Patrones de Diseño Estructurales

Fundamentos de Ingeniería de Software/Análisis y Diseño de Software

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

Facade

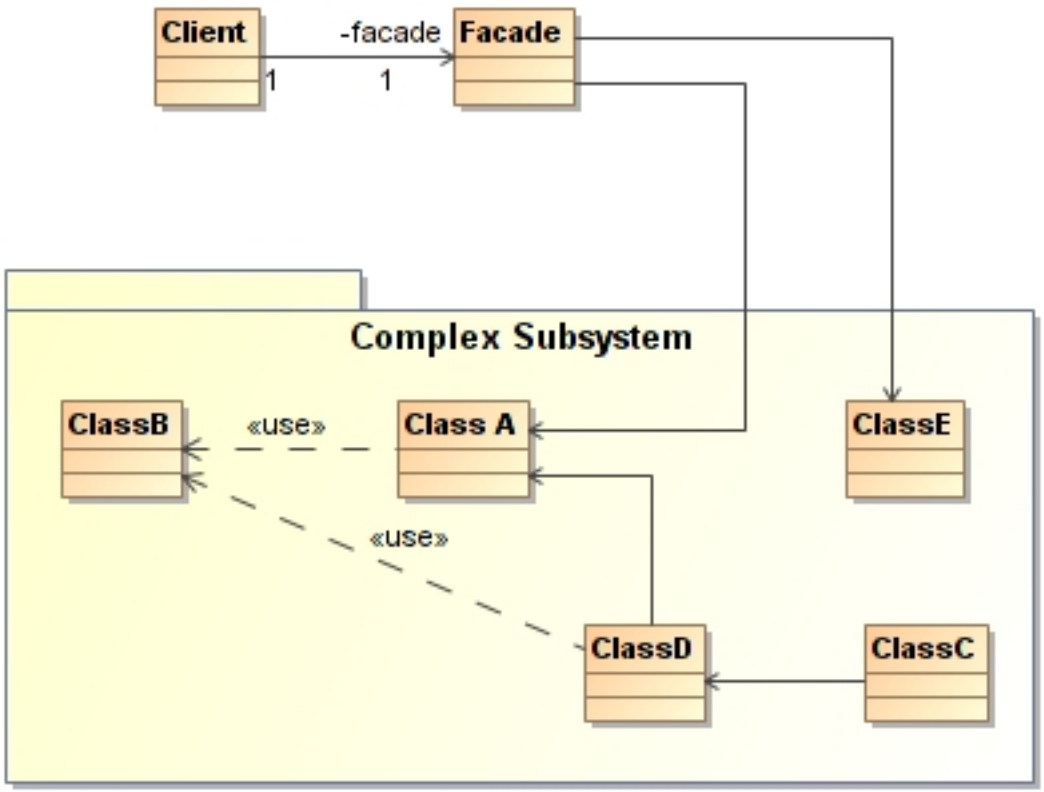
Facade [1]

- Patrón de diseño **estructural**
- Propósito: proveer una interfaz única de **fachada** para un conjunto de otras interfaces
  - Puede entenderse como un nivel más de abstracción que facilita el uso de subsistemas

Facade [2]

- La clase Facade provee un método de fachada
- Es de fachada porque en realidad está ocultando al cliente el trabajo con un subsistema
- Por ejemplo, la clase **TicketFacade** puede tener un método llamado **buyTicket ()** (método fachada) que en realidad hace llamados a otros métodos:
  - **checkAvailability ()** en clase **Ticket**
  - **assignPassenger ()** en clase **Ticket**
  - **registerPayment ()** en clase **Payment**

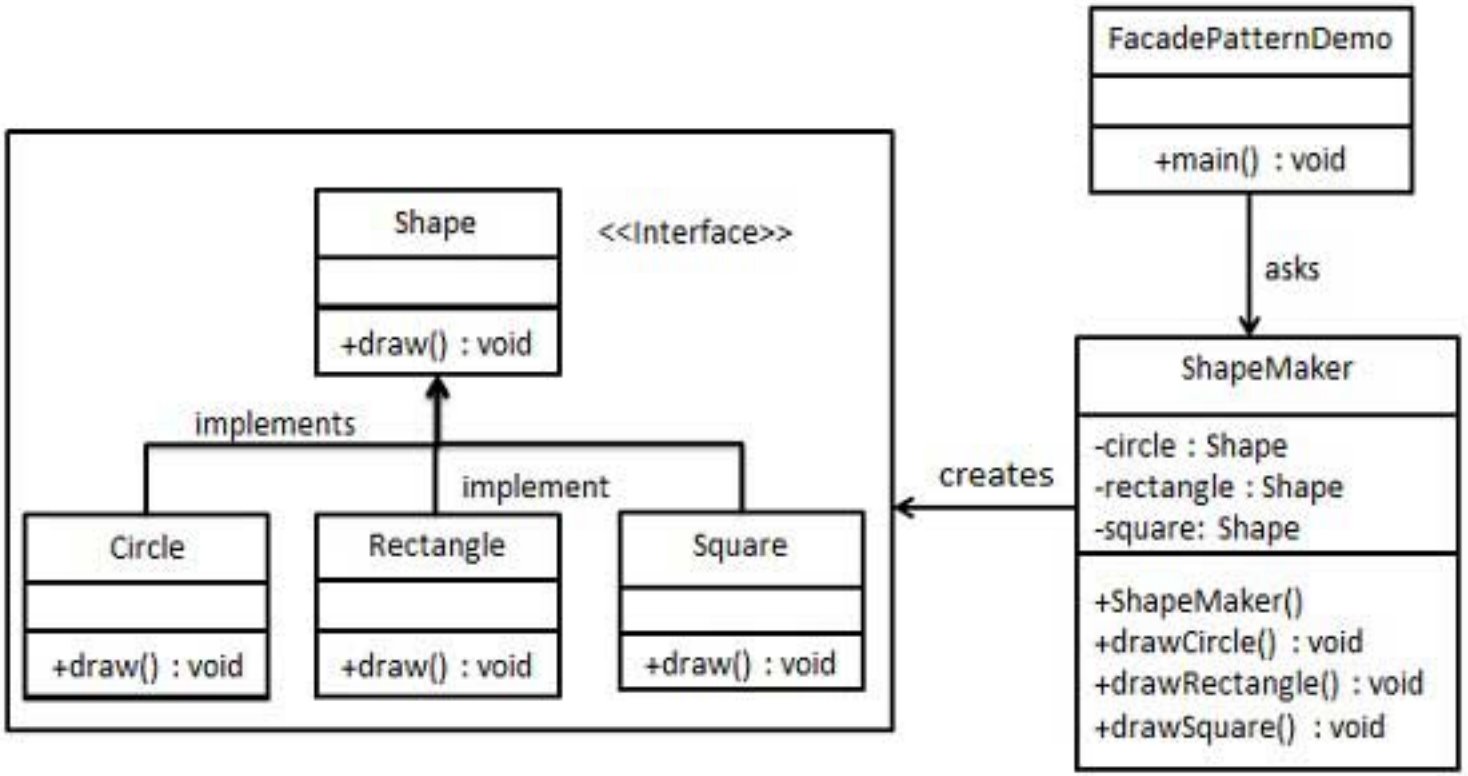
Facade [3]


Client
-facade
Facade
1
1
Complex Subsystem
ClassB
«use»
Class A
ClassE
«use»
ClassD
ClassC

Facade [3]

- Facade: Conoce las clases del subsistema y delega las peticiones de los clientes en los objetos del subsistema.
- Complex Subsystem: implementan su propia funcionalidad y manejan el trabajo asignado por el objeto Facade.

Facade [4]


FacadePatternDemo
+main() : void
asks
ShapeMaker
-circle : Shape
-rectangle : Shape
-square: Shape
+ShapeMaker()
+drawCircle() : void
+drawRectangle() : void
+drawSquare() : void
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
+draw() : void
Rectangle
+draw() : void
Square
+draw() : void

Facade [4]

```
public interface Shape {
    void draw();
}

public class Rectangle implements Shape {

    @Override
    public void draw() {

System.out.println("Rectangle::draw()");
    }
}
```

Facade [4]

```
public class Square implements Shape {

    @Override
    public void draw() {
       System.out.println("Square::draw()");
    }
}
public class Circle implements Shape {

    @Override
    public void draw() {
       System.out.println("Circle::draw()");
    }
}
```

Facade [4]

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

Facade [4]

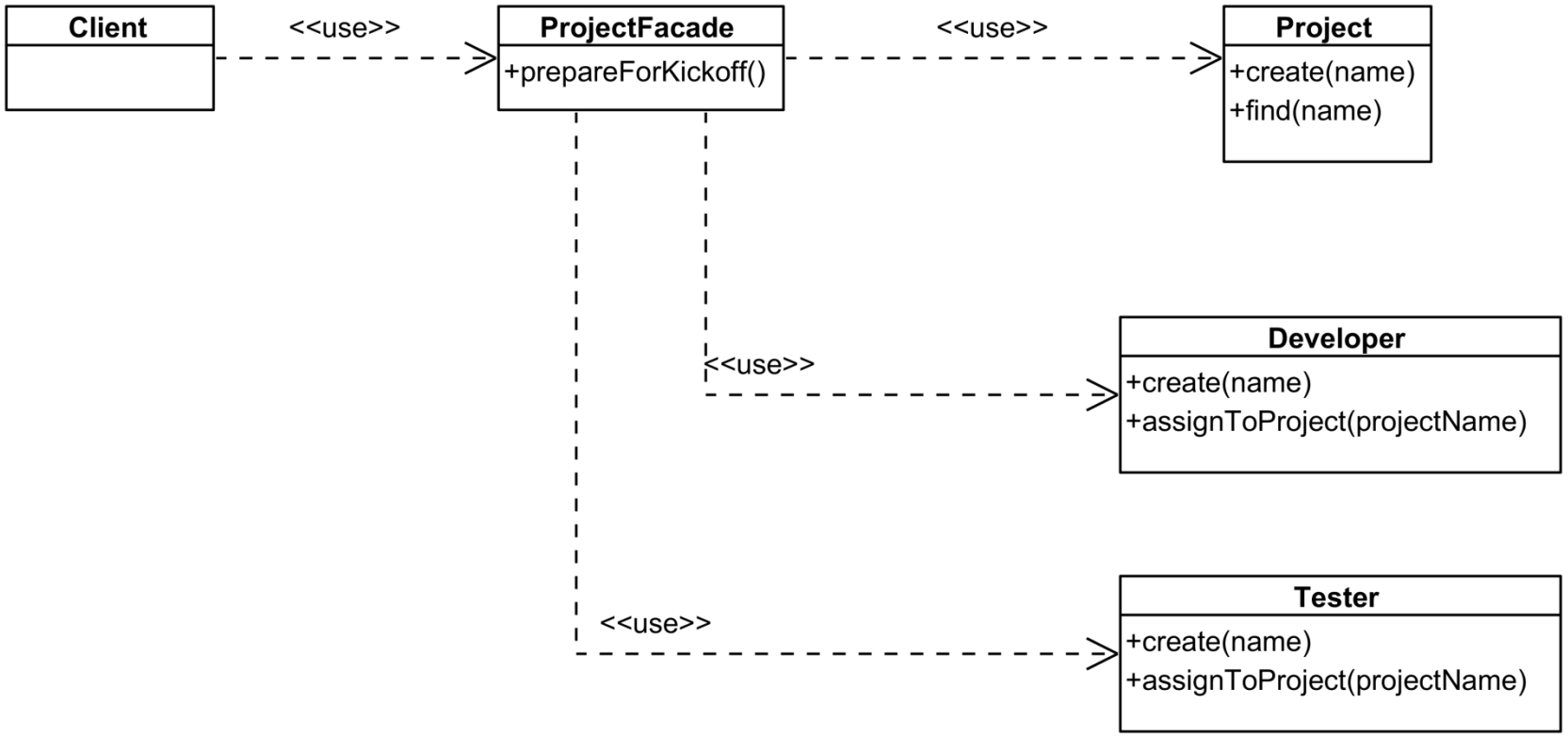
```
public class FacadePatternDemo {
   public static void main(String[]
args) {
      ShapeMaker shapeMaker = new
ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

Ejemplo: Facade con Java [1]

- El inicio de un proyecto *(kickoff)* requiere instrumentar el ambiente de desarrollo
- En el ejemplo veremos un caso muy sencillo de instrumentación de un **Issue Tracker**
- **prepareForKickoff()** es el **método fachada** que "oculta" al cliente:
  - Creación de un proyecto
  - Creación de un desarrollador y asignación al proyecto
  - Creación de un Tester y asignación al proyecto

Ejemplo: Facade con Java [2]



Ejemplo: Facade con Java [3]

```
public class ProjectFacade() {
        public void prepareForKickoff() {
                Project newProject = new Project();
                newProject.create(“FISW”);

                Developer newDeveloper = new Developer();
                newDeveloper.create(“Juan Pérez”);

                Tester newTester = new Tester();
                newTester.create(“John Doe”);

                newDeveloper.assignToProject(“FISW”);
                newTester.assignToProject(“FISW”);
        }
}
```

Ejemplo: Facade con Java [4]

- Algunas observaciones:
  - Podríamos haber usado los constructores, pero para el ejemplo es más transparente un método especial “create”
  - Las clases Project, Developer y Tester tienen implementaciones particulares que no afectan al cliente
    - Al cliente sólo le importa la **fachada!**

Bridge

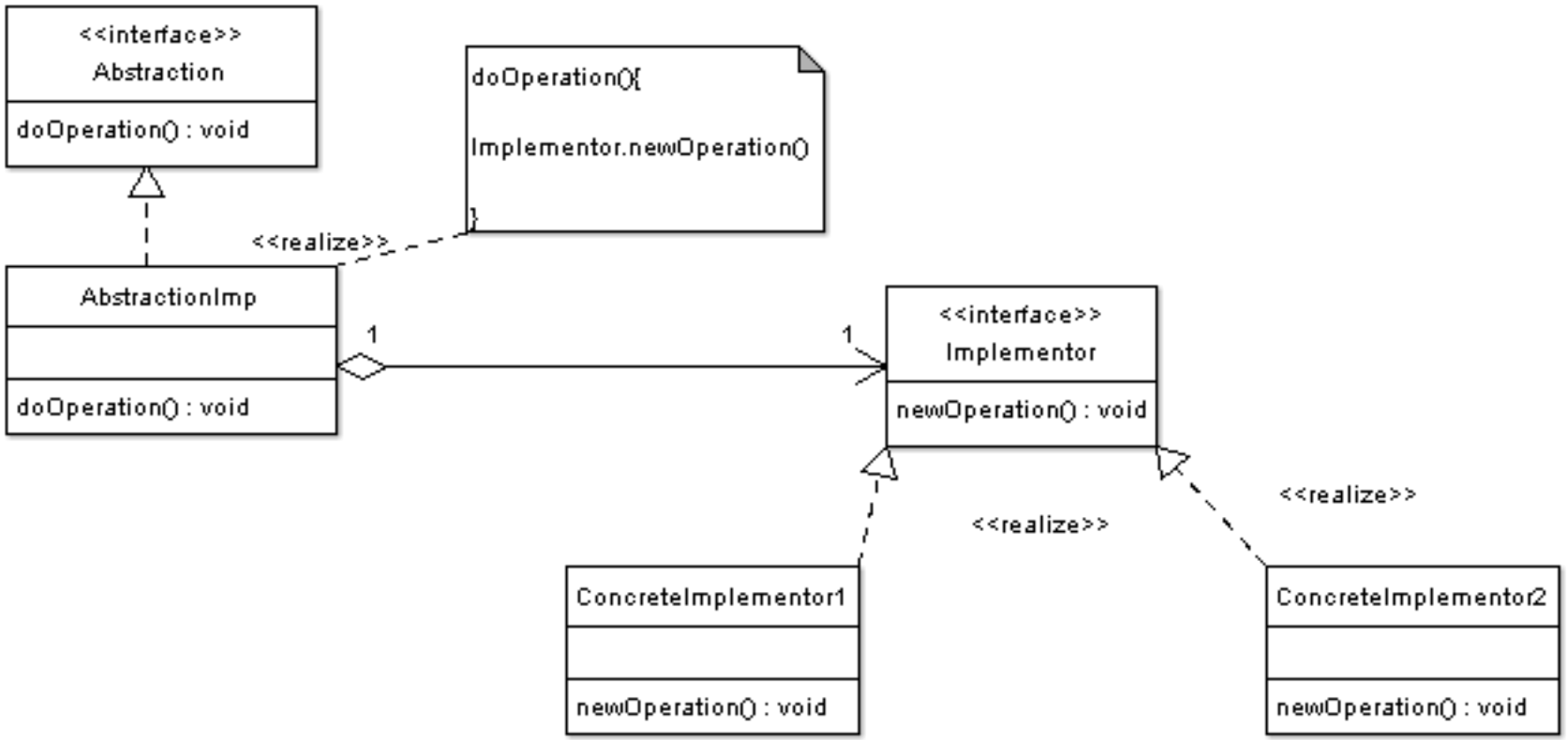
Bridge [1]

- Patrón de diseño **estructural**
- Propósito: desacoplar la **abstracción** de su **implementación** a fin de que ambas partes puedan variar de manera independiente
- **Abstracción** aquí significa "lo que ve el *cliente*"
  - Ej: Botones para *setear* velocidad crucero
    - Los botones son una **abstracción** para un conjunto de clases y métodos
    - Un **puente** nos permite utilizar distintas **abstracciones** (botones en "muscle car" versus botones en "city car") de manera independiente de la **implementación**

Bridge [2]

- Este patrón encontrará aplicación natural cuando existan distintas *abstracciones* para objetos que son similares (en su interior)
- Muy típico en la construcción de aparatos electrónicos:
  - Cámaras digitales de fotografía de distinto nivel son "similares internamente", pero con "abstracciones distintas"
    - La abstracción "controles novatos" impide que el usuario pueda acceder a funcionalidad avanzada
  - Lo mismo pasa con televisores, sintetizadores, tablets…

Bridge [3]


<<interface>>
Abstraction
doOperation() : void
doOperation(){
Implementor.newOperation()
}
<<realize>>
AbstractionImp
doOperation() : void
1
1
<<interface>>
Implementor
newOperation() : void
<<realize>>
<<realize>>
ConcreteImplementor1
newOperation() : void
ConcreteImplementor2
newOperation() : void

Bridge [3]

- Abstraction: define la abstracción de la interface
- AbstractionImpl: implementa la abstracción usando como referencia un objeto del tipo Implementor
- Implementor: esta clase define una interface para implementar las clases. Esta interface no necesita corresponder directamente a la abstracción de la interface y puede ser muy diferente.
- ConcreteImplementor: Implementa la interface Implementor

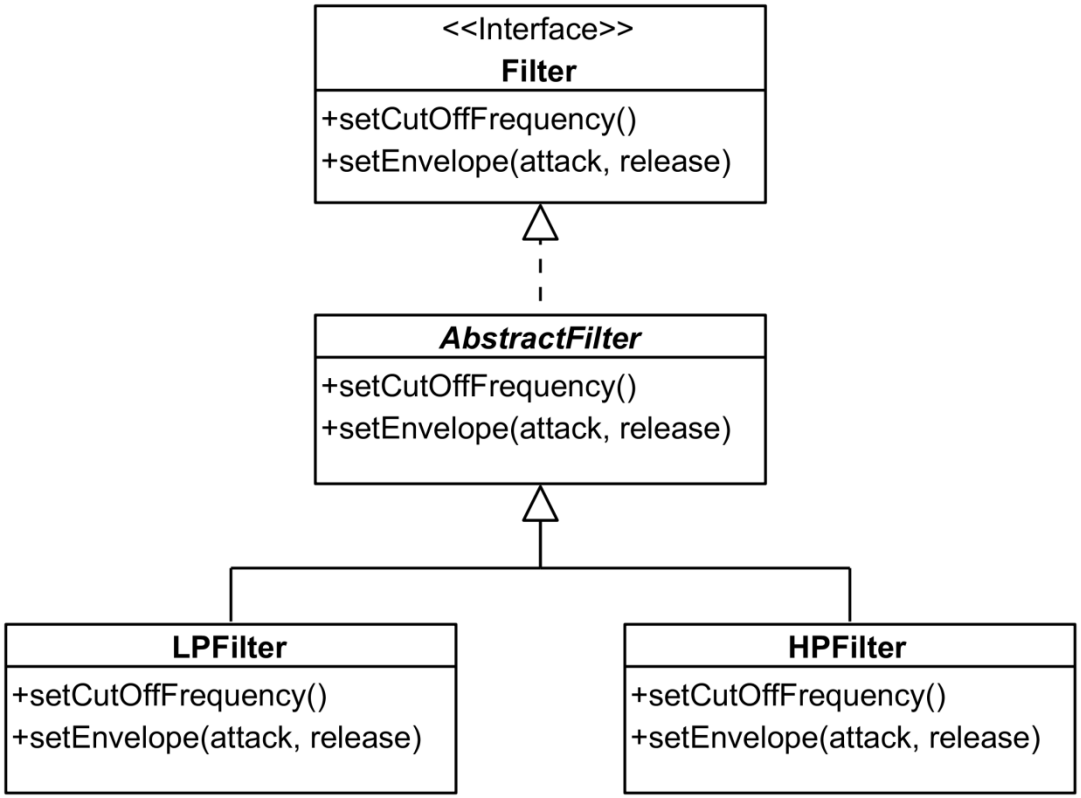
Ejemplo: Bridge con JAVA [1]

- Veremos un ejemplo en la construcción de sintetizadores de sonido, sean estos por software o por hardware
- Teclados sintetizadores de distintos precios ofrecen distintos paneles de controles (abstracciones)
- Pero en su interior, los motores de síntesis son similares o iguales (implementación)
- En los sintetizadores, un concepto difícil de entender para novatos es el de “filtro de ondas”

Ejemplo: Bridge con JAVA [2]

- Sintetizador para novatos:
  - Abstracción provee controles para:
    - El timbre
      - Cuán opaco o brillante es el sonido
    - Rapidez del filtro
      - Cuán rápido o lento actúa el filtro sobre la onda
- Sintetizadores para expertos:
  - Abstracción provee controles para:
    - Frecuencia
    - Envolvente (ataque, liberación)

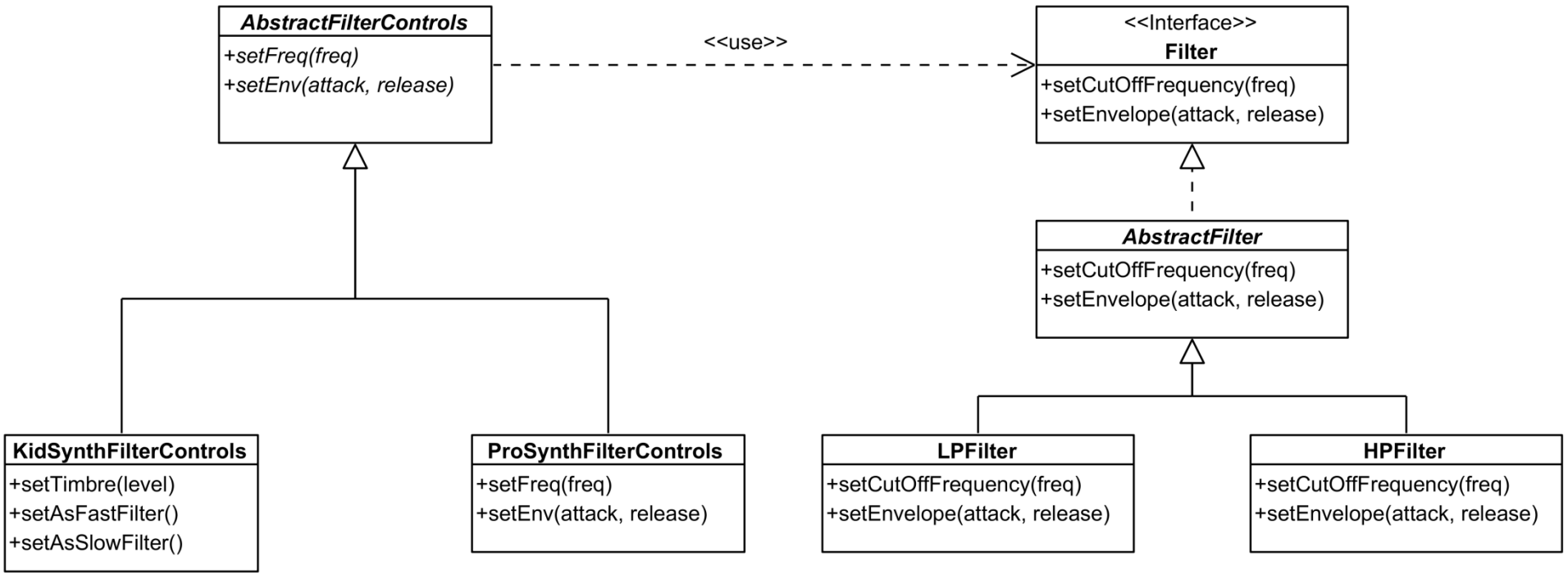
Ejemplo: Bridge con JAVA [3]



Ejemplo: Bridge con JAVA [4]

- El modelo de clases nos presenta una interfaz con dos métodos importantes para:
  - Establecer la frecuencia de corte del filtro
  - Establecer la envolvente del filtro
- Ambos métodos son heredados por dos clases filtros:
  - Filtro de paso bajo (LPF)
  - Filtro de paso alto (LPF)
- Hasta aquí todo ha sido **implementación**

Ejemplo: Bridge con JAVA [5]



Ejemplo: Bridge con JAVA [6]

- Con el patrón Bridge construimos un puente entre los controles **(abstracción)** de los dos tipos de sintetizadores y los filtros **(implementación)**
- Ambas abstracciones utilizan la misma implementación, pero pueden variar libremente
  - De hecho, las abstracciones pueden utilizar los mismos métodos o agregar nuevos
- **IMPORTANTE:** los métodos de una abstracción se implementan en términos de la superclase de abstracción (AbstractFilterControls) y NO en términos de la implementación (Filter)

Ejemplo: Bridge con JAVA [7]

```
// interfaz Filter
public interface Filter {
	public void setCutOffFrequency(float freq);
	public void setEnvelope(float attack, float release);
}

// implementación (abstracta) de Filter
// LPF y HPF podrán hacer override de los métodos según sea necesario
public abstract class AbstractFilter implements Filter {
	// constructor se encarga de inicializar objeto
	public void setCutOffFrequency(float freq) {
		// setear frecuencia de corte
	}
	public void setEnvelope(float attack, float release {
		// setear ataque y liberación
	}
}
```

Ejemplo: Bridge con JAVA [8]

```
// construimos la abstracción (controles)
public abstract class AbstractFilterControls {
	// aquí hacemos el puente a 'Filter'
	private Filter filter;

	// constructor es usado para inicializar la abstracción
	public AbstractFilterControls (Filter filter) {
		this.filter = filter;
	}

	public void setFreq() {
		filter.setCutOffFrequency(freq);
	}
	public void setEnv(attack, release) {
		filter.setEnvelope(attack, release);
	}
}
```

Ejemplo: Bridge con JAVA [9]

```
// extendemos la clase abstracta de controles; creamos controles
// para novatos y expertos

// caso fácil: controles de filtro para teclados 'pro' no
// requieren características especiales
public class ProSynthFilterControls extends
        AbstractFilterControls {
        public ProSynthFilterControls(Filter filter) {
                super(filter);
        }
        // y nada más
}
```

Ejemplo: Bridge con JAVA [10]

```
// caso complejo: controles para novatos
public class KidSynthFilterControls extends
        AbstractFilterControls {
        public KidSynthFilterControls(Filter filter) {
                super(filter);
        }
        // definimos los nuevos métodos
        public void setTimbre(int level) {
                // si nivel es bajo, setFreq(freqBajo)
                // si nivel es alto, setFreq(freqAlto)
        }
        public void setAsFastFilter() {
                setEnv(shortAttackRate, shortReleaseRate);
        }
        public void setAsSlowFilter() {
                setEnv(longAttackRate, longReleaseRate);
        }
}
```

Ejemplo: Bridge con JAVA [11]

```
// el objeto cliente (ej: la ventana de controles)
// puede hacer uso del patrón de esta forma:
LPFilter lpFilter = new LPFilter();
KidSynthFilterControls kidSynthControls = new
                        KidSynthFilterControls(lpFilter);

// queremos un sonido brillante, level = alto = 1
// con un filtro rápido
kidSynthControls.setTimbre(1);
kidSynthControls.setAsFastFilter();

// de manera análoga, el objeto cliente puede hacer uso de los
// controles “pro”
```

Adapter

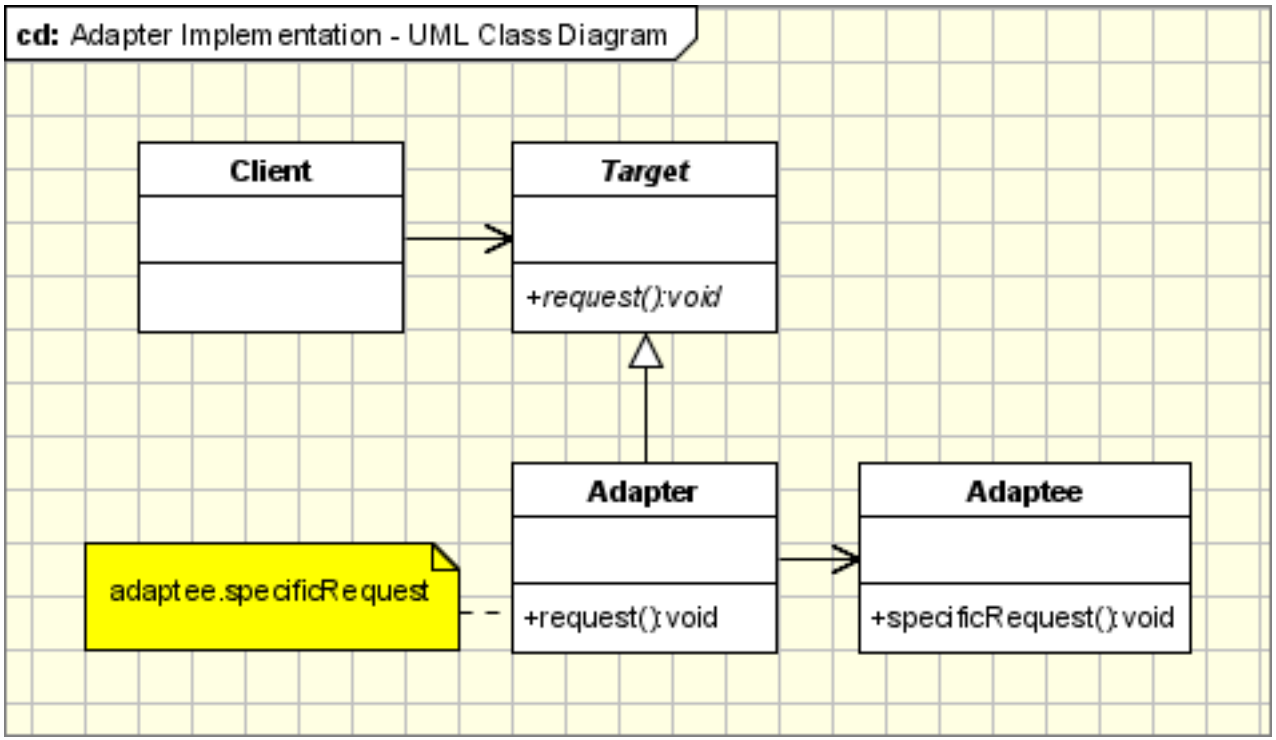
Adapter [1]

- Patrón de diseño **estructural**
- Propósito: convertir la interfaz de una clase en otra interfaz que un objeto cliente pueda entender (o esperar)
- Si no tenemos esta **adaptación**, las clases no podrán trabajar por incompatibilidad en las interfaces

Adapter [2]

- El patrón Adapter parte del supuesto de que existen dos clases que "hablan" en términos distintos
  - Por ejemplo, podríamos tener dos clases en dos sistemas distintos:
    - Clase **BusTicket** habla en términos de pasaje de bus
    - Clase **TravelTicket** habla en términos genéricos de un pasaje (puede ser de avión, bus, barco)
  - Una clase Adapter permitirá que **TravelTicket** (de un sistema de búsqueda de pasajes y hoteles) se comunique con la clase **BusTicket** (del sistema de una empresa de buses particular)

Adapter [3]


cd: Adapter Implementation - UML Class Diagram
Client
Target
+request():void
Adapter
+request():void
Adaptee
+specificRequest():void
adaptee.specificRequest

Adapter [3]

- Target: define el dominio especifico que el cliente usa
- Adapter: adapta la interface Adaptee a la interface Target
- Adaptee: define una interfaz existente que necesita ser adaptada
- Client: colabora con los objetos en función de la interface Target

Adapter [4]

- Vea el código del profesor

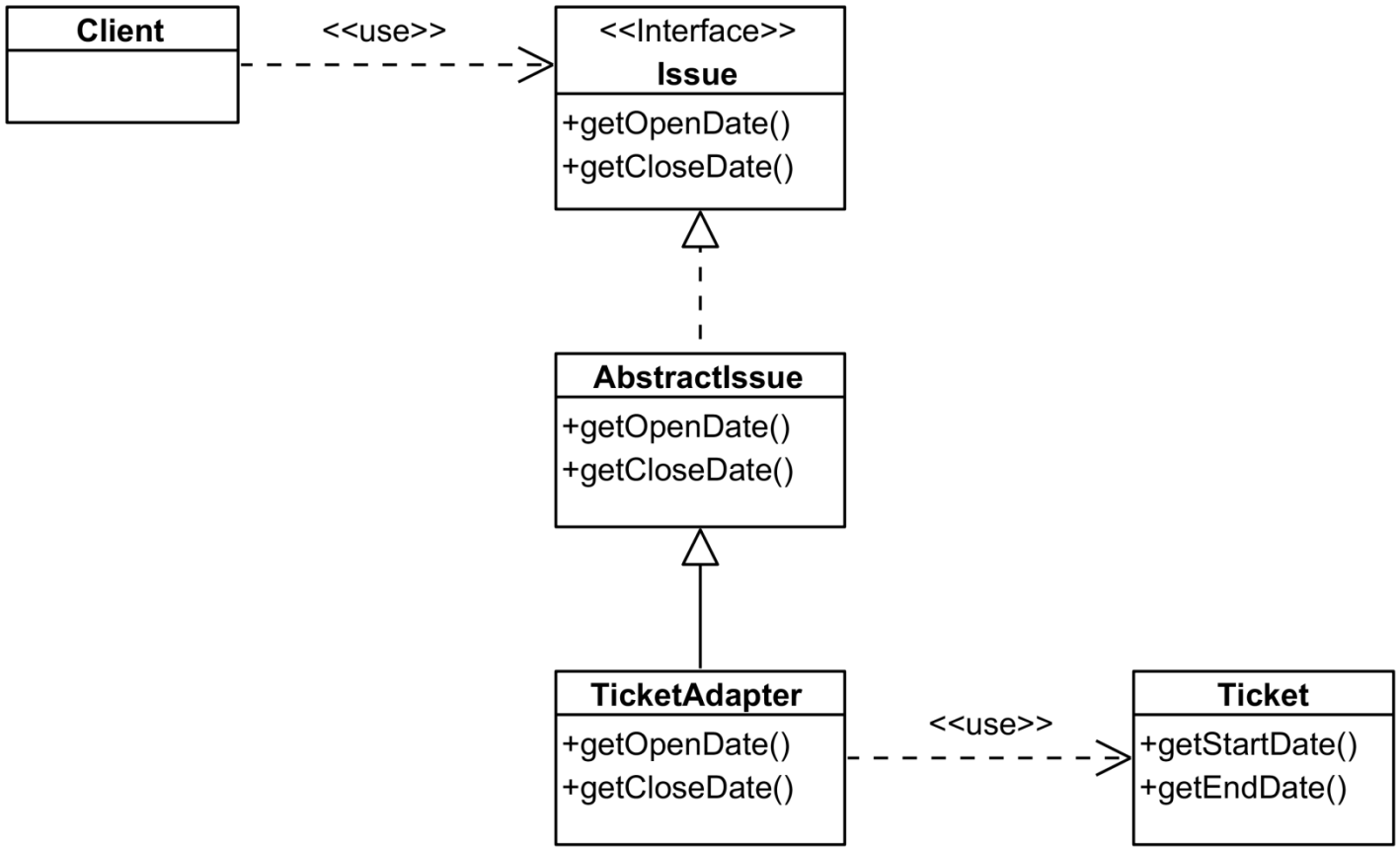
Ejemplo: Adapter con JAVA [1]

- Por diversos motivos, en las organizaciones puede haber más de un sistema de gestión de incidencias (Issue Trackers)
  - Issue Trackers manejan el concepto de Issue que se abre en una fecha openDate y se cierra en una fecha closeDate
  - Ticket Tracking Systems manejan el concepto de Ticket que se abre en una fecha startDate y se cierra en una fecha endDate

Ejemplo: Adapter con JAVA [2]

- ¿Cómo podemos hacer que un medidor que ya está construido para "hablar" con Issue Trackers ahora pueda "hablar" con Ticket Tracking Systems?
  - La solución es agregar una clase que **adapte** las interfaces
    - Esta clase "Adapter" hereda los métodos de la clase a la que nos queremos adaptar (ej: AbstractIssue)
    - La clase "Adapter" acepta en el constructor una referencia a la clase que adaptaremos

Ejemplo: Adapter con JAVA [3]



Ejemplo: Adapter con JAVA [4]

```
// nuevamente definimos la estructura de un Issue
public abstract class AbstratIssue implements Issue {
	private String openDate;
	private String closeDate;

	public AbstractIssue(String openDate, String closeDate){
		this.openDate = openDate;
		this.closeDate = closeDate;
	}
	public String getOpenDate(){
		return openDate;
	}
	public String getCloseDate(){
		return closeDate;
	}
}
```

Ejemplo: Adapter con JAVA [5]

```
// la referencia al constructor de la clase a la que nos adaptamos
public class TicketAdapter extends AbstractIssue {
	public TricketAdapter(Ticket ticket) {
		super(ticket.getStartDate(),
			ticket.getEndDate());
	}
}

// hacemos uso del patrón
Ticket ticket = new Ticket (...);

// pero la clase cliente no llama directamente al Ticket sino
// que llama al adaptador
TicketAdapter adapter = new TicketAdapter (ticket);

// trabajamos con el adaptador
adapter.getOpenDate();
```

FIN