Patrones de Diseño de Comportamiento

Fundamentos de Ingeniería de Software/Análisis y Diseño de Software

Pablo Cruz Navea-Gastón Márquez
Departamento de Informática
Universidad Técnica Federico Santa María

Strategy

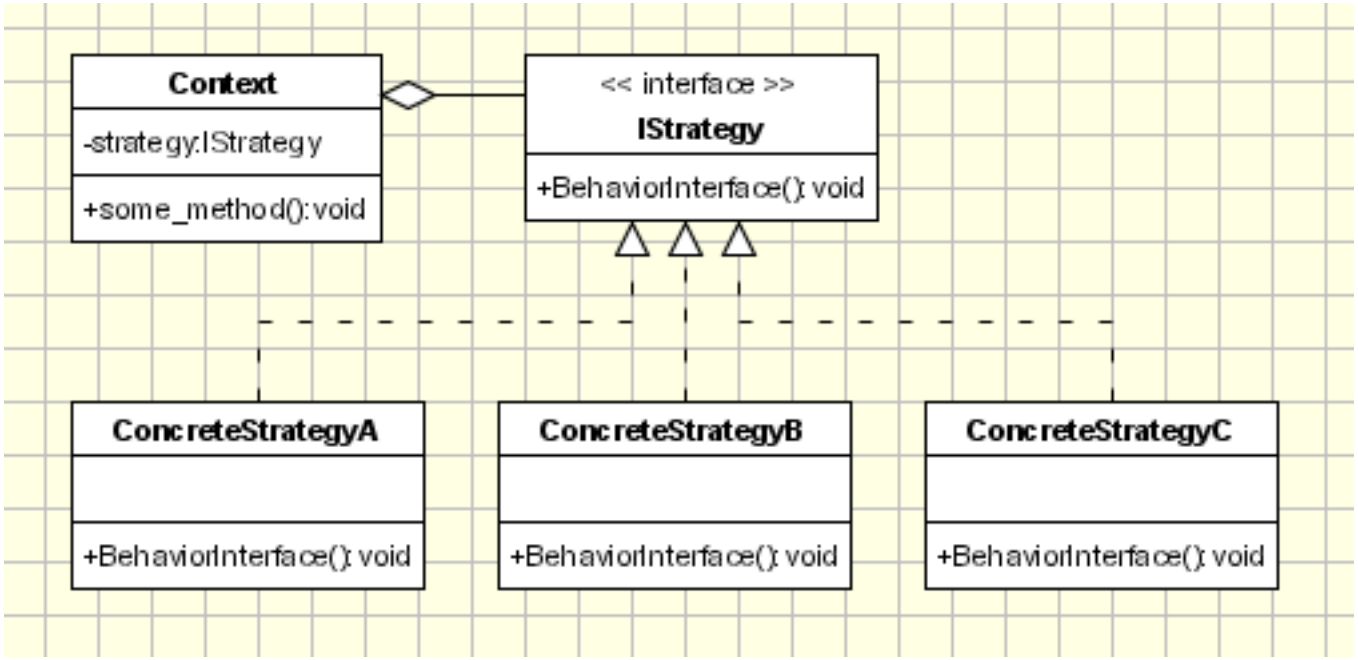
Strategy [1]

- Patrón de diseño **de comportamiento**
- Propósito: cuando existe una familia de algoritmos, encapsulamos cada uno de ellos para hacerlos intercambiables
- Con **Strategy** los algoritmos pueden variar independiente de los clientes que hacen uso de ellos
- **Strategy** aplica de manera natural a los casos en los que debe seleccionarse uno de los algoritmos según el contexto

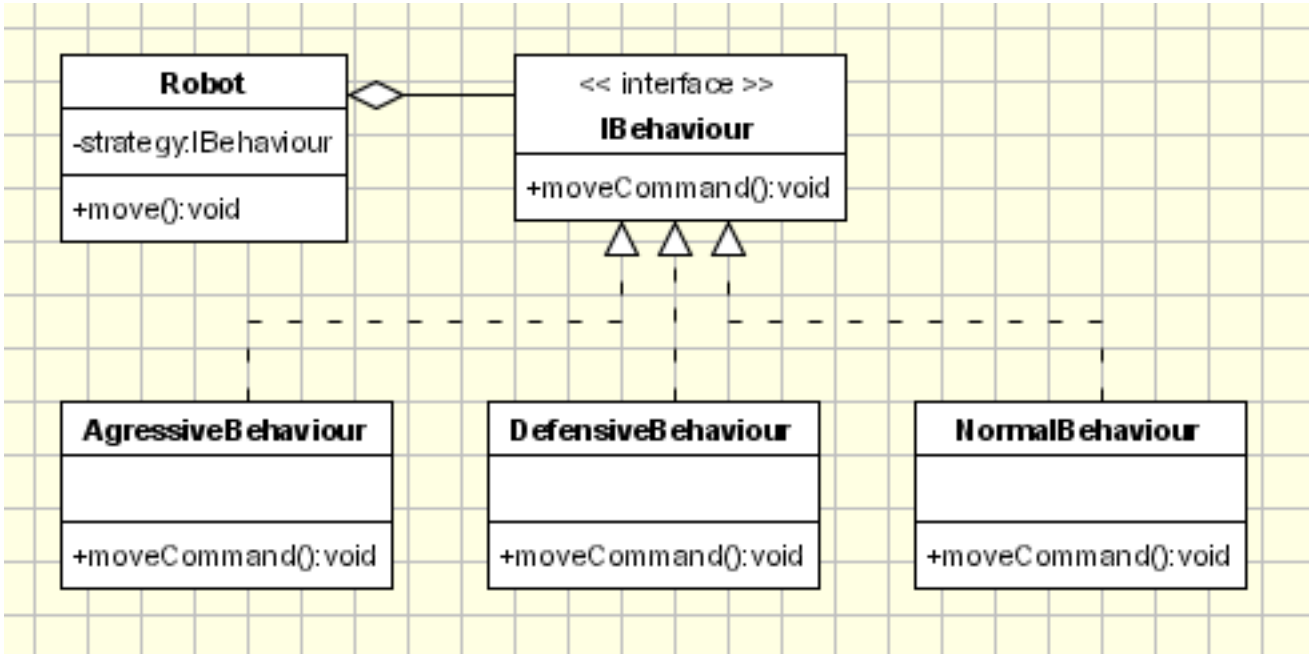
Strategy [2]

- “Seleccionar un algoritmo” podría ser resuelto fácilmente con una serie de estructuras *if-then-else*
- ¿Por qué usar un patrón de diseño para resolver este problema de selección de algoritmos?
  - Porque así los algoritmos quedan encapsulados en clases distintas y pueden variar libremente
    - Cada desarrollador puede encargarse de una clase “estrategia” sin entorpecer el trabajo de los demás desarrolladores

Strategy [3]


Context
-strategy:IStrategy
+some_method():void
<< interface >>
IStrategy
+BehaviorInterface(): void
ConcreteStrategyA
+BehaviorInterface(): void
ConcreteStrategyB
+BehaviorInterface(): void
ConcreteStrategyC
+BehaviorInterface(): void

Strategy [4]


Robot
-strategy:IBehaviour
+move():void
<< interface >>
IBehaviour
+moveCommand():void
AgressiveBehaviour
+moveCommand():void
DefensiveBehaviour
+moveCommand():void
NormalBehaviour
+moveCommand():void

Strategy [4]

```
public interface IBehaviour {
     public int moveCommand();
}

public class AgressiveBehaviour implements
IBehaviour{
     public int moveCommand()
     {
          System.out.println("\tAgressive
Behaviour: if find another robot attack it");
          return 1;
     }
}
```

Strategy [4]

```
public class DefensiveBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tDefensive Behaviour: if find
another robot run from it");
		return -1;
	}
}

public class NormalBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tNormal Behaviour: if find
another robot ignore it");
		return 0;
	}
}
```

Strategy [4]

```
public class Robot {
	IBehaviour behaviour;
	String name;

	public Robot(String name)
	{
		this.name = name;
	}

	public void setBehaviour(IBehaviour behaviour)
	{
		this.behaviour = behaviour;
	}

	public IBehaviour getBehaviour()
	{
		return behaviour;
	}
```

Strategy [4]

```
        public void move()
        {
                System.out.println(this.name + ": Based on current position" +
                                                "the behaviour object decide the
next move:");
                int command = behaviour.moveCommand();
                // ... send the command to mechanisms
                System.out.println("\tThe result returned by behaviour object " +
                                                "is sent to the movement mechanisms " +
                                                " for the robot '"  + this.name + "'");
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

Strategy [4]

```
public class Main {

      public static void main(String[] args) {

            Robot r1 = new Robot("Big Robot");
            Robot r2 = new Robot("George v.2.1");
            Robot r3 = new Robot("R2");

            r1.setBehaviour(newAgressiveBehaviour());
            r2.setBehaviour(newDefensiveBehaviour());
            r3.setBehaviour(new NormalBehaviour());

            r1.move();
            r2.move();
            r3.move();
```

Strategy [4]

```
System.out.println("\r\nNew behaviours: " +
                                   "\r\n\t'Big Robot' gets really scared" +
                                   "\r\n\t, 'George v.2.1' becomes really
mad because" +
                                   "it's always attacked by other robots" +
                                   "\r\n\t and R2 keeps its calm\r\n");

                r1.setBehaviour(new DefensiveBehaviour());
                r2.setBehaviour(new AgressiveBehaviour());

                r1.move();
                r2.move();
                r3.move();
        }
}
```

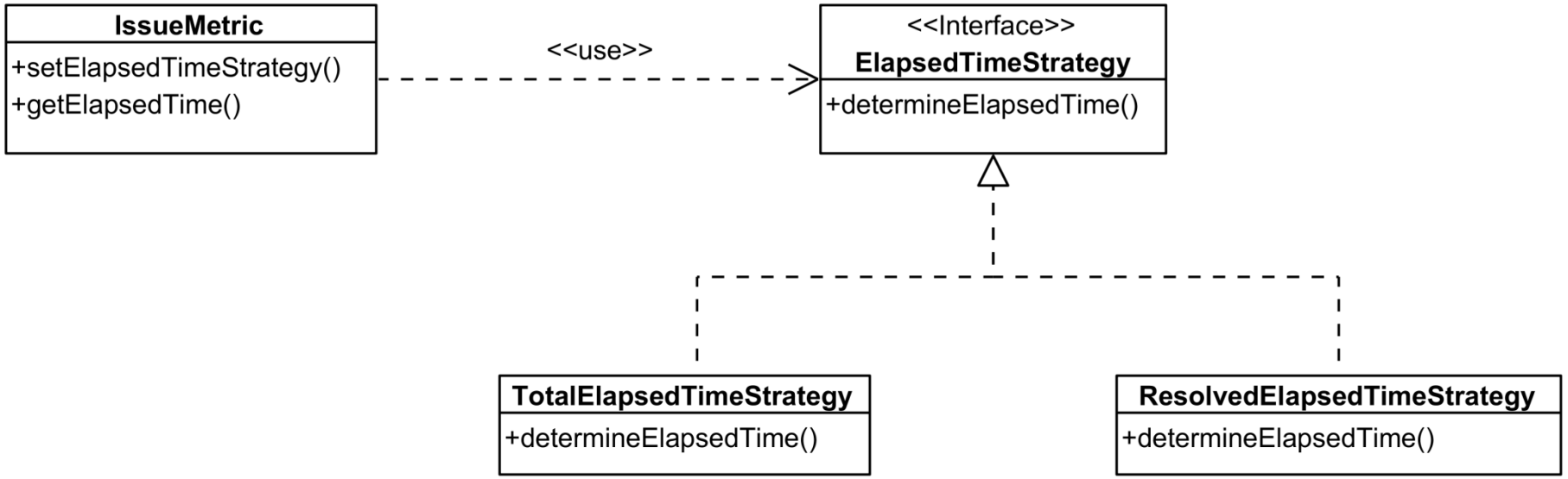
Ejemplo: Strategy con JAVA [1]

- En muchos ejemplos hemos visto el concepto de tiempo transcurrido *(elapsed time)* para los *issues* (incidencias)
- En la realidad, medir el tiempo transcurrido de un *issue* es un asunto un poco más complejo porque existen muchos tipos de *"elapsed time"*
  - ElapsedTime completo
    - Desde que el *issue* se creó hasta que se cerró
  - ElapsedTime de resolución
    - Desde que el *issue* se creó hasta que fue marcado como resuelto (sin probar)
  - ElapsedTime de asignación
    - Desde que el *issue* se creó hasta que fue asignado hacia un equipo de desarrollo

Ejemplo: Strategy con JAVA [2]

- ¿Cómo nos ayuda **Strategy** en este problema?
  - Encapsulando los distintos algoritmos en clases distintas
- Para el ejemplo, supondremos que un sistema debe medir el **tiempo transcurrido total** y el **tiempo transcurrido para la última resolución**
  - Tendremos dos clases que encapsulan los distintos algoritmos:
    - TotalElapsedTimeStrategy
    - ResolvedElapsedTimeStrategy
  - Ambas clases implementan una interfaz que provee el método determineElapsedTime()

Ejemplo: Strategy con JAVA [3]



Ejemplo: Strategy con JAVA [4]

```
// creamos una estrategia general para encontrar el tiempo
// transcurrido
public interface ElapsedTimeStrategy {
        public void determineElapsedTime(Issue issue);
}

// caso sencillo: estrategia 'tiempo total' es lo que hemos
// visto siempre en los ejemplos
public class TotalElapsedTimeStrategy implements
        ElapsedTimeStrategy {
        public void determineElapsedTime(Issue issue) {
                // obtener openDate desde issue
                // obtener closeDate desde issue
                // obtener closeDate - openDate
                return elapsedTime;
        }
}
```

Ejemplo: Strategy con JAVA [5]

```
// caso complejo: estrategia 'tiempo de resolución'
public class ResolvedElapsedTimeStrategy implements
        ElapsedTimeStrategy {
        public void determineElapsedTime(Issue issue) {
                // obtener openDate desde issue
                // obtener de alguna forma la última vez
                // que el issue fue marcado como status = resolved
                // obtener resolvedDate - openDate
                return resolvedElapsedTime;
        }
}
```

Ejemplo: Strategy con JAVA [6]

```
public class IssueMetric {
        private ElapsedTimeStrategy elapsedTimeStrategy;
        private Issue issue;
        // constructor puede ser usado para establecer
        // estrategia por defecto (si aplica) y para
        // recibir un 'issue'

        public void setElapsedTimeStrategy(ElapsedTimeStrategy
                                elapsedTimeStrategy) {
                this.elapsedTimeStrategy = elapsedTimeStrategy;
        }

        public int getElapsedTime() {
                return elapsedTimeStrategy.
                        determineElapsedTime(issue);
        }
}
```

Ejemplo: Strategy con JAVA [7]

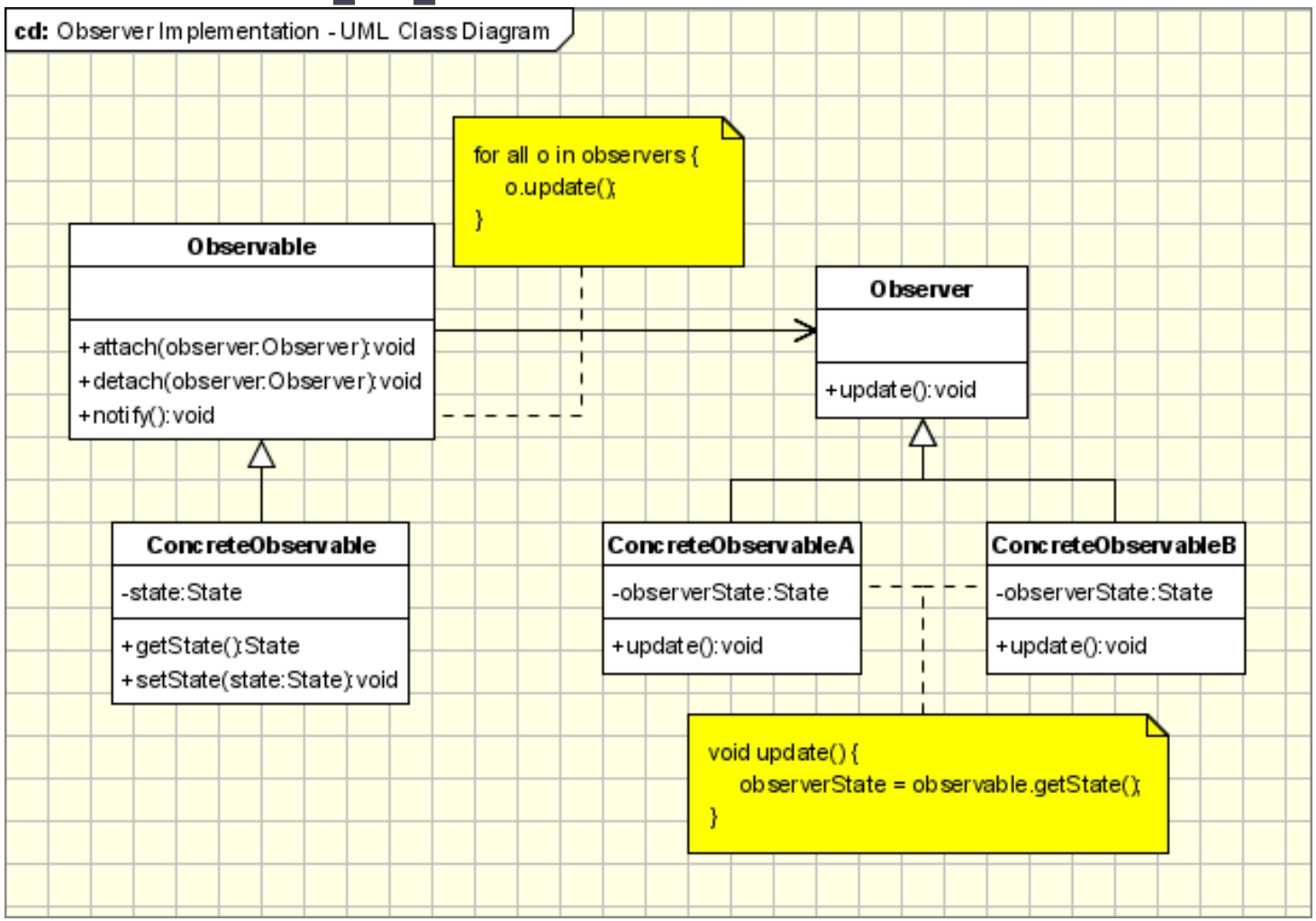
```
// hacemos uso de las estrategias
IssueMetric issueMetric = new IssueMetric();

issueMetric.setElapedTimeStrategy(new
                                        TotalElapsedTimeStrategy());
issueMetric.getElapsedTime();

// de manera análoga, usamos estrategia para tiempo transcurrido
// de resolución
```

Observer

Observer [1]

- Patrón de diseño **de comportamiento**
- Propósito: una dependencia uno-a-muchos entre objetos permite notificar a los objetos dependientes (muchos) cuando el objeto del que dependen (uno) haya cambiado su estado
- Existen dos tipos de clases importantes:
  - Observer (observador): clases que necesitan mantenerse informadas de lo que ocurra en alguna clase de interés (clase observada)
  - Observable: la clase que es observada (por las clases observadoras)

Observer [2]


cd: Observer Implementation - UML Class Diagram
for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}

Observer [3]

- Observable: es una clase abstracta o interface que define las operaciones attach o detach observadores del cliente (esta clase también es conocida como Subject).
- ConcreteObservable: es una clase que mantiene el estado del objeto y cuando cambia el estado, se le notifica a Observers
- Observer: interface o clase abstracta que define operaciones para utilizadas en ConcreteObservable
- ConcreteObservable: clase que implementa las operaciones definidas en Observer

Observer [4]

- Mirar el código del profesor

Ejemplo: Observer con JAVA [1]

- El patrón **Observer** es tan común (e importante) que muchos lenguajes ofrecen estructuras y clases apropiadas para implementarlo
- JAVA nos ofrece para la implementación:
  - java.util.Observable
    - Clase que define métodos apropiados para la clase que será observada
  - java.util.Observer
    - Interfaz que define el método update() para ser implementado por las clases observadoras

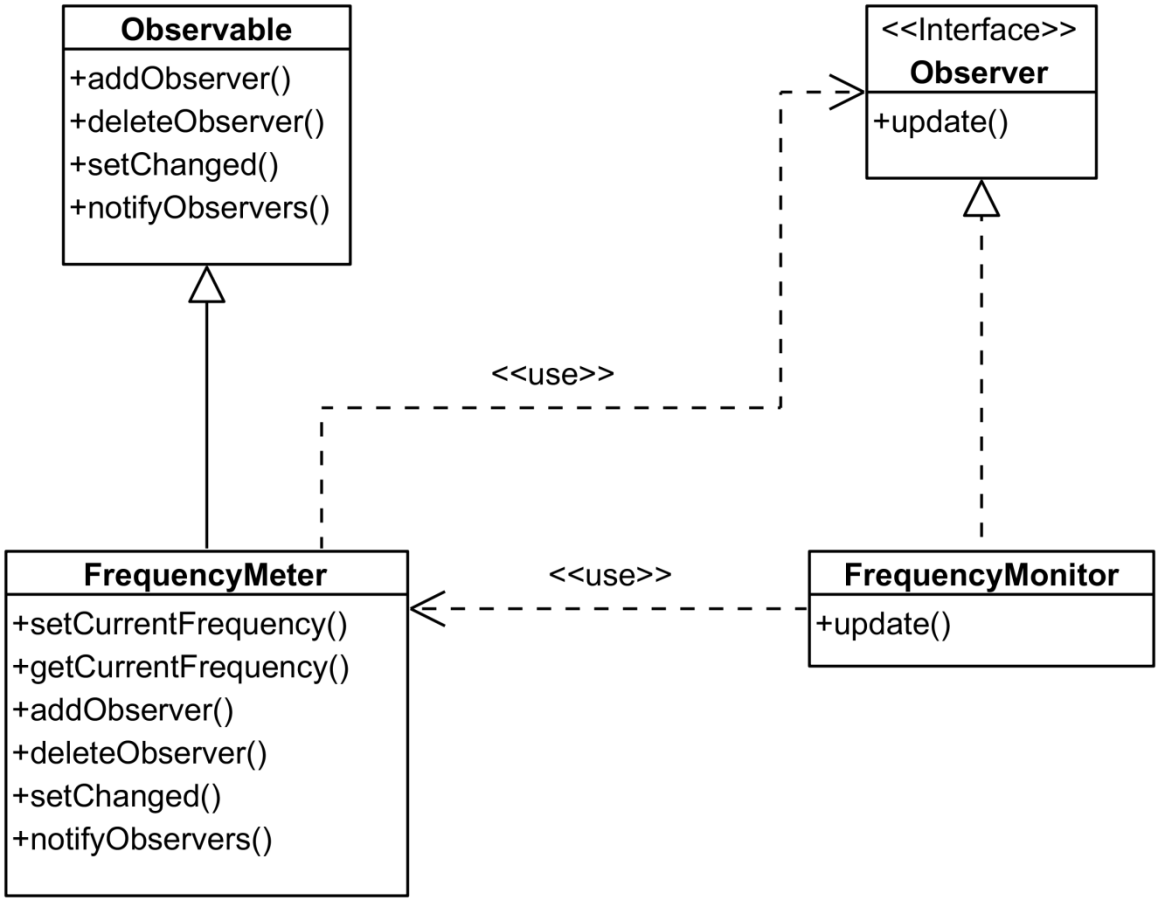
Ejemplo: Observer con JAVA [2]

- En el ejemplo daremos solución al problema que ocurre cuando se establecen frecuencias demasiado altas a un oscilador
- Recordemos que un oscilador puede tomar (teóricamente) frecuencias desde 20 [Hz] hasta 20.000 [Hz]
  - Pero en la realidad, setear frecuencias más allá de los 16.000 [Hz] es impráctico (y hasta peligroso)

Ejemplo: Observer con JAVA [3]

- Una clase FrequencyMeter será la clase observada
- Una clase FrequencyMonitor será la clase que observa al medidor
- Cuando la clase medidor note que se estableció una frecuencia mayor a 16.000 [Hz] notificará a la clase monitor (observadora)
- Si esta situación ocurre, la clase monitor detendrá inmediatamente el motor de síntesis

Ejemplo: Observer con JAVA [4]



Ejemplo: Observer con JAVA [5]

```
// el medidor de frecuencia hereda los métodos de la clase
// Observable que ofrece JAVA
public class FrequencyMeter extends Observable {
	private float currentFrequency;

	public void setCurrentFrequency(float curFreq) {
		currentFrequency = curFreq;
		// indicamos que la frecuencia cambió
		// y notificamos a los observadores
		setChanged();
		notifyObservers();
	}
	public float getCurrentFrequency() {
		return currentSpeed;
	}
}
```

Ejemplo: Observer con JAVA [6]

```
// la clase monitor de frecuencia se encarga de tomar las
// acciones necesarias cuando la clase observable le notifique
public class FrequencyMonitor implements Observer {
	public static final float MAX_FREQ = 16000;

	public void update(Observable fMeter, Object obj) {
		FrequencyMeter fMeter = (FrequencyMeter) fMeter;

		if (fMeter.getCurrentFrequency() > MAX_FREQ) {
			// alertar y detener motor de síntesis
		}
	}
}
```

Ejemplo: Observer con JAVA [7]

```
// hacemos uso del patrón
FrequencyMonitor fMonitor = new FrequencyMonitor();
FrequencyMeter fMeter = new FrequencyMeter();

// agregamos un observador a la clase medidor
fMeter.addObserver(fMonitor);

// intentamos setear una frecuencia más allá de lo permitido
fMeter.setCurrentFrequency(20000);

// el observador toma nota de esta frecuencia (por notificación)
// y detiene el motor de síntesis
// además, alerta con un mensaje de error
```

Observaciones finales

- Con el patrón de diseño **Observer** terminamos nuestro recorrido en esta gran unidad dedicada a los patrones de diseño
- Antes de finalizar la unidad es bueno detenerse a pensar qué es lo que hemos hecho todo este tiempo
- Discutiremos algunas buenas prácticas utilizadas…

Buenas prácticas [1]

- Hemos preferido usar **interfaces** por sobre **implementaciones** directas
  - Todos los patrones que hemos visto hacen uso intensivo de interfaces
  - Las interfaces pueden ser evitadas y podemos trabajar directamente con implementaciones
    - Pero el uso de interfaces nos da flexibilidad y formalidad en la interacción de los objetos

Buenas prácticas [2]

- La **herencia** ha sido usada como un recurso conceptual
  - Hemos evitado hacer uso de la herencia como una forma de ahorrar escritura de métodos
- Hemos mantenido un **bajo acoplamiento** entre los objetos/clases
  - De esta forma, la variación de un objeto tiene poco efecto sobre los demás
- Hemos mantenido una **alta cohesión** entre los objetos/clases
  - Esto es, cada clase/objeto tiene responsabilidad claramente definida en los modelos de clases

Cuándo usar patrones

- La respuesta por defecto es siempre!
- Cada vez que nos encontremos con problemas recurrentes debemos recurrir a los patrones para diseñar software
  - Esto nos convertirá en verdaderos Ingenieros de Software
  - Y facilitará las tareas de programación
- Sin embargo, toda regla tiene su excepción…
  - Hay que considerar el *overhead* que viene asociado con el diseño y el uso de patrones

FIN