questão. A Injeção de Propriedade às vezes é chamada de “Injeção de Setter”.

Qualquer implementação padrão provavelmente será uma implementação não funcional. Mas não precisa ser. Se você quiser fornecer uma implementação padrão funcional, tudo bem. No entanto, esteja ciente de que, ao usar Injeção de Propriedade e criar essa classe no construtor do objeto que o contém , você está se acoplando a essa implementação.
Mas não se desespere; existem maneiras de lidar com isso quando começamos a usar um Container. Mas ainda não estamos tão longe, daí o aviso.

Um exemplo, é claro, mostrará como as coisas são feitas. Vamos dar uma olhada no código que faz o que descrevi acima – uma classe de documento que tem um verificador gramatical opcional.

Primeiro, vamos começar com uma interface:

```
tipo
  IGrammarChecker = interface
  ['{9CA7F68C-8A42-4B8C-AD1A-14C04CAE0901}']
    procedimento CheckGrammar;
  fim
```

Agora, vamos implementá-lo duas vezes. Uma vez como um padrão de não fazer nada e novamente como um verificador gramatical “real”.

```
tipo
  TDefaultGrammarChecker = class(TInterfacedObject, IgrammarChecker) private

    procedimento CheckGrammar;
  fim

  TRealGrammarChecker = class(TInterfacedObject, IGrammarChecker)
    procedimento CheckGrammar;
  fim;

procedimento TDefaultGrammarChecker.CheckGrammar;
começar
  // não fazemos nada, mas vamos WriteLn apenas para provar que estivemos aqui
  WriteLn('Não faça nada'); fim;


procedimento TRealGrammarChecker.CheckGrammar;
begin WriteLn('Gramática foi verificada'); fim;
```

Ambas as implementações fazem um WriteLn, mesmo a implementação “no-op”.
Eu apenas queria ter certeza de que as coisas estavam funcionando corretamente. Novamente, TDefaultGrammarChecker deve ser uma implementação padrão não operacional que nos impedirá de verificar nil o tempo todo.

Agora precisamos de uma classe que tenha uma propriedade para o verificador gramatical.

```
  TDocumento = classe
  privado
    FTexto: string;
    FGrammarChecker: IGrammarChecker;
    procedimento SetGrammarChecker(const Valor: IGrammarChecker);
  construtor público Create(const aText: string); procedimento CheckGrammar;
    propriedade Text: string ler FText escrever FText; propriedade
    GrammarChecker: IGrammarChecker lê FGrammarChecker escreve
    SetGrammarChecker; fim;


procedimento TDocument.CheckGrammar;
comece FGrammarChecker.CheckGrammar;

fim;

construtor TDocument.Create(const aText: string); começar

  herdado Criar;
  FTexto := aTexto;
  FGrammarChecker := TDefaultGrammarChecker.Create;
fim;

procedimento TDocument.SetGrammarChecker(const Valor: IGrammarChecker); begin
Guard.CheckNotNull(Value, 'Valor em TDocument.SetGrammarChecker'); FGrammarChecker :=
  Valor;

fim;
```

Aqui estão algumas coisas a serem observadas sobre este código:

- Seu construtor toma o texto do documento como parâmetro. Em seguida, ele é exposto como uma propriedade de leitura/gravação, para que você possa alterá-lo, se desejar.
- O construtor também cria uma instância do verificador gramatical padrão. Observe novamente que isso cria uma dependência codificada – um dos perigos da Injeção de Propriedade. Mas a dependência é um padrão do-nothing e nos impede de verificar constantemente se há nil. • O setter para a propriedade GrammarChecker contém uma chamada Guard , garantindo

  que o valor interno para FGrammarChecker nunca pode ser nulo.

  Outra coisa que você pode notar é que a propriedade GrammarChecker poderia realmente ser uma propriedade “somente gravação” se você quisesse. Ou seja, você só pode definir o valor e nunca lê-lo.

> Você pode criar uma propriedade somente gravação quando o valor que está sendo gravado for usado apenas internamente e nunca será chamado por código fora da classe. Algo como o GrammarChecker pode se qualificar como uma propriedade somente de gravação.

Agora, aqui está um código que exercita tudo e mostra Injeção de Propriedade em açao:

```
procedimento principal;
Onde
    Documento: TDocumento;
começar
    Document := TDocument.Create('Este é o texto do documento.'); tente WriteLn(Document.Text); // Usa o
    verificador gramatical padrão sem operação Document.CheckGrammar; // Altere a dependência para usar
        o verificador gramatical "real" Document.GrammarChecker := TRealGrammarChecker.Create; // Agora
        o verificador gramatical é um "real" Document.CheckGrammar; // Isso irá -- e deve -- gerar uma
        exceção.




        Document.GrammarChecker := nil;
    finalmente
        Documento.Gratuito;
    do que;
do que;
```

Aqui estão algumas coisas a serem observadas sobre o código acima: * Ele cria um documento, pegando o texto gramatical padrão como um parâmetro do construtor. Ele chama Check Grammar, então grava isso no console. * Mas então usamos Injeção de Propriedade para injetar um verificador gramatical “real”, e quando chamamos CheckGrammar, a gramática é verificada “de verdade”. * Em seguida, tentamos definir o verificador gramatical como nil, mas isso gera uma exceção por causa da cláusula Guard .

Assim, a Injeção de Propriedades permite que você forneça dependências opcionais. Também permite alterar uma dependência, se necessário. Por exemplo, sua classe de documento pode receber textos de diferentes idiomas e, portanto, exigirá que o verificador gramatical seja alterado à medida que o idioma do documento for alterado. A injeção de propriedade permitirá isso.

Injeção de Método

E se a dependência que sua classe precisa for diferente na maior parte do tempo? E se a dependência for uma interface e você tiver várias implementações que queira passar para a classe? Você poderia usar a injeção de propriedade, mas estaria definindo a propriedade o tempo todo antes de chamar o método que utilizava a dependência que muda frequentemente, configurando a possibilidade de acoplamento temporal.

> Acoplamento Temporal é basicamente o mesmo que Connascence of Execu tion – a ideia de que a ordem de execução deve ocorrer de uma maneira específica para que as coisas funcionem corretamente.

Construtor e Injeção de Propriedade geralmente são usados quando você tem uma dependência que não vai mudar com frequência, então eles não são apropriados para uso quando sua dependência pode ser uma das muitas implementações.

É aqui que entra a injeção de métodos.

A injeção de método permite que você injete uma dependência diretamente no ponto de uso, para que você possa passar qualquer implementação que desejar sem ter que se preocupar em armazená-la para uso posterior. É frequentemente usado quando você passa outras informações que precisam de tratamento especial. Por exemplo:

```
unidade uPropertyInjectionMultiple;

interface

tipo

  TRreceita = classe
  privada
    FTexto: string;

  propriedade pública Texto: string ler FText escrever FText;
  fim;

  IFoodPreparer = interface
```

```
    ['{3900BE64-B0EC-4281-9D92-96B191FDE5BC}']
      procedimento PrepareFood(aRecipe: TRecipe);
    fim;

    TBaker = class(TInterfacedObject, IFoodPreparer) procedure
      PrepareFood(aReceita: TRreceita);
    fim;

    TShortOrderCook = class(TInterfacedObject, IFoodPreparer) procedure
      PrepareFood(aRecipe: TRecipe); fim;


    TCef = class(TInterfacedObject, IFoodPreparer) procedimento
      PrepareFood(aReceita: TRreceita); fim;


    TRestaurante = classe
    privado
      FNome: cadeia;

    construtor público Create(const aName: string):
      procedure PrepareFood(aRecipe: TRecipe; aPreparer: IFoodPreparer); Nome da
      propriedade : string lida FName;
    fim;

implementação

construtor TRestaurant.Create(const aName: string); começar
FNome := umNome;

fim;

procedimento TRestaurant.PrepareFood(aReceita: TRreceita; aPreparador: IFoodPreparer); begin
aPreparer.PrepareFood(aRecipe) end;



procedimento TBaker.PrepareFood(aReceita: TRreceita); begin
Writeln('Use habilidades de cozimento para fazer o seguinte: '
   end;                                                          + aReceita.Texto);


procedimento TShortOrderCook.PrepareFood(aReceita: TRreceita); começar

   Writeln('Use a grade para fazer o seguinte: '                 + aReceita.Texto);
fim;

procedimento TCef.PrepareAlimento(aReceita: TRreceita);
begin Writeln('Use habilidades culinárias bem treinadas para
   preparar o seguinte: ' end;                                                  + aReceita.Texto);


fim.
```

Aqui, temos a noção de uma receita, que pode exigir um preparador diferente dependendo daquela receita. Somente a entidade chamadora saberá qual será o tipo de preparador adequado para uma determinada receita. Por exemplo, uma receita pode exigir um cozinheiro de curto prazo e outra receita pode exigir um padeiro ou um chef. Não sabemos ao escrever o código que tipo de IFoodPreparer será necessário e, portanto, não podemos realmente passar a dependência no construtor e ficar presos a essa implementação.

Também é desajeitado definir uma propriedade toda vez que um IFoodPreparer novo ou diferente for necessário. E definir a propriedade de tal forma induz o acoplamento temporal (Connascence of Execution) e sofrerá com problemas de segurança de thread porque exigiria um bloqueio ao redor do código em um ambiente de thread.

A melhor solução é apenas passar o IFoodPreparer para o método no ponto de usar.

A injeção de método deve ser usada quando a dependência pode mudar a cada uso, ou pelo menos quando você não pode ter certeza de qual dependência será necessária no ponto de usar.

Aqui está um exemplo de uso de Injeção de Método quando a dependência precisa ser alterada toda vez que é usada. Imagine uma situação em que um robô de pintura de carros requer uma nova ponta de pistola de pintura após cada carro que pinta. Você pode começar assim usando Constructor Injection:

```
tipo

  IPaintGunTip = interface
    procedimento SprayCar(aColor: TColor); fim;


  TPaintGunTip = classe (TInterfacedObject, IPaintGunTip)
    procedimento SprayCar(aColor: TColor);
  fim;

  TCarPaintingRobot = classe
  privada
    FPaintGunTip: IPaintGunTip;

  construtor público Create (aPaintGunTip: IPaintGunTip);
    procedimento PaintCar (aColor: TColor);
  fim;

Construtor TCarPaintingRobot.Create (aPaintGunTip: IPaintGunTip); começar
Guard.CheckNotNull (aPaintGunTip, 'aPaintGunTip');
```

```
    FPaintGunTip : = aPaintGunTip;
fim;

procedimento TCarPaintingRobot.PaintCar(aColor: TColor); começar
FPaintGunTip.SprayCar(aColor);


   // Uh oh -- o que fazer agora? Como liberamos a ponta?
   // E mesmo se pudéssemos, e daí?
   // Como conseguiríamos um novo?

fim;

procedimento TPaintGunTip.SprayCar(aColor: TColor); begin
Writeln('Pulverize o carro com ', ColorToString(aColor)); fim;
```

> Ao implementar um método usando Method Injection, você deve incluir uma Guard Clause. A dependência será usada imediatamente e, é claro, se você tentar usá-la quando for nula, obterá uma violação de acesso imediata. Isso obviamente deve ser evitado.

Aqui, quando pintamos o carro, temos que pegar uma nova ponta de pistola de pintura. Mas como? Quando pintamos o carro, a dica não serve mais, mas é uma interface, e não temos como liberá-la manualmente, e mesmo que fizéssemos, o que faríamos na próxima vez que precisarmos pintar um carro? Não sabemos que tipo de dica é necessária para um determinado carro e, se separamos adequadamente nossas preocupações, nem sabemos nada sobre a criação de uma nova dica. O que fazer? Bem, use injeção de método em vez disso:

```
tipo

   IPaintGunTip = interface
      procedimento SprayCar(aColor: TColor); fim;


   TPaintGunTip = classe (TInterfacedObject, IPaintGunTip)
      procedimento SprayCar(aColor: TColor); fim;


   TCarPaintingRobot = procedimento
   público da classe PaintCar
      (aColor: TColor; aPaintGunTip: IPaintGunTip); fim;


procedimento TCarPaintingRobot.PaintCar (aColor: TColor; aPaintGunTip: IPaintGunTip);
```

```
comece aPaintGunTip.SprayCar (aColor);
fim;

procedimento TPaintGunTip.SprayCar(aColor: TColor);
begin WriteLn('Pulverize o carro com ', ColorToString(aColor));
   fim;
```

Agora, quando passamos a dependência diretamente para o método, a interface sai do escopo quando terminamos de pintar e a ponta da pistola de pintura é destruída. Além disso, na próxima vez que um carro precisar ser pintado, o consumidor passará em uma nova ponteira, que será liberada no momento do uso. Método de injeção para o resgate!

Assim, a injeção de método é útil em dois cenários: quando a implementação de uma dependência varia e quando a dependência precisa ser renovada após cada uso.
Em ambos os casos, cabe ao chamador decidir qual implementação passar para o método.

A injeção de dependência Recipiente

Até este ponto, fiz apenas uma breve referência à noção de um contêiner de injeção de dependência. Eu fiz isso de propósito por algumas razões. Primeiro, eu queria ter certeza de que você entendeu a injeção de dependência antes de falarmos sobre o contêiner de injeção de dependência. Como observei no início, a injeção de dependência e o uso de um contêiner de injeção de dependência são duas coisas muito diferentes. Você pode fazer a injeção de dependência sem nunca tocar em um contêiner de injeção de dependência.
É o que temos feito até agora. Segundo, há muita confusão sobre o que é um Container, como usá-lo, onde usá-lo e quando usá-lo. Eu queria adiar a resposta a essas perguntas o máximo possível para que você tivesse a ideia de injeção de dependência sem a confusão potencial do que o Container é e faz.

Mas chegou a hora, e agora vamos dar uma olhada no famoso – ou infame – Dependency Injection Container.

O que é um Container de Injeção de Dependência?

Respondendo à pergunta “O que é um contêiner de injeção de dependência?” tem sido difícil desde que a idéia se originou. Muitas pessoas têm visões diferentes de como responder a essa pergunta. Obviamente, tenho minha própria definição, mas antes de chegar a isso, falarei primeiro sobre o que um Container não é.

Um Container não é meramente um substituto para uma chamada para Create. Embora grande parte do que ele faz seja criar objetos, não é tão simples assim. Você certamente não deve simplesmente substituir todas as suas chamadas de construtor por chamadas para o Container (ou – estremecer – o ServiceLocator), e você certamente não deve usá-lo para criar variáveis de métodos locais. Por exemplo, o código a seguir deve ser considerado um antipadrão:

```
procedimento TWidgetManager.ProcessWidget(aWidget: TWidget);
Onde
  Processador de Widget: Processador de Widget;
começar
  WidgetProcessor := ServiceLocator.GetService<IWidgetProcessor>;
  WidgetProcessor.ProcessWidget(aWidget);
fim;
```

Aqui, apenas substituímos uma chamada para TWidgetProcessor.Create por uma chamada para o Container fornecido. Isso deve ser firmemente evitado. Muitas vezes, isso é chamado de padrão ServiceLocator (ou antipadrão, como acredito) devido à tendência de usar uma classe ServiceLocator para acessar o Container. Discutirei o antipadrão ServiceLocator em um capítulo posterior.

Um Container não é meramente um dicionário de implementações e serviços de classe. Se você vê dessa forma, torna-se um grande balde de variáveis globais, e todos concordam que variáveis globais são indesejáveis. Essa visualização é semelhante à visualização “substituição para Criar” na medida em que, se você a mantiver, verá o Container como nada mais do que “um lugar para obter objetos”, quando na verdade é mais do que isso.

O que é o Contêiner?

Um Container **é** um meio de compor o gráfico de objetos do seu aplicativo. Ele gerencia a criação de dependências juntamente com seus tempos de vida. Um Container cria e gerencia dependências conforme necessário, ou melhor, antes mesmo de serem necessárias. Pense em suas aulas como balões vazios em uma caixa. Quando seu aplicativo é iniciado, você pressiona um botão e todos os balões são preenchidos com ar e prontos para serem usados. Como veremos – e continuando a metáfora – você pode instruir o Container sobre a forma correta de encher os balões e a quantidade adequada de ar para inflar cada um. Você pode atrasar a inflação dos balões, escolher que tipo de balões são inflados e escolher quais balões são compostos em cachos. O Container dá a você controle total sobre seus balões, se você quiser. Assim, torna-se mais do que uma instância glorificada do padrão de fábrica e, em vez disso, é um contêiner poderoso e flexível – sim – para seus objetos.

Por que um contêiner é necessário?

Antes de responder a essa pergunta, quero salientar que um Container de Injeção de Dependência **nem** sempre é necessário. Às vezes, o bom e velho Construtor, Propriedade e Injeção de Método são suficientes, e um Container é um exagero. Dependendo da complexidade do seu gráfico de objetos, você pode criar seus próprios objetos manualmente ou simplesmente usar fábricas para criar suas classes.

Não há vergonha em fazer o que Mark Seamann chamou de “Pure DI”. Na verdade, Pure DI pode ser bastante esclarecedor. Se você fizer isso corretamente, você eventualmente chegará à Raiz de Composição (A Raiz de Composição será discutida um pouco mais abaixo) de sua aplicação, com uma visão muito clara de como seu Gráfico de Objetos é construído.
Quais objetos seu aplicativo requer e como eles são criados serão fáceis de ver.

Pure DI também é fortemente tipado. Todos esses parâmetros de Injeção de Construtor têm um tipo anexado a eles e, se a digitação não estiver correta, o compilador irá reclamar.
O feedback em tempo de compilação é imediato e, portanto, nos permitirá evitar bugs que podem resultar quando dependemos de feedback em tempo de execução.

Dado que você não, estritamente falando, “precisa” de um DI Container, você provavelmente vai querer usar um. Você provavelmente acabará querendo um Container porque uma aplicação de qualquer consequência terá um gráfico de objeto bastante extenso na Raiz de Composição. Criar todos esses objetos pode ser complicado. Um Container pode liberar você de ter que criar tudo manualmente.

Talvez esta pergunta já tenha ocorrido a você: “Se eu continuo pedindo minhas dependências e nunca as crio, onde diabos **elas** são criadas?” Bem, essa é uma pergunta muito boa. A resposta é o que você provavelmente já imaginou – todos eles são criados no Container. O Container gerencia sua criação e sua vida útil. Ele pode, com apenas uma chamada (falaremos sobre isso abaixo), fazer tudo e gerenciar tudo o que você precisa para toda a sua aplicação.

Onde usar o contêiner

Talvez esta seja outra pergunta que você já teve: “Se eu continuar a 'empurrar' a criação de meus objetos via injeção de dependência, todas as minhas chamadas de criação não terminarão no

arquivo DPR? São muitas chamadas de criação!” Bem, você se pergunta isso porque é exatamente isso que vai acontecer. Se você usar a Injeção de Dependência de maneira dedicada – e deveria – todos os seus construtores terminarão no que é chamado de “Raiz de Composição” do seu aplicativo. Em um aplicativo Delphi, este é o bloco principal do arquivo DPR – o local onde todos os aplicativos Delphi são iniciados. E é aí que você deve usar o Container. É lá que você deve fazer uma única chamada para o Container para Resolver todo o gráfico de objetos para sua aplicação.

Você deve atrasar a conexão de suas classes – sua composição de objeto – o máximo possível. Quanto mais você adiar, mais flexível poderá ser e mais liberdade terá para decidir como fazê-lo. Fazer isso – atrasar as coisas – é o que torna possível o uso de um DI Container. Você pode atrasar as coisas para um único ponto na raiz do seu aplicativo e, em seguida, deixar o DI Container fazer a composição para você.

Quando usar o recipiente

A questão de quando usar o Container tem duas respostas.

A primeira é “o tempo todo”. Você deve sempre usar o Dependency Injection e um Dependency Injection Container em qualquer aplicação de consequência que você construir. Deve ser a maneira normal, aceita e cotidiana de criar aplicativos. Deve fazer parte da sua caixa de ferramentas tanto quanto números inteiros, classes e listas.

Eu aludi à segunda resposta acima: você deve usar o Container uma vez e apenas uma vez na raiz composta do seu aplicativo. Neste ponto, você provavelmente está se perguntando como isso é possível – há muitas classes! – mas asseguro-lhe que pode e deve ser feito. Discutirei isso nos próximos capítulos.

Mas é claro que o uso de um Dependency Injection Container tem custos associados e, naturalmente, você deve considerar esses custos antes de tomar a decisão de usar um. O principal custo do uso de um DI Container é o tempo necessário para aprendê-lo. (Na verdade, você tem que comprar e ler um livro inteiro sobre o assunto!) Planejar e projetar para o uso de um DI Container não vem de graça. Além disso, há a sobrecarga de adicionar uma estrutura não trivial ao seu aplicativo.

Outro custo envolvido com o uso de um DI Container é a perda de verificação de tipo em tempo de compilação. Isso não é trivial. O compilador é sua primeira linha de defesa contra erros,

e se você usar a velha e simples injeção de dependência, o compilador geralmente lhe dirá se você cometeu um erro. No entanto, com um DI Container, você perde isso. Como seu gráfico de objeto é criado e vinculado em tempo de execução, você receberá erros de tempo de execução se cometer um erro. Se você esquecer de registrar uma implementação para uma determinada interface, não saberá disso até tentar obter essa implementação ausente em tempo de execução. Este não é um custo pequeno. Você definitivamente precisará tomar cuidado para garantir que seu código seja composto adequadamente para evitar isso.

No entanto, se alguém usa um DI Container corretamente, os benefícios devem superar em muito os custos. Esses benefícios são sobre os quais falarei no restante deste livro.

Capacidades e Funcionalidade

Que recipiente usar?

Você deve usar um contêiner que você cria para si mesmo. O framework Spring for Delphi fornece um GlobalContainer singleton, mas o uso dele deve ser evitado, pois é, bem, uma variável global. Em vez disso, você deve criar sua própria instância de um contêiner e usá-la para registrar e resolver suas implementações.

Registro de Objeto Simples

Registrar uma classe com o Container é tão fácil quanto pode ser:

```
MyContainer.RegisterType<TMyClass>;
```

É isso. Agora seu Container conhece o TMyClass e pode recuperar uma instância do TMyClass sempre que precisar. Se outra classe tiver TMyClass como dependência, o Container saberá como obtê-lo automaticamente. Então, por exemplo, digamos que você registre outra classe que tenha TMyClass injetado no construtor:

```
tipo
    TAnotherClasss = class
      construtor Create(aMyClass: TMyClass); fim;

    ....

    MyContainer.RegisterType<TAnotherClass>
```

Então, se você fizer referência a TAnotherClass, o Container simplesmente fornecerá uma instância de TMyClass para seu construtor. Não há necessidade de você fazer nada. O Container verá sua necessidade, encontrará o construtor mais específico que possui todas as dependências que conhece e criará a classe para você. Na verdade, ele fará tudo isso antes mesmo de você pedir. Tudo isso faz parte de “criar seu gráfico de objetos” para você. Este é um exemplo simples de como o Container é mais do que apenas um saco de classes esperando para ser usado.

Implementação de Interface

Mencionei anteriormente que você deve sempre codificar em uma abstração – geralmente uma interface – e não uma implementação. Bem, o Container torna isso muito fácil de fazer. Ao registrar uma interface e uma implementação juntas, você pode dizer ao Container que uma determinada classe implementa uma interface e, como acima, quando você precisar de uma implementação para uma determinada interface, o Container a fornecerá:

```
MyContainer.RegisterType<ILogger, TLogger>;
```

> Leitores astutos podem notar que esta é uma sintaxe diferente para registrar classes em interfaces de meus livros anteriores. Anteriormente, você os registrava usando o método Implements , mas a maneira acima de fazer o registro agora é o método preferido para associar uma classe e uma interface.

O código acima basicamente diz “Registre a interface ILogger e use a implementação TLogger para isso”. Desta forma, você pode usar interfaces ao fazer Injeção de Dependência, e o Container irá apoiá-lo em sua busca para sempre codificar contra abstrações.

Várias implementações por interface

Mas e se você tiver mais de uma classe que implementa a mesma interface? Bem, você pode registrar classes pelo nome:

```
MyContainer.RegisterType<ILogger, TFileLogger>('file');
MyContainer.RegisterType<ILogger, TConsoleLogger('console');
MyContainer.RegisterType<ILogger, TDatabaseLogger('database');
```

Dado o exposto, você pode escolher a implementação desejada entre os três registros com base no nome.

Gerenciamento vitalício

Um dos principais recursos do Container é a capacidade de gerenciar o tempo de vida dos objetos que ele fornece a você. No Delphi, é claro, você é responsável por garantir o descarte adequado da memória alocada pela construção de um objeto. Se você tiver o Container alocando o objeto para você, poderá instruir o Container sobre como gerenciar o tempo de vida desse objeto.

No Delphi, você geralmente é responsável por liberar qualquer objeto que você criar. Na VCL, há a exceção de que qualquer objeto VCL pertencente a outro objeto VCL será liberado pelo objeto proprietário. Além disso, se seu objeto for referenciado como uma interface, o compilador fará a contagem de referência e liberará automaticamente seu objeto quando a referência sair completamente do escopo e a contagem de referência for

para zero.

Mas se você transferir a responsabilidade de criar seu gráfico de objetos para o Depen dency Injection Container, então o Container assume a tarefa de gerenciar o tempo de vida dos objetos que ele cria. Isso significa que você não precisa fazer isso porque o Container é responsável pelo tempo de vida de um objeto. Isso também significa que você precisa informar ao Container como você gostaria de ter o tempo de vida de seus objetos gerenciados.

Você pode fazer isso com chamadas para o Container conforme definido pela tabela abaixo:

Tipo	Descrição
AsTransiente	Este é o padrão. Ele cria uma instância por solicitação e essa solicitação dura enquanto a variável estiver no escopo. Em outras palavras, uma nova instância é criada para cada solicitação.
Como Único	Cria uma única instância da classe e retorna essa instância para cada solicitação feita para essa classe. Essa única classe é usada para todas as referências ao objeto.
AsSingeltonPerThread	O mesmo que AsSingleton, mas cria uma instância por thread.
Como Agrupado	Cria um pool de objetos de tamanho configurável pelo usuário e distribui instâncias desse pool.

Até agora você deve ter notado que o Spring4D Container usa a interface fluente. Uma olhada no código-fonte mostra que cada chamada para o Container sempre retorna uma instância de TRegistration<T>, permitindo que você encadeie chamadas para o Container juntas. TRegistration<T> é uma classe que gerencia toda a funcionalidade do Container, ao mesmo tempo em que permite criar uma única declaração que define todo um registro de uma classe. Assim, por exemplo, você pode ter declarações como as seguintes:

```
    MyContainer.RegisterType<IFarma, TRifle>('rifle').AsSingleton.InjectProperty('MetalSight', 'sight'\
).InjectField('Clip');
```

que registrará uma classe em uma interface e a declarará como singleton, além de injetar um valor de propriedade e um valor de campo. É uma maneira poderosa e fácil de ler para declarar dependências. Interfaces fluentes geralmente são lidas como frases.

Se você deseja que sua referência seja um singleton - ou seja, uma instância para todas as chamadas para o registro fornecido - declare-o da seguinte maneira:

```
MyContainer.RegisterType<IWeapon, TSword>.AsSingleton;
```

Isso garantirá que sempre que você solicitar uma IWeapon, você receberá de volta a mesma instância exata do TSword para todas as solicitações.

AsSingletonPerThread faz o que seu nome indica – ele fornece a mesma instância para cada solicitação dentro de um determinado thread. Pode haver várias instâncias da implementação, mas cada thread sempre terá sua própria instância.

AsTransient é o comportamento padrão. AsTransient criará uma nova instância para cada solicitação feita. Supondo que você esteja usando uma referência de interface, essa instância transitória viverá enquanto a interface permanecer no escopo (ou seja, enquanto sua contagem de referência for maior que zero, presumindo que você esteja usando uma contagem de referência “normal”).

Transient é a menos eficiente das opções de gerenciamento de tempo de vida, porque pode fazer com que o Container crie um grande número de classes que vivem por muito tempo.

AsPooled criará um pool de instâncias para uso mediante solicitação. Você pode determinar o número mínimo e máximo de itens no pool. Quando um item é resolvido pelo Container, ele será recuperado do pool de itens. Todos os itens do pool são criados quando o Container é construído e, portanto, todos estão disponíveis quando o programa é iniciado. Use o pooling quando a criação for cara e você quiser pagar esse custo antecipadamente ou quando tiver um número limitado de recursos e quiser garantir que não sejam criadas mais instâncias da classe que representa o recurso do que recursos. Itens agrupados nunca são compartilhados, então esse também pode ser um motivo para escolhê-los.

Delegar

Às vezes, o construtor de sua classe pode não cooperar perfeitamente com o Container e ser uma dependência resolvível. Considere a seguinte classe:

```
tipo
  IWindowsUser = interface
  ['{432973CE-CDDF-45CC-9BA0-EC089F23EAF4}']
    função GetUserName: string;
    propriedade UserName: string lida GetUserName;
  fim;

  TWindowsUser = classe (TInterfacedObject, IWindowsUser ) privado

    FUserName: string;
    função GetUserName: string;
  construtor público Create(const
    aUserName: string); propriedade Username: string read
    GetUsername;
  fim;
```

Primeiro, declaramos uma interface, IWindowsUser, que representa um nome de usuário em um sistema Windows. Também fornecemos uma classe de implementação que recebe uma string como parâmetro para o construtor. Gostaríamos que essa interface fosse implementada dentro do nosso Container, então declaramos o seguinte:

```
MyContainer.RegisterType <IWindowsUser, TWindowsUser> .DelegateTo (função: TWindowsUser
                                                                    begin
                                                                      Resultado := TWindowsUser.Create(GetLocalUserName); fim)
```

Em seguida, registramos TWindowsUser como implementando a interface IWindowsUser . Lembre-se, para que uma classe tenha seus parâmetros de construtor resolvidos automaticamente, você precisa registrar essa classe com o Container. No entanto, aqui há um problema: o construtor usa uma string como parâmetro – uma string que muda toda vez que um novo usuário faz logon no Windows. Assim, não está claro como o Container deve criar uma instância de TWindowsUser. Portanto, definimos para o Container exatamente como queremos que TWindowsUser seja construído para nós por meio de uma chamada para DelegateTo. Esse método usa uma função anônima que retorna um TWindowsUser, dando a você a oportunidade de informar ao Container como criar um TWindowsUser.

Como uma observação lateral, GetLocalUserName é declarado da seguinte forma:

```
função GetLocalUserName: string;
Onde
  aComprimento:
  DWORD; aUserName: array [0 .. Max_Path - 1] de Char;
começar
  aLength := Max_Path; se
  não GetUserName(aUserName, aLength) então comece
  a aumentar Exception.CreateFmt('Win32 Error %d: %s',
    [GetLastError, SysErrorMessage(GetLastError)]); fim; Resultado := string(aUserName); fim;
```

Use DelegateTo quando o Container não souber como criar uma instância de uma classe. Isso pode acontecer porque os parâmetros do construtor não podem ser resolvidos pelo Container ou porque as informações necessárias ao construtor dependem de informações externas que não podem ser armazenadas no Container.

Há algumas ressalvas ao usar DelegateTo. Você deve ter muito cuidado para limitar o uso de DelegateTo e encontrar maneiras de o Container resolver esse problema. Por outro lado,

você pode facilmente acabar com muitas chamadas para DelegateTo , que apenas chamam o Container. Isso realmente se torna apenas outra forma de Pure DI, mas com a sobrecarga do contêiner incluída.

Conclusão

Esse é o básico do Container. Um DI Container é uma ferramenta extremamente útil. Ele pode permitir que você, em um único ponto com uma única linha de código, componha seu gráfico de objeto inteiro. Assim, sua aplicação pode ser composta por muitas classes fracamente acopladas que solicitam suas dependências e que nada sabem do próprio Container. Mais tarde, abordarei alguns tópicos mais avançados que sobrecarregam o que você pode fazer com o Container, mas por enquanto você deve ter ferramentas suficientes para dar uma olhada em um exemplo básico que ilustra como o Container funciona e como ele deve ser usado . É o que farei no próximo capítulo.

Um exemplo passo a passo

Até agora, vimos exemplos simples que mostram os recursos específicos do Container. Mas como tudo isso realmente funciona? Do que realmente estamos falando aqui? Neste capítulo, vamos dar uma olhada em um exemplo que começa “normal” – como as coisas são feitas tradicionalmente – e então passar para mostrar como ele pode ser refatorado para ser limpo e desacoplado, bem como fácil de testar e manter aproveitando o poder de um Container de Injeção de Dependência.

O início

O código para este capítulo pode ser encontrado em: http://bit.ly/diidcode

Primeiro, começaremos na raiz do nosso aplicativo de demonstração, a chamada para DoOrderProcess ing, que é chamada no arquivo DPR do projeto:

```
procedimento DoOrderProcessing;
Onde
    Ordem: TOOrdem;
    Processador de Pedidos: TOProcessador de Pedidos;
começar
    Ordem := TOOrder.Criar; tente
    OrderProcessor :=
        TOrderProcessor.Create; tente se OrderProcessor.ProcessOrder(Order)
        então comece WriteLn('Pedido processado com sucesso....');



            fim;
        finalmente
            OrderProcessor.Free; fim;
        finalmente


        Pedido. Gratuito;
    do que;
do que;
```

Aqui, vemos um código bastante típico. Um pedido é criado e processado por um processador de pedidos. Uma vez que o processamento é feito, o processador é liberado. Código muito simples e básico. Você provavelmente já fez algo assim um milhão de vezes.

Como é o TOrderProcessor ?

```
tipo
   TOOrderProcessor = classe
   privado
      FOrderValidator: TOOrderValidator;
      FOEntrada: TOEntrada; construtor
   público Criar; destruidor Destruir;
      sobrepor; função
      ProcessOrder(aOrder: TOrder):
      Boolean; fim;


TOrderProcessor constructor.Create ;
começar
   FOrderValidator : = TOrderValidator.Criar; FOrderEntry :
   = TOrderEntry.Criar;
fim;

destruidor TOrderProcessor.Destroy; começar

   FOrderValidator.Free;
   ParaEntrada.Gratuita;
   herdado; fim;


função TOrderProcessor.ProcessOrder (aOrder: TOrder): Boolean;
Onde
   OrderIsValid: Boolean;
começar
   Resultado := Falso;
   OrderIsValid := FOrderValidator.ValidateOrder(aOrder); se OrderIsValid
   então
   begin
      Result := FOrderEntry.EnterOrderIntoDatabase(aOrder); fim; WriteLn('O
   pedido foi processado...'); fim;
```

Aqui estão algumas coisas a serem observadas sobre a declaração e implementação do TOrder Processador:

- O construtor não tem parâmetros. Como resultado, as duas dependências, TOrder Validator e TOrderEntry, são codificadas por meio de suas chamadas de construtor no

construtor de TOrderProcessor. Nós nos acoplamos fortemente às implementações fornecidas dessas duas classes.

- Imediatamente isso torna essa classe difícil de testar porque não podemos substituir essas duas dependências por falsificações. • Como é normal, temos que gerenciar o tempo de vida das dependências manualmente, resultando em um destruidor para a classe.

Apresentando a injeção de construtor

Vamos dar alguns passos para melhorar as coisas. Primeiro, usaremos a injeção de dependência para injetar as dependências do TOrderProcessor – especificamente algumas Injeções de Construtor:

```
tipo
  TOOrderProcessor = classe
  privado
    FOrderValidator: TOOrderValidator;
    FORDERENTRY: TOOrderEntry;

  construtor público Create (aOrderValidator: TOrderValidator; aOrderEntry: TOrderEntry); função ProcessOrder
    (aOrder: TOrder): Booleano;
  fim;

Construtor TOrderProcessor.Create (aOrderValidator: TOrderValidator; aOrderEntry: TOrderEntry); begin FOrderValidator : =
aOrderValidator; FOOrderEntry : = aOrderEntry; fim;



função TOrderProcessor.ProcessOrder (aOrder: TOrder): Boolean;
Onde
  OrderIsValid: Boolean;
começar
  Resultado := Falso;
  OrderIsValid := FOrderValidator.ValidateOrder(aOrder); se OrderIsValid
  então
  begin
    Result := FOrderEntry.EnterOrderIntoDatabase(aOrder); fim; WriteLn('O
  pedido foi processado...'); fim;
```

Observe que não somos mais responsáveis pelo tempo de vida das dependências dentro da função – o chamador é – e, portanto, o destruidor se foi.

Isso, é claro, muda um pouco nosso procedimento DoOrderProcessing , pois “retrocedemos” a criação para mais perto da base do aplicativo:

```
procedimento DoOrderProcessing;
Onde
  Ordem: TOOrdem;
  Processador de Pedidos: TOProcessador de Pedidos;
  Validador de Pedidos: TOValidador de Pedidos;
  OrderEntry: TOOrderEntry;
começar
  Ordem := TOOrder.Criar; tente
  OrderValidator :=
    TOrderValidator.Create; OrderEntry := TOrderEntry.Create;
    OrderProcessor := TOrderProcessor.Create(OrderValidator,
    OrderEntry); tente se OrderProcessor.ProcessOrder(Order) então comece WriteLn('Pedido processado
    com sucesso....'); fim;



    finalmente
      OrderProcessor.Free;
      OrderValidator.Free;
      OrderEntry.Free;
    fim;
  finalmente
    Order.Free;
  fim; fim;
```

Aqui, criamos o validador de pedidos e as classes de entrada de pedidos na “raiz” da aplicação. (DoOrderProcessing é o que chamamos no arquivo DPR). Já com esta simples mudança, tornamos o TOrderProcessor um pouco mais flexível. Enquanto ainda nos acoplamos às implementações específicas, o próprio TOrderProcessor não está mais vinculado especificamente a qualquer instância de TOrderValidator e TOrderEntry. Pode aceitar uma falsificação dessa classe, ou qualquer descendente das respectivas classes. Não é muito, mas é alguma coisa.

Codificando para uma interface

Como afrouxamos ainda mais esse acoplamento? Apresentamos interfaces, é claro.
Lembre-se, sempre queremos codificar para uma interface, não para uma implementação. Até agora, codificamos para uma implementação e você pode ver as limitações que isso está causando – estamos vinculados a essas implementações. Vamos nos desamarrar.

Primeiro, vamos declarar algumas interfaces:

```
  IOrderValidator = interface
  ['{CF8834A3-F815-4F6B-A177-7AB801BEC95E}']
    função ValidateOrder(aOrder: TOrder): Boolean; fim;


  IOrderEntry = interface
  ['{406EA68D-0733-429E-9E48-73BC660B1C72}'] função
    EnterOrderIntoDatabase(aOrder: TOrder): Boolean;
  fim;

  IOrderProcessor = interface
  ['{C690B9D5-8C26-4DFE-AD27-2D7A4610ACBC}']
    função ProcessOrder(aOrder: TOrder): Boolean;
  fim;
```

Uma vez que estes tenham sido declarados, podemos codificar contra eles em vez de contra implementações:

```
tipo
  TOrderProcessor = classe (TInterfacedObject, IOrderProcessor) privado

    FOrderValidator: IOrderValidator;
    FOEntrada: IOEntrada; construtor
  público Create (aOrderValidator:
    IOrderValidator; aOrderEntry: IOrderEntry); função ProcessOrder (aOrder: TOrder): Booleano;

  fim;

construtor TOrderProcessor.Create (aOrderValidator: IOrderValidator; aOrderEntry: IOrderEntry); começar

  FORDERVALIDATOR := aOrderValidator;
  FOOrderEntry := aOrderEntry;
fim;

função TOrderProcessor.ProcessOrder (aOrder: TOrder): Boolean;
Onde
  OrderIsValid: Boolean;
começar Resultado := False;
  OrderIsValid :=
  FOrderValidator.ValidateOrder(aOrder); se OrderIsValid então

  começar
    Resultado := FOrderEntry.EnterOrderIntoDatabase(aOrder);
  fim;
  WriteLn('O pedido foi processado...');
fim;
```

Primeiro, alteramos todas as referências ao validador de pedidos e às classes de entrada de pedidos para serem interfaces em vez de tipos de classe. Isso significa que agora podemos realmente passar

qualquer implementação - incluindo falsificações para fins de teste - para TOrderProcessor. Como estamos codificando em uma interface, também podemos remover as chamadas para Free. Veja como é a chamada para DoOrderProcessing agora:

```
procedimento DoOrderProcessing;
Onde
  Ordem: TOOrdem;
  Processador de Pedidos: IOrderProcessor;
  Validador de Pedidos: IOrderValidator;
  OrderEntry: IOrderEntry; começar
OrderValidator : = TOrderValidator.Create;
  OrderEntry : = TOrderEntry.Create;


  Ordem := TOOrder.Criar; tente
  OrderProcessor :=
    TOrderProcessor.Create(OrderValidator, OrderEntry); se OrderProcessor.ProcessOrder(Order)
    então
    begin
      WriteLn('Pedido processado com sucesso...');
    fim;

  finalmente
  Order.Free; fim;
fim;
```

Isso já é muito mais simples. Ainda estamos codificados para as implementações específicas de TOrderValidator e TOrderEntry, mas estamos mais uma etapa removida desse acoplamento porque suas referências são interfaces. O código já está parecendo muito mais limpo como resultado.

Essas duas chamadas Create podem ser “recuadas” ainda mais – vamos colocá-las diretamente na raiz da composição para que DoOrderProcessor não exija o acoplamento rígido a elas. Aqui está o novo DoOrderProcessing:

```
procedimento DoOrderProcessing (aOrderValidator: IOrderValidator; aOrderEntry: IOrderEntry);
Onde
    Ordem: TOOrdem;
    Processador de Pedidos: IOrderProcessor;
começar
    Ordem := TOOrder.Criar; tente
    OrderProcessor :=
        TOrderProcessor.Create(aOrderValidator, aOrderEntry); se OrderProcessor.ProcessOrder(Order) então
        comece WriteLn('Order processado com sucesso....'); fim; finalmente




        Pedido. Gratuito;
    do que;
do que;
```

e aqui está o código no arquivo DPR que agora o chama:

```
Onde
    Validador de Pedidos: IOrderValidator; OrderEntry:
    IOrderEntry; comece a tentar OrderValidator : =
TOrderValidator.Create; OrderEntry : =
    TOrderEntry.Create; DoOrderProcessing
        (OrderValidator, OrderEntry); ReadLn;



    exceto
        em E: Exceção do
            WriteLn(E.ClassName, ': ', E.Message);
    fim;
fim.
```

Neste ponto, levamos a Injeção de Dependência o mais longe possível. Nós empurramos a criação de nossos objetos até o Composite Root do nosso aplicativo – neste caso, o bloco principal do arquivo DPR no Delphi. Aqui criamos todas as nossas dependências logo no início do arquivo DPR. Não podemos empurrar as coisas mais para trás do que isso.

Agora, isso é ótimo e tudo – toda a nossa criação de dependência é centralizada na raiz do aplicativo – mas isso pode levar a um grande problema. Se nossa aplicação ficasse muito mais complicada e continuássemos seguindo esse padrão, teríamos uma enorme pilha de coisas sendo criadas no arquivo DPR. Isso seria bom, mas poderia ficar muito desajeitado muito rápido. Desajeitado não é uma palavra que eu gostaria de descrever meu código. Ah, se ao menos houvesse uma maneira de gerenciar toda essa criação!

Entre no Contêiner

Claro, existe uma maneira de gerenciar toda essa criação – o Dependency Injection Container. Podemos pegar toda aquela criação de classes que iriam se acumular e colocar tudo no Container. Vamos fazer isso agora.

Vamos criar uma nova unidade e colocar todo o registro lá. Dessa forma, tudo fica centralizado, mas bem escondido. Chamaremos a unidade de uRegistration.pas e faremos com que fique assim:

```
unidade uRegistro;

interface

procedimento RegisterClassesAndInterfaces;

implementação

usa
      Spring.Container
    , uOrderEntry
    , uValidador de Pedidos
    , uOrderProcessor
    ;

procedimento RegisterClassesAndInterfaces (aContainer: TContainer); comece
umContainer.RegisterType <IOrderProcessor, TOrderProcessor> .AsSingleton;
  aContainer.RegisterType <IOrderValidator, TOrderValidator> .AsSingleton; aContainer.RegisterType
  <IOrderEntry, TOrderEntry> .AsSingleton; aContainer.Build;


fim;

fim.
```

O que fizemos aqui é bastante simples. Primeiro, registramos o tipo de processador TOrder com o contêiner. Isso permite que o Container saiba sobre a classe e, como veremos, permite que o Container resolva todas as suas dependências automaticamente. Além disso, registramos as duas classes para as quais temos dependências como implementações específicas. Isso fará com que o contêiner resolva referências às interfaces usando as classes concretas registradas nelas. O Container, quando solicitado por uma instância de IOrderValidator, poderá fornecer uma instância válida de TOrderValidator para implementá-lo. Muito legal. Finalmente, como todas as classes podem ser

singletons – eles sempre serão solicitados a fazer exatamente a mesma coisa – nós os registramos com a chamada de gerenciamento vitalício do AsSingleton.

E agora que registramos essas classes e interfaces, coisas legais acontecem. A primeira é que o arquivo DPR se torna realmente simples. O coração da DPR acaba ficando assim:

```
Onde
  Recipiente: TContainer;
começe tente Container :=
  TContainer.Create; tente
    RegisterClassesAndInterfaces(Container);
    DoOrderProcessing; ReadLn; finalmente
      Container.Free; fim; exceto em E: Exceção do
      Writeln(E.ClassName, ': ', E.Message);
  fim;
fim.
```

Há algumas coisas a serem observadas neste código:

- Se você baixou e executou o aplicativo de demonstração deste capítulo, notará que todo o código é executado conforme o esperado. À primeira vista, isso pode ser uma surpresa. Mas é claro que tudo será explicado.
- As variáveis para as duas dependências desapareceram - não precisamos eles mais como veremos.
- Os parâmetros em DoOrderProcessing também desapareceram. Também não precisamos deles.
- Nada está sendo criado no código. Isso é um pouco misterioso. Mas, novamente, tudo vai ser explicado.
- Há uma chamada para Container.Build. Essa chamada é necessária para que o Container saiba construir o gráfico do objeto. Isso é tudo que você precisa para que o Container faça sua mágica. Você deve fazer essa chamada uma vez e apenas uma vez na raiz do aplicativo, como fizemos aqui. Chamá-lo mais de uma vez apenas fará com que o processo seja executado duas vezes e desperdiçará ciclos de CPU.

As coisas realmente legais acontecem dentro da chamada para DoOrderProcessing. Aqui está em sua encarnação atual:

```
procedimento DoOrderProcessing(aContainer: TContainer);
Onde
    Ordem: TOOrdem;
    Processador de Pedidos: IOrderProcessor;
começar
    Ordem := TOOrder.Criar; tente
    OrderProcessor :=
        aContainer.Resolve<IOrderProcessor>; se OrderProcessor.ProcessOrder(Order)
        então comece WriteLn('Order processado com sucesso....');


        fim;
    finalmente
        Pedido. Gratuito;
    do que;
do que;
```

Observe, novamente, que os parâmetros desapareceram. Em seguida, observe que a referência a OrderProcessor é satisfeita por uma única chamada para aContainer.Resolve. Esta única chamada faz algumas coisas. Primeiro, você notará que ele instancia completamente uma implementação de IOrderProcessor – neste caso, uma instância de TOrderProcessor.
E é aí que acontece a “mágica” que faz do Container mais do que uma fábrica.

Lembre-se da declaração de TOrderProcessor?

```
TOrderProcessor = classe (TInterfacedObject, IOrderProcessor) privado

    FOrderValidator: IOrderValidator;
    FOEntrada: IOEntrada; construtor
público Create (aOrderValidator:
    IOrderValidator; aOrderEntry: IOrderEntry); função ProcessOrder (aOrder: TOrder): Booleano; fim;
```

Como você pode ver, o construtor usa duas dependências. Portanto, todo esse código é executado e em nenhum lugar criamos instâncias para anexar a esses dois parâmetros de interface. O que está acontecendo aqui?

O que está acontecendo é que o Container é esperto o suficiente para saber o que fazer. Como você registrou implementações para ambas as interfaces, o Container pode

veja o que você registrou e crie uma instância de TOrderProcessor instanciando corretamente seus parâmetros de construtor. É como o Container diz para si mesmo:

"OK. Me pediram para criar uma implementação para IOrderProcessor. Vejo que tenho TOrderProcessor registrado como implementando IOrderProcessor. Então, eu preciso criar uma instância de TOrderProcessor. Ei, olha, tem um construtor! Isso é ótimo, mas, uh oh, tem duas dependências que preciso resolver. O primeiro é IOrderValidator. Ei, eu tenho uma classe registrada para essa interface! E eu tenho um para IOrderEntry também! Que alivio! Isso significa que posso criar instâncias dessas classes para as interfaces, passar essas duas implementações para o construtor de TOrderProcessor e retornar isso como uma implementação de IOrderProcessor.

E então eu terminei!"

Tudo isso acontece "automagicamente" dentro do Container. Rapaz, esse Container com certeza é inteligente, não é?

Como resultado dessa inteligência, conseguimos fazer algumas coisas:

- Reduza drasticamente a quantidade de código que temos que escrever. Como o contêiner faz toda a criação, não temos chamadas para Create. Quanto menos código tivermos que escrever, menos código teremos que manter. Isso é uma vitória. • Tornar nosso código pouco acoplado. Observe que todas as classes são independentes, cada uma em suas próprias unidades. Podemos colocar as interfaces em sua própria unidade, e o único lugar em que as coisas se juntam é na unidade que registra tudo com o container, e mesmo aí a relação é meramente entre uma interface e a implementação. Isso é muito fino e, portanto, resulta em acoplamento fraco. Dito de outra forma, só temos a Connascência do Nome com que nos preocupar.

- O teste fica muito fácil porque cada classe é independente e declara claramente suas dependências. Dependências claramente declaradas significam que você pode substituir facilmente essas dependências por falsificações, tornando as classes facilmente testáveis.

  A propósito, se você registrar uma classe que tenha mais de um construtor, o container irá olhar para cada um e encontrar o mais específico que ele pode resolver. Ou seja, se você tiver um construtor com uma dependência resolvível, bem como um com duas dependências resolvíveis, então o

container usará o segundo. No entanto, recomendo que sua classe tenha apenas um construtor que declare todas as dependências da classe.

Conclusão

E é assim que o Container pode construir um gráfico de objeto inteiro enquanto faz apenas uma chamada Resolve para o Container. Fazer isso resulta em um código muito legal. Ao usar o Container para fazer todo o nosso trabalho, evitamos escrever muito código de canalização, como criar e destruir objetos, permitindo que nos concentremos em escrever a lógica de negócios. E como eu te acertei agora, seu código será fracamente acoplado e, portanto, fácil de manter e testar. â€ƒ

Uso avançado de contêiner

Até agora, mostrei a você o básico da injeção de dependência, incluindo injeção de construtor, injeção de propriedade (ou setter) e injeção de método. Dei uma olhada no básico de como funciona o contêiner de injeção de dependência e mostrei um exemplo básico do contêiner fazendo sua “mágica”. Agora é hora de dar uma olhada em algumas coisas avançadas que o container pode fazer para tornar seu código ainda mais poderoso.

Várias implementações em tempo de execução

Como mencionei antes, você pode registrar várias implementações em uma única interface. Nesta seção, vamos dar uma olhada nisso.

Quem não ama frutas? Bem, a fruta tem que ser cultivada em algum lugar, e uma vez crescida, essa fruta tem que ser colhida. Mas existem muitas maneiras diferentes de colher frutas. Então, se quisermos modelar a colheita de frutas, faz sentido declarar uma interface:

```
tipo
  IFruitPicker = interface
  ['{DD861C60-D9A0-411C-8448-2E3B798026DB}']
    procedimento PickFruit;
  fim;
```

Agora que temos uma interface para programar, podemos colher frutas da maneira que quisermos. Aqui estão três implementações:

```
tipo
  THumanFruitPicker = class (TInterfacedObject, IFruitPicker)
    procedimento PickFruit;
  fim;

  TMechanicalFruitPicker = class(TInterfacedObject, IFruitPicker)
    procedimento PickFruit;
  fim;

  TAndroidFruitPicker = class (TInterfacedObject, IFruitPicker)
    procedimento PickFruit;
  fim;

procedimento THumanFruitPicker.PickFruit;
begin WriteLn('Colha com cuidado a fruta.....'); fim;



procedimento TMechanicalFruitPicker.PickFruit; begin
WriteLn('Colha a fruta com um dispositivo mecânico...');

fim;

procedimento TAndroidFruitPicker.PickFruit; begin
WriteLn('Colha a fruta com robôs semelhantes a
  androides....'); fim;


fim.
```

Agora podemos ter humanos, máquinas e robôs andróides colhendo as frutas. Como podemos registrar todos esses métodos de colheita de frutas com o recipiente e escolher um para colher a fruta? Bem, da seguinte forma:

```
procedimento RegisterFruitPickers(aContainer: TContainer); começar
aContainer.RegisterType<IFruitPicker, THumanFruitPicker>('humano').AsDefault;
  aContainer.RegisterType<IFruitPicker, TMechanicalFruitPicker>('mecânico');
  aContainer.RegisterType<IFruitPicker, TAndroidFruitPicker>('android');

fim;
```

Observe que cada um dos três registros inclui uma string exclusiva que identifica o registro. Dessa forma, podemos escolher a implementação que queremos pelo nome em tempo de execução. Observe também que o THumanFruitPicker foi declarado com o método AsDefault . Isso significa que THumanFruitPicker será o usado se um

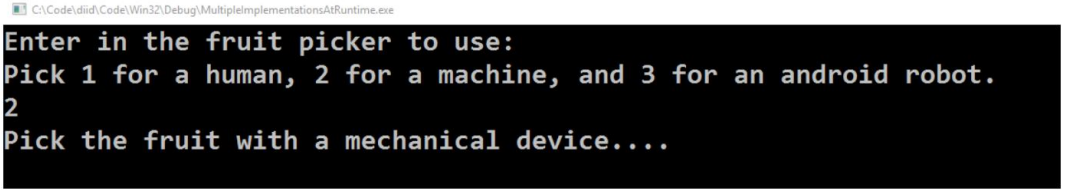
solicitação é feita para um IFRuitPicker sem especificar um nome.

Aqui está o arquivo DPR que permite escolher qual colhedora de frutas usar:

```
tipo
  TPickerType = (humano, mecânico, android);

Onde
  Nome: cadeia;
  Seleção: inteiro;
  Seletor de Frutas: IFruitPicker;
  Recipiente: TContainer;


começe tente Container := TContainer.Create;
    tente RegisterFruitPickers(Container);


      WriteLn('Digite o colhedor de frutas para usar: '); WriteLn('Escolha
      1 para um humano, 2 para uma máquina e 3 para um robô androide.'); ReadLn(Seleção);


      Nome := GetEnumName(TypeInfo(TPickerType), Seleção - 1); FruitPicker :=
      Container.Resolve<IFruitPicker>(Nome); FruitPicker.PickFruit;

    finalmente
      Container.Free;
    fim;
  exceto
    em E: Exceção do
      WriteLn (E.ClassName, ': ', E.Message); fim; ReadLn;

fim.
```

Aqui declaramos uma enumeração simples que define os três colhedores de frutas. Em seguida, permitimos que você escolha um número para o apanhador de frutas que deseja e use uma pequena informação de tipo (da unidade TypInfo.pas) para obter uma string que usamos para resolver a interface IFruitPicker com a implementação que queremos.

Executando o aplicativo e selecionando “2”, o selecionador de máquina, obtém o seguinte resultado no console:



Resultados da colheita da máquina de colheita de frutas em tempo de execução

Assim, você pode registrar várias implementações para uma determinada interface e escolher a que quiser quando quiser.

Inicialização lenta

Quando você chama Container.Build, o Container irá construir todo o gráfico do objeto de uma vez. Isso é bom – seu aplicativo deve estar pronto para executar as coisas quando elas forem necessárias. No entanto, às vezes, uma determinada implementação pode ser cara para criar, ou talvez seja uma implementação que provavelmente não será usada durante a execução normal do seu aplicativo. Nesse caso, você pode querer fazer uma inicialização “preguiçosa” da implementação. Inicialização lenta significa que a construção de um determinado objeto é atrasada até que seja realmente solicitada. Dessa forma, talvez você possa economizar alguns recursos que não precisam ser alocados imediatamente ou impedir que esses recursos sejam alocados se raramente forem usados.

O Spring4D Framework fornece um meio para você fazer uma inicialização lenta por meio do registro Lazy<T> . Você pode declarar seu tipo como Lazy e a estrutura não criará o tipo fornecido até que seja solicitado.

Eu acho que um exemplo está em ordem. As conexões de banco de dados geralmente levam tempo para serem criadas e, muitas vezes, você deseja criá-las apenas quando necessário. Vamos dar uma olhada em um exemplo de inicialização lenta de uma conexão de banco de dados.

Como sempre, começaremos declarando uma interface:

```
tipo
  IDatabaseConnector = interface
  ['{8E00247F-F910-41C1-9122-523F2CB6E5CB}']
    procedimento Connect(const aName: string);
  fim;
```

Esta é apenas uma demonstração, portanto, não faremos conexões reais com o banco de dados, mas sim simularemos isso causando um pouco de atraso ao fingir “conectar”. Então, dado isso, aqui está uma implementação da nossa interface IDatabaseConnector :

```
  TDatabaseConnector = class(TInterfacedObject, IDatabaseConnector) private

    FConectado: Booleano;
  público
    construtor Criar;
    procedimento Connect(const aName: string);
    propriedade Connected: leitura booleana FConnected write FConnected;
  fim;

construtor TDatabaseConnector.Create; comece
herdado Criar; FConectado := False;
  Connect(Self.ClassName); fim;


procedimento TDatabaseConnector.Connect(const aName: string); começar

  se não conectado então
  begin
    WriteLn('Agora conectando com ', aName);
    Sono(3000); WriteLn('Conectado! Desculpe por
    demorar tanto!'); Conectado := Verdadeiro; fim; fim;
```

Essa classe implementa a interface IDatabaseConnector , mas demora um pouco para se conectar. Ele “se conecta” gravando no console, aguardando três segundos e, em seguida, gravando novamente no console que está concluído. Novamente, esta é uma conexão de demonstração, mas é suficiente para nossos propósitos, como você verá.

A verdadeira diversão começa quando declaramos esse tipo em uma classe que realmente o usa. Por exemplo, aqui está uma classe, TDatabaseConnectionManager, que aceita um parâmetro de conector IDatabaseC no construtor, que preenche um campo no objeto.

```
  TDatabaseConnectionManager = classe
  privada
    FDatabaseConnector: IDatabaseConnector;

  construtor público Create(aDatabaseConnector: IDatabaseConnector); fim;

construtor TDatabaseConnectionManager.Create(aDatabaseConnector: IDatabaseConnector);
começar
  herdado Criar;
  FDatabaseConnector := aDatabaseConnector; fim;
```

Essa classe, quando criada e transmitida a uma instância de IDatabaseConnector, armazenará essa instância para uso posterior. Mas se registrarmos tudo com o Container:

```
procedimento RegisterStuff(aContainer:TContainer); começar

  aContainer.RegisterType <TDatabaseConnector, IDatabaseConnector>;
  aContainer.RegisterType <TDatabaseConnectionManager>; aContainer.Build; fim;
```

a implementação de IDatabaseConnector será criada automaticamente pelo Container. Mas e se você não quiser que isso aconteça? Bem, você pode declarar a variável um pouco diferente:

```
  TDatabaseConnectionManagerLazy = classe
  privada
    FDatabaseConnector: Lazy<IDatabaseConnector>;

  construtor público Create(aDatabaseConnector: Lazy<IDatabaseConnector>);
    procedimento ConnectToDatabase; fim;

construtor TDatabaseConnectionManagerLazy.Create(aDatabaseConnector: Lazy<IDatabaseConnector>); começar

  herdado Criar;
  FDatabaseConnector := aDatabaseConnector;
fim;

procedimento TDatabaseConnectionManagerLazy.ConnectToDatabase;
começar FDatabaseConnector.Value.Connect(Self.ClassName); fim;
```

A única diferença aqui é a declaração de FDatabaseConnector, que é declarada como Lazy<IDatabaseConnector> em vez de apenas IDatabaseConnector. O mesmo vale para o parâmetro para o construtor. Ao declará-lo como Lazy, ele instrui o Container a criar a conexão sob demanda, em vez de imediatamente. Essa demanda ocorre quando você solicita a propriedade Value da variável inicializada lentamente.

Vamos exercitar tudo isso e ver como funciona.

Aqui está um procedimento chamado Main que é chamado pelo arquivo DPR :

```
procedimento Main(aContainer: TContainer);
Onde
  DatabaseConnectionManager: TDatabaseConnectionManager;
  DatabaseConnectionManagerLazy: TDatabaseConnectionManagerLazy;
começar
  aContainer.RegisterType <TDatabaseConnector, IDatabaseConnector>;
  aContainer.RegisterType <TDatabaseConnectionManager>; aContainer.RegisterType
  <TDatabaseConnectionManagerLazy>; aContainer.Build;


  WriteLn('Tudo está registrado');
  ReadLn;

  WriteLn('Prestando para criar DatabaseConnectionManager');
  DatabaseConnectionManager := aContainer.Resolve<TDatabaseConnectionManager>;

  WriteLn('Observe que a conexão é feita sem que nada seja feito de sua parte.'); WriteLn('A conexão ocorre como
  parte da mágica do Container.'); WriteLn('Tudo pronto'); WriteLn; ReadLn;



  WriteLn('Prestando para criar DatabaseConnectionManagerLazy');
  DatabaseConnectionManagerLazy := aContainer.Resolve<TDatabaseConnectionManagerLazy>;

  WriteLn('Observe que a conexão não é feita até que seja especificamente solicitada por você pressionando Return.'); ReadLn;
  WriteLn('Ok, agora você pediu a conexão, e ela será feita'); DatabaseConnectionManagerLazy.ConnectToDatabase;



  WriteLn;
  WriteLn('Tudo pronto');
  WriteLn;
fim;
```

Isso é um pouco complicado de demonstrar, então vamos fazer isso passo a passo.

- Primeiro, tudo é registrado: a classe TDatabaseConnector como implementação

a interface IDatabaseConnector e as duas classes que usam a interface, uma "regularmente" e outra "preguiçosamente".

- Feito isso, o aplicativo informa que tudo está registrado e chama ReadLn para que você veja que tudo está registrado e aguardando uso. • Então, quando você pressiona a tecla Enter, ele informa que está prestes a criar um TDatabaseC onectionManager e faz isso.
- Neste ponto, você deve ver a conexão sendo feita. Não há atraso - a conexão é feita da maneira certa, mesmo antes de você fazer qualquer coisa. Isso acontece porque o Container cria tudo automaticamente na chamada para Build.
  Observe que leva três segundos para que a conexão seja feita enquanto simulamos um demora na conexão.
- Em seguida, porém, é TDatabaseConnectionManagerLazy. Essa classe declara a conexão do banco de dados como Lazy e, portanto, não é criada até que você a solicite especificamente. Podemos resolver uma instância dele por meio do Container sem o Container criar e conectar. Ele só se conecta quando realmente chamamos ConnectToDatabase e primeiro solicitamos o IDatabaseConnector. Você pode ver isso em ação quando você pressiona Return e nada acontece até que você pressione Return uma segunda vez.
- O aplicativo relata tudo isso para você à medida que está acontecendo, portanto, se você ler a saída com atenção, poderá ver o que está acontecendo à medida que acontece. Lembre-se, você precisa aguardar os três segundos para que a conexão seja feita. Se desejar, você pode esperar muito tempo para garantir que a conexão não seja feita até que seja especificamente solicitada.
- Você pode dizer – ambas as vezes – quando a conexão é criada porque o o aplicativo grava "All Done" no console quando terminar.

Existem apenas duas diferenças básicas entre as duas classes do gerenciador de conexões. O primeiro – TDatabaseConnectionManager – não faz nada de especial – apenas armazena a referência a IDatabaseConnector para uso posterior. É a segunda classe – TDatabaseConnectionManagerLazy – que tem as diferenças. Primeiro, ele declara sua referência a IDatabaseConnector como Lazy, garantindo que ele seja criado apenas quando referenciado pela primeira vez no código. Segundo, ele faz essa referência real no método ConnectToDatabase . (Observe que o registro Lazy tem uma propriedade chamada Value que contém a referência real à instância criada lentamente.) É nesse ponto que a implementação é atribuída e a classe realmente instanciada.

Registrando Fábricas

Ok, está tudo ótimo até agora. Mas você provavelmente notou um fato inconveniente: todas as nossas dependências são interfaces registradas. E se nosso construtor precisar de strings ou inteiros ou outros tipos primitivos?

Bem, o Container tem um meio de lidar com isso. Ele permite que você registre uma “fábrica” para informar ao Container como criar a classe com parâmetros de construtor que não estão registrados.

Implementação de uma cafeteira

Que tal imaginarmos uma interface para uma cafeteira que gerencie qual café é feito e por quanto tempo o café é preparado. E claro, aquela cafeteira precisa saber fazer café, daí a seguinte interface:

```
  ICoffeeMaker = interface
  ['{73436E03-EF65-44F5-9606-F706156CBEB5}']
    procedimento MakeCoffee;
  fim;
```

Mas como dissemos, a implementação precisa de um tipo de café e um tempo de preparo, então temos isso:

```
tipo
  TCoffeeMaker = class(TInterfacedObject, ICoffeeMaker) private

    FCoffeeBrand: string;
    FBrewingMinutes: inteiro;
  construtor público Create(const
    aCoffeeBrand: string; const aBrewingMinutes: integer); procedimento MakeCoffee;

  fim;

construtor TCoffeeMaker.Create(const aCoffeeBrand: string; const aBrewingMinutes: integer); começar

  herdado Criar;
  FCoffeeBrand := aCoffeeBrand;
  FBrewingMinutes := aBrewingMinutes;
fim;

procedimento TCoffeeMaker.MakeCoffee;
```

```
begin
  WriteLn('Despeje água quente sobre o ', FCoffeeBrand, ' es.'); fim;      para que seja fermentado por ', FBrewingMinutes, ' minut\
```

Agora essa classe, como está, não precisa ser colocada no Container, mas podemos facilmente imaginar a noção de, digamos, uma classe TKitchen que tomaria TCoffeeMaker como uma dependência e, portanto, a classe precisaria ser registrada com o recipiente para que ele resolva corretamente. Mas, novamente, o construtor recebe uma string e um inteiro. O que fazer?

Bem, podemos registrar uma fábrica para dizer ao Container como criar a cafeteira.
Este tipo de fábrica assume a forma de um método anônimo:

```
tipo
  {$M+}
  TCoffeeMakerFactory = referência à função(const aCoffeeBrand: string; const aBrewingMinutes: integ\
er): ICoffeeMaker;
  {$M-}
```

É fácil pensar em uma função anônima como uma fábrica. É um “plano” para criar uma coisa específica, uma implementação para ICoffeeMaker – neste caso, TCoffeeMaker.
Observe também que a assinatura do blueprint corresponde à do construtor para nossa classe de implementação. Isso não é uma coincidência, como você pode imaginar. Para que o método anônimo seja uma fábrica no que diz respeito ao Container, ele deve ter o {$METHODINFO} ativado. Uma vez declarada a função anônima, podemos registrá-la como fábrica com o Container:

```
procedimento RegisterStuff (aContainer: TContainer); comece
aContainer.RegisterType <ICoffeeMaker, TCoffeeMaker> .AsDefault;

  aContainer.RegisterFactory <TCoffeeMakerFactory>;
  aContainer.Build;
fim;
```

Primeiro, fazemos o registro normal do TCoffeeMaker na interface do ICoffeeMaker . Em seguida, passamos a função anônima TCoffeeMakerFactory para o método RegisterFactory do Container. O contêiner é inteligente o suficiente para observar o tipo de resultado da função anônima na Fábrica e descobrir qual interface ele deve resolver. Agora podemos obter as informações que precisamos e resolver a fábrica para criar a classe correta:

```
Onde
  CoffeeName: string;
  BrewingMinutes: inteiro;
  CoffeeMakerFactory: TCoffeeMakerFactory;
  Cafeteira: ICafeteira;
begin
  Write('Que tipo de café você quer fazer? '); ReadLn(CoffeeName);


  Write('Quantos minutos? ');
  ReadLn(Minutos de Infusão);

  CoffeeMakerFactory := Container.Resolve<TCoffeeMakerFactory>(); CoffeeMaker :=
  CoffeeMakerFactory(CoffeeName, BrewingMinutes); CoffeeMaker.MakeCoffee;

fim;
```

Este código é um pouco interessante e, portanto, você deve observar o seguinte:

- A variável CoffeeMakerFactory é do tipo TCoffeeMakerFactory, ou seja, uma referência à função anônima que é a fábrica. Essa linha de código diz “Consiga-me a fábrica para uma cafeteira”. Se você tiver várias implementações, poderá registrá-las e recuperá-las pelo nome. • A chamada para Resolve toma como tipo parametrizado TCoffeeMakerFactory.

  Observe que ele não usa a interface ICoffeeMaker como você poderia esperar. Ele também usa os parênteses para informar ao compilador que está retornando a função anônima e não um procedimento de objeto.

- Uma vez que tenhamos uma referência à função anônima, podemos chamá-la com os parâmetros necessários. Ele retornará uma interface, ou seja, ICoffeeMaker.
  Agora, isso é um pouco de Container Magic. O Container basicamente dá uma olhada na referência da fábrica e encontra uma implementação registrada do resultado da função fábrica que possui um construtor que corresponde à assinatura do método da fábrica. Em seguida, ele cria essa classe com os parâmetros passados e retorna uma interface para a implementação instanciada.
- A partir daí, você pode chamar o método MakeCoffee da interface e obter os resultados esperados.

A propósito, se você está realmente se sentindo aventureiro, pode fazer tudo isso em uma linha de código:

```
Container.Resolve<TCoffeeMakerFactory>()(CoffeeName, BrewingMinutes).MakeCoffee;
```

Agora temos uma classe cadastrada no Container que pode ser resolvida com seus parâmetros construtores, seja qual for o seu tipo. Agradável.

Mas agora você está pensando (eu sei disso porque você é muito inteligente, e é claro que você está pensando isso) “E se eu tiver, digamos, duas implementações de ICoffeeMaker e elas tiverem listas de parâmetros de construtor diferentes?” Isso não é um problema. Você só precisa trabalhar um pouco mais. Primeiro, aqui está uma implementação diferente com uma lista de parâmetros de construtor diferente:

```
  TCupCoffeeMaker = class(TInterfacedObject, ICoffeeMaker) estrito privado

    FCupType: string;

  construtor público Create(const aCupType: string);
    procedimento MakeCoffee; fim;


construtor TCupCoffeeMaker.Create(const aCupType: string); comece
herdado Criar; FCupType := aCupType;


fim;

procedimento TCupCoffeeMaker.MakeCoffee;
begin WriteLn('Coloque o ' end;
                                + FCupType +       ' xícara na cafeteira e pressione o botão "Brew"');
```

Esta é uma cafeteira estilo “xícara” que só precisa do tipo de xícara no construtor, então aqui está como nossas declarações de fábrica se parecem agora:

```
tipo
  {$M+}
  TCoffeeMakerFactory = referência à função(const aCoffeeBrand: string; const aBrewingMinutes: integ\
er): ICoffeeMaker;
  TCupCoffeeMakerFactory = referência à função(const aCupType: string): ICoffeeMaker;
  {$M-}
```

Aqui está como registramos tudo:

```
procedimento RegisterStuff(aContainer: TContainer); begin
aContainer.RegisterType<ICoffeeMaker,
   TCoffeeMaker>('regular'); aContainer.RegisterType<ICoffeeMaker,
   TCupCoffeeMaker>('copo');

   aContainer.RegisterFactory<TCoffeeMakerFactory>.AsFactory('regular');
   aContainer.RegisterFactory<TCupCoffeeMakerFactory>.AsFactory('cup'); aContainer.Build;

fim;
```

e aqui está como chamamos tudo:

```
Write('Que tipo de copo você quer fazer?');
ReadLn(NomeCopa); CupCoffeeMakerFactory :=
aContainer.Resolve<TCupCoffeeMakerFactory>(); Cafeteira :=
XícaraCoffeeMakerFactory(CupName); CoffeeMaker.MakeCoffee;
```

Como de costume, aqui está uma lista de coisas a serem observadas:

- Primeiro, as chamadas RegisterType têm nomes, pois há mais de uma classe implementando o ICoffeeMaker.
- Em seguida, observe que a chamada para RegisterFactory possui uma chamada para AsFactory que recebe como parâmetro o nome da implementação dessa fábrica. Isso é necessário para que o Container saiba qual implementação associar a qual fábrica.
- Finalmente, o resto do código é o que você espera. Você obtém uma referência à fábrica (que obviamente é um método anônimo) e a chama, passando o tipo de copo que você solicitou anteriormente.

Registrando Parâmetros Primitivos

Na seção anterior vimos como você pode registrar uma função anônima como uma fábrica para criar objetos com parâmetros arbitrários em seus construtores. Nesta seção, veremos uma maneira mais direta de fazer isso usando atributos no próprio construtor. Com efeito, podemos registrar tipos primitivos por nome e depois resolvê-los usando atributos e o contêiner. Vamos dar uma olhada em como isso funciona.

Aqui está uma classe de pessoa simples:

```
tipo
  TPerson = classe
  privado
    FOcupação: string;
    FNome: cadeia;
    FAge: inteiro;
    função GetName: string; função
    GetAge: inteiro; função
    GetOcupação: string; construtor público
  Create([Inject('name')]aName: string;
    [Inject('age')]aAge: integer; [Inject('occupatio\
n')]aOcupação: string); nome da
    propriedade: string lida GetName; propriedade
    Idade: inteiro lido GetAge; propriedade
    Ocupação: string lida GetOccupation;
  fim;
```

Não é nada notável, exceto que cada um dos parâmetros do construtor é marcado com um atributo [inject] . Eles têm nomes anexados como um parâmetro que os identificará ao contêiner. Esses atributos apontarão para registros de seus tipos primitivos no contêiner. Esses registros ficam assim:

```
procedimento Main(aContainer: TContainer);
Onde
  TempName: TPerson;
começar
  Aleatória;
  aContainer.RegisterType<TPerson>;
  aContainer.RegisterType<string>('name').DelegateTo(
                                        função: string
                                        começar
                                          Resultado := GetLocalUsername;
                                        fim
                                        ) .Como padrão;
  aContainer.RegisterType <string> ('ocupação').
                                        função: string
                                        começar
                                          Resultado := 'encanador'; // Isso pode ser recuperado de qualquer lugar, de\
 curso.
                                        fim
                                        );

  aContainer.RegisterType<inteiro>('idade').DelegateTo(
                                        função: inteiro começar

                                          Resultado := Aleatório(100);
                                        fim
                                        );
  aContainer.Build;
```

```
  TempName := aContainer.Resolve<TPerson>;
  WriteLn(TempName.Name, ' é ', TempName.Age, '          anos', ' e é um ', TempName.Occupation);
  ReadLn;
fim;
```

Não deve haver nada estranho aqui - você provavelmente pode descobrir o que está acontecendo. Os tipos que estão sendo registrados são primitivos – duas strings e um inteiro – e são registrados usando os mesmos valores de string que vimos nos atributos. Em seguida, chamamos DelegateTo para definir uma função anônima que informa ao contêiner como obter as strings e o inteiro. (Observe que usamos a chamada GetLocalUserName do capítulo anterior para obter o valor do nome.)

A diferença aqui de todo o código até este ponto é que estamos registrando tipos primitivos. Anteriormente, registramos classes, mas aqui registramos tipos primitivos arbitrários. Você pode registrar qualquer tipo que desejar, desde que forneça um nome para o registro e um método anônimo para informar ao contêiner qual deve ser o valor da primitiva. Como mostrado, você rotula os parâmetros (eles podem ser parâmetros em um método, se desejar) e os valores são definidos automaticamente para nós pelo Container. Agora você tem outra maneira de tornar uma classe completamente resolvível dentro do contêiner.

Atributos

Começo esta seção assumindo que você sabe como os atributos funcionam no Delphi. (Se não, sinta-se à vontade para ler tudo sobre eles no meu livro, “Coding in Delphi”.) O framework Spring for Delphi fornece vários atributos que podem ser usados no lugar de chamadas de registro.

[Injetar] Atributo

A estrutura Spring4D define a classe de atributo InjectAttribute. Isso, é claro, resulta no atributo [Inject] . Esse atributo permite rotular qualquer número de elementos de linguagem em uma classe, incluindo construtores, métodos, propriedades e parâmetros. (Também permite que você marque campos com ele, mas como veremos na próxima seção, Field Injection é um antipadrão.)

Quando um membro é assim marcado, é o equivalente a adicionar os métodos Injectxxxx às chamadas de registro de classe/interface. Assim, a finalidade do atributo [Inject] é registrar um determinado membro de classe como sendo injetado no contêiner com base em seu tipo.

Considere o seguinte código:

```
tipo

  ICavalo = interface
  ['{BDBB0FE7-D369-4EC7-B2A0-FC012136B87E}']
    procedimento Passeio;
  fim;

  ICowboy = interface
  ['{337002BB-2219-46A2-BF28-5CFD8A6873AD}']
    procedimento SetHorse(aValue: IHorse);
    função GetHorse: IHorse;
    procedimento DoCowboyStuff;
    propriedade Cavalo: IHorse ler GetHorse escrever SetHorse; fim;


  THorse = class(TInterfacedObject, IHorse) public

    procedimento Passeio;
  fim;

  TCowboy = class(TInterfacedObject, ICowboy) privado

    FCavalo: ICavalo;
    procedimento SetHorse(aValue: IHorse);
    função GetHorse: IHorse;
  público
    procedimento DoCowboyStuff;
    propriedade Cavalo: IHorse ler GetHorse escrever SetHorse; fim;


procedimento BeACowboy (aContainer: TContainer);

implementação

procedimento BeACowboy (aContainer: TContainer);
Onde
  Cowboy: ICowboy;
começar
  Cowboy := aContainer.Resolve<ICowboy>;
  Cowboy.DoCowboyCoisas;
fim;

procedimento TCowboy.DoCowboyStuff;
começar
```

```
    WriteLn('Yippee Kay Yay!'); Passeio
    a cavalo;
fim;

função TCowboy.GetHorse: IHorse; começar

    Resultado := FHorse;
fim;

procedimento TCowboy.SetHorse(aValue: IHorse); begin
FHorse := aValue ; fim;



procedimento THorse.Ride;
começar
    WriteLn('Galope ao longo da pradaria!');
fim;
```

e o código de registro:

```
procedimento RegisterStuff(aContainer: TContainer); begin
aContainer.RegisterType<ICowboy, TCowboy>.InjectProperty('Horse');
    aContainer.RegisterType<THorse, IHorse>; aContainer.Build;


fim;
```

Não há nada neste código que não deva ser familiar. O “trabalho” é feito no código de registro com a chamada para InjectProperty. Isso informa ao Container que haverá uma propriedade chamada 'Horse' que será resolvida pela chamada de registro fornecida. Isso permite que você nunca chame Create em qualquer lugar da sua classe e, em vez disso, deixe o Container fazer esse trabalho para você.

Agora, você pode fazer quase exatamente a mesma coisa com o seguinte:

```
TCowboy = class(TInterfacedObject, ICowboy) privado

    FCavalo: ICavalo;
    procedimento SetHorse(aValue: IHorse);
    função GetHorse: IHorse;
público
    procedimento DoCowboyStuff;
    [Injetar]
    propriedade Cavalo: IHorse ler GetHorse escrever SetHorse; fim;
```

e, em seguida, simplifique o registro da seguinte forma:

```
procedimento RegisterStuff(aContainer: TContainer); começar

    aContainer.RegisterType <TCowboy, ICowboy>;
    aContainer.RegisterType <THorse, IHorse>;
    aContainer.Build;
fim;
```

Digo “quase” a mesma coisa, pois ao usar o atributo, você especifica precisamente qual propriedade deseja associar ao registro IHorse . Se você usar InjectProperty, o Container irá associar isso a qualquer propriedade em qualquer classe registrada no container que tenha uma propriedade chamada “Horse” do tipo IHorse. O atributo, por outro lado, limita a associação à propriedade marcada com o atributo. Esta é uma diferença sutil, mas importante.

Atributos vitalícios

Anteriormente, discutimos como o Container pode ser instruído a gerenciar o tempo de vida das instâncias que ele retorna e atribui a você. Como mostrei na época, isso foi feito com uma ligação durante o processo de registro:

```
Container.RegisterType<IWeapon, TSword>.AsSingleton;
```

Esse código informa ao contêiner para sempre retornar a mesma instância de TSword sempre que recuperar uma implementação de TSword. No entanto, você pode fazer a mesma coisa com o atributo [Singleton] anexando-o à classe TSword :

```
[Singleton]
TSword = class(TInterfacedObject, IWeapon)
  procedimento Empunhar;
fim;
```

Isso informará ao Container que a implementação de IWeapon deve ser um singleton – ou seja, quando o Container fornece uma instância de IWeapon, ele deve sempre retornar exatamente a mesma instância.

Conclusão

O contêiner Spring4D Dependency Injection é bastante poderoso e capaz. Ele pode fazer mais do que simplesmente combinar interfaces e implementações, como vimos.

Injeção de dependência Antipadrões

Até agora, tentei mostrar a você bons padrões e práticas para fazer Injeção de Dependência e usar o Contêiner de Injeção de Dependência. Há definitivamente maneiras corretas que as coisas devem ser feitas. E, inversamente, definitivamente existem maneiras pelas quais as coisas não devem ser feitas. Este capítulo cobre algumas dessas maneiras erradas – comumente chamadas de antipadrões – que você pode ficar tentado a usar, mas não deveria. Esses antipadrões são aqueles que foram testados, mas mostraram falta de produção de código limpo, bom e bem escrito.

Localizador de serviço

Facilmente, o antipadrão mais conhecido, controverso e mais abusado é o antipadrão ServiceLocator . Eu o chamo de conhecido e controverso porque por muito tempo ServiceLocator foi realmente aceito como um padrão útil para fazer Injeção de Dependência. Eu o chamo de abuso porque as pessoas ainda o usam e acreditam que é um padrão útil. Argumentarei aqui que ServiceLocator é de fato um antipadrão e que você não deve usá-lo em seus aplicativos.

ServiceLocator é tentador. É super fácil substituir suas chamadas para criar com uma chamada para ServiceLocator.Resolve<IMyInterface>. É muito natural usar o ServiceLocator sempre que você precisar de uma implementação de uma interface.

Mas como vimos, ServiceLocator não é necessário. O Container pode fazer 99% do trabalho para o qual você está tentado a usar o ServiceLocator . Como o Container pode resolver automaticamente qualquer dependência registrada que outra classe registrada possa ter, você pode usar o ServiceLocator apenas uma vez na raiz do seu aplicativo. O Container é totalmente capaz de conectar todo o seu gráfico de objeto antes que o aplicativo seja iniciado. Você precisa dessa chamada de resolução na raiz do seu aplicativo, mas se torna um pequeno preço a pagar pelo enorme benefício que o Container traz.

Existem várias outras razões pelas quais o ServiceLocator deve ser evitado.

- O ServiceLocator é um singleton e singletons são variáveis globais. Variáveis globais devem ser evitadas a todo custo.
- Se você usar o ServiceLocator como um substituto para suas chamadas Create , estará fazendo com que o Container não seja nada mais do que um grande bucket de variáveis globais. Se você pode pegar qualquer instância de classe de qualquer lugar em seu aplicativo, você está usando o Container como tal e, como eu disse, as variáveis globais devem ser evitadas.
- Usar o ServiceLocator em vez de passar dependências oculta essas dependências. Um dos principais propósitos da Injeção de Construtor é declarar abertamente as dependências de uma classe. Se você usar o ServiceLocator para criar dependências em vez de passá-las para uma classe, você subverte os benefícios da injeção de construtor.
- Ao usar o ServiceLocator, você cria um erro de tempo de execução em vez de um erro de tempo de compilação, e os erros de tempo de compilação são muito preferíveis. A injeção de construtor usada incorretamente resultará em um erro do compilador. Usar o ServiceLocator incorretamente resultará em um erro em tempo de execução. Você deve preferir o primeiro. • Ao usar o ServiceLocator , você não aproveita a capacidade de composição do Container. O Container pode ser configurado para compor corretamente seu gráfico de objetos. Ele pode até ser configurado para fornecer diferentes composições por diferentes motivos. Mas se você simplesmente pegar implementações aleatoriamente do Container com ServiceLocator, você perde todos os benefícios de usar o Container para compor seus objetos corretamente.

Em suma , o ServiceLocator deve ser visto como um antipadrão e evitado.
Em vez disso, confie no contêiner para resolver todos os seus objetos e faça uma única chamada Resolve na raiz composta do seu aplicativo. É muito tentador usá-la porque, à primeira vista, parece uma técnica muito útil, mas, como vimos, essa tentação não passa de ouro de tolo e na verdade leva a problemas.

Injeção de campo

O que é injeção de campo

Injeção de campo é um tipo de injeção de dependência em que você injeta uma dependência definindo um valor de campo em uma classe. Aqui está um exemplo simples e explicativo:

```
unidade uFieldInjection;

interface

usa
        Primavera.Contêiner.Comum
    ;

tipo
  Ibrake = interface
  ['{74DBE39C-F52F-42C4-B7CB-8009F7EDF1E1}']
    Procedimento StopVehicle ;
  fim;

  IEngine = interface
  ['{0BC34CC8-DE81-4073-9CA4-A160CFB9A64A}']
    procedimento PropelVeículo;
  fim;

  ICar = interface
  ['{B2C1C9FB-E388-4F0B-9197-2BCAB5A2A396}']
    procedimento Drive;
  fim;

  TBrakes = class(TInterfacedObject, IBrak)
    Procedimento StopVehicle ;
  fim;

  TEngine = classe (TInterfacedObject, IEngine)
    procedimento PropelVeículo;
  fim;

  TCar = class(TInterfacedObject, ICar) privado

    Fbrakes: Ibrake;
     [Injetar]
    FE Motor: IE Motor;
  procedimento público
    Drive;
  fim;

procedimento MakeCarGo (aContainer: TContainer);
```

```
implementação

usa
      Spring.Container
    ;

procedimento MakeCarGo (aContainer: TContainer);
Onde
  Carro: ICar;
begin Car :=
  aContainer.Resolve<ICar>; Car.Drive;

fim;

procedimento TBrakes.StopVehicle;
begin WriteLn('Pise no pedal do freio e
  faça o carro parar'); fim;


procedimento TEngine.PropelVehicle;
begin WriteLn('Queime gasolina e faça o
  carro andar');
fim;

procedimento TCar.Drive;
começar
  FEEngine.PropelVehicle;
  FBrakes.StopVehicle;
fim;
```

e o código de registro:

```
procedimento RegisterStuff(aContainer: TContainer); começar

  aContainer.RegisterType<TBrakes, Ibrake>;
  aContainer.RegisterType<TEngine, IEngine>;
  aContainer.RegisterType<ICar, TCar>.InjectField('FBrakes'); fim;
```

Aqui estão algumas coisas a serem observadas sobre o código acima:

- Utiliza Field Injection para dois campos, FBrakes e FEngine. Ao registrar o TCar como implementando a interface ICar , ele anexa uma chamada ao InjectField que registra FBrakes como um campo que será injetado na criação. Para registrar o campo FEngine , utiliza-se o atributo [Inject] . Ambas as técnicas são exatamente idênticas e fazem a mesma coisa.

- Como o TCar está registrado no container e existem classes cadastradas para os tipos dos campos, tudo fica “conectado” automaticamente e uma chamada para MakeCarGo se comporta como esperado.

Por que a injeção de campo é uma má ideia?

A injeção de campo pode ser muito atraente. Pode parecer “mais limpo” do que a injeção de construtor porque requer menos código. Pode parecer fácil porque basta um único atributo. Mas não se engane – Field Injection é um antipadrão. Aqui está o porquê:

- Ele quebra o encapsulamento permitindo acesso a um membro privado. Quebrar o encapsulamento é um grande não-não no mundo da programação orientada a objetos. Assim é brincar com membros privados. Os campos são privados por um motivo e devem ser gerenciados internamente pela classe, não por uma entidade externa. Os valores dos campos são normalmente definidos por um construtor ou um método setter. Você não deve definir um valor de campo com base em algum valor de fora da classe. Em nosso exemplo acima, os dois campos são privados, mas obtêm seus valores do Container, que tem permissão para acessá-los. Não é bom.
- Além de violar o encapsulamento, Field Injection também oculta as dependências de uma classe. Se você conhece ou visualiza apenas a interface pública de uma classe, não verá todas as dependências que uma classe possui, pois os campos estão ocultos em seções privadas. Você pode não saber que a dependência está mesmo lá. Com construtor e injeção de propriedade, as dependências estão lá para serem vistas e conhecidas. Com Field Injection, o usuário da classe pode nunca ver a dependência e pode até tentar usar a classe antes de ter certeza de que a dependência está presente para uso. Isso é uma violação de acesso esperando para acontecer. Uma olhada na interface pública do TCar não revela dependências. O construtor não recebe parâmetros e não há propriedades que possam ser definidas. Se tudo o que você tivesse fosse a interface pública do TCar, você permaneceria felizmente ignorante do que o TCar realmente precisa e faz. • Além disso, como você testa uma classe que tem uma dependência privada? Como você zomba dessa dependência? Você realmente não pode. Quando você usa injeção de campo, você está basicamente escrevendo uma classe não testável, e isso parece uma ideia tola. • Field Injection permite dependências circulares. Se você passar um valor de campo de alguma entidade externa, não poderá ter certeza de que a referência não está criando

uma dependência circular. A Injeção de Construtor evita isso ao nunca permitir que as referências internas “escapem” para criar uma dependência circular.

- Finalmente, Field Injection, especialmente se você usá-lo para muitos campos em uma classe, pode esconder a complexidade de uma classe e fazer você pensar que a classe é mais simples do que realmente é. Afinal, o que é outro campo? No entanto, se você injetar suas dependências por meio do construtor, elas logo podem se acumular e fica óbvio que você está violando o Princípio da Responsabilidade Única. Seria muito fácil adicionar IRadio e ITransmission e ISeats e uma tonelada de outros campos ao TCar, tudo sem ter que fazer muito além de registrar as classes. O construtor permaneceria simples e tudo pareceria bem. Mas, infelizmente, não é. Se você está projetando um carro, você vai querer quebrar uma estrutura de classe bastante séria para manter a declaração TCar de nível muito alto, com dependências caindo em cascata em dependências mais baixas e assim por diante. Se o seu carro tivesse uma centena de parâmetros no construtor, isso seria complicado. Não é uma boa ideia, mas pode ser menos doloroso ter cem campos. Mas é claro que cada um desses campos está oculto e torna a classe mais difícil de testar, então você deve evitar esses campos injetados.

Bottom Line: Use Constructor Injection e Property Injection, e não use Field Injection.

Sobreinjeção do Construtor

Eu insinuei isso um pouco acima ao discutir a injeção de campo. Construtor sobre injeção é a tendência de continuar passando referências para trás em direção à raiz composta no construtor de cada classe subsequente, fazendo com que esses construtores inchem e assumam muitos parâmetros. Imagine ClassA que depende de ClassB, que por sua vez depende de ClassC, depois ClassD e assim por diante. Você pode acabar com um construtor que se parece com isso:

```
construtor TClassG.Create (aClassA: TClassA; aClassB: TClassB; aClassC: TClassC; aClassD: TClassD; aCl \ assE: TClassE;
aClassF: TClassF);
```

Isso claramente não é desejável, embora se possa entender como isso pode acontecer se for assíduo no uso de Injeção de Construtor. Se você está fazendo isso, é hora de

dê um passo para trás e perceba que sua hierarquia de classe tem problemas, o principal dos quais é provável que sua classe não esteja seguindo o Princípio da Responsabilidade Única. Uma classe que recebe tantas dependências, mesmo que esteja apenas passando essas dependências para outras classes, provavelmente está tentando fazer demais. Procure dividir suas aulas em turmas menores e mais focadas que fazem uma coisa e apenas uma coisa. Isso deve evitar a superinjeção de construção.

A sobreinjeção de construtor também pode ocorrer de maneira mais sutil. Considere o seguinte código:

```
unidade uOverInjeção;

interface

tipo
  IBankingService = interface
  ['{E367001A-94D1-4694-A0B9-FB0B3FD822ED}']
    procedimento DoBankingStuff; fim;


  IMailingService = interface
  ['{ED17BB98-CC8C-4DBF-9466-C20F6BBC4AE2}']
    procedimento MailPayrollInfo;
  fim;

  TEmployee = class(TObject) private

    FLastName: string;
    FFirstName: string;
    FWantsMail: Boolean;
  propriedade pública
    WantsMail: Boolean lê FWantsMail escreve FWantsMail; propriedade FirstName:
    string lida FFirstName escreve FFirstName; propriedade LastName: string ler
    FLastName escrever FLastName; fim;


  TMailingService = class(TInterfacedObject, IMailingService) procedimento
    MailPayrollInfo;
  fim;

  TBankService = class(TInterfacedObject, IBankingService)
    procedimento DoBankingStuff;
  fim;

  TPayrollSystem = classe
  privada
    FBankService: IBankService;
    FMailingService: IMailingService;
  público
    construtor Create (aBankingService: IBankingService; aMailingService: IMailingService);
```

```
        procedimento DoPayroll (aEmployee: TEmployee); fim;


implementação

construtor TPayrollSystem.Create (aBankingService: IBankingService; aMailingService: IMailingService);
começar
    FBankService : = aBankingService;
    FMailingService := aMailingService; fim;


procedimento TPayrollSystem.DoPayroll(aEmployee: TEmployee); begin
WriteLn('Fazendo folha de pagamento'); FBankingService.DoBankingStuff; se
    aEmployee.WantsMail então comece



        FMailingService.MailPayrollInfo;
    do que;
do que;

procedimento TBankingService.DoBankingStuff; begin
WriteLn('Fazendo coisas bancárias'); fim;



procedimento TMailingService.MailPayrollInfo; begin
WriteLn('Enviar informações da folha de pagamento'); fim;


fim.
```

Isso parece ótimo. Mas observe que, embora haja duas dependências passadas, uma dessas dependências – IMailingService – nem sempre é usada. No entanto, IMailingService é sempre **necessário.** A classe TMailingServices está sendo “gananciosa” e pedindo mais do que sempre precisa. O construtor de uma classe deve solicitar apenas as dependências que realmente precisa, e não as dependências que apenas precisa. Este tipo de Sobreinjeção de Construtor deve ser considerado um Code Smell. Em vez disso, você pode considerar usar algum outro método, como uma Fábrica, para obter uma instância do serviço de correio.

Devemos também reconhecer isso como uma violação (simples, mas ilustrativa) do Princípio da Responsabilidade Única. Ou seja, a classe TPayrollSystem está fazendo muito. Está fazendo a folha de pagamento e enviando coisas. Em vez disso, provavelmente deve se preocupar apenas com a folha de pagamento e depois ter uma dependência que se preocupe com o “envio”. Em outras palavras, provavelmente deve haver uma noção de ISendPayrollInfo e, em seguida, vários

implementações baseadas no que o funcionário deseja – correio tradicional, e-mail, entrega em mãos, o que for. O TPayrollSystem não deve se preocupar com a forma como o envio acontece.

Assim, existem dois tipos de sobre-injeção de construtor antipadrão. Primeiro, há a passagem de muitos parâmetros de dependência no construtor (começo a ficar nervoso quando o número chega a três…). Isso deve causar uma reconsideração do design da classe à luz do Princípio da Responsabilidade Única. A segunda é a passagem de dependências que não são usadas diretamente pela classe, ou dependências que nem sempre são necessárias. Isso deve ser considerado um Code Smell e refatorado também.

Componentes VCL no contêiner

Discuti anteriormente a noção de Injetáveis e Criativos. Algumas classes – Creatables – devem ser criadas manualmente e não colocadas no Container. WI falou sobre como certas classes como classes RTL – TStringList, TList, TStream, etc. – eram Creatables e não deveriam ser registradas no Container. Bem, outro grupo inteiro de classes que nunca deve ser registrado no Container é qualquer controle VCL que descenda de TComponent. Isso é particularmente verdadeiro para TForm e TDataModule.
Há várias razões para isso:

- Os controles VCL possuem seu próprio sistema de gerenciamento de vida útil e colocá-los no Container causa confusão, tanto por parte do desenvolvedor quanto do Container, quanto a quem possui os componentes e quando devem ser destruídos.

- Muitos controles VCL são visuais e, como tal, precisam ser gerenciados por seus contêineres visuais (como formulários, painéis, caixas de grupo e similares). Colocar esses componentes no Dependency Injectino Container estraga esses relacionamentos de propriedade. O Dependency Injection Container e os controles visuais não combinam bem e não devem ser usados juntos.
- É difícil. Você tem que passar por muitos aros até mesmo para fazê-lo funcionar. Mantenha as coisas simples e reserve o contêiner para classes de negócios e outros objetos que você mesmo escreve para permitir que seu aplicativo funcione.

Vários construtores

Uma classe envolvida com Injeção de Dependência deve ter apenas um construtor definitivo. Esse construtor deve declarar todas as dependências que a classe requer.
Lembre-se, os parâmetros de um construtor devem ser a lista definitiva de dependências sem as quais uma classe não pode viver. Se uma classe tem mais de um construtor, então ela está declarando mais de um conjunto de dependências obrigatórias, e isso não faz nenhuma diferença.
senso.

Vários construtores também tornarão difícil, se não impossível, para o Container resolver como criar uma classe corretamente. Mesmo que todos os parâmetros sejam resolvíveis para vários construtores, a questão de qual construtor o Container deve selecionar não é clara. Um DI Container pode ter regras específicas para selecionar um de vários construtores, mas essa decisão pode não ser clara para o desenvolvedor. E as alterações na classe podem resultar em um caminho de código diferente, sem o conhecimento do desenvolvedor. Não é bom.

Misturando o contêiner com seu código

> Primeiro, eu fui culpado por este, e só recentemente o reconheci como um antipadrão quando Stefan Glienke o sugeriu como tal. Você provavelmente encontrará um código de demonstração meu na internet que faz o que estou prestes a dizer que é a maneira errada de fazer as coisas. Esforcei-me para limpar meu código de demonstração – o código para este livro deve estar correto – e espero ter feito isso completamente (tenho muito código de demonstração por aí…). Infelizmente - viver e aprender.

Misturar o registro de suas classes e interfaces com seu código colocando os registros na seção de inicialização de sua unidade é um anti-padrão. Ao fazer isso, você acopla sua classe ao Container e, conforme discutimos, você deve evitar acoplamentos desnecessários.

Por exemplo: O protocolo de teste de unidade afirma que você deve testar suas classes isoladamente. Se você fizer seu cadastro na seção de inicialização , não poderá testar seu código sem envolver o Container.

Em vez disso, crie uma unidade separada e coloque seu código de registro na seção de inicialização dessa unidade e, em seguida, use essa unidade apenas no arquivo DPR , incluindo-a no projeto. Isso permitirá que você registre tudo o que precisa ser registrado, mantendo o Container desacoplado do seu código. O código de exemplo para este livro ilustra essa técnica.

Conclusão

Assim como existem maneiras certas de fazer as coisas com a injeção de dependência, também existem maneiras erradas de fazer as coisas. Este capítulo discutiu alguns antipadrões que são tentadores, mas não são uma boa ideia ao fazer a injeção de dependência. Fique longe deles e você evitará muitos problemas.

Um Simples, Útil e Completo Exemplo

Introdução

Ok, tivemos muita teoria e muitos exemplos básicos, então agora é hora de juntar tudo em uma demonstração real e funcional que realmente faz algo útil.

O exemplo a seguir é um aplicativo simples que permite visualizar arquivos. É expansível, pois você pode escrever mais visualizadores de tipo de arquivo e adicioná-los ao aplicativo em tempo de design. Por padrão, ele fornece um visualizador de arquivos de texto e um visualizador para os tipos de gráficos mais populares.

E, claro, usa Injeção de Dependência como seu padrão de design básico. O uso principal é via Injeção de Construtor, mas também incluí algumas Injeções de Propriedade.

O código para esta demonstração pode ser encontrado em: http://bit.ly/NickFileViewer

Interfaces

Naturalmente, o aplicativo depende de interfaces e não de implementações dessas interfaces (já fiz esse ponto o suficiente?). Então aqui estão essas interfaces, em sua própria unidade, é claro:

```
unidade uFileDisplayerInterfaces;

interface

usa
  Vcl.ExtCtrls
  ;

tipo
  IDisplayOnPanel = interface
  ['{C334B1AE-F562-4EA6-B98D-BB52F1CBE7A7}']
    procedimento DisplayOnPanel(const aPanel: TPanel); fim;


  IDisplayFile = interface
  ['{3437F0E6-2974-4C1A-BA07-2598A2774855}']
    procedimento DisplayFile(const aFilename: string; const aPanel: TPanel);
  fim;

  IFileExtensionGetter = interface
  ['{9E28B18F-3CDF-4F6E-B629-D3E02E0E0E6C}']
    função GetExtension(const aFilename: string): string; fim;


  IFilenameGetter = interface
  ['{48E1FFD8-73EA-43DF-B722-4A86206BDFCE}'] função
    GetFilename: string; fim;


  IFileDisplayerRegistry = interface
  ['{7211F4E0-0E7E-4216-912D-069E21660DE1}']
    procedimento AddDisplayer(aExt: string; aDisplayer: IDisplayFile); função
    GetDisplayer(aExt: string): IDisplayFile; função GetExtensions: TArray<string>;

  fim;

implementação

fim.
```

O aplicativo tem três dependências principais:

1. **IDisplayFile** – Esta é a interface responsável por realmente exibir o arquivo. Ele requer o nome do arquivo e um TPanel no qual exibir o que quer que seja exibido.

2. **IFileExtensionGetter** – Esta interface é projetada para obter a extensão de um determinado arquivo que é necessária para saber como lidar com o arquivo. Nossa implementação simplesmente chama uma chamada de função de System.IOUtils, mas pode ser determinada da maneira que você quiser .

3. **IFilenameGetter** – O objetivo desta interface é fornecer um meio para obter um nome de arquivo a ser exibido. Em nossa implementação, o usuário é avisado por meio de uma caixa de diálogo aberta, mas um nome de arquivo pode ser obtido por qualquer meio, desde que a interface seja implementada corretamente.

O Exibidor de Arquivos

Vamos dar uma olhada na classe principal que faz todo o trabalho e recebe as dependências. É uma classe autônoma que usa Injeção de Construtor para gerenciar suas dependências:

```
unidade uFileDisplayer;

interface

usa
      Vcl.ExtCtrls
    , uFileDisplayerInterfaces
    ;

tipo

  TFileDisplayer = class(TInterfacedObject, IDisplayOnPanel) private

    FFilenameGetter: IFilenameGetter;
    FFileExtensionGetter: IFileExtensionGetter;
    FFileDisplayer: IDisplayFile;
    procedimento ClearPanelChildren(const aPanel: TPanel);

  construtor público Create(aFilenameGetter: IFilenameGetter; aFileExtensionGetter: IFileExtensionGetter); procedimento
    DisplayOnPanel(const aPanel: TPanel);
  fim;

implementação

usa
      System.Classes,
      uFileDisplayerRegistry
    ;

procedimento TFileDisplayer.ClearPanelChildren(const aPanel: TPanel);
Onde
  Componente: TComponent; i:
  inteiro; begin for i := 0 to
aPanel.ControlCount - 1 do

  begin
    Component := aPanel.Controls[i] as TComponent;
```

```
        Componente.Free;
    do que;
do que;

construtor TFileDisplayer.Create(aFilenameGetter: IFilenameGetter; aFileExtensionGetter: IFileExtensi\ onGetter); começar

    herdado Criar;
    FFilenameGetter : = aFilenameGetter;
    FFileExtensionGetter := aFileExtensionGetter; fim;

procedimento TFileDisplayer.DisplayOnPanel(const aPanel: TPanel);
Onde
    LEXT: string;
    LFfilename: string;
begin
    ClearPanelChildren(aPanel);
    LFilename := FFilenameGetter.GetFilename; LExt :=
    FFileExtensionGetter.GetExtension(LFilename); FFileDisplayer :=
    FileDisplayerRegistry.GetDisplayer(LExt); FFileDisplayer.DisplayFile(LFilename,
    aPanel); fim;

fim.
```

Como de costume, vamos dar uma olhada em algumas coisas a serem observadas sobre o código acima:

- A classe TFileDisplayer implementa a interface IDisplayOnPanel . Acho essa aula muito bonita. Ele solicita suas dependências, usa essas dependências para fazer uma única coisa e faz tudo sem realmente implementar nada. Encantador.
- O construtor injeta duas dependências como interfaces – um IFilenameGetter e um IFileExtensionGetter. Observe que o construtor é muito simples e apenas guarda essas referências de interface para uso posterior.
- O método DisplayOnPanel é aquele que implementa IDisplayOnPanel, e é o método onde as dependências são utilizadas. Ele reúne as informações necessárias, obtém a extensão, usa a extensão para obter o Displayer correto e, em seguida, chama DisplayFile nesse displayer.
- Ele usa uma classe chamada FileDisplayerRegistry para gerenciar os visualizadores de arquivos. Bem falar sobre essa classe em um minuto.
- E é isso. Tudo o que precisamos agora é implementar displayers, e então podemos conectar esta classe thin com a interface de usuário VCL, e temos um funcionamento

inscrição. Isso é bom, porque podemos acoplar vagamente à interface do usuário e fazer a maior parte do trabalho codificando contra abstrações. Observe que a classe acima faz exatamente isso. Cada linha de código na classe está codificando em uma interface. Ele não sabe ou se preocupa com como essas interfaces são implementadas.

Aqui está a implementação que estamos usando para IFilenameGetter:

```
unidade uFilenameGetter;

interface

usa uFileDisplayerInterfaces;

tipo
  TFilenameGetter = class(TInterfacedObject, IFilenameGetter) private

    função GetFilename: string;
  fim;

implementação

usa
      Vcl.Dialogs ,
    Spring.Container
    ;

função TFilenameGetter.GetFilename: string; começar
PromptForFileName(Result); fim;



fim.
```

e para IFileExtensionGetter:

```
unidade uFilenameExtensionGetter;

interface

usa uFileDisplayerInterfaces;

tipo
  TFileExtensionGetter = class(TInterfacedObject, IFileExtensionGetter) função privada
  GetExtension(const aFilename: string): string; fim;


implementação

usa
      System.IOUtils ,
    Spring.Container
    ;

function TFileExtensionGetter.GetExtension(const aFilename: string): string; begin Resultado :=
TPath.GetExtension(aFilename); fim;


fim.
```

Ambos são bastante simples e autoexplicativos. Como eles estão implementando interfaces, você pode implementá-los facilmente de maneira diferente, se desejar, e desde que atendam ao contrato da interface, eles funcionarão bem dentro do aplicativo.
É por isso que você codifica para uma interface e não para uma implementação. Se você deseja obter seu nome de arquivo de um banco de dados, você pode fazer isso sem alterar nada além da implementação da interface específica.

Construindo Expositores

Ok, então a próxima interface que temos que implementar é IDisplayFile. Esta interface é aquela onde o trabalho real é feito. Para nosso aplicativo, implementamos dois – um que pode exibir arquivos de texto e outro que pode exibir arquivos gráficos.

Aqui está um que pode exibir texto – TTextFileDisplayer:

```
unidade uTextFileDisplayer;

interface

usa
      uFileDisplayerInterfaces
    , Vcl.ExtCtrls
    ;

tipo
  TTextFileDisplayer = class(TInterfacedObject, IDisplayFile)
    procedimento DisplayFile(const aFilename: string; const aPanel: TPanel); fim;


implementação

usa
      Vcl.StdCtrls
    , Vcl.Controls
    ;

procedimento TTextFileDisplayer.DisplayFile(const aFilename: string; const aPanel: TPanel);
Onde
  Memorando:
TMemo; begin
  Memo := TMemo.Create(aPanel);
  Memo.Parent := aPanel;
  Memo.Align := alCliente;
  Memo.ReadOnly := Verdadeiro;
  Memo.Lines.LoadFromFile(aFilename);
fim;

fim.
```

Essa classe é simples (como todas as classes deveriam ser, certo?). Ele implementa o único método de IDisplayFile adicionando um TMemo ao painel que é passado e abrindo o arquivo de texto nele. Não poderia ser mais simples. Outro belo exemplo de uma classe fazendo uma coisa e fazendo bem.

Aqui está a implementação muito semelhante para exibir gráficos:

```
unidade uPictureDisplayer;

interface

usa
      uFileDisplayerInterfaces
    , Vcl.ExtCtrls
    ;

tipo
  TPictureDisplayer = class(TInterfacedObject, IDisplayFile)
    procedimento DisplayFile(const aFilename: string; const aPanel: TPanel); fim;


implementação

usa
      System.Classes
    , Vcl.Gráficos
    , Vcl.Controls
    , Vcl.Imaging.JPEG ,
    Vcl.Imaging.PngImage ,
    Vcl.Imaging.GIFImg
    ;

procedimento TPictureDisplayer.DisplayFile(const aFilename: string; const aPanel: TPanel);
Onde
  Imagem: TImage;
begin Image :=
  TImage.Create(aPanel); Image.Parent :=
  aPanel; Image.Align := alCliente;


  Image.Picture.Bitmap := TBitmap.Create;
  Image.Picture.LoadFromFile(aFilename);

fim;

fim.
```

Não é necessária muita explicação para isso – a classe simplesmente aproveita os recursos do TImage para permitir a abertura de qualquer arquivo Bitmap, JPEG, GIF ou PNG. Bonito.

Amarrando tudo junto

Usamos a Injeção de Construtor para criar uma classe principal que gerencia as coisas para nós. Implementamos classes que nos permitem visualizar arquivos. Então agora é hora de amarrar tudo junto.

Registrando extensões

Primeiro, precisamos de um local para conter todos os visualizadores de arquivos. Como temos alguns visualizadores de arquivos e extensões de arquivo que nos informam quais desses visualizadores usar para essas extensões, um IDictionary atende muito bem aos nossos propósitos. Podemos envolvê-lo em um registro, para que armazenemos e recuperemos os pares. Como queremos codificar em uma interface, criamos o IFileDisplayerRegistry (visto acima) que faz o trabalho que queremos. Aqui está a implementação:

```
unidade uFileDisplayerRegistry;

interface

usa
      uFileDisplayerInterfaces ,
    Spring.Collections
    ;

tipo
  TFileDisplayerRegistry = class(TInterfacedObject, IFileDisplayerRegistry) private

    FDictionary: IDictionary<string, IDisplayFile>;
    FDefaultDisplayer: IDisplayFile;
  público
    construtor Criar;
    procedimento AddDisplayer(aExt: string; aDisplayer: IDisplayFile); função
    GetDisplayer(aExt: string): IDisplayFile; função GetExtensions: TArray<string>; //
    Injeção de propriedade: Temos um displayer padrão, mas você pode fornecer o seu
    próprio // se quiser. propriedade DefaultDisplayer: IDisplayFile ler FDefaultDisplayer escrever FDefaultDisplayer; fim;



  função FileDisplayerRegistry: IFileDisplayerRegistry;

implementação

usa
      Spring.Container
    , uDefaultDisplayer
    ;

 Onde
   FDR: IFileDisplayerRegistry;

função FileDisplayerRegistry: IFileDisplayerRegistry; começar se FDR = nil
então


  começar FDR := TFileDisplayerRegistry.Create;
```

```
    fim;
    Resultado := FDR;
fim;

procedimento TFileDisplayerRegistry.AddDisplayer(aExt: string; aDisplayer: IDisplayFile); começar
FDictionary.Add(aExt, aDisplayer); fim;



construtor TFileDisplayerRegistry.Create; begin
FDictionary := TCollections.CreateDictionary<string,
    IDisplayFile>; FDefaultDisplayer := TDefaultFileDisplayer.Create;

fim;

função TFileDisplayerRegistry.GetDisplayer(aExt: string): IDisplayFile; começar

    // nunca retorna nil. Se um Displayer não for encontrado, retorne o padrão.
    FDictionary.TryGetValue(aExt, Resultado);
    se resultado = nil então
    começar
        Resultado := FDefaultDisplayer;
    do que;
do que;

função TFileDisplayerRegistry.GetExtensions: TArray <string>;
Onde
    Extensão: cadeia; e:
    inteiro; begin
SetLength(Result,
    FDictionary.Count); e := 0;

    para Extensão em FDictionary.Keys comece
    Result[i] := FDictionary.Keys.ElementAt(i); Inc(i);


    fim;

fim;

fim.
```

A classe TFileDisplayerRegistry tem três métodos. Dois são para gerenciar a conexão entre as extensões de arquivo e os exibidores que podem exibir esse tipo específico de arquivo. Você pode adicionar e recuperar expositores. O terceiro método é usado para obter uma lista de extensões suportadas para exibição ao usuário para que ele possa saber quais tipos de arquivos podem ser abertos.

Ele também possui uma única propriedade – DefaultDisplayer – que segue o padrão Property Injection. Ele permite que você defina um valor para um visualizador de arquivos padrão que será usado em

o evento de que não há extensão registrada para o arquivo escolhido. A classe fornece um exibidor padrão (visto abaixo) que é retornado quando nenhum exibidor registrado pode ser encontrado. (Observe que GetDisplayer segue a regra “Nunca retornar nil” sempre retornando uma implementação de IDisplayFile , não importa o que aconteça). Além disso, se você deseja fornecer seu próprio visualizador de arquivos padrão, pode fazê-lo definindo a propriedade DefaultDisplayer – exatamente como falamos no capítulo Injeção de propriedades. Veja, este material realmente funciona.

Observe também que a classe TFileDisplayerRegistry segue o padrão de registro e, portanto, é exposta como um Singleton. O padrão de registro praticamente exige que a classe de registro resultante seja um singleton, apesar do fato de que o padrão de singleton caiu em desuso (é realmente apenas uma variável global glorificada...). Fiz assim para fins de demonstração – uma versão mais robusta do aplicativo gerenciaria o registro sem recorrer a isso.

Aqui está o exibidor padrão, que apenas exibe algumas informações simples sobre o arquivo selecionado:

```
unidade uDefaultDisplayer;

interface

usa
      uFileDisplayerInterfaces
    , Vcl.ExtCtrls
    ;

tipo
  TDefaultFileDisplayer = class(TInterfacedObject, IDisplayFile) função privada estrita
  GetFileSize(aFilename: string): Int64; procedimento DisplayFile(const aFilename:
    string; const aPanel: TPanel);

  fim;

implementação

usa
      Vcl.StdCtrls
    , Vcl.Controls
    , System.IOUtils ,
    System.SysUtils
    ;

procedimento TDefaultFileDisplayer.DisplayFile(const aFilename: string; const aPanel: TPanel);
Onde
  Memorando: TMemo;
começar
```

```
  Memo := TMemo.Create(aPanel);
  Memo.Parent := aPanel; Memo.Align :=
  alCliente; Memo.ReadOnly := Verdadeiro;


  Memo.Lines.Add('Nome do arquivo: ' + aNome do arquivo);
  Memo.Lines.Add(' Tamanho do arquivo: '    + IntToStr(GetFileSize(aFilename)) + Memo.Lines.Add('  'bytes'); +
  Tempo de criação: '                        DateTimeToStr(TFile.GetCreationTime(aFilename)));
  Memo.Lines.Add(' Horário do último acesso: '          + DateTimeToStr(TFile.GetLastAccessTime(aFilename)));
  Memo.Lines.Add('Última hora de gravação: '          + DateTimeToStr(TFile.GetLastWriteTime(aFilename))); fim;

fim.
```

Para registrar todos esses expositores e classes, o projeto contém uma unidade chamada uRegistration.pas que possui um procedimento que faz todo o cadastro das extensões e expositores. Observe que você pode registrar vários tipos de arquivo no mesmo exibidor:

```
procedimento RegisterDisplayers;
Onde
  TextFileDisplayer: IDisplayFile;
  PictureFileDisplayer: IDisplayFile;
começar
  TextFileDisplayer : = TTextFileDisplayer.Create;
  FileDisplayerRegistry.AddDisplayer ('.txt', TextFileDisplayer); FileDisplayerRegistry.AddDisplayer
  ('.pass', TextFileDisplayer); FileDisplayerRegistry.AddDisplayer ('.dpr', TextFileDisplayer);
  FileDisplayerRegistry.AddDisplayer ('.dproj', TextFileDisplayer); FileDisplayerRegistry.AddDisplayer
  ('.xml', TextFileDisplayer);


  PictureFileDisplayer := TPictureDisplayer.Create;
  FileDisplayerRegistry.AddDisplayer('.jpg', PictureFileDisplayer); FileDisplayerRegistry.AddDisplayer('.bmp',
  PictureFileDisplayer); FileDisplayerRegistry.AddDisplayer('.png', PictureFileDisplayer);
  FileDisplayerRegistry.AddDisplayer('.gif', PictureFileDisplayer);

fim;
```

Registrei vários arquivos de texto diferentes que podem ser visualizados usando o player TTextFileDis.

Cadastrando Interfaces e Implementações.

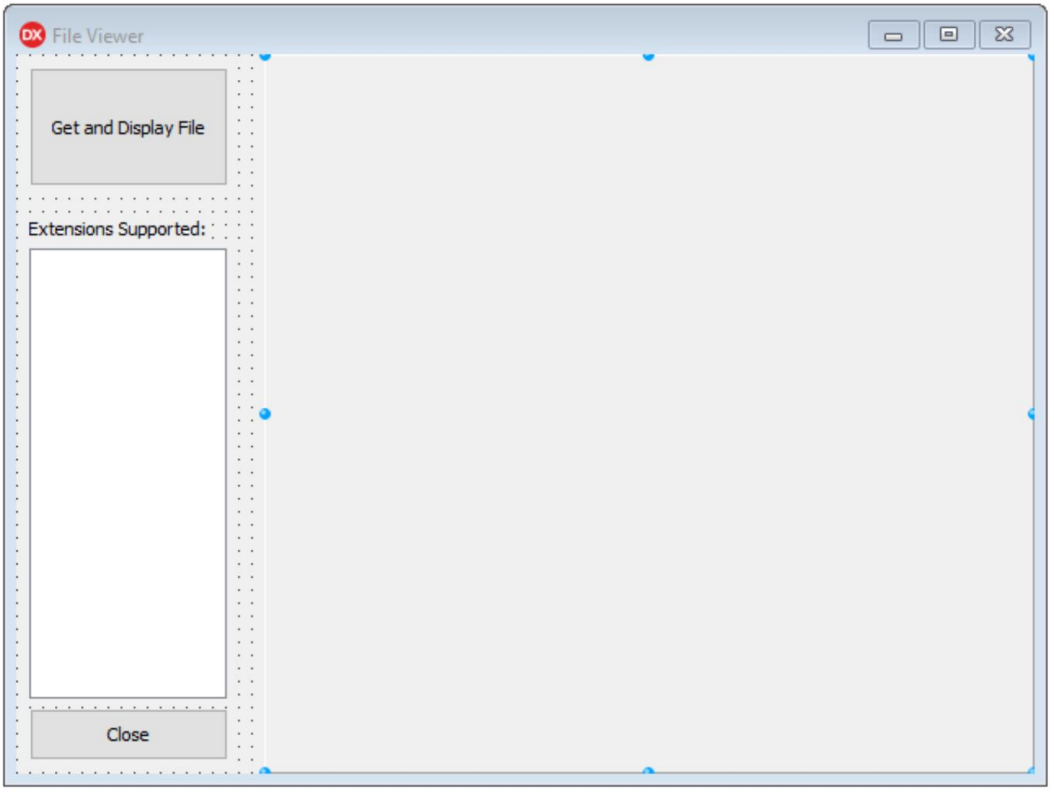
Claro, a verdadeira mágica acontece dentro do Dependency Injection Container. É lá que as interfaces são conectadas às suas implementações, os objetos necessários são criados e tudo é magicamente conectado. Aqui está o código de registro da unidade uRegistration :

```
procedimento RegisterInterfaces(aContainer: TContainer); começar

  aContainer.RegisterType<IDisplayOnPanel, TFileDisplayer>;
  aContainer.RegisterType<IFileExtensionGetter, TFileExtensionGetter>;
  aContainer.RegisterType<IFilenameGetter, TFilenameGetter>; aContainer.Build;

fim;
```

Feito isso, você não precisa criar nada – o contêiner fará todo o trabalho para você. Você notou que o aplicativo não cria nenhum de seus objetos de negócios (salve o singleton TFileDisplayerRegistry)? Isso porque o Container faz toda a criação e conexão de todas as referências de interface. (Lembra quando falamos sobre injetáveis? Bem, a maioria das coisas no aplicativo são injetáveis.
Sobre o único criável é o IDictionary no registro de extensão de arquivo.)

Conectando-se à interface do usuário

Claro, nada disso é útil a menos que você possa expô-lo para o usuário. Aqui está um formulário simples em tempo de design que nos permitirá exibir arquivos:



O aplicativo File Viewer no Form Designer

O código para fazer tudo funcionar é pateticamente simples. Já fizemos todo o trabalho pesado; agora é só aproveitar nossa bela arquitetura para mostrar um arquivo na aplicação.

Em primeiro lugar, adicionamos uma variável privada ao formulário do tipo IDisplayOnPanel:

```
tipo
  TFileDisplayerForm = class(TForm)
    Botão1: TBotão;
    Painel1: TPanel;
    Botão2: TBotão;
    ListBox1: TListBox;
    Label1: TLabel;
    procedimento Button1Click(Remetente: TObject);
    procedimento Button2Click(Remetente: TObject);
    procedimento FormCreate(Remetente: TObject);
  privado
      FContainer: TContainer;
      FileDisplayer: IDisplayOnPanel;
  fim;
```

Em seguida, no evento OnCreate do formulário , adicionamos o seguinte código:

```
procedimento TFileDisplayerForm.FormCreate(Remetente: TObject);
Onde
  Extensões: TArray<string>;
  Ext: cadeia; s:
  cadeia;
começar
  FContainer := TContainer.Create;
  FileDisplayer := FContainer.Resolve<IDisplayOnPanel>; Extensões :=
  FileDisplayerRegistry.GetExtensions; para Ext em Extensões faça

  begin
    s := Format('*.%s', [Ext]);
    ListBox1.Items.Add(s); fim; fim;
```

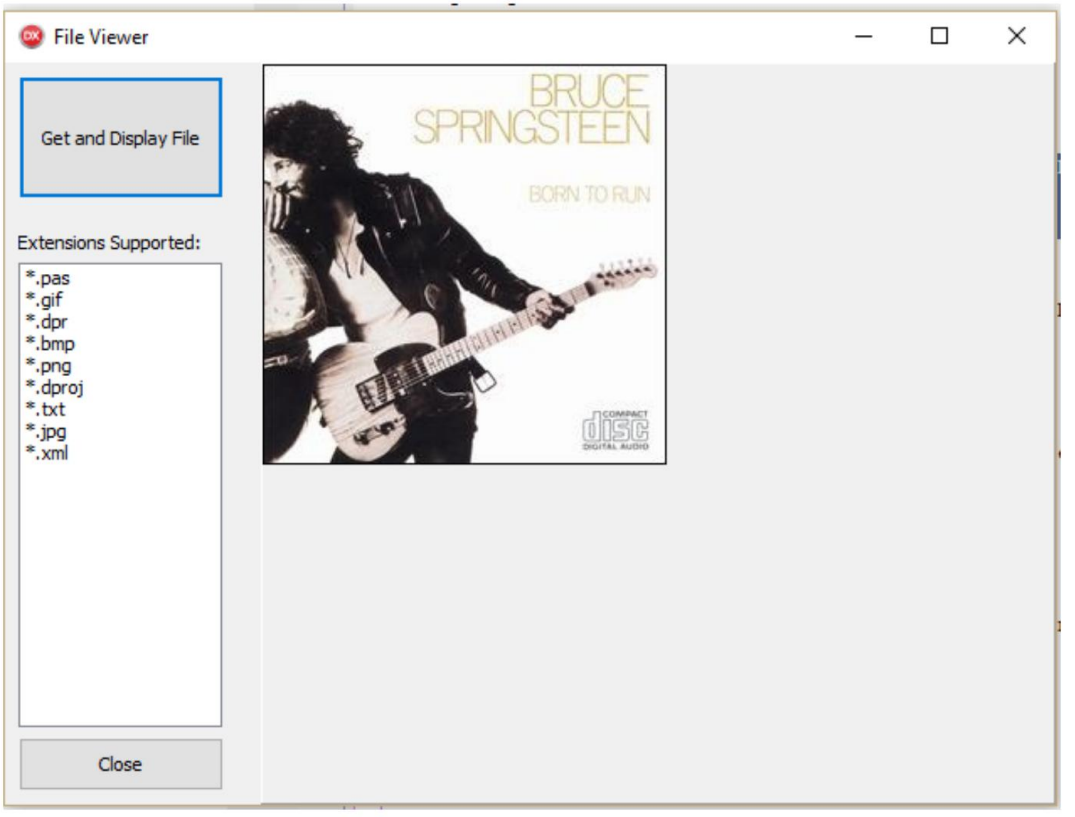
Aqui fazemos nossa única ligação para o Resolve (lembre-se, só recebemos uma ligação) e preenchemos a caixa de listagem com ramais qualificados. Não há muita coisa acontecendo lá.

A carne das coisas – e é de fato uma carne em fatias finas – está por trás do botão “Obter e exibir arquivo”:

```
procedimento TFileDisplayerForm.Button1Click(Sender: TObject); começar

  FileDisplayer.DisplayOnPanel(Panel1);
fim;
```

Isso é tudo para a interface do usuário. A sério. Há um pouco de funcionalidade em nosso aplicativo, mas não muito disso está vinculado à interface do usuário. E é assim que deve ser.

E agora aqui está o aplicativo exibindo uma imagem do meu álbum favorito de todos os tempos:



O aplicativo File Viewer exibindo um arquivo

Maneiras de melhorar este aplicativo

Este aplicativo é uma demonstração - mostra certas técnicas de injeção de dependência em ação. No entanto, por ser um aplicativo de demonstração, deixou as coisas bastante simples. Eu não queria obscurecer os principais pontos a serem feitos, tornando o aplicativo excessivamente complicado.

Aqui estão algumas coisas que você pode considerar adicionar ao aplicativo:

- Obviamente, você pode escrever displays adicionais para diferentes tipos de

arquivos. Cada arquivo pode ser exibido de alguma forma e, desde que você possa colocar os resultados em um TPanel, não há limite para o que você pode adicionar.

- A adição de novos tipos de arquivo atualmente requer uma recompilação. Você poderia facilmente criar um esquema que permitisse conectar visualizadores de arquivos dinamicamente.
- Há pouco ou nenhum tratamento de erros no aplicativo, embora seja provável que pouco precise ser adicionado. O aplicativo garante que nada seja nulo, portanto, não há necessidade de verificar isso. No entanto, o aplicativo não verifica, por exemplo, se o arquivo realmente existe antes de tentar exibi-lo. • Você pode melhorar a implementação de obtenção de arquivos permitindo que o usuário selecione a extensão em que está interessado e filtrando a caixa de diálogo de pesquisa para esses tipos de arquivo.

- Você pode considerar substituir o registro de extensão de arquivo por uma implementação que não use um Singleton.

Conclusões

Aqui estão alguns pensamentos enquanto encerramos:

- Todas as interfaces estão em uma única unidade, então você pode depender dessa unidade e não das unidades que implementam as interfaces. Claro, você pode acabar com um monte de unidades, mas como é tão fácil olhar para cada uma e descobrir o que ela faz, torna-se um pequeno preço a pagar.

- Cada uma dessas unidades contém uma classe que faz uma coisa. Seguir o Princípio da Responsabilidade Única mantém as coisas limpas e simples. O resultado final é que cada classe tem um propósito claro e compreensível. E ainda mais importante, cada classe é testável. • Todo o cadastro, tanto para o cadastro do displayer quanto para o cadastro do Dependency Injection Container, fica em uma unidade própria. É aqui que as coisas são acopladas – a unidade uRegistration é onde as unidades de implementação são usadas.

  Esse é o único “ponto de contato” para interfaces e implementações. Isso é tão fracamente acoplado quanto você pode obter.
- No final, temos um aplicativo limpo, extensível e pouco acoplado. Manter, atualizar e aprimorar esse aplicativo seria muito simples. Adicionar as coisas descritas acima não deve ser difícil.

Espero que agora você possa ver os benefícios de construir seu aplicativo com interfaces e Injeção de Dependência.

Pensamentos finais

Aí está. Espero que você tenha achado este livro útil. Espero que tenha mudado a maneira como você pensa sobre a construção de seus aplicativos. Espero que você veja a sabedoria de codificar contra abstrações e injetar suas dependências, e espero ter realizado o que me propus a fazer no prefácio.

Escrever e manter código já é difícil o suficiente sem emaranhar tudo em uma bola de lã bagunçada. Se você pudesse escrever seu código de uma maneira que tornasse as coisas independentes, unindo essas coisas livremente e resultando em um aplicativo poderoso, mas fácil de testar e manter, você faria isso, não faria? Espero tê-lo convencido de que a injeção de dependência pode fazer exatamente isso.

Como eu disse no prefácio – Injeção de Dependência é uma ideia muito simples: apenas entregue suas dependências às suas classes, normalmente através do construtor. Se essa ideia básica é tudo o que você aprendeu neste livro, fico feliz. Essa ideia simples pode transformar seu código do mundano para o sublime.

A injeção de dependência fez um mundo de diferença – tudo para melhor – na forma como escrevo código. Eu imploro a você – deixe-o fazer o mesmo por você.