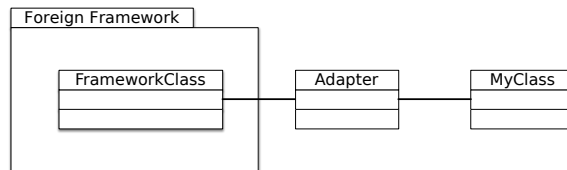# Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Adapter Pattern

The Adapter Design Pattern
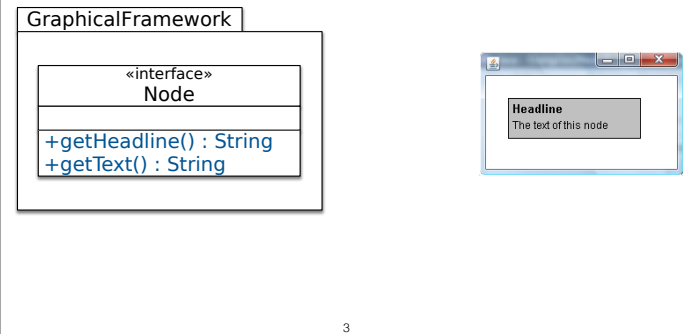
Fit foreign components into an existing design.

Goal

We want to reuse existing frameworks or libraries in our software, even if they do not match with our design.

We do not want to change our design to adhere to the structure of the reused components.
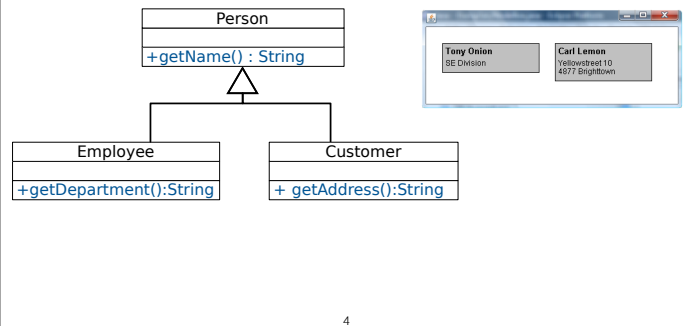
The Adapter Design Pattern - Illustrated

GraphicalFramework

«interface»
Node

+getHeadline() : String
+getText() : String

Headline
The text of this node

We have acquired the framework **GraphicalFramework**.

`GraphicalFramework` provides the interface **Node** to draw rectangles with a headline and text to the screen.

Drawing is done by the framework, we just need to provide the data via the interface **Node**.

Desired Usage of the Framework

Person

+getName() : String

Tony Onion
SE Division

Carl Lemon
Yellowstreet 10
4877 Brighttown

Employee

+getDepartment():String

Customer

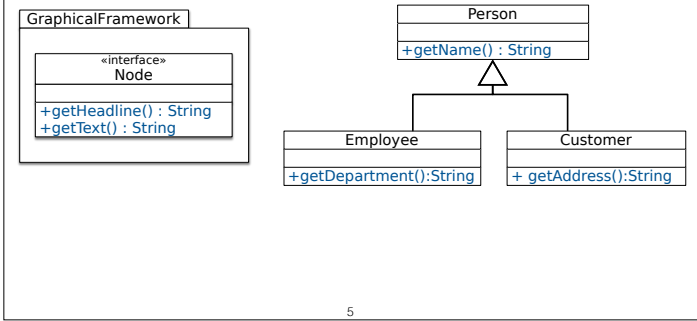+ getAddress():String

Our own design represents different kinds of persons.
We want to draw our data to the screen:
- Name and department of `Employee`.
- Name and address of `Customer`.

Adapting the Framework

GraphicalFramework

«interface»
Node

+getHeadline() : String
+getText() : String

Person

+getName() : String

Employee

+getDepartment():String

Customer
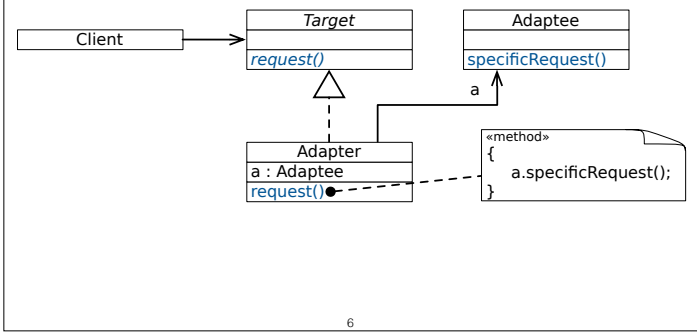
+ getAddress():String

5

We will create adapters to use the functionality of GraphicalFramework for our classes.

We have to adapt Employee and Customer to fit with Node.

Object Adapter

Client

Target

request()

Adaptee

specificRequest()

a

Adapter

a : Adaptee

request()

«method»
{
    a.specificRequest();
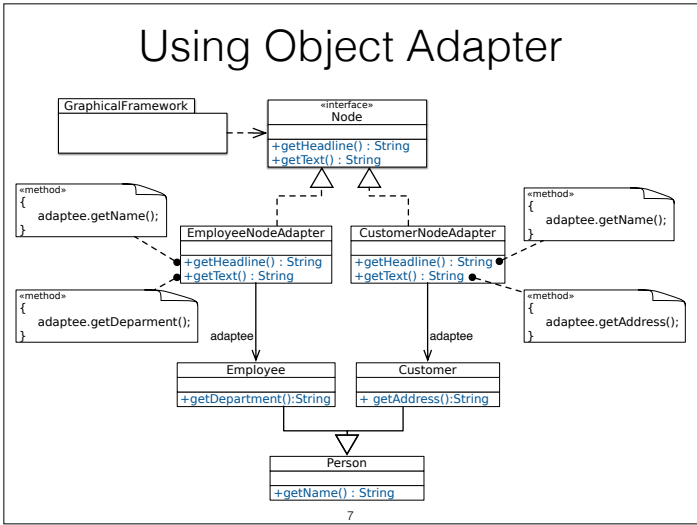}

6

- Adaptee is wrapped by Adapter to fit in the interface of Target.
- Adapter forwards calls of Client to request() to the specific methods of Adaptee (e.g, specificRequest()).
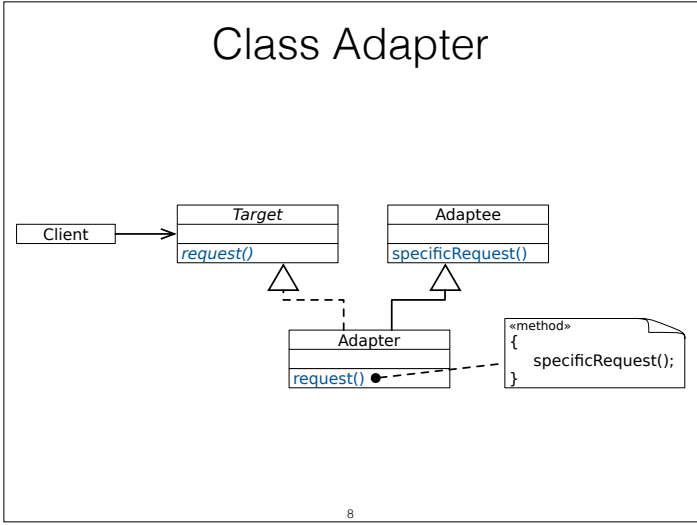
Using Object Adapter

GraphicalFramework

«interface»
Node
+getHeadline() : String
+getText() : String

«method»
{
  adaptee.getName();
}

«method»
{
  adaptee.getName();
}

EmployeeNodeAdapter
+getHeadline() : String
+getText() : String

CustomerNodeAdapter
+getHeadline() : String
+getText() : String

«method»
{
  adaptee.getDeparment();
}

«method»
{
  adaptee.getAddress();
}

adaptee

adaptee

Employee
+getDepartment():String

Customer
+ getAddress():String

Person
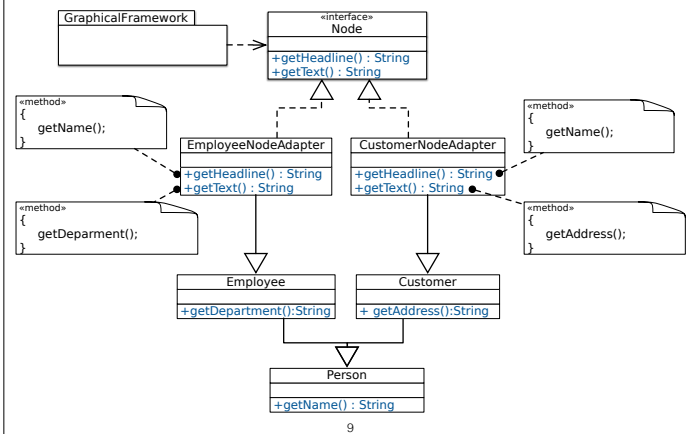+getName() : String

7

- **Advantages**:
  - Adapter works with Adaptee and any subclass of it.
  - Adapter can add functionality to Adaptee and its subclasses.

- **Disadvantages**:
  - Cannot override methods in Adaptee.
  - Cannot reuse Adapter with subclasses of Target.
  - Adapter and Adaptee are different objects.
    (Need to maintain relation between Adaptee and his Adapter)

Class Adapter

Client

Target
request()

Adaptee
specificRequest()

Adapter
request()

«method»
{
   specificRequest();
}

8

Instead of having Adaptee as an attribute, Adapter inherits from Adaptee.

Using Class Adapter

Advantages:
- Behavior of Adaptee can be overridden.
- Adapter and Adaptee are the same object, no forwarding.

Disadvantages:
- Adapter cannot be used with subclasses of Adaptee or Target.
- Multiple inheritance may be required.
  In Java: At least one of Target and Adaptee must be an Interface.

Takeaway

- Adapter is an effective means to adapt existing behavior to the expected interfaces of a reusable component or framework.

- Two variants: **Object and Class Adapter**

  - Both have their trade-offs.

  - Both have problems with the reusability of the adapter.

Pimp-my-Library
Idiom/Pattern (Scala)

Transparently add functionality to "fixed"
library classes.

Pimp-my-Library Idiom/Pattern (Scala)

Solution Idea

- Define a conversion function to convert your object into
  the required object and make this conversion `implicit` to
  let the compiler automatically perform the conversion
  when needed.
  (*Transparent* generation of object adapters.)

Example Scenario

- We want to be able to repeat a certain operation multiple times and want to store the result in some given mutable store/collection.

In Scala's (2.10) mutable collections do not define a common method to add an element to them.

In the following we develop a generalization of the previously shown `repeat` method. This variant enables the developer to specify the target data store.

Implementing a `repeatAndStore` method (naïve approach)

```scala
object ControlFlowStatements {
    import scala.collection.mutable.Set
    abstract class MutableCollection[T, C[T]](val underlying: C[T]) {
        def +=(elem: T): Unit
    }
    implicit def setToMutableCollection[T](set: Set[T]) =
        new MutableCollection(set) {
            def +=(elem: T) = set += (elem)
        }

    def repeatAndStore[T, C[T]](
        times: Int)(
            f: ⇒ T)(collection: MutableCollection[T, C]): C[T] = {
        var i = 0; while (i < times) { collection += f; i += 1 }
        collection.underlying
    }
}
```

What are the drawbacks of the solution?

Implementing a `repeatAndStore` method (naïve approach)

```scala
object ControlFlowStatements {
    import scala.collection.mutable.Set
    abstract class MutableCollection[T, C[T]](val underlying: C[T]) {
        def +=(elem: T): Unit
    }
    implicit def setToMutableCollection[T](set: Set[T]) =
        new MutableCollection(set) {
            def +=(elem: T) = set += (elem)
        }

    def repeatAndStore[T, C[T]](
        ti
```

```scala
object CFSDemo extends App {
    import ControlFlowStatements._

    val nanos =
        repeatAndStore(5) {
            System.nanoTime()
        }(new scala.collection.mutable.HashSet[Long]())
    }
```

What is the type of **nanos**?

15

The solution has two issues:

1. The repeatAndStore method requires a MutableCollection which is basically an implementation-internal type.
2. It returns the original collection to make the usage easier, but important type information is lost (the HashSet has become a Set).

Implementing a `repeatAndStore` method.

```scala
import scala.collection.mutable.{Set,HashSet,Buffer,ArrayBuffer}
object ControlFlowStatements{

    trait Mutable[-C[_]] {
        def add[T](collection: C[T], elem: T): Unit
    }

    implicit object Set extends Mutable[Set] {
        def add[T](collection: Set[T], elem: T) { collection += elem }
    }

    implicit object Buffer extends Mutable[Buffer] {
        def add[T](collection: Buffer[T], elem: T) { collection += elem }
    }

    def repeat[T, C[T] <: AnyRef: Mutable](
        times: Int)(f: ⇒ T)(collection: C[T]): collection.type = {

        var i = 0
        while (i < times) { implicitly[Mutable[C]].add(collection, f); i += 1 }
        collection
    }
}
```

16

Implementing a `repeatAndStore` method.

```scala
import object CFSDemo extends App {
object   import ControlFlowStatements._

   tra   val nanos_1: Set[Long] =
}          repeat(5){ System.nanoTime() }(new HashSet[Long]())

   imp   val nanos_2: Buffer[Long] =
}          repeat(5){ System.nanoTime() }(new ArrayBuffer[Long]())

   imp   val nanos_3: nanos_1.type =
          repeat(5) {System.nanoTime() }(nanos_1)

   } }
def repeat[T, C[T] <: AnyRef: Mutable](
      times: Int)(f: ⇒ T)(collection: C[T]): collection.type = {
      var i = 0
      while (i < times) { implicitly[Mutable[C]].add(collection, f); i += 1 }
      collection
   }
}
```