README.md

Travaux pratiques - Injection de dépendances

Lors de la première séance du module, nous avons utilisé *Twitter* pour illustrer les concepts principaux de l'injection de dépendances avec *Guice*. Maintenant, nous avons traiter une version étendue de cet exemple, ce qui nous permettra de remarquer plus facilement l'utilité de ce patron de conception ainsi que d'explorer d'autres caractéristiques plus avancées de *Guice*.

Partie 1 : préparation

1. Importez le code fourni dans Eclipse.
2. Vous pouvez étudier le code et exécuter la classe méthode principale (classe `tp.Main`) pour mieux comprendre le fonctionnement du programme.

Partie 2 : le code fourni

1. Quel est-il le type d'injection de dépendances utilisé dans le projet fourni ?
2. Pour mieux comprendre les différents niveaux de dépendances entre les classes/objets, dessinez un diagramme de classes illustrant les dépendances entre classes (**pas d'interface!**).
3. Quels sont-ils les avantages et inconvénients de l'architecture adoptée dans projet fourni ?
4. Si on devrait changer des dépendances, dans combien d'endroits dans le code faudrait il changer ?
    - par exemple: *SmsTwitter* -> *GSMTwiter*, *Md5Cryptographer* -> *DummyCryptographer*, *FileLoginData* -> *MemoryLoginData*, ...

Partie 3 : *Guice* basique

1. Transformez l'injection de dépendances du projet fourni afin d'utiliser *Guice*. Pour ce faire, il faut: *Annoter* les dépendances dans toutes les classes dépendantes:

```
public class TwitterClient {

  @Inject
  private Twitter twitter;

  @Inject
  private Shortener shortener;
  ...
}
```

or

```
public class TwitterClient {
 ...

 @Inject
 public TwitterClient(Login loginManager, Shortener shortener, Twitter twitter) {
    this.loginManager = loginManager;
    this.shortener = shortener;
    this.twitter = twitter;
 }
 ...
}
```

    - Créer une classe module définissant/concentrant toutes les dépendances

```
public class Module1 extends AbstractModule {

  @Override
  protected void configure() {
        bind(Login.class).to(LoginConsole.class);
        bind(Shortener.class).to(TinyUrlShortener.class);
        bind(Twitter.class).to(SmsTwitter.class);
        ...


  }
  ...
}
```

Attention : utilisez `bind(MonInterface.class).toInstance(monInstance);` pour associer une interface à une instance specifique.

- Utiliser le *module* ci-dessus pour créer un injecteur qui servira comme fabrique générique :

```
public class Main {

  public static void main(String[] args) {
     ...
     Injector injector = Guice.createInjector(new Module1());
     TwitterClient client = injector.getInstance(TwitterClient.class);
     ...
  }
}
```

2. Définissez un module contenant la configuration ci-dessous et testez-le

Type Statique	Type Dynamique
Shortener	DummyUrlShortener
Twitter	SmsTwitter
Cryptographer	DummyCryptographer
LoginData	MemoryLoginData
Connection	RestConnection
Transmitter	IosTransmitter

3. Définissez un autre module contenant la configuration ci-dessous et testez-le

Type Statique	**Type Dynamique**
Shortener	TinyUrlShortener
Twitter	MockTwitter
Cryptographer	Md5Cryptographer
LoginData	MemoryLoginData
Connection	SoapConnection
Transmitter	IosTransmitter

4. Dans cette version avec *Guice*, quelles classes sont-elles devenues inutiles ?
5. Dans combien d'endroits dans le code faudrait il changer si l'on voudrait modifier les dépendances ?

Partie 4 : *Guice* avancé

Injection nommée

Si notre application a plusieurs dépendances du même type `MonInterface`, dans différentes classes, l'association `bind(MonInterface.class).to(MaClasse.class);` sera appliquée à toutes les dépendances. Pour contourner cette limite, nous pouvons nommer les dépendances et s'en servir pour associer une classe à une dépendance spécifique :

```
public class ClassDependante {
    ...
    @Inject
    @Named("NomDeLaDependance")
    private  MonInterface maDependance;
    ...
}

public class Module1 extends AbstractModule {

    @Override
    protected void configure() {
        ...

        bind(MonInterface.class).annotatedWith(Names.named("NomDeLaDependance")).to(MaClasse.class);
        ...
    }
    ...
}
```

1. Modifiez l'application pour que seules les dépendances du type `Appendable` des classes `SmsTwitter` et `AndroidTransmitter` soient associées à `System.err` au lieu de `System.out`
2. Créez une nouvelle dépendance du type `Twitter` dans `TwitterClient` de sorte que si l'appel à la méthode *send* de la première dépendance renvoie *false*, le programme re-essaye avec la deuxième. Modifiez la classe module pour que les deux dépendances soient associées à deux classes différentes.

Méthodes @Provides

Lorsque nous avons besoin d'un code pour créer un objet, nous pouvons définir une méthode dans la classe module et l'annoter avec `@Provides`. Cette méthode est appelée à chaque fois que l'injecteur a besoin de créer un objet du type en question. Par exemple, la méthode `provideMonInterface` ci-dessous est appelée pour créer un objet `MaClasse` qui sera associé aux dépendances de type `MonInterface` (pour `MaClasse implements MonInterface`). Ces méthodes sont également utiles pour injecter des dépendances selon une condition qui doit être vérifiée à l'exécution.

```
public class Module1 extends AbstractModule {
   @Override
   protected void configure() {
      ...
   }

   @Provides
   MonInterface provideMonInterface(){
        MaClasse monObj = new MaClasse();
        monObj.setAttribute1("Valeur_Arbritaire");
        return monObj;
    }


}
```

1. Modifiez votre implementation pour que l'URL (voir la méthode `Connection.setUrl`) soit définie juste après la creation d'un objet du type `Connection`.