**CES-28 Fundamentos de Eng. de Software**

**Dependency Injection Pattern**

[Atividade em grupo: até 2 alunos]

[Introdução]

Estou propondo experimentar exemplos de programação com o padrão de injeção de dependência (Dependency Injection Pattern). Entregar o material produzido no TIDIA, quer seja código ou diagrama de classes ou de sequência. O objetivo é trabalhar os exemplos e ter consciência dos conceitos e técnicas empregados.

[Tarefa]

Estudar e experimentar os recursos sobre Dependency Injection Pattern em Java.

[Processo]

1. Montar grupo com até 2 integrantes e ler o Recurso [A]
2. Dado o código do Recurso [B], entender o funcionamento, implementar as classes apresentadas no Eclipse ou outro IDE (Interactive Development Environament) e desenvolver uma classe Main para testar o funcionamento.

🡪 Ao final, mostrar funcionando para o professor e, ao final, depositar esse código com a saída dele junto com os demais resultados na atividade TIDIA correspondente!

1. Desenhar o diagrama de classes e o diagrama de sequência (Main) em UML correspondentes ao código do item 2.

🡪 Mostrar e depositar TIDIA

1. Considerando agora que o relacionamento das classes envolvidas nos itens anteriores é por agregação, com injeção dependência por construtor. Criar novo pacote e implementar adaptando o código do recurso [B] para essa nova realidade; providenciar classe Main para testar o funcionamento.

🡪 Mostrar e depositar TIDIA

1. Desenhar o diagrama de classes e de sequências (Main), somente se eles forem diferentes dos do item 3 acima.

🡪 Mostrar e depositar TIDIA

1. Suponha agora que eu queira incluir um novo serviço de mensagem para celular: SMSService. Criar uma interface MessageService, cujas implementações são EmailService e SMSService. Fazer um acoplamento abstrato entre a classe MyApplication com a interface MessageService, com injeção de dependência por construtor. Essa injeção de dependência é chamada de DI manual.

Faça as alterações necessárias nos códigos existentes + o código novo, implemente uma classe de teste e teste tudo.

🡪 Mostrar e depositar TIDIA

1. Desenhar o diagrama de classes e de sequências (Main), somente se eles forem diferentes dos do item 3 acima.

🡪 Mostrar e depositar TIDIA

1. Outras formas de resolver esse problema é com o uso dos padrões Método de Fábrica e Service Locator. O problema aqui é que o desacoplamento das dependências ainda não é muito grande e há certa dificuldade de se testar com o uso de mocks.
2. Leia o Recurso [C] que mostra o padrão Dependency Injection.
3. Dado o código do Recurso [D], entender o funcionamento, implementar as classes apresentadas no Eclipse ou outro IDE (Interactive Development Environament) e desenvolver uma classe Main para testar o funcionamento.

🡪 Mostrar e depositar TIDIA

1. Desenhar o diagrama de classes e o diagrama de sequência (Main) correspondentes ao código do item 10.

🡪 Mostrar e depositar TIDIA

1. A injeção de dependência empregada no código exemplo por construtor ou por setter?!? Quer seja uma forma ou outra, como o Main fica com a forma que não foi empregada?!?

🡪 Mostrar e depositar TIDIA

1. Como fica uma representação diagramática mais livre (não UML) da DI neste caso, ilustrando como as dependências são criadas em tempo de execução?

🡪 Mostrar e depositar TIDIA

1. Quero colocar mais um recurso de mensagem para o Facebook. Mostrar o que precisa ser feito para se fazer essa inclusão, ou seja, que classes precisam ser criadas e modificadas.

🡪 Mostrar e depositar TIDIA

1. Frameworks para o padrão DI: Spring, Google Guice, JEE CDI, MS Unity etc. Ironicamente, o uso de DI pode encorajar dependência de um framework de DI.
2. Listar vantagens e desvantagens encontradas na literatura do recurso [C] ou outro qualquer, com especial atenção aos dois últimos textos que apontam problemas para o uso do padrão DI.

🡪 Mostrar e depositar TIDIA

**[Recursos]**

[A] Conceitos preliminares sobre injeção de dependência:

* A injeção de dependência diminui o acoplamento entre uma classe e sua dependência.
* A injeção de dependência adia a criação do serviço da associação cliente-servidor (serviço) correspondente do tempo de compilação para o de execução!
* Serviço—Um objeto que realiza uma função bem definida quando chamado
* Cliente—Qualquer consumidor de um serviço; um objeto que chama um serviço para realizar uma função bem definida
* Dependência—Um serviço específico que é exigido por outro objeto para preencher sua função
* Dependente—Um objeto cliente que precisa de uma ou mais dependências para realizar sua função
* Um objeto não pode funcionar adequadamente sem suas dependências

[B] Aplicação exemplo:

**public** **class** MyApplication {

**public** **void** processMessages(String message, String receiver) {

// Do some msg validation, manipulation logic etc.

\_email.sendEmail(message, receiver);

}

**private** EmailService \_email = **new** EmailService();

}

**public** **class** EmailService {

**public** **void** sendEmail(String message, String receiver){

// Logic to send email

System.*out*.println("Email sent to "+ receiver);

System.*out*.println("----------------------------------");

System.*out*.println("Message: " + message);

}

}

[C] Dependency Injection Pattern, que requer o seguinte:

* Componentes de serviços que refinam **interface de serviços** – classes de dependências das classes consumidoras!
* **Classes consumidoras ou clientes**, que devem ser escritas em função da interface de serviços – classes dependentes das classes de serviços
* Classes injetoras: inicializarão os serviços e em seguida inicializarão as classes consumidoras e injetarãos os serviços correspondentes nelas (classes consumidoras); também chamadas de **containers, assemblers e construction codes!**
* <http://www.martinfowler.com/articles/injection.html>
  + Muito bom: mais longo, mas cobre histórico, variações e especialmente vantagens e desvantagens!
* <https://en.wikipedia.org/wiki/Dependency_injection>
* <https://msdn.microsoft.com/en-us/library/ff921152.aspx>
  + curto, formato de catalogo de DP
* <http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection>
  + não gostei da analogia e explicações, mas traz exemplos de implementações em diferentes frameworks.
* THE EVIL SIDE
* <http://www.natpryce.com/articles/000783.html>
  + curto, explica alguns exageros no uso do conceito
* <http://www.tonymarston.net/php-mysql/dependency-injection-is-evil.html>
  + mais longo, mas ótima leitura. Volta pro básico. Perceba que ele não é contra DI per se, mas é contra usar DI por default, sem que haja necessidade! Deixa claro os custos da DI, e advoga que só vale a pena quando há necessidade!

[D] Aplicação exemplo ampliada com DI Pattern:

**// Serviços**

**public** **interface** MessageService {

**void** sendMessage(String msg, String rec);

}

**public** **class** EmailServiceImpl **implements** MessageService {

@Override

**public** **void** sendMessage(String msg, String rec) {

// logic to send email

System.*out*.println("Email sent to " + rec + " with Message=" + msg);

}

}

**public** **class** SMSServiceImpl **implements** MessageService {

@Override

**public** **void** sendMessage(String msg, String rec) {

// logic to send SMS

System.*out*.println("SMS sent to " + rec + " with Message=" + msg);

}

}

// Classes Consumidoras

**public** **interface** Consumer {

**public** **void** processMessages(String msg, String rec);

}

**public** **class** MyDIApplication **implements** Consumer {

**private** MessageService service;

**public** MyDIApplication(MessageService svc) {

**this**.service = svc;

}

@Override

**public** **void** processMessages(String msg, String rec) {

// Do some msg validation, manipulation logic etc

**this**.service.sendMessage(msg, rec);

}

}

// Classes Injetoras

**public** **interface** MessageServiceInjector {

**public** Consumer getConsumer();

}

**public** **class** EmailServiceInjector **implements** MessageServiceInjector {

@Override

**public** Consumer getConsumer() {

**return** **new** MyDIApplication(**new** EmailServiceImpl());

}

}

**public** **class** SMSServiceInjector **implements** MessageServiceInjector {

@Override

**public** Consumer getConsumer() {

**return** **new** MyDIApplication(**new** SMSServiceImpl());

}

}