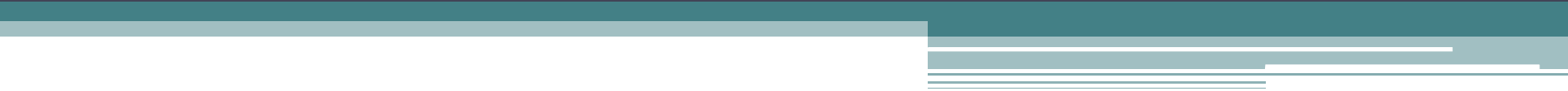


Patrón Adapter

Análisis & Diseño de Software/Fundamentos de Ingeniería de Software



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



Contexto

- Cargador de batería
- Todos los países tienen distintas políticas sobre la electricidad
 - 120V (US)
 - 240V(India)
- Por lo tanto, un cargador de batería que se *adapte* al país
- Se podría tener una clase llamada Volt (para medir los volts) y una clase Socket (que genere voltios constantes de 120V)

Contexto

```
public class Volt {  
    private int volts;  
    public Volt(int v){  
        this.volts=v;  
    }  
  
    public int getVolts() {  
        return volts;  
    }  
  
    public void setVolts(int volts) {  
        this.volts = volts;  
    }  
}
```

Contexto

```
public class Socket {  
    public Volt getVolt(){  
        return new Volt(120);  
    }  
}
```

Contexto

- Ya tenemos las clases. Ahora debemos crear un *adaptador* que pueda producir 3 volts, 12 volts y 120 volts (por defecto).
- Por lo anterior, es necesario crear una interface con los métodos solicitados

Contexto

```
public interface SocketAdapter {  
  
    public Volt get120Volt();  
  
    public Volt get12Volt();  
  
    public Volt get3Volt();  
}
```

- Ahora, ¿cómo podemos adaptar la interface SocketAdapter dependiendo del país?
- ¿Alguna solución?

Adapter [1]

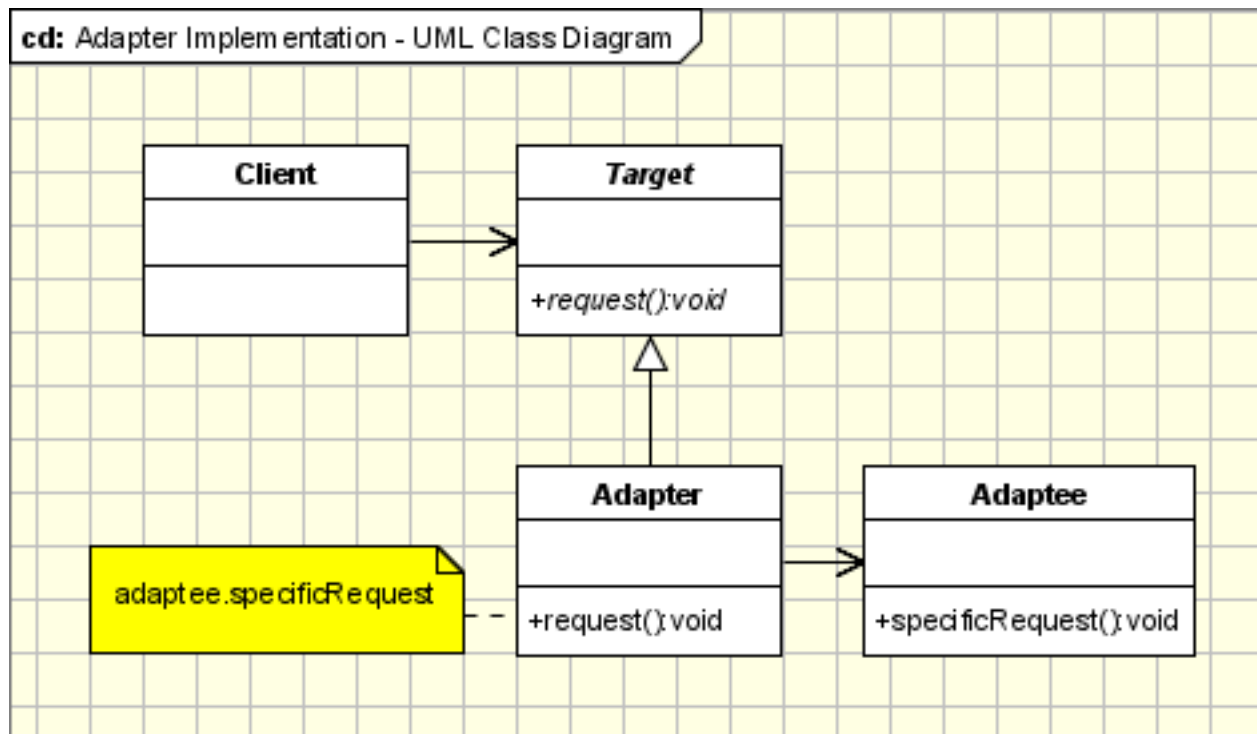
- Patrón de diseño estructural
- Propósito: convertir la interfaz de una clase en otra interfaz que un objeto cliente pueda entender (o esperar)
- Si no tenemos esta adaptación, las clases no podrán trabajar por incompatibilidad en las interfaces



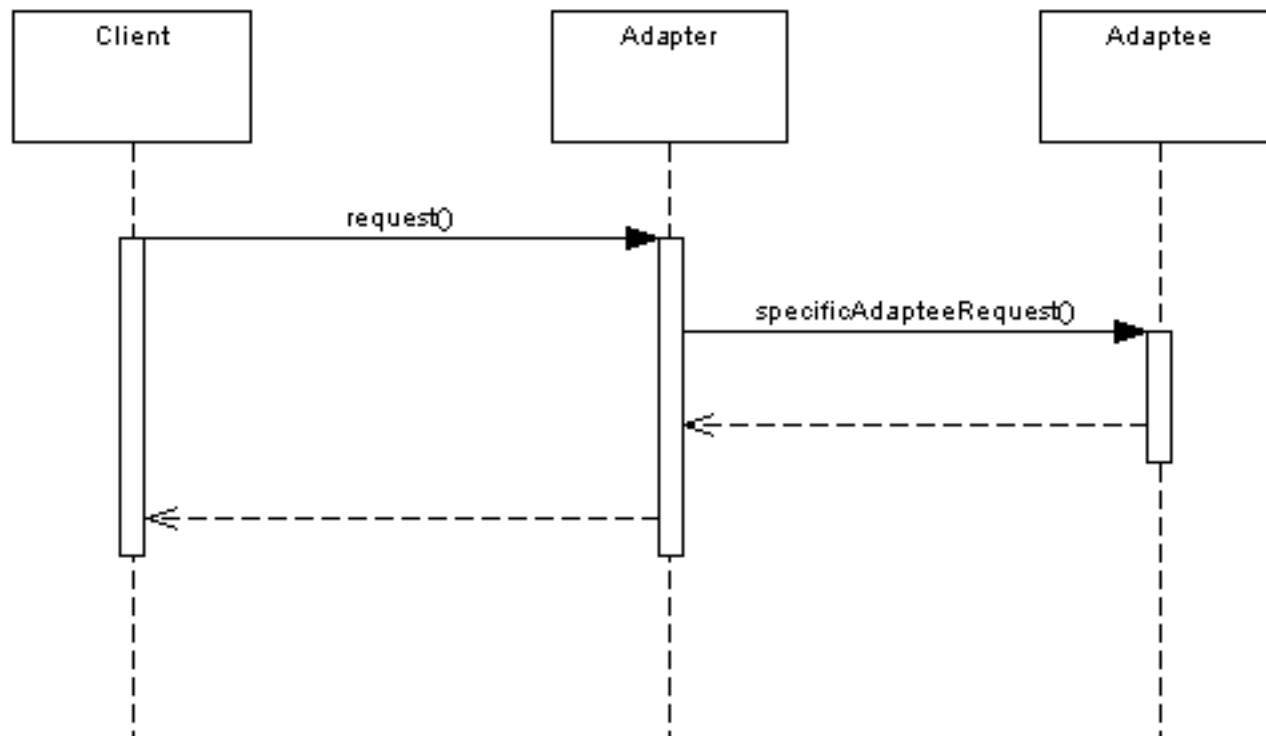
Adapter [2]

- 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 comuniquen con la clase BusTicket (del sistema de una empresa de buses particular)

Adapter [3]



Adapter [4]



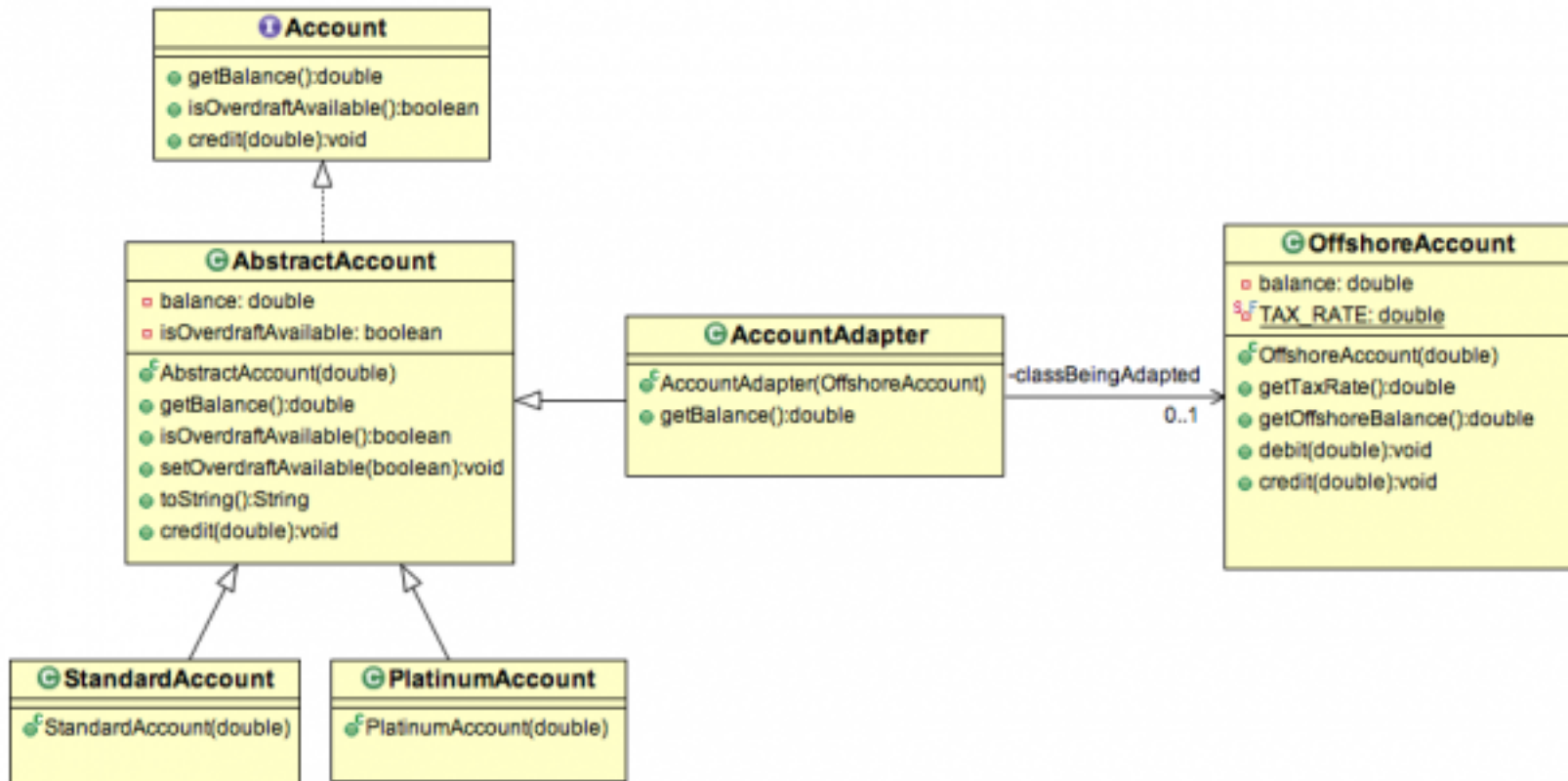
Adapter [3]

- Target: define el dominio específico 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

Ejemplo

- Un cierto banco ofrece servicios a nivel mundial. Para las cuentas offshore, el impuesto es un 0.03%
- En India, el banco ofrece dos tipos de cuenta, Standard y Platinum. Los impuestos internacionales no aplican en India, lo que produce un problema para las cuentas offshore del banco.
- Por lo anterior, el banco solicita *adaptar* estas cuentas incompatibles para que puedan trabajar juntas.

Ejemplo



Ejemplo

```
public class OffshoreAccount {
    private double balance;

    /** The tax for the country where the account is */
    private static final double TAX_RATE = 0.04;

    public OffshoreAccount(final double balance) {
        this.balance = balance;
    }

    public double getTaxRate() {
        return TAX_RATE;
    }

    public double getOffshoreBalance() {
        return balance;
    }

    public void debit(final double debit) {
        if (balance >= debit) {
            balance -= debit;
        }
    }

    public void credit(final double credit) {
        balance += credit;
    }
}
```

Ejemplo

```
public interface Account {  
    public double getBalance();  
    public boolean isOverdraftAvailable();  
    public void credit(final double credit);  
}
```

```
public class PlatinumAccount extends AbstractAccount {  
  
    public PlatinumAccount(final double balance) {  
        super(balance);  
        setOverdraftAvailable(true);  
    }  
}
```

```
public class StandardAccount extends AbstractAccount {  
  
    public StandardAccount(final double balance) {  
        super(balance);  
        setOverdraftAvailable(false);  
    }  
}
```


Ejemplo

```
public class AbstractAccount implements Account {
    private double balance;
    private boolean isOverdraftAvailable;

    public AbstractAccount(final double size) {
        this.balance = size;
    }

    @Override
    public double getBalance() {
        return balance;
    }

    @Override
    public boolean isOverdraftAvailable() {
        return isOverdraftAvailable;
    }

    public void setOverdraftAvailable(boolean isOverdraftAvailable) {
        this.isOverdraftAvailable = isOverdraftAvailable;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + " Balance=" + getBalance()
            + " Overdraft:" + isOverdraftAvailable();
    }

    @Override
    public void credit(final double credit) {
        balance += credit;
    }
}
```

Ejemplo

```
public class AccountAdapter extends AbstractAccount {  
    // Adaptee – The class we are adapting from  
    private OffshoreAccount offshoreAccount;  
  
    public AccountAdapter(final OffshoreAccount offshoreAccount) {  
        super(offshoreAccount.getOffshoreBalance());  
  
        // holds adaptee reference  
        this.offshoreAccount = offshoreAccount;  
    }  
  
    @Override  
    public double getBalance() {  
        final double taxRate = offshoreAccount.getTaxRate();  
        final double grossBalance = offshoreAccount.getOffshoreBalance();  
  
        final double taxableBalance = grossBalance * taxRate;  
        final double balanceAfterTax = grossBalance - taxableBalance;  
        return balanceAfterTax;  
    }  
}
```

Ejemplo

```
public class AdapterTest {  
    public static void main(String[] args) {  
  
        StandardAccount sa = new StandardAccount(2000);  
        System.out.println("Account Balance= " + sa.getBalance());  
  
        //Calling getBalance() on Adapter  
        AccountAdapter adapter = new AccountAdapter(new  
OffshoreAccount(2000));  
        System.out.println("Account Balance= " +  
adapter.getBalance());  
    }  
}
```

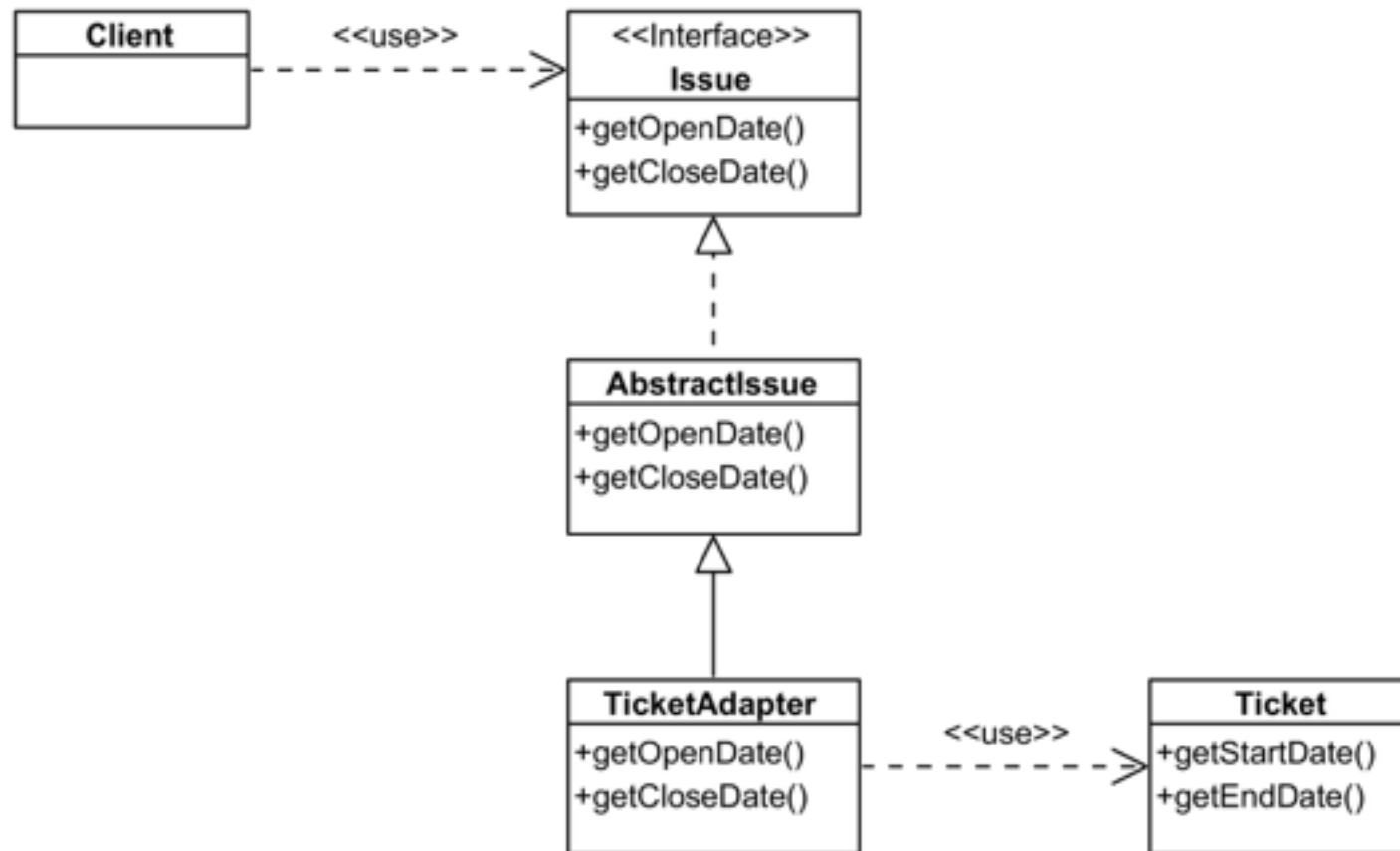
Ejemplo 2

- 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 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 2



Ejemplo 2

```
// 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;
    }
}
```

Volvamos al problema del contexto

- ¿Ya está lista la solución?

Solución

```
//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket implements SocketAdapter{

    @Override
    public Volt get120Volt() {
        return getVolt();
    }

    @Override
    public Volt get12Volt() {
        Volt v= getVolt();
        return convertVolt(v,10);
    }

    @Override
    public Volt get3Volt() {
        Volt v= getVolt();
        return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
        return new Volt(v.getVolts()/i);
    }

}
```

Solución

```
public class AdapterPatternTest {  
  
    public static void main(String[] args) {  
        testClassAdapter();  
    }  
  
    private static void testClassAdapter() {  
        SocketAdapter sockAdapter = new SocketClassAdapterImpl();  
        Volt v3 = getVolt(sockAdapter, 3);  
        Volt v12 = getVolt(sockAdapter, 12);  
        Volt v120 = getVolt(sockAdapter, 120);  
        System.out.println("v3 volts using Class Adapter="+v3.getVolts());  
        System.out.println("v12 volts using Class Adapter="+v12.getVolts());  
        System.out.println("v120 volts using Class Adapter="+v120.getVolts());  
    }  
  
    private static Volt getVolt(SocketAdapter sockAdapter, int i) {  
        switch (i){  
            case 3: return sockAdapter.get3Volt();  
            case 12: return sockAdapter.get12Volt();  
            case 120: return sockAdapter.get120Volt();  
            default: return sockAdapter.get120Volt();  
        }  
    }  
}
```

FIN