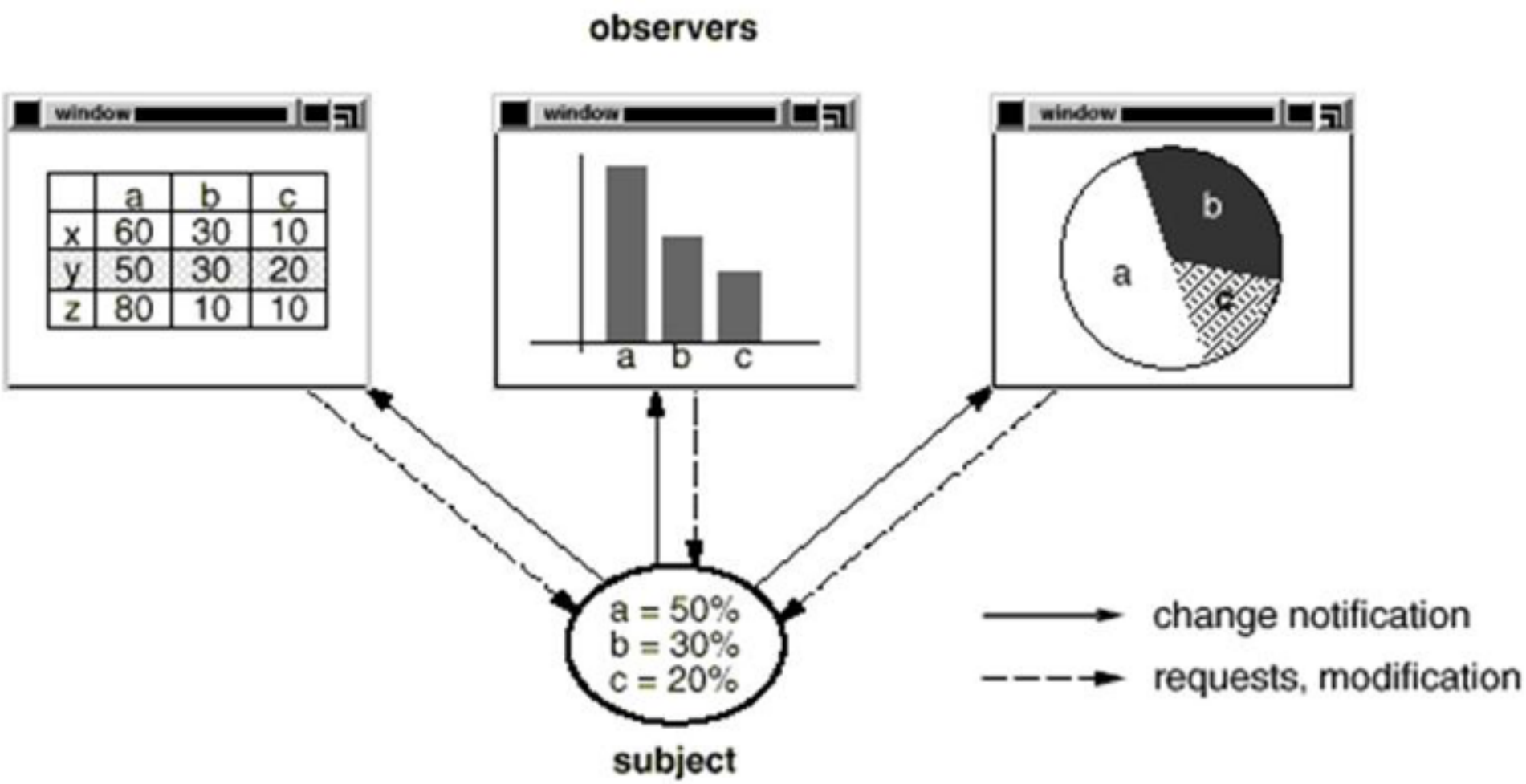
Patrones de Diseño: Observer

Análisis & Diseño de Software/Fundamentos de Ingeniería de Software

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

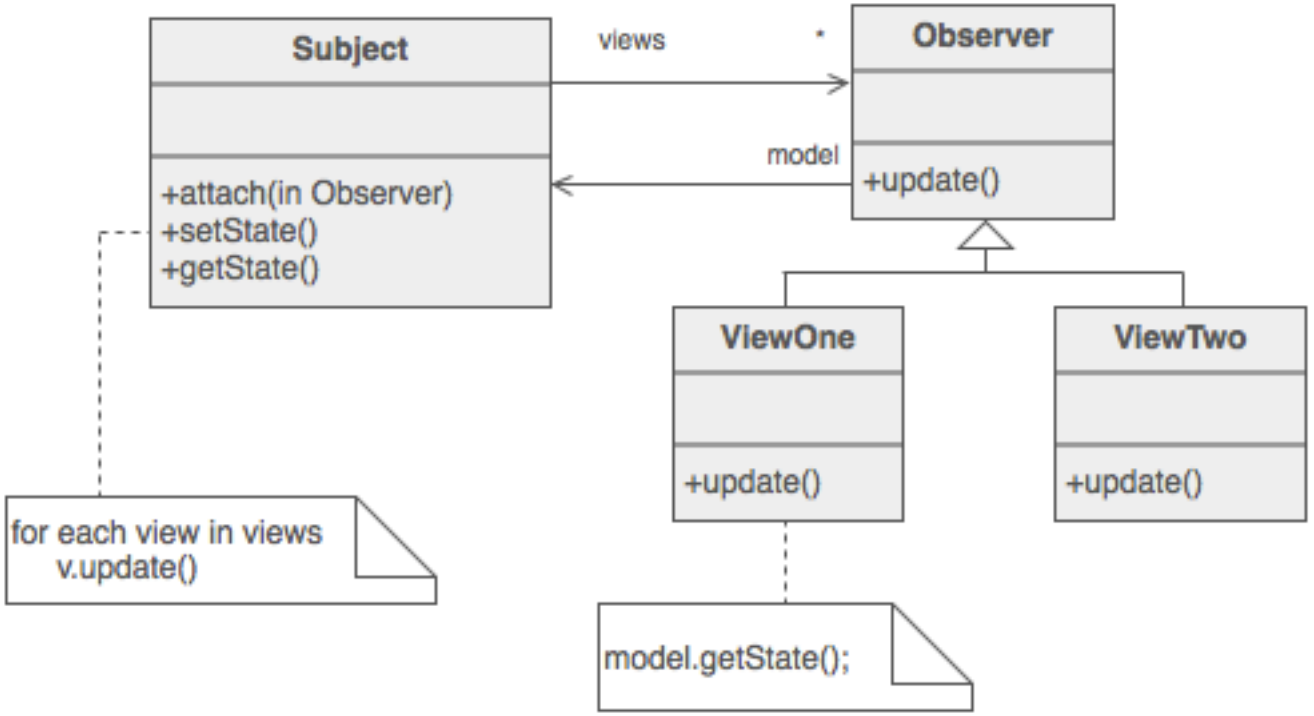

observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

Observer Design Pattern Real Time Example

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]


Subject
views
*
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

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 (https://docs.oracle.com/javase/7/docs/api/java/util/Observable.html)
    - Clase que define métodos apropiados para la clase que será observada
  - java.util.Observer (https://docs.oracle.com/javase/7/docs/api/java/util/Observer.html)
    - Interfaz que define el método update() para ser implementado por las clases observadoras

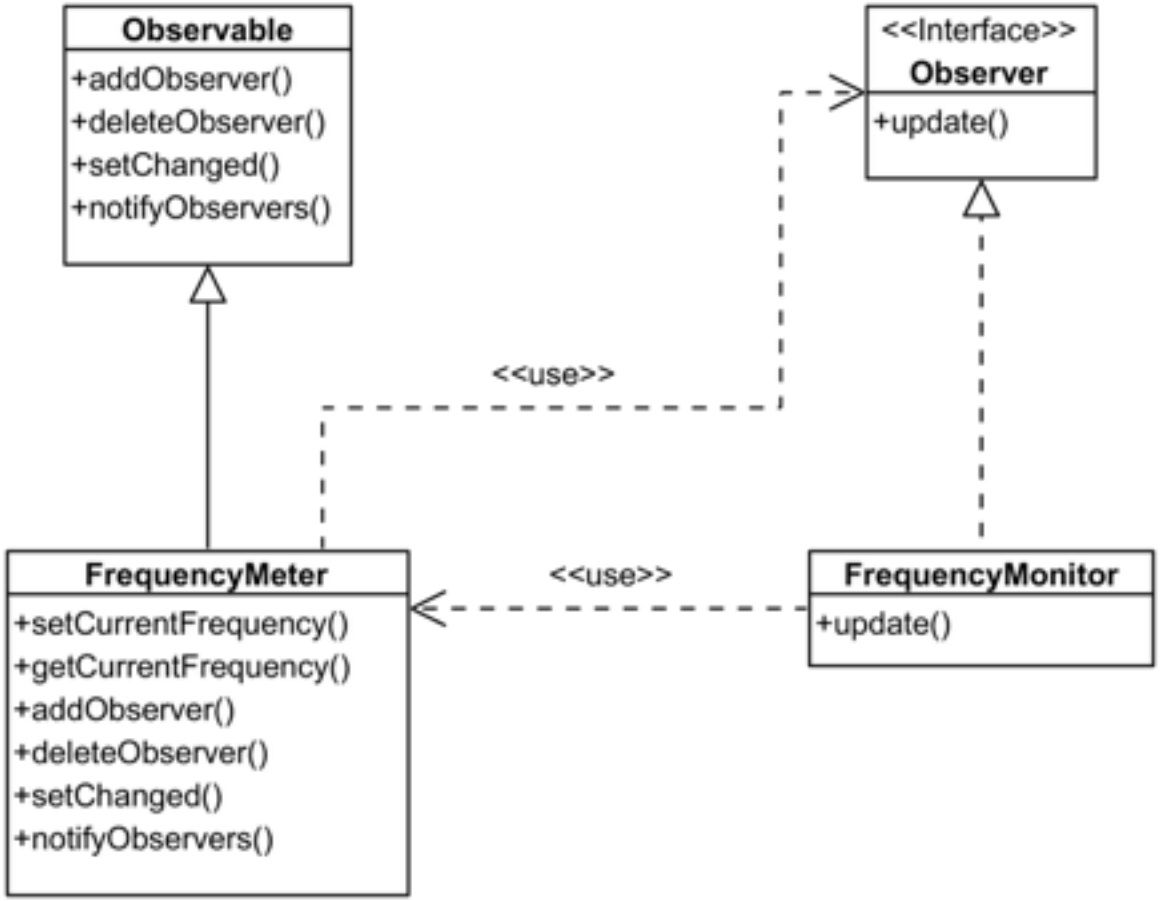
Ejemplo: Observer con JAVA [2]

- 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
		}
	}
}
```

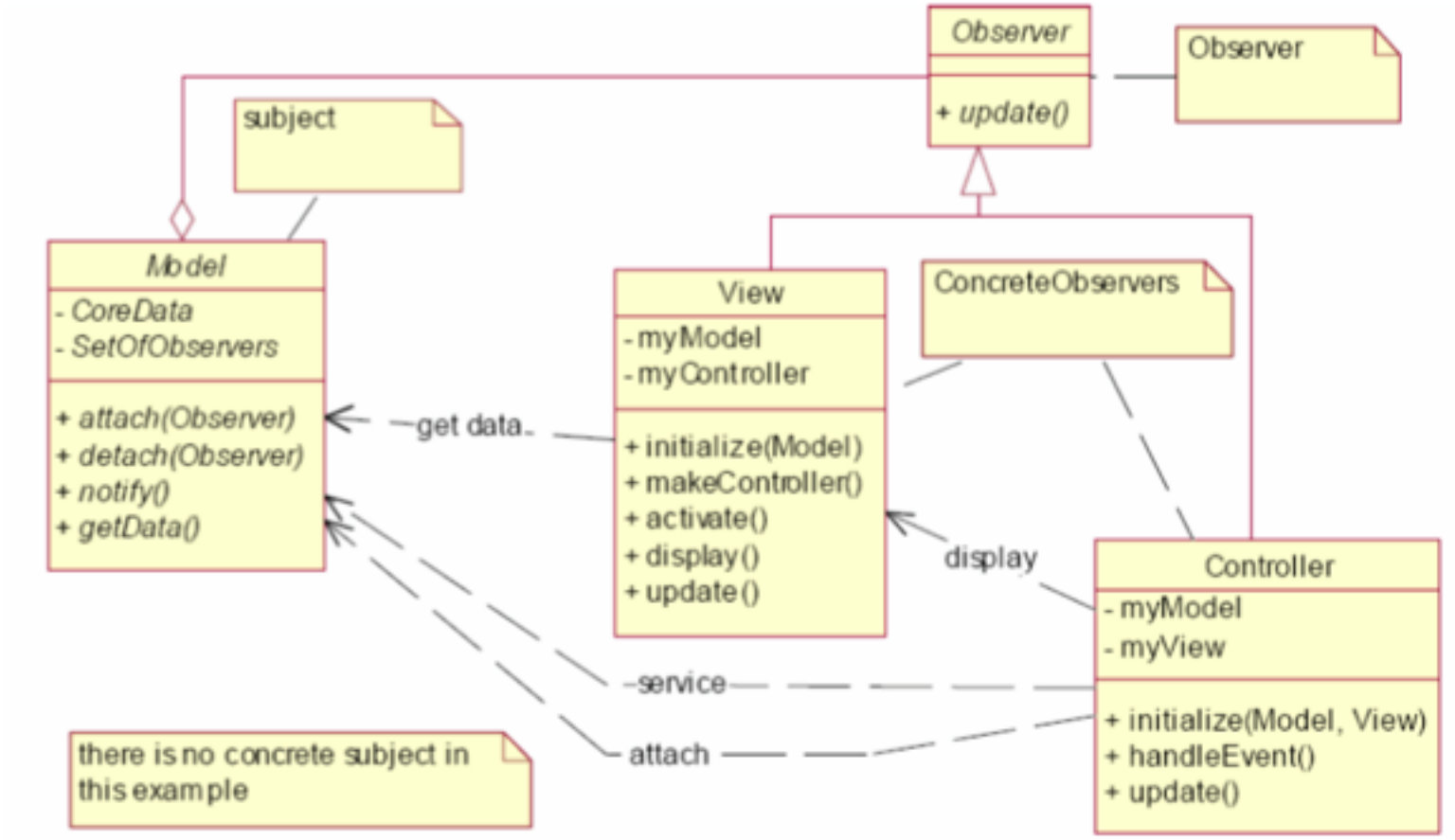
Ejemplo: Observer con JAVA [7]

```
// 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
```

Desafío [1]

- Proponga una variante del Modelo MVC para que pueda incluir el patrón Observer.
- El nuevo modelo debe cumplir las propiedades de ambos modelos previos.

Desafío [2]


Observer
+ update()
Observer
subject
Model
- CoreData
- SetOfObservers
+ attach(Observer)
+ detach(Observer)
+ notify()
+ getData()
get data
View
- myModel
- myController
+ initialize(Model)
+ makeController()
+ activate()
+ display()
+ update()
ConcreteObservers
display
Controller
- myModel
- myView
+ initialize(Model, View)
+ handleEvent()
+ update()
service
attach
there is no concrete subject in this example

Desafío [3]

- Vea el siguiente código

```
public interface Observer {
    void update(Observable o, Object arg);
}
```

- ¿Por qué el método update en la interface Observer incluye la referencia a un objeto que está observando? ¿No es que acaso Observer sabe cuál es el objeto que está observando?

Desafío [4]

- Vea el siguiente código

```
public interface Observer {
    void update(Observable o, Object arg);
}
```

- Observer puede que esté observando más de un objeto. El parámetro Observable identifica el objeto que está siendo actualizado

Desafío [5]

- Sea un grupo de clases que permiten la creación y envío de mensajes de email y que entre otras incluye clases que representan el cuerpo del mensaje, los anexos, la cabecera, el mensaje, la firma y una clase encargada de enviar el mensaje. El código cliente debe interactuar con instancias de todas estas clases para el manejo de los mensajes, por lo que debe conocer en qué orden se crean esas instancias; cómo colaboran esas instancias para obtener la funcionalidad deseada y cuáles son las relaciones entre las clases.
- Proponga un (unos) patrón(es) de diseño con el objetivo de:
  - Reducir las dependencias del código entre las clases descritas
  - Reducir la complejidad del código
- Finalmente, dibuje el diagrama de clases de su solución e indique el(los) patrón(es) seleccionado(s)

FIN