P4

Francisco Javier Hernández Martín, Jose Luis Bueno Pachón, Carlos Marín Corbera, Carmen González Ortega, Altair Bueno Calvente

UNIVERSIDAD
DE MÁLAGA

15 jan 2022

Contents

Preámbulo

Para la realización de esta práctica, hemos optado por utilizar un estilo de programación estandarizado para el lenguaje de programación Java, conocido como **Google Java Style Guide**[1]

Ejercicio 1: Los interfaces selectivos

Apartado A

En Java contamos con los **modificadores de acceso** para exportar de forma selectiva las distintas clases y atributos del sistema. Por ejemplo, para crear una clase `Foo` con visibilidad pública escribimos lo siguiente:

```
// Modificador de acceso público
// vvv
public class Foo {/* ... */}
```

En Java, los distintos modificadores de acceso son:

- **Visibilidad privada (`private`)**: Es la visibilidad más restrictiva. El elemento marcado con el atributo `private` será solo visible desde la clase en la que se declara.
- **Visibilidad de paquete**: Es la visibilidad por defecto de Java. Aquellos elementos que no han sido explícitamente marcados serán considerados visibles desde el paquete. Esto significa que el elemento se comporta con visibilidad pública en aquellas clases del mismo paquete, pero de forma privada con las clases externas al paquete.
- **Visibilidad protegida (`protected`)**: Al igual que la visibilidad de paquete, la visibilidad protegida impide que las clases externas accedan a aquellos elementos marcados con el atributo `protected`, aunque permite a sus subclases y clases del mismo paquete acceder a dichos elementos.
- **Visibilidad pública (`public`)**: Permite que el elemento marcado con el atributo `public` sea visible desde cualquier clase.

En la siguiente tabla se resumen las características principales de las distintas visibilidades de Java:

[1] https://google.github.io/styleguide/javaguide.html

Modificador	Misma clase	Mismo paquete	Distinto paquete, subclase	Distinto paquete
Privado	Sí	No	No	No
De paquete	Sí	Sí	No	No
Protegido	Sí	Sí	Sí	No
Público	Sí	Sí	Sí	Sí

Por otra parte, el lenguaje Eiffel profundiza más en el mecanismo de exportación selectiva, otorgando al programador de un control granular sobre el acceso a los elementos de las clases. Los distintos modificadores de acceso Eiffel, según Wikipedia[2], son:

- **Visibilidad pública (`feature`)**: Al igual que en Java, permite que sea visible desde cualquier clase.
- **Visibilidad protegida (`feature {}`)**: Al igual que en Java, permite que sea visible desde sus subclases y clases del mismo paquete.
- **Visibilidad *selectiva* (`feature {class1,class2,class3...}`)**: Permite que dicho elemento sea visible a las clases seleccionadas, actuando como privada para las demás clases.

Podemos observar que ambos lenguajes tienen distintos modificadores de acceso; Java incluye visibilidad **privada** y **de paquete**, mientras que Eiffel incluye ***selectiva***, siendo esta última un preciso que las demás visibilidades. También se puede diferenciar entre ambos lenguajes según su metodología para controlar el acceso: los modificadores de acceso de Java dependen del rango o alcance (misma clase, subclases, paquete, fuera del paquete…) y los modificadores de Eiffel dependen de quién intente acceder a ese elemento/característica.

[2] https://en.wikipedia.org/wiki/Eiffel_(programming_language)#Scoping

Apartado B

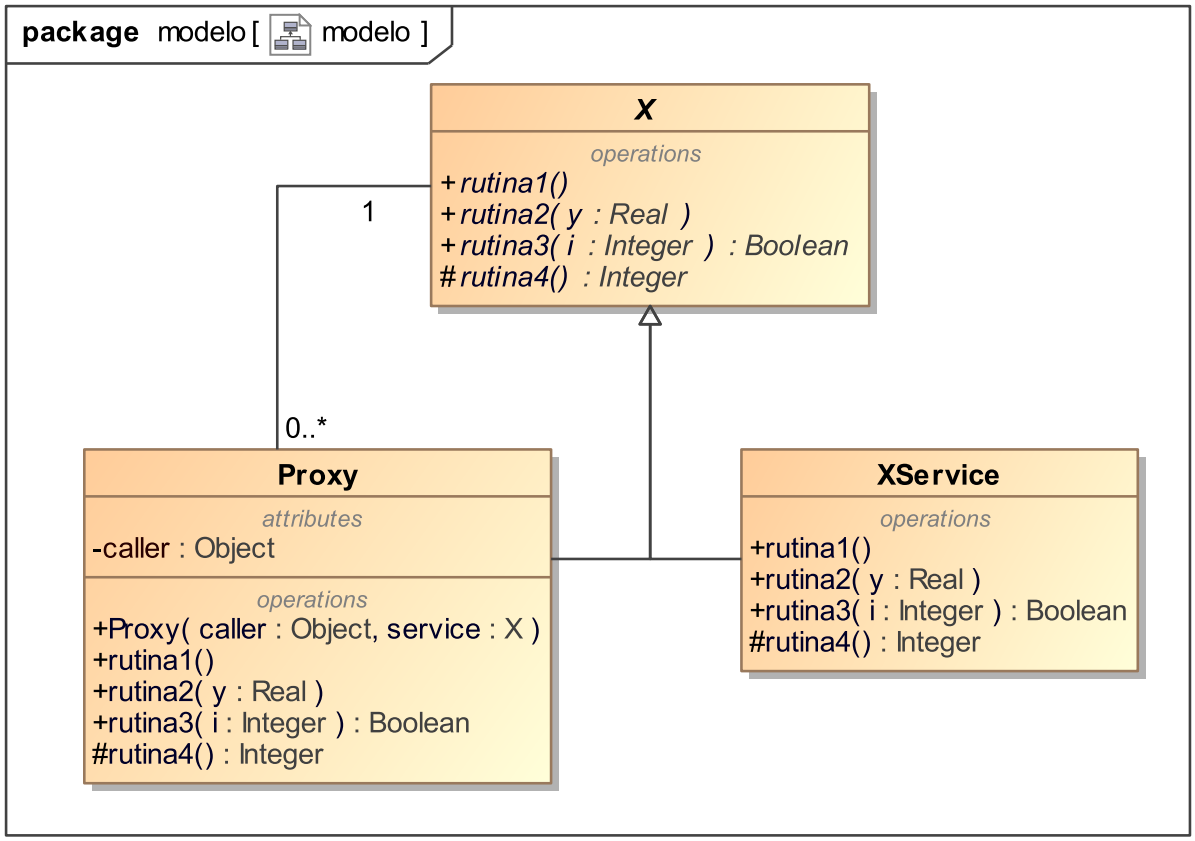

Figure 1: Modelo

Para poder emular el mecanismo de exportación selectiva de Eiffel, debemos controlar el acceso al servicio (la clase X) y para ello utilizamos el Patrón **Representante** o **Proxy**.

Se generan así dos clases que hereden de una clase abstracta `X`; `Proxy` (encargada de controlar el acceso) y `XService` (encargada de ofrecer el servicio). `Proxy` contará, además de los métodos de X, con un constructor que recibe como parámetro el objeto *caller*. La clase `Proxy` se encargará de verificar si la clase *caller* cuenta con los permisos suficientes para poder realizar dicha llamada, y en caso contrario, lanzará una excepción de tipo `IllegalCallerException`

Las clases `A`, `B` y `C` se encontrarán en otro paquete para proteger el acceso a los métodos de `XService`. El método `rutina1` será visible para todos, los métodos `rutina2` y `rutina3` serán visibles desde el exterior, pero su ejecución solo será satisfactoria en caso de que la clase *caller* cumpla los requisitos establecidos. Por último, el método `rutina4` será solo accesible desde las subclases

La ventaja que nos proporciona aplicar este Patrón es que las clases `A`, `B` y `C` no interactúan directamente con `XService`, relegando el control del acceso a la clase `Proxy` y dejando que `XService` únicamente se encargue de aplicar los métodos. La desventaja que podemos observar es que al no existir soporte en Java para el control granular de permisos, el acceso se comprueba en tiempo de ejecución en vez de en tiempo de compilación.

Ejercicio 2: Triestables

Ejercicio 3: Cliente de correo e-look

Código Java

Ej1

A.java

```
package Ej1;

public class A {}
```

B.java

```
package Ej1;

public class B {}
```

C.java

```
package Ej1;

public class C {}
```

Proxy.java

```
package Ej1.x;
```

```
import Ej1.A;
import Ej1.B;
import Ej1.C;

public class Proxy extends X {
  private Object caller;
  private X service;

  public Proxy(Object caller) {
    this.caller = caller;
    this.service = new XService();
  }

  @Override
  public void rutina1() {
    service.rutina1();
  }

  @Override
  public void rutina2(double y) {
    if (!(caller instanceof A) && !(caller instanceof B))
      throw new IllegalCallerException("Caller must be instance of class A or B");
    service.rutina2(y);
  }

  @Override
  public boolean rutina3(int i) {
    if (!(caller instanceof A) && !(caller instanceof C))
      throw new IllegalCallerException("Caller must be instance of class A or C");
    return service.rutina3(i);
  }

  @Override
  protected int rutina4() {
    return service.rutina4();
  }
}
```

X.java

```
package Ej1.x;

public abstract class X {
  public abstract void rutina1();

  public abstract void rutina2(double y);

  public abstract boolean rutina3(int i);

  protected abstract int rutina4();
}
```

XService.java

```
package Ej1.x;

class XService extends X {

  @Override
  public void rutina1() {
    // ...
  }

  @Override
  public void rutina2(double y) {
    // ...

  }

  @Override
  public boolean rutina3(int i) {
    // ...
    return false;
  }

  @Override
```

```
  protected int rutina4() {
    // ...
    return 0;
  }
}
```

Ej3

DateSortStrategy.java

```
package Ej3;

public class DateSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    return e1.getDate().compareTo(e2.getDate()) > 0;
  }
}
```

Email.java

```
package Ej3;

import java.util.Date;

public class Email {
  private String from;
  private String subject;
  private Date date;
  private Priority priority;
  private String text;

  public Email(String from, String subject, Date date, Priority priority, String text) {
    this.from = from;
    this.subject = subject;
    this.date = date;
    this.priority = priority;
```

```
    this.text = text;
  }

  public String getFrom() {
    return from;
  }

  public String getSubject() {
    return subject;
  }

  public Date getDate() {
    return date;
  }

  public Priority getPriority() {
    return priority;
  }

  public String getText() {
    return text;
  }

  public String toString() {
    return "From: "
        + from
        + "\n"
        + "Subject: "
        + subject
        + "\n"
        + "Date: "
        + date
        + "\n"
        + "Priority: "
        + priority
        + "\n"
        + "Text: "
        + text;
```

```
  }
}
```

FromSortStrategy.java

```
package Ej3;

public class FromSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    return e1.getFrom().compareTo(e2.getFrom()) > 0;
  }
}
```

Mailbox.java

```
package Ej3;

import java.util.ArrayList;
import java.util.List;

public class Mailbox {
  private SortStrategy strategy;
  private List<Email> email;

  public Mailbox(SortStrategy strategy) {
    email = new ArrayList<>();
    this.strategy = strategy;
  }

  public void addMail(Email e) {
    email.add(e);
  }

  public void show() {
    for (Email e : email) {
      System.out.println(e.toString() + "\n");
```

```
    }
  }

  public void setStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    for (int i = 2; i <= email.size(); i++) {
      for (int j = email.size(); j >= i; j--) {
        if (strategy.before(email.get(j), email.get(j - 1))) {
          Email temp = email.get(j);
          email.set(j - 1, temp);
          email.set(j, email.get(j - 1));
        }
      }
    }
  }
}
```

Main.java

```
package Ej3;

import java.util.Date;

public class Main {
  public static void main(String[] args) {
    SortStrategy e = new SubjectSortStrategy();
    Mailbox m = new Mailbox(e);

    Email e1 = new Email("av", "abc", new Date(), Priority.P1, "Un saludo, Antono");
    Email e2 = new Email("bav", "bcd", new Date(), Priority.P3, "Un saludo");
    Email e3 = new Email("inshallah", "zzz", new Date(), Priority.P2, "pls");

    m.addMail(e1);
    m.addMail(e2);
```

```
    m.addMail(e3);

    System.out.println("----------------------------SUBJECT SORT----------------------------

    m.show();

    System.out.println("------------------------------DATE SORT-----------------------------

    m.setStrategy(new DateSortStrategy());
    m.show();

    System.out.println("----------------------------PRIORITY SORT---------------------------

    m.setStrategy(new PrioritySortStrategy());
    m.show();

    System.out.println("-----------------------------FROM SORT------------------------------"

    m.setStrategy(new FromSortStrategy());
    m.show();
  }
}
```

Priority.java

```
package Ej3;

public enum Priority {
  P1,
  P2,
  P3
}
```

PrioritySortStrategy.java

```
package Ej3;
```

```
public class PrioritySortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    return e1.getPriority().compareTo(e2.getPriority()) > 0;
  }
}
```

SortStrategy.java

```
package Ej3;

public interface SortStrategy {
  boolean before(Email e1, Email e2);
}
```

SubjectSortStrategy.java

```
package Ej3;

public class SubjectSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    return e1.getSubject().compareTo(e2.getSubject()) > 0;
  }
}
```