Trabalho final de Ambientes Virtuais de Execução

Semestre de Inverno 2010/2011

Objectivos

Este trabalho tem como objectivo a consolidação de conhecimentos dos mecanismos e construções do sistema de tipos do .Net estudados nas aulas teóricas, nomeadamente a utilização de *delegates*, genéricos e reflexão .

Extensibilidade e adaptabilidade de aplicações

Um dos requisitos no desenvolvimento de uma aplicação é que esta seja extensível e adaptável, de tal forma que futuras alterações possam sejam incorporadas com o menor impacto possível tanto no tempo de desenvolvimento como no código fonte da aplicação. A programação orientada por objectos (POO) tem um papel central neste requisito, mas por si só não resolve todos os aspectos deste problema.

Considere o excerto da classe `EMailer`, responsável pelo processamento de mensagens de correio electrónico:

```
public class EMailer {
  private ISpellChecker spellChecker;
  public EMailer() {
    this.spellChecker = new PortugueseSpellChecker();
  }
  public void send(String text) { .. }
}
```

O construtor de `EMailer` instancia o *spell checker* usado. A criação explícita do *spell checker* obriga à colocação *hard-coded* do nome da classe concreta a usar, impossibilitando a sua substituição (por exemplo para um `EnglishSpellChecker`) sem alteração do código.

Para resolver o problema pode-se usar um *service locator* [1] para funcionar como *factory* [2] de objectos. Por exemplo:

```
public class EMailer {
  private SpellChecker spellChecker;
  public EMailer() {
    this.spellChecker = locator.getInstance("SpellChecker");
  }
  public void send(String text) { .. }
}
```

Desta forma é possível alterar o *spell checker* a utilizar em `Emailer` através da reconfiguração do *locator*. Outra das mais valias da utilização do *service locator* é a possibilidade de especificar, por exemplo, a semântica de activação. Por cada invocação de `getInstance` deve ser retornado o mesmo objecto *spell checker* (solitário) ou são retornadas diferentes instâncias?

Embora funcional, a utilização do *service locator* dispersa-se pelo código, criando um comprometimento adicional e dificultando, por exemplo, a criação de testes.

Outra alternativa de resolução do problema original é passar os objectos de que `EMailer` depende na fase de construção deste. Assim:

```
public class EMailer {
   private ISpellChecker spellChecker;
   public EMailer(ISpellChecker spellChecker) {
          this.spellChecker = spellChecker;
   }
}

…
Emailer em = new EMailer(new PortugueseSpellChecker());
```

O problema agora é que as dependências de `EMailer` passaram a ter de ser conhecidas pelos seus clientes (por quem instancia o `EMailer`). Este conhecimento viola de alguma forma o princípio de encapsulamento, pilar da programação orientada por objectos. O problema agudiza-se com o aumento das dependências. Por exemplo, se `EMailer` dependesse de um *logger*, de uma base de dados de contactos e de um protocolo de transporte específicos, estes teriam igualmente de ser passados no construtor, tornando incómoda a utilização e dificultando alterações pontuais.

Estas questões podem ser resolvidas com a utilização de uma infraestrutura de injecção de dependências [1], de forma a automatizar a construção de objectos. Numa infraestrutura desta tipo, existe (tal como num *service locator*) uma fase inicial de registo de resolução de dependências (*binding*). Após o registo a infraestrutura é usada para criar instâncias dos tipos registados(*injection*). A seguir apresenta-se a possível configuração de um *binder* para resolver a classe EMailer:

```
public class MyBinder : Binder {

       protected override void InternalConfigure() {
           Bind<ISpellChecker, PortugueseSpellChecker>();
           Bind<EMailer, Emailer>();
       }
}
```

Para criar objectos utiliza-se o injector de dependências (*injector*), devidamente configurado com determinado *binding*:

```
public class Injector {
        private Binder _myBinder;

        public Injector(Binder myBinder) {
            _myBinder = myBinder;
            _myBinder.Configure();
        }

        public T GetInstance<T>() { T t; … ; return t; }
    }
}
…
Injector  injector = new Injector(new MyBinder());
EMailer em = injector.GetInstance<EMailer>();
```

As dependências de construção ficam centralizadas no `Binder` e não espalhadas pelo código, o que é uma vantagem. Se o Binder for colocado num *assembly* próprio, é possível a sua reconfiguração sem alterar o resto

da aplicação. Por outro lado, através do *binder* é possível especificar outros aspectos como a semântica de activação e valores de parâmetros de construção de tipos intrínsecos.

Descrição do trabalho

Pretende-se neste trabalho implementar uma infra-estrutura, distribuída na forma de um *assembly* .NET, que tem as funcionalidades de injector de dependências e de *Service Locator*.

Como anexo a este trabalho, são disponibilizadas duas soluções do *Visual Studio*, para o VS2008 e para o VS2010, cada uma com 2 projectos:

`DIChelas.SampleTypes` – Projecto que inclui os tipos que serão usados nos testes à infra-estrutura.

`DIChelas.Tests` – Projecto que inclui os testes unitários (`DIChelasTests`). Os testes unitários são configurados através um `Binder` específico para projecto de testes (`MyBinder`) e um *custom resolver* também para o projecto de testes (`MyCustomResolver`).

Note-se que o projecto de testes não irá compilar, pois referencia tipos do *assembly* da infraestrutura(`DIChelas`) que terão de ser implementados. A especificação da interface e funcionamento da API do `DIChelas` deve ser inferida da sua utilização nos testes presentes em `DIChelas.Tests` e da classe `MyBinder`.

Referências

[1] Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html

[2] Wikipedia, Factory (software concept), http://en.wikipedia.org/wiki/Factory_(software_concept)

Notas

O Framework de testes unitários utilizado (NUnit) não está integrado no VS2008 e VS2010. Para integrar a execução dos testes com estas versões do Visual Studio, poderá optar pelas seguintes alternativas:

- Instalar o *plugin* `TestDriven.Net` (em http://www.testdriven.net/download.aspx).
- Instalar o *plugin* `Resharper`(em http://www.jetbrains.com/resharper/download). Neste caso apenas está disponível uma versão *trial* com licença para 30 dias. Contudo poder-se-á disponibilizar uma licença anual a quem estiver interessado.

Ambas as alternativas acrescentam ao menu de contexto de uma classe ou projecto que inclua uma classe de teste, uma opção para executar os respectivos testes unitários , diferindo apenas na visualização dos resultados.

Prazo de Entrega

A data limite para a entrega do trabalho é o dia 14 de Fevereiro de 2010