Pràctica de Patrons de Disseny

Quan dissenyem un sistema orientat a objectes, com distribuïm la funcionalitat en molts objectes amb responsabilitats diferents, un dels problemes que hem de resoldre és el de la seva interconnexió.
Què volem dir amb interconnexió? Suposem que tenim un conjunt d'*interfícies* que poden ser implementades de vàries maneres per diverses *classes concretes* (en el nostre cas tot aquest codi serà part dels tests que farem per provar que la nostra implementació funciona) .

```
1 // Interfícies
2 interface InterfaceA { ... }
3 interface InterfaceB { ... }
4 interface InterfaceC { ... }
5 interface InterfaceD { ... }
6
7 // Implementacions
8 class ImplementationA1 implements InterfaceA {
9    private InterfaceB b;
10   private InterfaceC c;
11   public ImplementationA1(InterfaceB b, InterfaceC c) {
12         this.b = b; this.c = c;
13   }
14 }
15
16 class ImplementationB1 implements InterfaceB {
17   private InterfaceD d;
18   public ImplementationB1(InterfaceD d) {
19         this.d = d;
20  }
21 }
22
23 class ImplementationC1 implements InterfaceC {
24   private String s;
25   public ImplementationC1(String s) {
26         this.s = s;
27  }
28 }
29
30 class ImplementationD1 implements InterfaceD {
31   private int i;
32   public ImplementationC1(int i) {
33         this.i = i;
34  }
35 }
```

Aspectes a comentar:

- Les classes concretes depenen de les interfícies i no de les altres classes concretes
- Les classes concretes són independents entre sí; només depenen de les interfícies
- Cada classe concreta requereix que al seu constructor se li passin els objectes que implementen les interfícies de les quals depèn

El problema de la *interconnexió* consisteix en, per a una configuració concreta del sistema, decidir quines són les implementacions que s'utilitzaran de cadascuna de les interfícies i fer que aquestes implementacions es passin com a paràmetres per a construir els objectes que les necessiten. En un sistema real, per exemple, podríem estar decidint quin motor de base de dades s'utilitza, quin sistema de plantilles, etc., etc.

Patró Service Locator

Hi ha varis patrons que serveixen per aconseguir això i, potser el més senzill de tots, és l'anomenat Service `Locator` (o localitzador de serveis).

Primera versió

Una possibilitat és crear un *registre* (el `ServiceLocator`) on associarem un `String` a cadascuna de les *implementacions* de cadascuna de les interfícies.
Hi ha varis dissenys possibles d'aquest registre depenent de les propietats que es demanen a les coses que es treuen del registre:

- Si quan es demana per una clau, sempre es vol el mateix objecte, es pot guardar directament la instància
- Si el es què vol és un objecte diferent (però de la implementació enregistrada), existeixen vàries possibilitats:
  - Guardar instàncies i clonar-les
  - Guardar factories de les implementacions

Nosaltres farem servir la darrera possibilitat: enregistrar *factories* de les diferents implementacions. Això també ens permetrà no lligar les implementacions amb el `ServiceLocator` ja que aquest serà utilitzat només per les *factories*.
Per tant, definirem les següents interfícies i classes al paquet `servicelocator`:

```
1 public class LocatorError extends Exception { ... }
2
3 public interface Factory {
4    Object create(ServiceLocator sl) throws LocatorError;
5 }
6
7 public interface ServiceLocator {
8    void setService(String name, Factory factory)
9                throws LocatorError;
10   void setConstant(String name, Object value)
11               throws LocatorError;
12   Object getObject(String name)
13               throws LocatorError;
14 }
```

Els mètodes de la classe `ServiceLocator` es comporten de la següent manera:

- `setService` instal·la una factoria donant-li un nom (i llença l'excepció `LocatorError` si ja hi ha alguna cosa enregistrada amb aquest nom)
- `setConstant` instal·la un valor de tipus `Object` donant-li un nom (i llença l'excepció `LocatorError` si ja hi ha alguna cosa enregistrada amb aquest nom)
- `getObject`, si el nom ha estat associat a una constant, retorna l'`Object` associat i, si ha estat associat a una factoria, retorna l'`Object` creat per aquesta factoria. Llença l'excepció `LocatorError` si no hi ha cap cosa sota aquest nom.
  Fixeu-vos en que, en cas d'estar associat a una factoria, cada vegada que es crida, retorna un objecte diferent.

Respecte de la interfície `Factory` indicar que el mètode `create` usa el `ServiceLocator` que rep per a obtenir les implementacions de les interfícies que necessita. També pot llençar l'excepció `LocatorError` si hi ha problemes al buscar dependències al `ServiceLocator` o si aquestes no implementen la interfície que es demana (veure l'exemple de factoria a la següent secció)..

Implementació de les factories

Definirem una factoria per a cadascuna de les classes concretes sobre les que treballarem. Per exemple, la factoria que crearà instàncies de la classe `ImplemetationA1` és:

```
1 class FactoryA1 implements Factory {
2   public InterfaceA create (ServiceLocator sl)
3                     throws LocatorError {
4     try {
5       InterfaceB b = (InterfaceB) sl.getObject("B");
6       InterfaceC c = (InterfaceC) sl.getObject("C");
7       return new ImplementationA1(b, c);
8     } catch (ClassCastException ex) {
9       throw new LocatorError(ex);
10    }
11  }
12 }
```

I, en alguna part del codi, haurem de configurar el `ServiceLocator`, associant els noms amb les constants o factories que després s'usaran des de les factories a l'hora d'obtenir les dependències.

Implementació de ServiceLocator

Definirem dues implementacions de `ServiceLocator`:

- `SimpleServiceLocator`
- `CachedServiceLocator`

La diferencia entre un i l'altre és que en el segon, tant si hem enregistrat una constant com una factoria, quan demanem un objecte associat al nom **sempre es retorna la mateixa instancia**. En la primera implementació això només passa quan havíem enregistrat el nom amb una constant, si hem enregistrat una factoria, sempre obtenim objectes diferents.

Segona versió

Fixeu-vos que, a la versió anterior, no tenim cap mena de control sobre la configuració del `ServiceLocator` (tot acaben sent `Objects`) i, per tant, el compilador és incapaç de detectar si ens equivoquem en la configuració.
Aquest segon apartat està encaminat a aconseguir **seguretat de tipus** en temps de compilació i a explorar coses com

- la classe genèrica `Class<T>` per usar "type literals" per marcar el tipus en temps d'execució (veure Class Literals as Runtime-Type Tokens)
- evitar l'ús de "raw types" per a referir-se a una classe genèriques sense instanciar el paràmetre de tipus (veure Raw Types)
- la directiva `-Xlint:unchecked` del compilador per assenyalar l'ús del genèrics. Normalment no està activada per a permetre que el codi antic, sense genèrics, no doni problemes (veure Unchecked Error Messages)
- l'anotació `@SuppressWarnings("unchecked")` per marcar instruccions que el compilador detecta com a warnings però que, pel context del programa, sabem que no generen problemes (veure Predefined Annotation Types)

Les classes i interfícies per aquesta versió aniran al paquet `servicelocator2` i seran:

```
13 public class LocatorError extends Exception { ... }
14
15 public interface Factory<T> {
16    T create(ServiceLocator sl) throws LocatorError;
17 }
18
19 public interface ServiceLocator {
20    <T> void setService(Class<T> klass, Factory<T> factory)
21                throws LocatorError;
22    <T> void setConstant(Class<T> klass, T value)
23                throws LocatorError;
24    <T> T getObject(Class<T> klass)
25                throws LocatorError;
26 }
```

Fixeu-vos que ara les factories podran fer:

```
27 class FactoryA1 implements Factory<InterfaceA> {
28   public InterfaceA create (ServiceLocator sl)
29                     throws LocatorError {
30     InterfaceB b = sl.getObject(InterfaceB.class);
31     InterfaceC c = sl.getObject(InterfaceC.class);
32     return new ImplementationA1(b, c);
33   }
34 }
```

ja que ara no hi haurà la possibilitat d'un `ClassClastException` quan estem demanant objectes al `ServiceLocator`.

També tindrem dues versions d'aquest nou `ServiceLocator`

- `SimpleServiceLocator`
- `CachedServiceLocator`

Amb les mateixes característiques descrites abans.

Es demana

- Projecte IntelliJ amb el codi de les quatre implementacions diferents de `ServiceLocator`
  - `servicelocator`
    - `LocatorError`
    - `Factory`
    - `ServiceLocator`
    - `SimpleServiceLocator`
    - `CachedServiceLocator`
  - `servicelocator2`
    - `LocatorError`
    - `Factory<T>`
    - `ServiceLocator`
    - `SimpleServiceLocator`
    - `CachedServiceLocator`

- Classes de test que comprovin el correcte funcionament de les vostres implementacions
    - En aquest codi definireu les interfícies i implementacions de les classes de les vostres proves, les seves factories, etc., etc.
    - Intenteu evitar que hi hagi codi duplicat als tests (recordeu que les classes de test són classes com les altres).
- Petit informe que descrigui el vostre disseny, les principals dificultats en que us heu trobat, com les heu solucionat, consideracions de disseny, etc., etc.

Enllaços

Com al transformar al PDF, en alguns lectors, no funcionen els enllaços, aquí teniu els enllaços utilitzats anteriorment.:

- Class Literals as Runtime-Type Tokens
    - https://docs.oracle.com/javase/tutorial/extra/generics/literals.html
- Raw Types i Unchecked ErrorMessages
    - https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html
- Predefined Annotation Types
    - https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html