InterBit
Training & Consulting Ltd.

Adapter Pattern

Chapter Content

InterBit
Training & Consulting Ltd.

- Adapter Pattern Overview
- Class & Object adapters
- Object Adapter UML Diagram
- Class Adapter UML Diagram
- Adapter Pattern Examples
- Object adapter vs. Class adapter
- Two way adapters

Adapter Pattern Overview

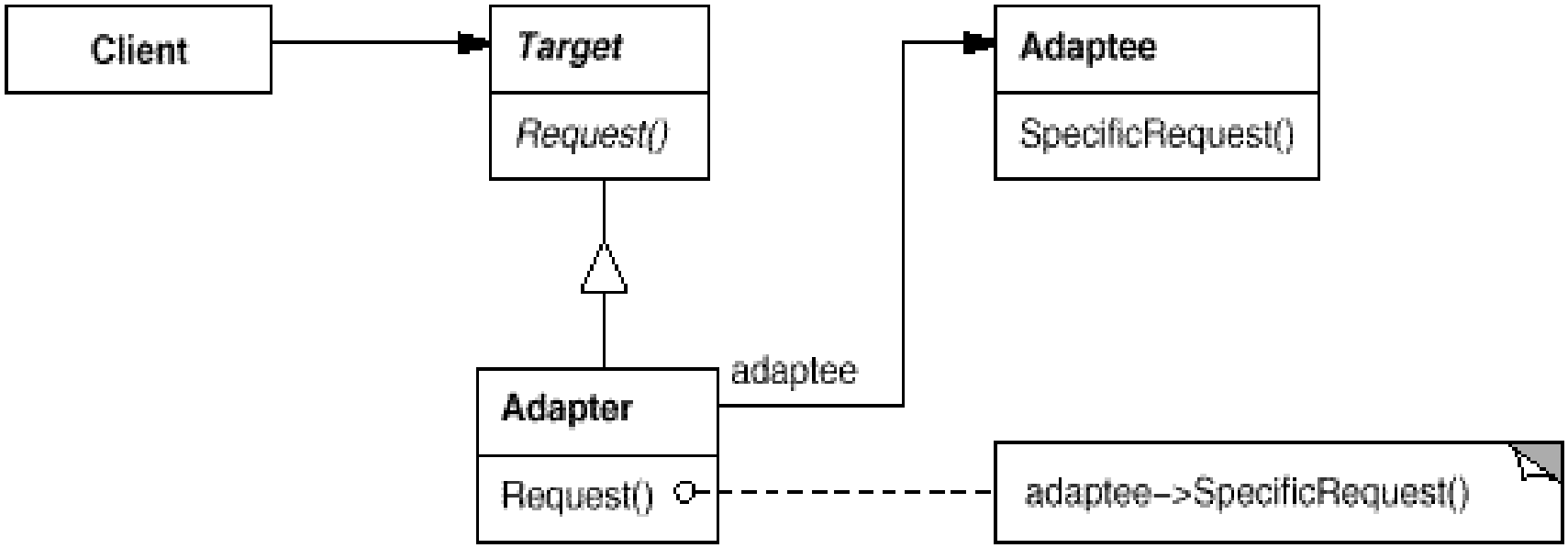
InterBit Training & Consulting Ltd.

- **Converts the interface of an existing class into another interface.**
- Use it to:
  - Adapt old classes to a new API (common!).
  - Incorporate existing classes into frameworks which require a slightly different interface.

Class & Object adapters

- How can one force a new interface on a class?
  - **Inheritance**:
    - new class *extends* the original one.
  - **Composition**:
    - new class *contains* (wraps) an instance of the original one.
- We shall soon discuss the pros & cons of each approach.

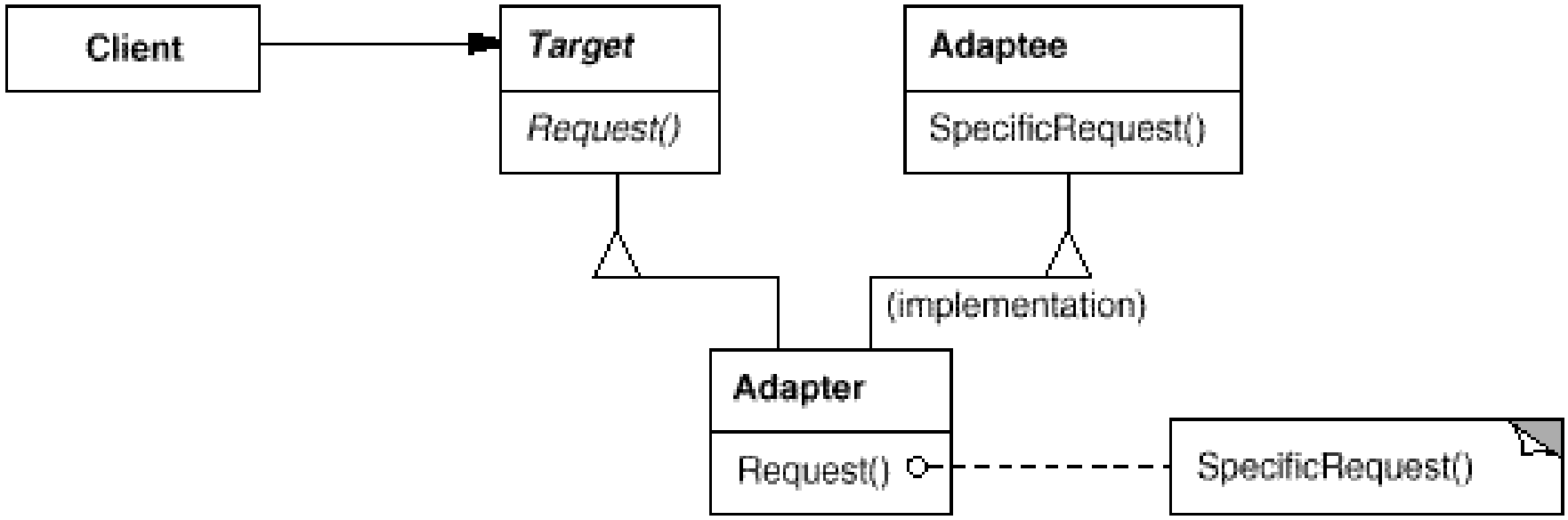
Object Adapter UML Diagram



Adaptee has some useful logic, but it doesn't implement the Target interface.

Composition solution: Adapter contains an Adaptee, and implements the Target.

Class Adapter UML Diagram



Adaptee has some useful logic, but it doesn't implement the Target interface.

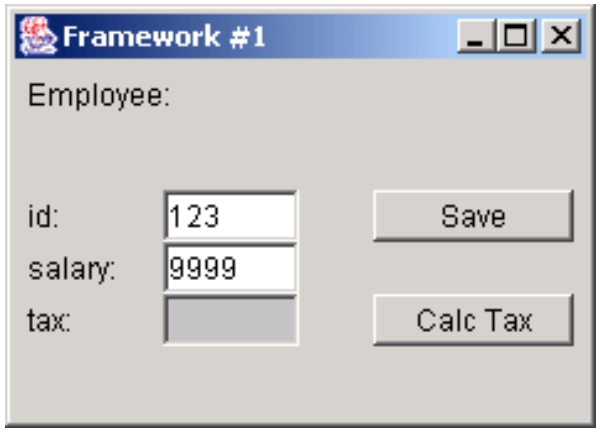
Inheritance solution: Adapter extends Adaptee, and implements Target .

Adapter Example - motivation

InterBit Training & Consulting Ltd.

```
class Employee {
     private int id;
     private double salary;
     public void saveToDB() {…}
     public double taxInNIS() {…}          // tax in Sheqels
     ...
}
```

Employee is incorporated into framework #1 (including, say, GUI), which relies on saveToDB(), calcTaxInNIS()



saveToDB()

taxInNIS

Example – motivation (cont.)

- Now suppose Employee needs to fit into a new framework, which relies on a different interface:

```
Interface NewEmpInterface {
     void store();
     double taxInDollars();
}
```

store()

taxInDollars

Solution (A): object adapter

```
// Object adapter, wraps an employee:
class EmployeeAdapter implements NewEmpInterface {
   private Employee emp= new Employee();
   public void store()  {
     emp.save();
   }
   public double taxInDollars() {
     return emp.taxInNIS()* getDollarRate();
   }
   private double getDollarRate() { … }  // returns approx. 5
}
```

Adapter:
Wraps employee in order to fit it into the new interface

- EmployeeAdapter **contains** an Employee.

Solution (B): class adapter

```
// Class adapter, inherits from Employee:
class EmployeeAdapter extends Employee implements NewEmpInterface {

   public void store()  {
     save();
   }
   public double taxInDollars() {
     return  taxInNIS()* getDollarRate();
   }
   private double getDollarRate() { … }
}
```

Adapter:
Extends employee in order to fit it into the new interface

- EmployeeAdapter **extends** Employee.

Object Vs. Class adapter

InterBit Training & Consulting Ltd.

- **Object adapters (composition):**
  - Flexible! The same adapter can be used with different internal **subclasses of employee.**
    - As opposed to class adapters which are committing to the concrete Employee which they extend.
  - Don’t suffer from the following inheritance problem : ending up with several methods for the same functionality (save / store).

Object vs. Class adapter (cont.)

InterBit Training & Consulting Ltd.

- **Class adapters (inheritance):**
  - Possibly less allocations.
  - Less coding (relying on some methods to be automatically inherited).

Two way adapters

- An adapter may fit several interfaces at the same time.
  - Used when the same object needs to work with several API's, possibly simultaneously.

```
class EmployeeAdapter implements Interface1, Interface2, Interface3 {
   private Employee emp= new Employee();
   public void store()   { emp.saveToDB();}
   public void save()    { emp.saveToDB();}
   public void persist() { emp.saveToDB();}
}
```