Summer Term 2018

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

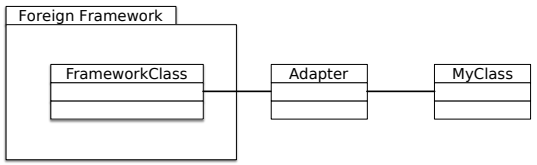
Adapter Pattern

1

Intent

The Adapter Design Pattern

Fit foreign components into an existing design.



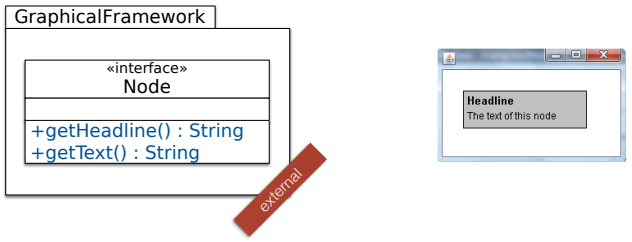
2

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



3

3

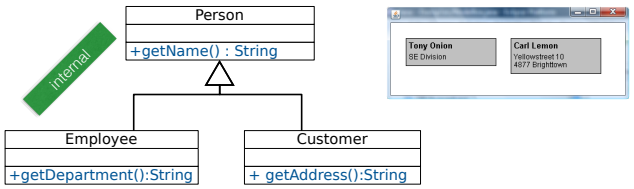
We have acquired the framework **GraphicalFramework**.

`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`.

(Similar: we have the class String from the JDK… but we need additional functionality.)

Desired Usage of the Framework



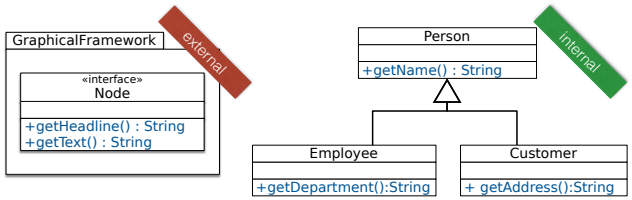
4

4

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



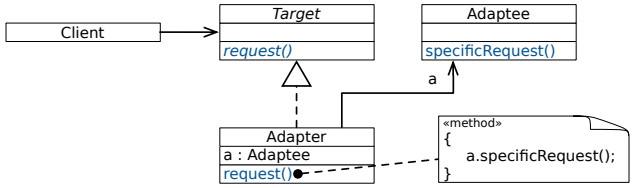
5

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

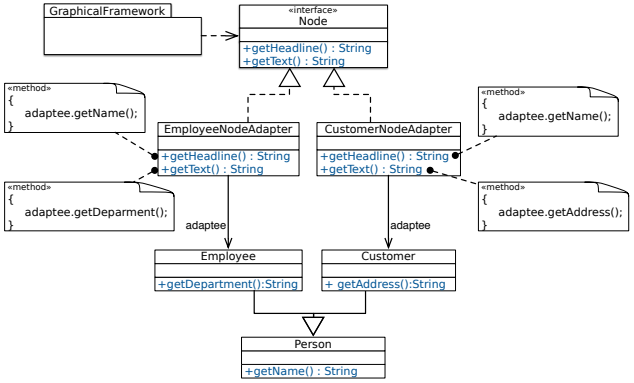


6

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



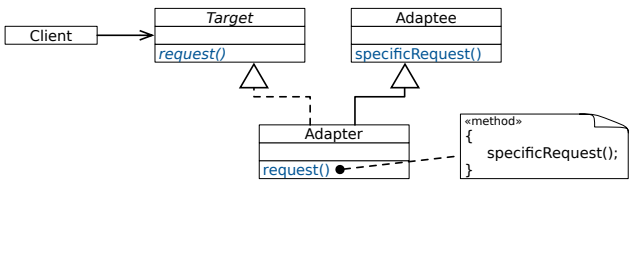
7

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

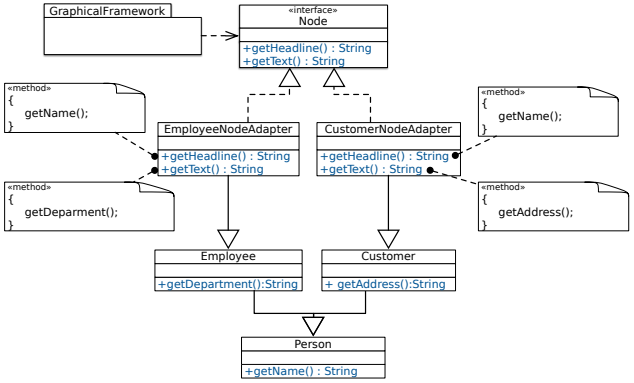


8

8

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

Using Class Adapter



9

9

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.

10

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.)

Adding `fold` to `java.lang.String`

A String is basically an ordered sequence of chars. Hence, we expect all standard collection operations.

- Definition of the “Adapter”:

```
Context {
implicit class RichString(val string: String) extends AnyVal {
  def foldIt[T](start:T)(f:(T,Char) => T) : T = {
    var r = start
    for(i <- 0 until string.length) r = f(r,string.charAt(i))
    r
  }
}
}
```

- As soon as the class `RichString` is in *scope*, we can now write:

```
"abc".foldIt("Result:")(_ + _.toShort)
```

13

13

Due to the inheritance from “AnyVal” this solution is actually allocation-free(!); i.e., no instance of the class RichString will be created at runtime.

Advanced 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.

14

14

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 (initial idea)

```
object ControlFlowStatements {

    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

    }

}
```


15

Issue: How do we design `MutableCollection`?

Implementing a `repeatAndStore` method (naïve approach)

```
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
    }
}
```


16

What are the drawbacks of the solution?

```
// the following code is included for easy copy and paste
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
    }
}
```

Implementing a `repeatAndStore` method (naïve approach)

```
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
        va
        co
    }
}
```

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

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

What is the type of nanos?

17

17

The solution has three issues:

1. The `repeatAndStore` method requires a `MutableCollection` which is basically an implementation-internal type.
2. It returns the original collection to make usage easier, but important type information is lost (the HashSet has become a Set).
3. Every call creates a new instance of the wrapper object.

Implementing a `repeatAndStore` method.

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

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

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

    implicit object BufferLike 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
    }
}
```

18

18

Explanation

Mutable is a so-called type class. It defines operations that must be supported (here) by a type C. Here, we next make the types “Set” and “Buffer” members of the type class Mutable (recall that an object does not define a type!)

Implementation Detail

“`: Mutable`” is basically equivalent to adding an implicit parameter “`evidence : Mutable[C]`”

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

Implementing a `repeatAndStore` method.

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

    trai
    }

    impl
    }

    impl
    }

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

```
import ControlFlowStatements._

val nanos_1: Set[Long] =
  repeat(3){ System.nanoTime() }(new HashSet[Long]())

val nanos_2: Buffer[Long] =
  repeat(3){ System.nanoTime() }(new ArrayBuffer[Long]())

val nanos_3: nanos_1.type =
  repeat(3) {System.nanoTime() }(nanos_1)
```