PROGRAMACIÓN CON ASPECTOS

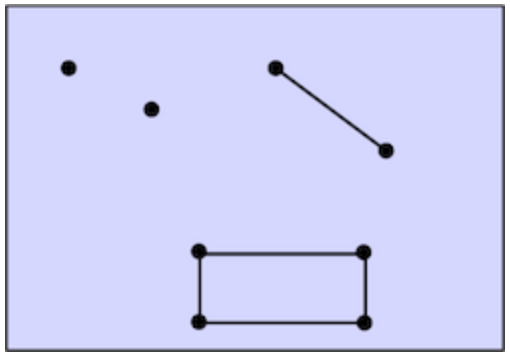
Ortogonalidad con aspectos

CASO PRÁCTICO: Editor de figuras

Ejemplo: editor de figuras

```
class Line implements FigureElement {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) { this.p1 = p1; }
  void setP2(Point p2) { this.p2 = p2; }
}

class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
}
```

Hay que actualizar la pantalla tras mover los objetos:



Hay una colección de figuras que cambian periódicamente.

Se deben monitorizar los cambios para refrescar el display.

Implementamos una clase que monitoriza los cambios en las figuras:

```
class MoveTracking {
  private static boolean flag = false;

  public static void setFlag() {
    flag = true;
  }

  public static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }
}
```

¿Qué dependencias aparecen?

¿Qué dependencias aparecen?

- `Line` ⟶ `MoveTracking`
- `Point` ⟶ `MoveTracking`

Implementaciones sin aspectos

Primero vemos algunas implementaciones con las dependencias anteriores, intentando resolver la no ortogonalizad, pero sin usar aspectos.

- Versión 1: solo detecta el cambio de los extremos de una línea
- Versión 2: también detecta el cambio de coordenadas de un punto
- Versión 3: monitoriza las figuras que cambian, evitando el refresco de todas

Versión 1 sin aspectos

Solo detecta el cambio de los extremos de una línea: `Line` --> `MoveTracking`

```
class Line implements FigureElement {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    MoveTracking.setFlag();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    MoveTracking.setFlag();
  }
}
```

```
class Point implements FigureElement {
  private int x = 0, y= 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Versión 2 sin aspectos

También detecta el cambio de coordenadas de un punto

- `Line` ⟶ `MoveTracking`
- `Point` ⟶ `MoveTracking`

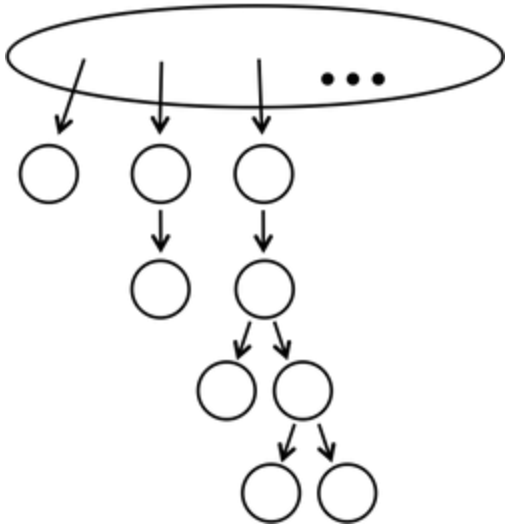
```
class MoveTracking {
  private static boolean flag = false;

  public static void setFlag() {
    flag = true;
  }

  public static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }
}
```

```
class Line implements FigureElement {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    MoveTracking.setFlag();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    MoveTracking.setFlag();
  }
}
```

```
class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    MoveTracking.setFlag(); //añadido
  }
  void setY(int y) {
    this.y = y;
    MoveTracking.setFlag(); //añadido
  }
}
```

Versión 3 sin aspectos

Las colecciones de figuras son complejas. Las estructuras de objetos son jerárquicas y se producen eventos asíncronos:



Versión 2: un cambio en cualquier elemento provocará un refresco de todas las figuras

Mejor monitorizar las figuras que cambian...

Modificamos la versión 2 para cambiar el método `setFlag` por `collectOne`

```
class MoveTracking {
  private static Set movees =
                          new HashSet();

  public static void collectOne(Object o) {
    movees.add(o);
  }

  public static Set getmovees() {
    Set result = movees;
    movees = new HashSet();
    return result;
  }
}
```

Indicamos la figura que se mueve:

```
class Line implements FigureElement {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    MoveTracking.collectOne(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    MoveTracking.collectOne(this);
  }
}
```

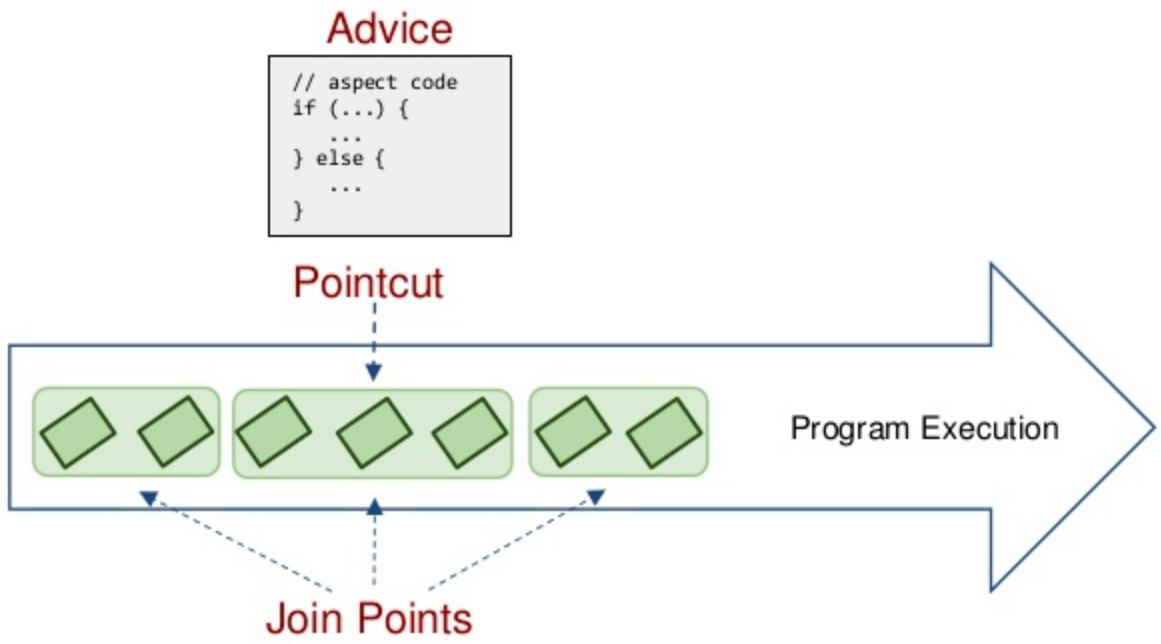
```
class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    MoveTracking.collectOne(this);
  }
  void setY(int y) {
    this.y = y;
    MoveTracking.collectOne(this);
  }
}
```

La no ortogonalidad de `MoveTracking` con respecto a `Line` y `Point` hace que la solicitud de un cambio de implementación (el seguimiento de los cambios en las figuras para el refresco en pantalla) provoque un cambio en los otros módulos (clases).

El cambio de implementación del seguimiento de los cambios para el refresco en pantalla ha dado lugar a modificaciones en todas las clases: `Line`, `Point` y `MoveTracking`

Programación orientada a aspectos

La **programación orientada a aspectos** (*AOP*) es un paradigma de programación cuyo objetivo es incrementar la modularidad (ortogonalidad) de las implementaciones mediante la separación de aspectos *transversales* (*cross-cutting concerns*).



- **aspect** = modularización de un aspecto de interés (*concern*) que afecta a varias clases o módulos
- **joinpoint** = especificación declarativa de un punto en la ejecución de un programa (por ejemplo, la ejecución de un método, el manejo de una excepción, etc.)
- **advice** = acción a tomar por la especificación de un aspecto dado en un determinado *joinpoint*.
  - Interceptan la ejecución de un *joinpoint*. Hay una cadena de interceptores alrededor de cada *joinpoint*.
  - Tipos de *advice*: *after*, *before*, *around*, etc.
- **pointcut** = predicado que define cuándo se aplica un *advice* de un aspecto en un *jointpoint* determinado. Se asocia un *advice* con la expresión de un *pointcut* y se ejecuta el *advice* en todos los *joinpoint* que cumplan la expresión del *pointcut*.

Implementación con aspectos

En el ejemplo anterior, las clases `Line` y `Point` no se ven afectadas:

```
class Line implements FigureElement {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}
```

```
class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Vamos a eliminar las dependencias (Δ ortogonalidad) implementando aspectos...

Versión 1 con aspectos

Eliminar dependencia `Line` ⇢ `MoveTracking`

```
aspect MoveTracking {
  private boolean flag = false;
  public boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }

  pointcut move():
    call(void Line.setP1(Point)) || call(void Line.setP2(Point));

  after(): move() {
    flag = true;
  }
}
```

Versión 2 con aspectos

Eliminar dependencias `Line` -/-> `MoveTracking` y `Point` -/-> `MoveTracking`

```
aspect MoveTracking {
  private boolean flag = false;
  public boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }

  pointcut move():
    call(void Line.setP1(Point)) || call(void Line.setP2(Point)) ||
    call(void Point.setX(int))   || call(void Point.setY(int));

  after(): move() {
    flag = true;
  }
}
```

Ejemplos de pointcut:

```
call(void Figure.set*(..))

call(public * Figure.* (..))
```

Versión 3 con aspectos

- `Line` ⊥ `MoveTracking`
- `Point` ⊥ `MoveTracking`

Versión más ortogonal. Todos los cambios están concentrados en un solo aspecto.

```
aspect MoveTracking {
  private Set movees = new HashSet();
  public Set getmovees() {
    Set result = movees;
    movees = new HashSet();
    return result;
  }

  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void Line.setP1(Point)) || call(void Line.setP2(Point)) ||
     call(void Point.setX(int))   || call(void Point.setY(int)));

  after(FigureElement fe): move(fe) {
    movees.add(fe);
  }
}
```

Lecturas recomendadas de AspectJ

- Lenguaje de AspectJ
- Introducción a AspectJ

Ejercicios: AspectJ y Spring AOP

- Introducción a AspectJ
- Introducción a Spring AOP