Summer Semester 2015

Software Engineering Design & Construction

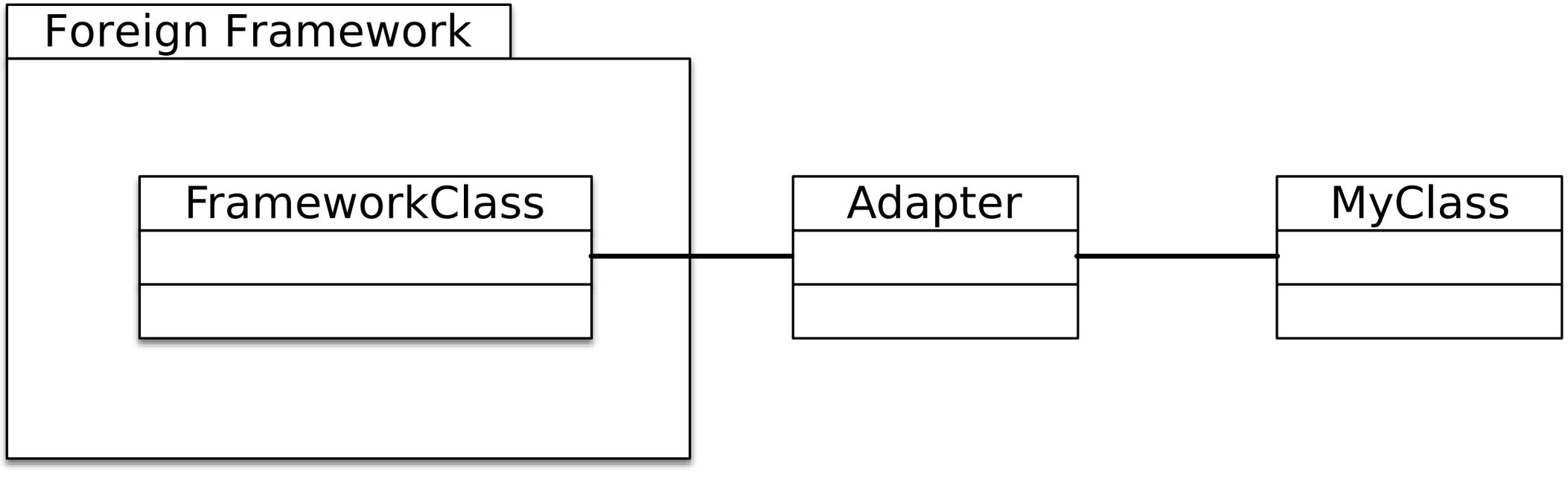
Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Adapter Pattern

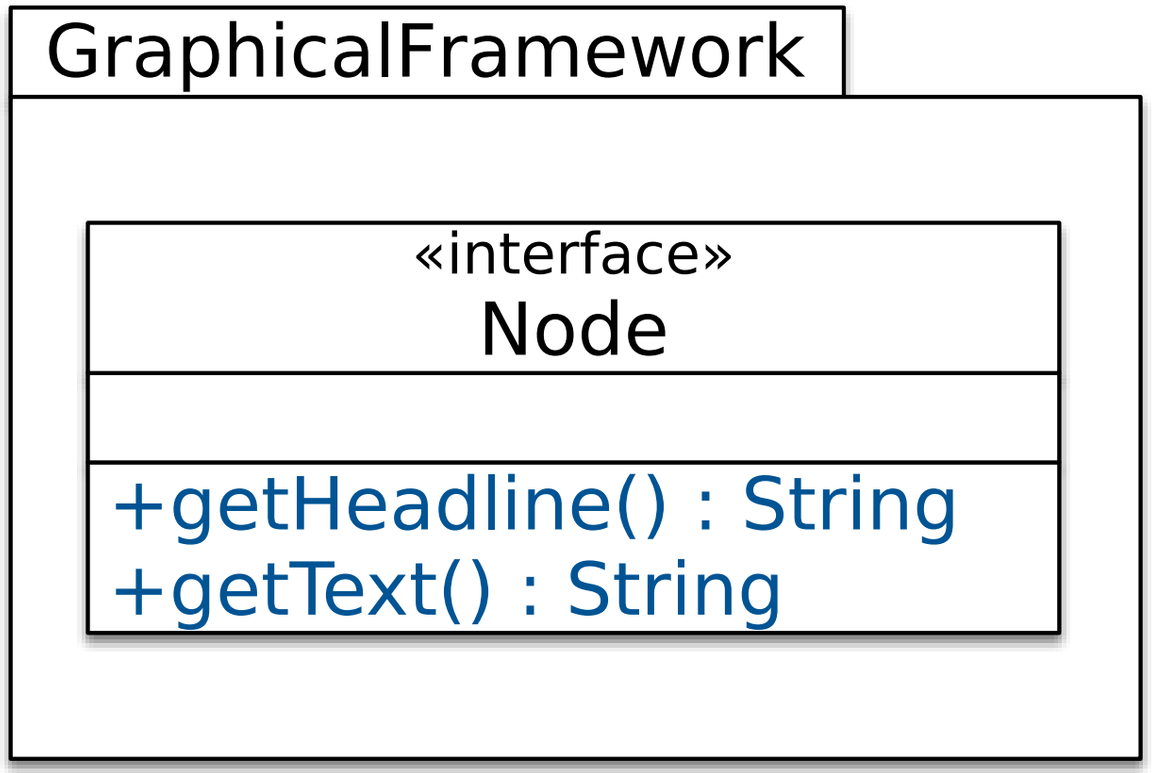
Intent

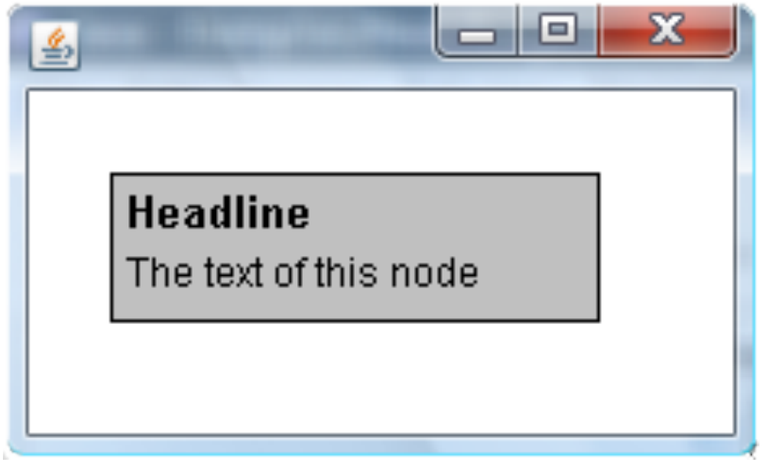
The Adapter Design Pattern

Fit foreign components into an existing design.

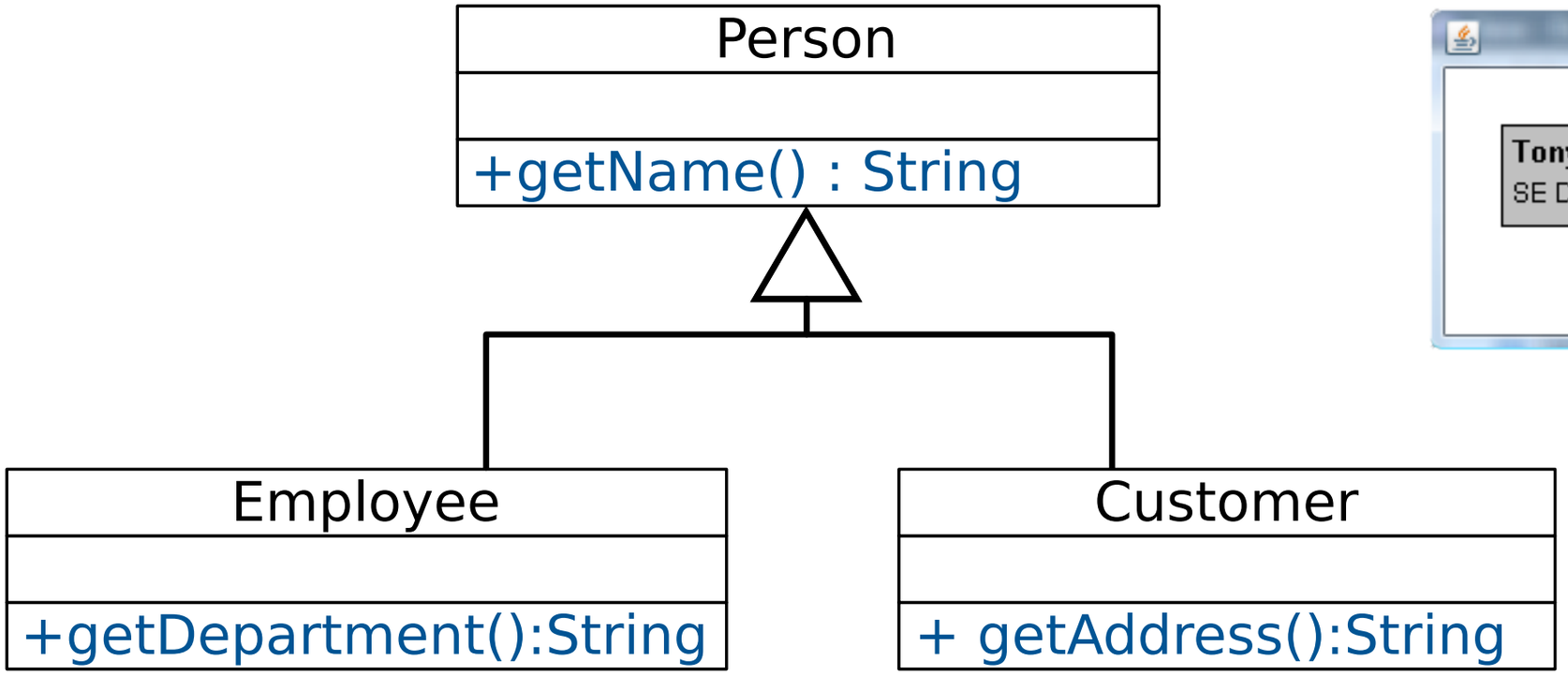


The Adapter Design Pattern - Illustrated

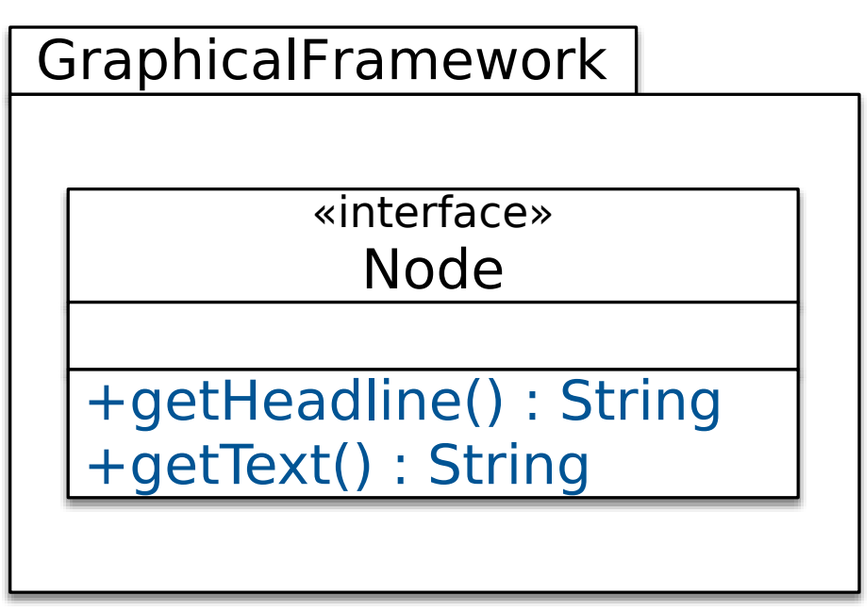


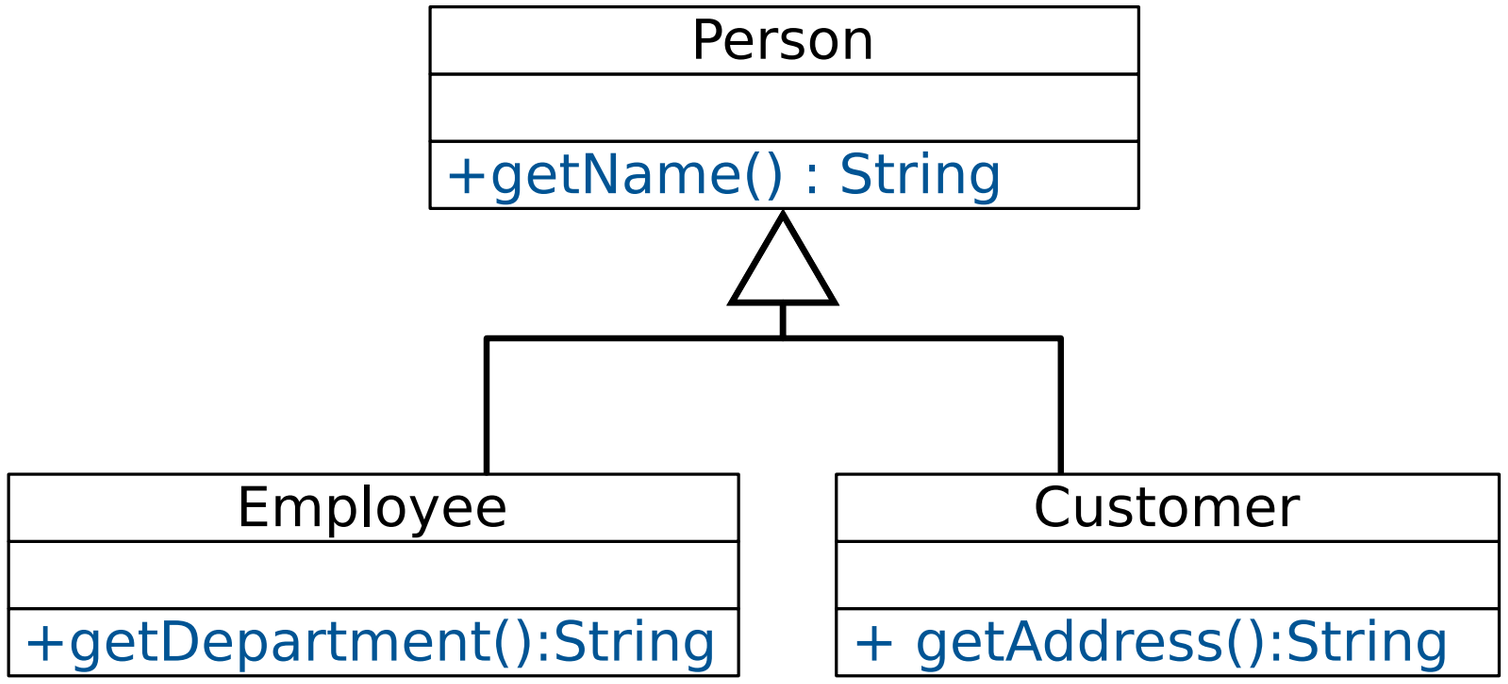


Desired Usage of the Framework

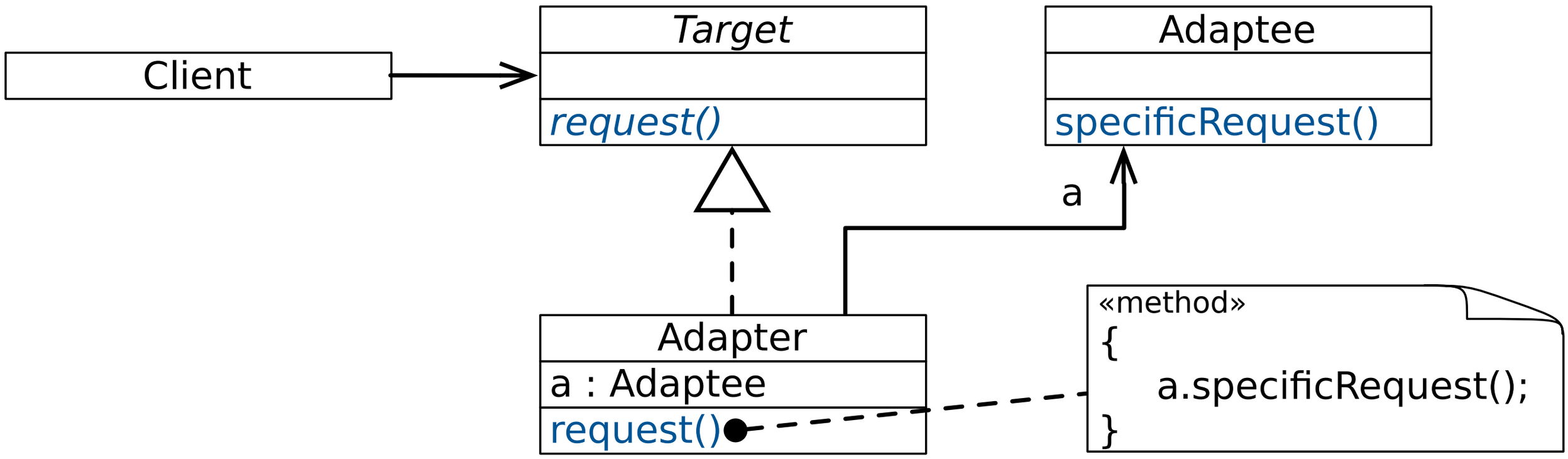


Adapting the Framework

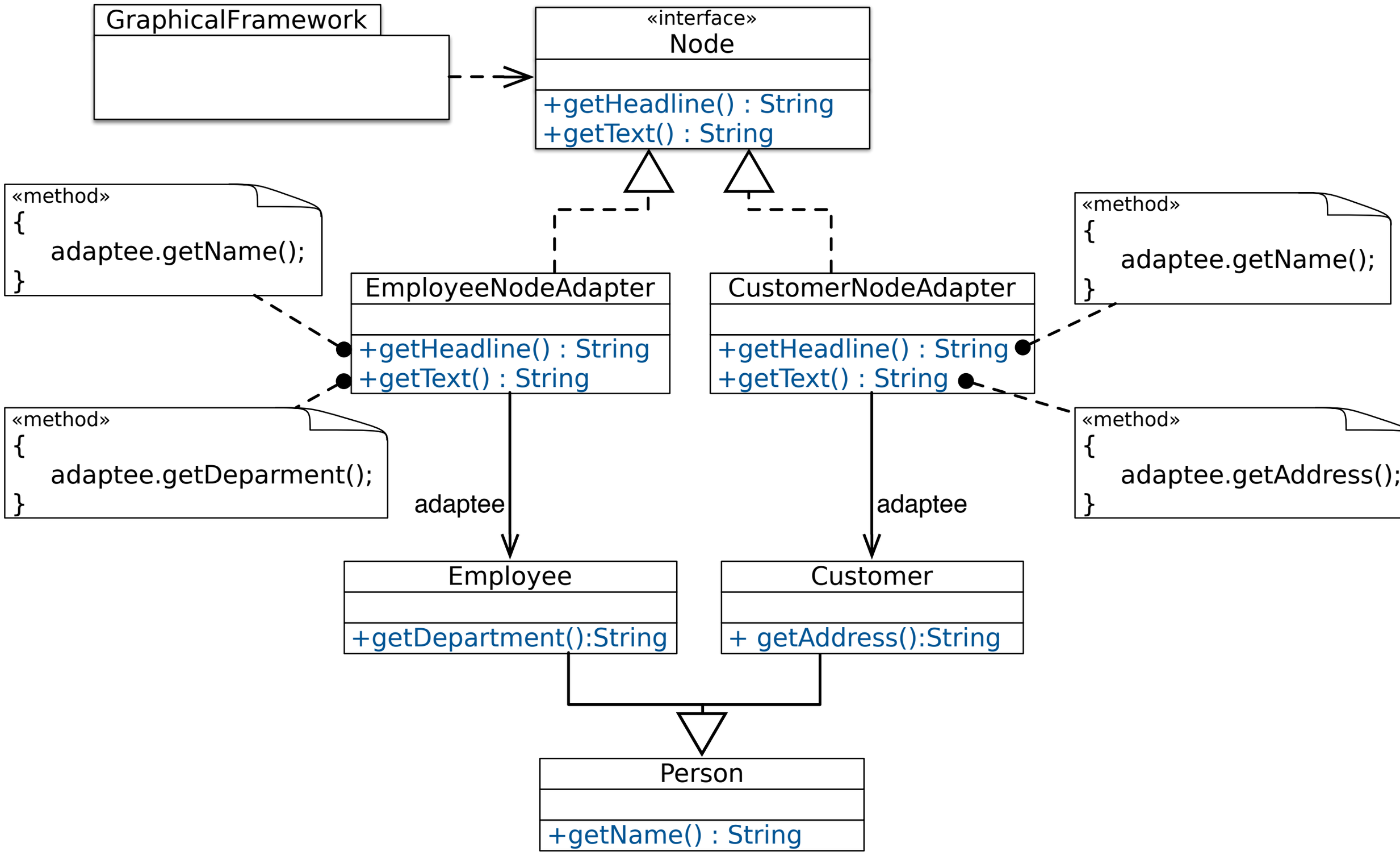




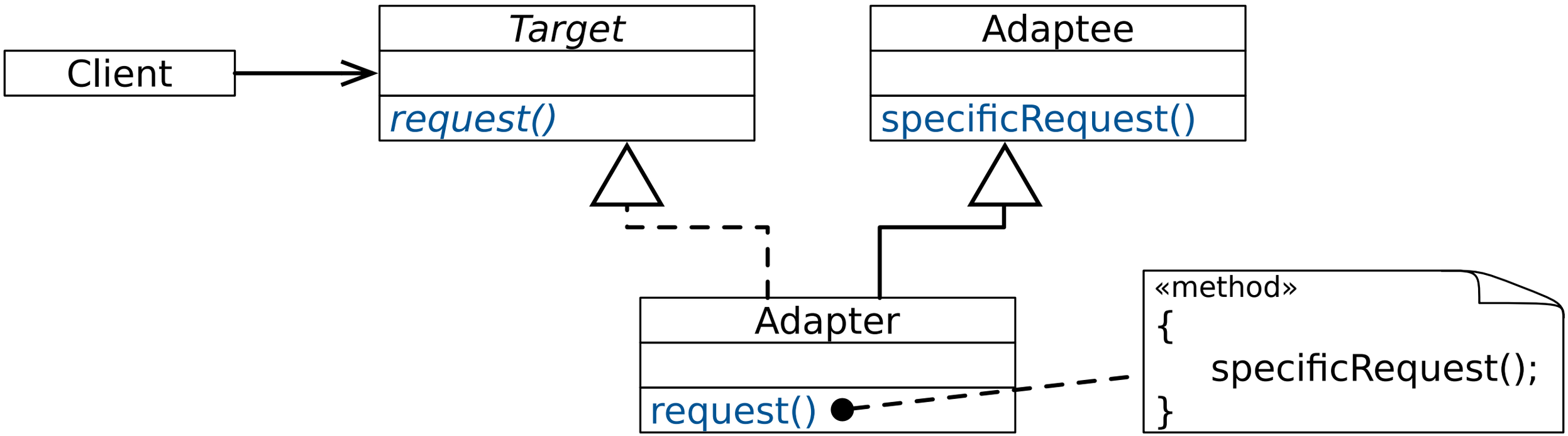
Object Adapter



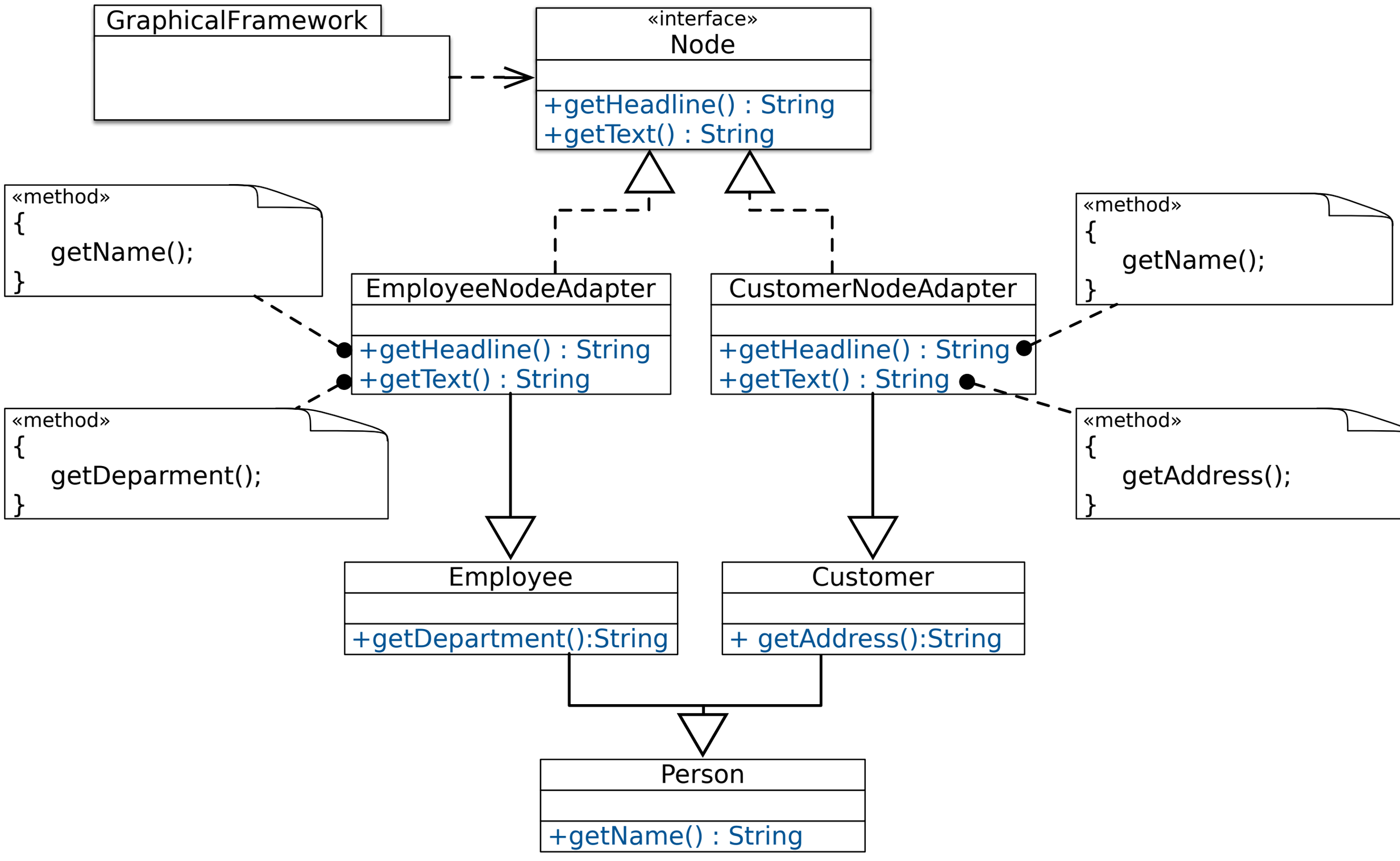
Using Object Adapter



Class Adapter



Using Class Adapter



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.

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
    }
}
```

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?

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 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
    }
}
```

Implementing a `repeatAndStore` method.

```
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]())
    }
          val nanos_3: nanos_1.type =
    imp     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
    }
}
```