Adapter

Cel:

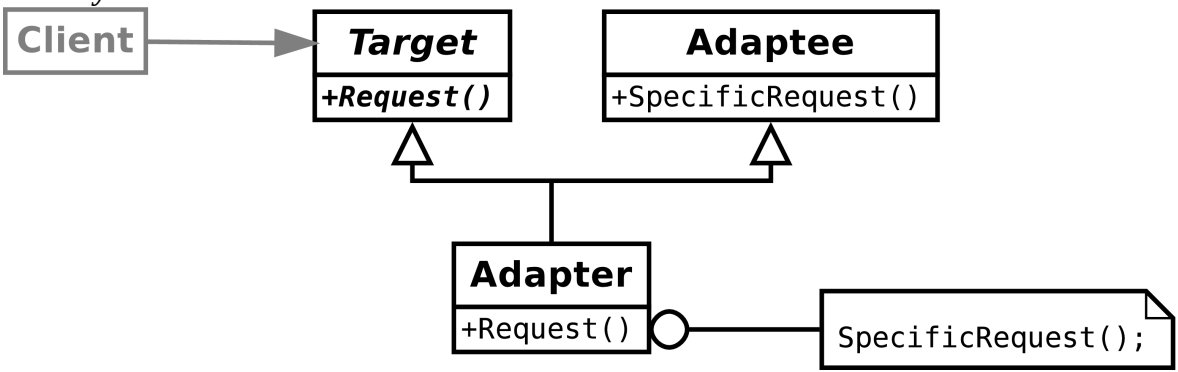
Zamienia interfejs klasy na wymagany przez klienta – pozwala współpracować klasom, które inaczej nie mogłyby tego robić z powodu niezgodności interfejsów.

Zastosowanie:

- Gdy chcemy zmienić interfejs klasy:
  - chcemy użyć istniejącej klasy, lecz jej interfejs nie pasuje do naszych wymagań,
  - chcemy stworzyć klasę, która będzie mogła współpracować z klasami o niekompatybilnych interfejsach.

Struktura:

klasowy:



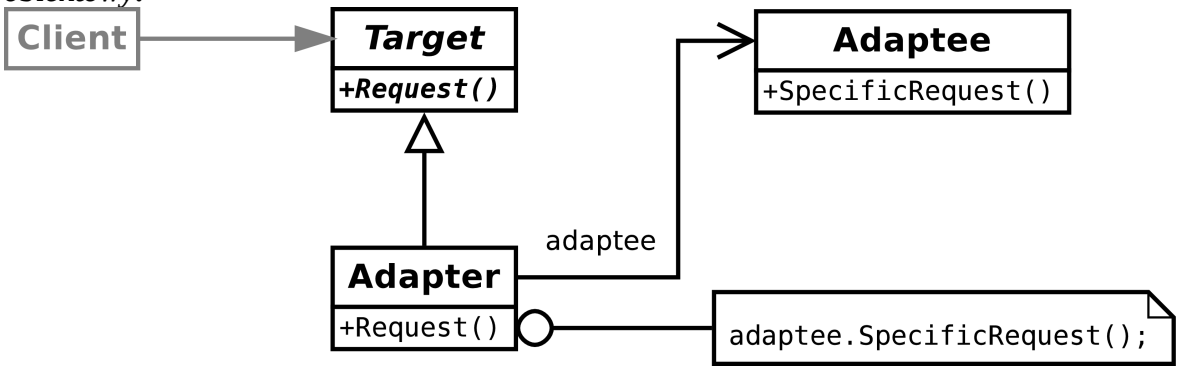
Składniki:

- **Target**
  - interfejs oczekiwany przez klienta (Client)
- **Adaptee (obiekt adaptowany)**
  - definiuje istniejący interfejs, który wymaga adaptacji
- **Adapter**
  - żądania klienta oddelegowuje do metod klasy Adaptee

```
interface Target {
     void Request();
}
class Adaptee {
     public void SpecificRequest() { /* ... */ }
}
class AdapterKlasowy extends Adaptee implements Target {
     public void Request() {
          super.SpecificRequest();
     }
}
// ... Client ...
Target o = new AdapterKlasowy();
o.Request();
```

Struktura:

obiektowy:



Składniki:

- **Target**
  - interfejs oczekiwany przez klienta (Client)
- **Adaptee (obiekt adaptowany)**
  - definiuje istniejący interfejs, który wymaga adaptacji
- **Adapter**
  - żądania klienta oddelegowuje do obiektu klasy Adaptee

```
interface Target {
     void Request();
}
class Adaptee {
     public void SpecificRequest() { /* ... */ }
}
class AdapterObiektowy implements Target {
     private Adaptee adaptee;
     AdapterObiektowy(Adaptee adaptee) {
          this.adaptee = adaptee;
     }
     public void Request() {
          adaptee.SpecificRequest();
     }
}
// ... Client ...
Target o = new AdapterObiektowy(new Adaptee());
o.Request();
```

Konsekwencje:

1. **obiektowy adapter** – raz napisany Adapter może współpracować również z klasami potomnymi Adaptee; trudniej jest wpłynąć funkcjonalność, gdyż obiektem adaptowanym posługujemy się jedynie przez publiczny interfejs

```
interface Stack {
     void push(double value); // ...
}
class Stos {
     void włóż(double wartość) { /* ... */ } // ...
}
class StosEx extends Stos { /* ... */ }
class StackAdapter implements Stack {
     private Stos stos;
     public StackAdapter(Stos stos) { this.stos = stos; }
     public void push(double value) {
          stos.włóż(value);
     }
     // ...
}
Stack stack = new StackAdapter(new StosEx());
stack.push(2.5);
```

2. **klasowy adapter** – dziedziczy z konkretnej klasy, którą adaptuje, może dowolnie modyfikować interfejs oraz funkcjonalność, nie wymaga tworzenia dodatkowych obiektów, ani użycia wskaźników

```
interface Stack {
     void push(double value); // ...
}
class Stos {
     void włóż(double wartość) { /* ... */ } // ...
}
class StackAdapter extends Stos implements Stack {
     public boolean empty() {
          return super.first == null;
     }
     // ...
}
Stack stack = new StackAdapter(new StosEx());
System.out.println(stack.empty());
```

3. zadania **adaptera** – niekiedy to tylko zmiana nazw metod, niekiedy nowe rodzaje operacji

```
interface Machine {
     boolean operation(int a, int b);
     boolean operation(double x);
}
class RealMachine {
     public String operation(String input);
}
class MachineProxy implements Machine {
     private RealMachine machine;
     public MachineProxy(RealMachine machine) {
          this.machine = machine; }
     public boolean operation(int a, int b) {
          String output = machine.operation (a + ", " + b);
          return Boolean.valueOf(output);
     }
     public boolean operation(double x) {
          String output = machine.operation ((int)x + ", 0");
          return Boolean.valueOf(output);
     }
}
```

4. *pluggable adapters* – sposób tworzenia użytecznych klas, które mogą współpracować z wieloma klasami, pod warunkiem że implementują oczekiwany przez nas interfejs

```
interface Viewable {
     int size();
     Object getValue(int idx);
}

class MyView {
     private Viewable object;
     public MyView(Viewable object) {
          this.object = object;
     }
     public void show() {
          System.out.print("*");
          int size = object.size();
          for(int i = 0; i < size; ++i)
               System.out.print(" " +
                       object.getValue(i).toString() + " *");
          System.out.println();
     }
}
```

```
class ViewableArray implements Viewable {
     private int[] ar;
     public ViewableArray(int[] ar) { this.ar = ar; }
     public int size() { return ar.length; }
     public Object getValue(int idx)  {
          return Integer.valueOf(ar[idx]);
     }
}


// ... Client ...

     int[] ar = new int[10];
     // ... wypełnianie wartościami
     MyView view = new MyView(new ViewableArray(ar));
     view.show();
```

Inny przykład:

```
class MyTableModel extends AbstractTableModel
{
     private MyData data;
     public MyTableModel(MyData data) {
          this. data = data;
     }
     public String getColumnName(int column) {
          return data.headers.get(column).toString();
     }
     public int getRowCount() {
          return data.size();
     }
     public int getColumnCount() {
          return data.headers.size();
     }
     public Object getValueAt(int row, int column) {
          return data.get(row).get(column);
     }
}
```

```
// ... Client ...

    JFrame frame = new JFrame("Moja baza danych");
    MyData database = new MyData();

    JTable table = new JTable(new MyTableModel(database));

    frame.pack();
    frame.setVisible(true);
```

Powiązania:

- Brigde – zbliżona struktura, lecz inne cele: oddzielić interfejs od implementacji
- Decorator – modyfikuje biekt, nie zmieniając interfejsu
- Proxy – definiuje zastępczą reprezentację obiektu, lecz nie zmienia interfejsu