AA 19/20 – Relatório Architectural Designs

João Marques (A81826), José Pereira (A82880), Ricardo Petronilho (A81744)

16 de Março de 2020

1 INTRODUÇÃO

O presente relatório relata a investigação sobre os *Architectural Patterns*, *Dependency Injection* e *Inversion of Control*, no âmbito da Unidade Curricular Arquitecturas Aplicacionais do 4º ano do Mestrado integrado em Engenharia Informática da Universidade do Minho.

Em anexo será ilustrada a implementação do **Dependency Injection**.

2 PADRÃO ARQUITECTURAL - DEPENDENCY INJECTION

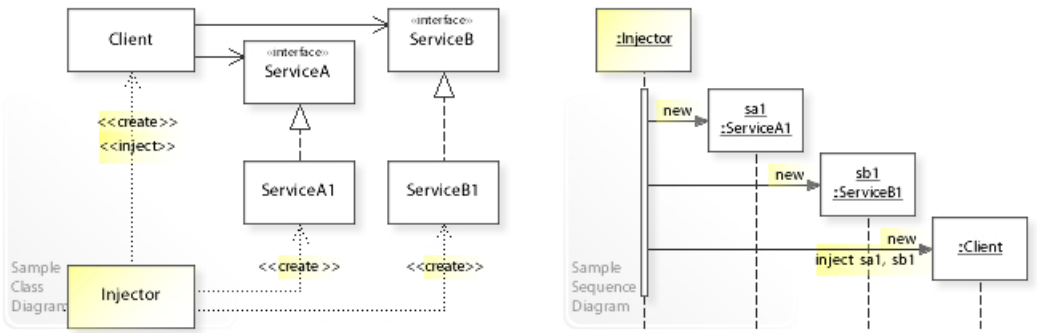

Figure 1. Dependency-injection.

O *pattern* arquitectural *dependency-injection* consiste numa técnica utilizada na construção de aplicações, onde um objecto declara as dependências de outro.

Uma **dependência** representa-se como um objecto, sendo este usado por exemplo como uma funcionalidade/serviço de uma aplicação, por outro lado, objectos que usem outros (objectos/funcionalidades/serviços) são considerados **clientes**.

A expressão "*dependency injection*" utilizada na nomeação deste *pattern*, deve-se ao facto do objecto **Injector**, criar e injectar/passar as **dependências/serviços** ao objecto **Client** como argumento, para que este os possa utilizar. Tal como se pode verificar na figura 1, no diagrama de sequência, a criação por parte do **Injector** dos serviços, seguida da injecção dos mesmos no **Client**, também criado pela classe **injectora**.

Os principais objectivos deste *pattern* arquitectural consiste na redução das dependências, bem como a separação das responsabilidades (criação e uso de objectos), com a finalidade de aumentar a fiabilidade e reutilização de código. Mas poder-se-á pensar o porquê da necessidade da separação destas responsabilidades. A razão consiste que o **Client** não deve criar os **objectos/serviços** através dos construtores destes, isto porque, no ponto de vista de evolução de código, se houver alterações à instanciação destas classes, vai ser necessário modificar o código do **Client**.

A solução encontrada, como já foi referido anteriormente, foi a injecção dos **serviços**, de forma a "impedir" que o **Client** saiba como são construídos, sendo isto da responsabilidade do **Injector**. O **Client** não executa qualquer código do **Injector** (não dependendo do mesmo), apenas conhece

as interfaces dos **serviços**, definindo-se nestas os métodos que o **Client** pode utilizar do respectivo **serviço**, tal como se pode verificar na figura 1.

Aparentemente, o *dependency-injection* parece muito semelhante ao **Abstract Factory**, no entanto, a grande diferença consiste em não ser necessário instanciar no **Client** o **factory**/**injector**, para a criação dos **objectos**/**serviços**, reduzindo assim as dependências. Por outro lado, o *dependency-injection* injecta/cria serviços, mas também pode apenas injectar serviços já existentes, sendo desta forma, diferente do **Abstract Factory**, que apenas cria objectos.

O *dependency-injection*, pode ser aplicado de diferentes formas, sendo a diferença entre elas, a forma como se injecta os **objectos**/**serviços**. Apesar de existirem várias formas de aplicar o *pattern*, apenas serão apresentadas as três mais importantes: *constructor injection*, *setter injection* e *interface injection*. O *constructor injection*, passa os **objectos**/**serviços**, pelo construtor do **Client**. O *setter injection*, define um método na classe **Client**, para efectuar a passagem dos **objectos**/**serviços**. Por fim, *interface injection*, bastante semelhante ao anterior, no entanto, a diferença consiste que neste caso, define-se uma interface, com o método para a passagem dos **objectos**/**serviços**, para "obrigar" o **Client** a defini-lo.

No entanto, o *pattern*, não consegue ser perfeito, contendo algumas desvantagens, das quais, dificuldades de análise/leitura do código, por parte dos *developers*, visto que, separa a criação do comportamento dos objectos. Do mesmo modo, devido ao facto de se mover a complexidade para fora das classes e passa-la para a interligação entre as mesmas torna-se mais complicada a sua gestão, entre outras desvantagens.

3 PADRÃO ARQUITECTURAL - INVERSION OF CONTROL

O IoC - **I**nversion **O**f **C**ontrol - é um padrão arquitectural muito usado nas aplicações de média ou alta complexidade que exigem **facilidade de manutenção**. Desta forma, para se perceber no que consiste o padrão IoC pode-se fazer uma comparação a um software tradicional (sem IoC).

Num software tradicional o engenheiro de software, desde o início do projecto, tem total controlo na decisão arquitectural do mesmo, isto é, toda a arquitectura é decidida e implementada por ele. Existem vantagens neste procedimento como **maior controlo e compreensão detalhada da arquitectura**. No entanto na perspectiva da **reutilização e manutenção do código este procedimento torna-se complicado e pouco viável**, uma vez que, para cada projecto diferente, a arquitectura base será também diferente, mesmo que utilizem as mesmas estruturas de dados genéricas.

O padrão IoC resolve precisamente o problema anterior, tornando possível criar e utilizar a mesma arquitectura para a base de todos os projectos, sendo que o papel do engenheiro de software consiste em "preencher os espaços em branco", isto é, **implementar porções de código específicas ao contexto do problema que o sistema pretende resolver**.

As **frameworks** conseguem reutilizar a mesma arquitectura base para todos os projectos, precisamente implementado o padrão IoC. Analisando qualquer projecto, por exemplo, utilizando Spring ou Laravel, têm exactamente a mesma base arquitectural. Desta forma para se implementar o IoC as frameworks podem utilizar o padrão DI - Dependency Injection - referido anteriormente.

Assim, note-se que o IoC não é exclusivamente usado nem destinado apenas a frameworks, é possível implementar o IoC num simples exemplo ou sistema, no entanto anteriormente foi dado relevo ao seu uso em frameworks uma vez que é o padrão que possibilita a própria criação da mesma.

4 EXEMPLO PRÁTICO - DEPENDENCY INJECTION

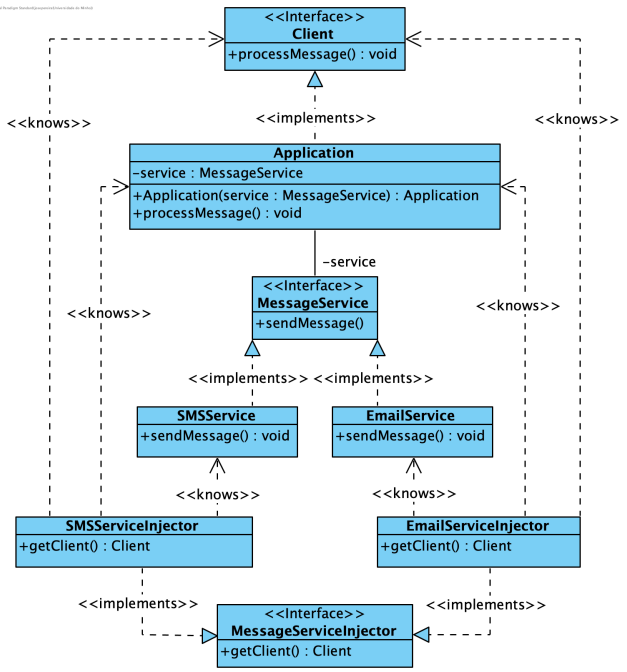

Figure 2. Diagrama de classes da implementação.

Com a finalidade de uma melhor percepção do *dependency injection*, a equipa implementou uma aplicação do mesmo a um problema real (A). Desta forma, o problema em questão consiste no envio de mensagens a partir de serviços diferentes, isto é, SMS e email. Como já foi referido acima, na secção 2, existem três entidades importantes a realçar: cliente, injector e os serviços.

O cliente consiste na classe **Application**, que implementa a interface **Client**, este contém o serviço para o poder utilizar, no entanto, esta classe não é responsável por criar-lo, apenas conhece o seu tipo genérico, isto é, a interface **MessageService**. Ainda em relação à classe do cliente, importa realçar a importância de se ter definido o construtor, que recebe como argumento um serviço do tipo mais genérico **MessageService**, visto que, este exemplo segue o **método do construtor**, isto é, os serviços são enviados através deste, tal como foi explicado na secção 2.

O injector tem o papel de criar o cliente e enviar os serviços a este. Desta forma, decidiu-se definir, uma classe injectora para cada serviço, isto é, **SMSServiceInjector** e **EmailServiceInjector**. Nestas classes, existe o método **getClient()**, que tal como o nome sugere, vai criar o objecto **Client**, passando-lhe o serviço. O serviço enviado, pode ser criado no momento, ou já existir.

Os serviços, contém as funcionalidades a executar pelo cliente, sendo o seu tipo mais genérico, e que o **Client** conhece, **MessageService**. Apesar de apenas ter-se definido uma interface, segundo o diagrama da figura 1, cada serviço poderá ter uma interface independente. No entanto, para este exemplo decidiu-se definir um tipo comum para o serviço, visto que facilita o processo. A interface **MessageService**, define os métodos que se pretende que o **Client** possa utilizar do serviço, sendo neste caso o **sendMessage()**. Cada serviço contém uma classe específica, que implementa a **MessageService** e os respectivos métodos.

Assim, o mais importante a reter deste exemplo, são os objectivos do *dependency injection*, ou seja, o **Client** (**Application**), apenas usa os serviços, não os cria, e não depende da classe que os cria, isto é, as classes **injectoras** (**SMSServiceInjector** e **EmailServiceInjector**). Do mesmo modo, tal como já foi dito anteriormente, os serviços não necessitam de ser criados no momento, podem já existir.

5 CONCLUSÃO

Após a conclusão desta pesquisa, percebe-se a utilidade do *Dependency Injection* e *Inversion of Control*, nas aplicações atuais, visto que estão em constante evolução.

Tal como se viu ao longo deste relatório, estes *patterns* caracterizam-se por facilitarem o processo de evolução de código, onde após alterações no mesmo, reduz-se a necessidade de serem alteradas outras partes.

Em relação ao *Inversion of Control*, percebe-se a sua utilidade, visto que, se existir uma arquitectura previamente bem definida e comum a vários projectos, torna-se fácil a análise e trabalho sobre estes.

6 ANEXOS

A DEPENDENCY INJECTION

```java
public interface MessageService {
    void sendMessage(String msg);
}

// ------------------------------------------------------------

public class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("SMS: " + message);
    }
}

// ------------------------------------------------------------

public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Email: " + message);
    }
}

// ------------------------------------------------------------

public interface Client {
    void processMessage(String msg);
}

// ------------------------------------------------------------

public class Application implements Client {
    MessageService service;

    public Application(MessageService service) {
        this.service = service;
    }

    @Override
    public void processMessage(String msg) {
        this.service.sendMessage(msg);
    }
}

// ------------------------------------------------------------
```

```
public interface MessageServiceInjector {
    public Client getClient();
}

// ------------------------------------------------------

public class SMSServiceInjector implements MessageServiceInjector {
    @Override
    public Client getClient() {
        return new Application(new SMSService());
    }
}

// ------------------------------------------------------

public class EmailServiceInjector implements MessageServiceInjector {
    @Override
    public Client getClient() {
        return new Application(new EmailService());
    }
}

// ------------------------------------------------------

public class Main {
    public static void main(String[] args) {
        String msg = "Hello";
        MessageServiceInjector injector = null;
        Client client = null;

        //Send Email
        injector = new EmailServiceInjector();
        client = injector.getClient();
        client.processMessage(msg);

        //Send SMS
        injector = new SMSServiceInjector();
        client = injector.getClient();
        client.processMessage(msg);
    }
}

// ------------------------------------------------------

Output:
Email: Hello
SMS: Hello
```