Modularisierung und Softwarearchitekturen

O.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Pree

Fachbereich Informatik
cs.uni-salzburg.at

UNIVERSITÄT SALZBURG

Inhalt

- Der Begriff Softwaremodul (= Softwarekomponente)
- Erwünschte Eigenschaften von Modulen
- Ausprägungen von Modulen (ADS, ADT)
- Beschreibung von Softwarearchitekturen
- Analyse von Softwarearchitekturen
- Mehrdimensionale Modularisierung durch Aspektorientierte Programmierung (AOP)

Der Begriff Softwaremodul (= Softwarekomponente)

Definition

Wir definieren einen *Modul (Softwarekomponente)* als *ein Stück Software mit einer Programmierschnittstelle.*

Man unterscheidet zwischen der **Schnittstelle eines Moduls und dessen Implementierung.**

Das mögliche Zusammenspiel mehrerer Module ist durch ihre Programmierschnittstellen festgelegt. Genau genommen ist zwischen zwei Arten von Schnittstellen zu unterscheiden: Die Programmierschnittstelle ist die ***Exportschnittstelle***, die angibt, welche Operationen und Daten ein Modul anderen Modulen zur Verfügung stellt. Die ***Importschnittstelle*** gibt an, was ein Modul von anderen Modulen benutzt. Wenn wir nachfolgend nicht explizit zwischen Export- und Importschnittstelle unterscheiden, meinen wir mit Schnittstelle die Exportschnittstelle.

Modul als Mittel zur Abstraktion

M. Reiser und N. Wirth (1992) beschreiben das Modulkonzept wie folgt:
It provides mechanisms for:

(1) structuring of a program into independent units;

(2) the declaration of variables that keep their value for the duration the module is active (that is, in memory) – these variables are called global *to the module;*

(3) export of variables *and procedures to be used in other modules.*

The module therefore provides the facilities for abstractions.

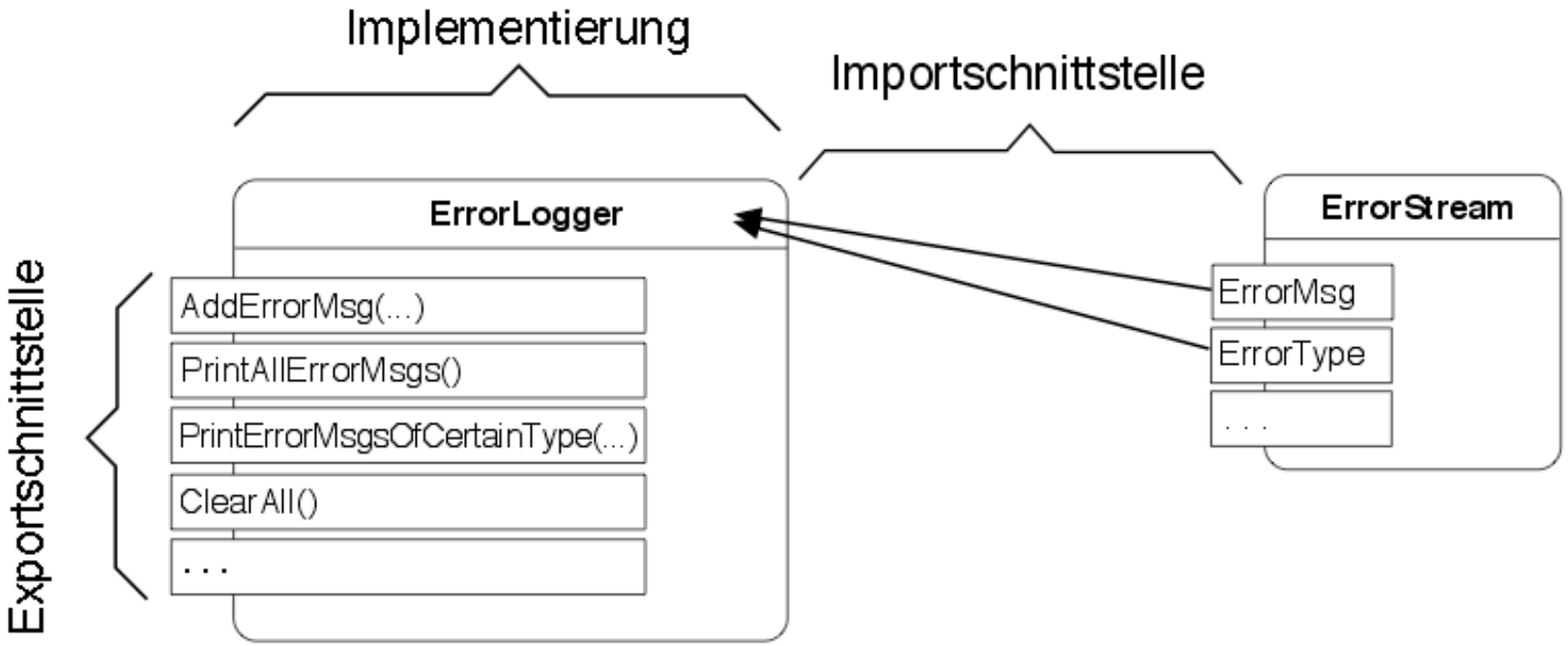
Modul als Ausdrucksmittel zum Modellieren

Durch die geschickte Zusammenfassung von Funktionen, Prozeduren und Daten können Abstraktionen gebildet werden, die den Entitäten der realen Welt, die es in einem Softwaresystem abzubilden gilt, entsprechen.

Beispiele dafür sind:

- ein Bankkonto,
- die GPS-Navigationseinheit eines Helikopters,
- die Dateienorganisation eines PCs oder die
- Interaktionselemente von grafischen Benutzungsschnittstellen.

Beispiel: Modul zum Speichern von Fehlermeldungen



Modulschnittstelle von ErrorLogger (angelehnt an Modula-2)

```
DEFINITION MODULE ErrorLogger;
  FROM ErrorStream
     IMPORT ErrorMsg, ErrorType; /* Importschnittstelle */
  PROCEDURE AddErrorMsg(↓em: ErrorMs);
  PROCEDURE PrintAllErrorMsgs();
  PROCEDURE PrintErrorMsgsOfCertainType(↓et: ErrorType);
  PROCEDURE ClearAll();
  . . .
END ErrorLogger.
```

Modulimplementierung von ErrorLogger (angelehnt an Modula-2)

```
IMPLEMENTATION MODULE ErrorLogger;
  VAR errors: ARRAY [0..cMaxNoOfStoredErrors–1] OF ErrorMsg;
  PROCEDURE AddErrorMsg(↓em: ErrorMs)
  BEGIN
     ... /* füge em am nächsten freien Platz im Feld errors ein */
  END AddErrorMsg;
  . . .
END ErrorLogger.
```

ARRAY => statisch festegelegte Obergrenze für die Anzahl der speicherbaren Fehlermeldungen

Vorteil der Trennung von Schnittstelle und Implementierung

Die Implementierung des Moduls kann verbessert oder geändert werden, ohne dass die Schnittstelle zu ändern ist.

Zum Beispiel könnte im Modul ErrorLogger die ARRAY-Struktur durch eine verkettete Liste ersetzt werden.

Das Konzept von *Information Hiding* (siehe im nachfolgenden Abschnitt) trägt zu einer stabilen Schnittstelle eines Moduls bei.

Erwünschte Eigenschaften von Modulen

Stabile und verständliche Modulschnitt-stellen durch Information Hiding (I)

Das Entwurfsprinzip *Information Hiding* geht auf David L. Parnas (1972) zurück.

Demnach sind Module so zu entwerfen, dass die Datenstrukturen vor dem Benutzer verborgen werden.

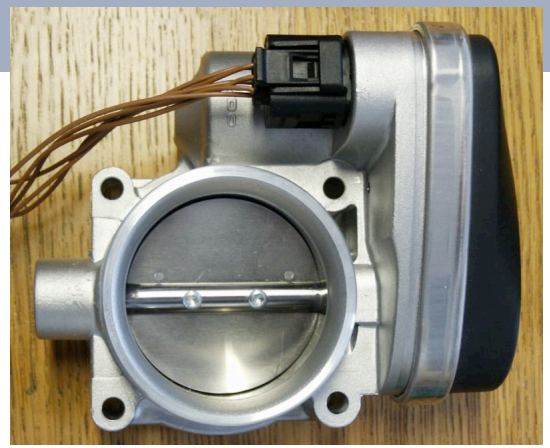
Der Zugriff auf Daten und ihre Manipulation ist nur über Zugriffsprozeduren, die in der Modulschnittstelle angeführt sind, möglich.

Stabile und verständliche Modulschnitt-stellen durch Information Hiding (II)

Eine Verallgemeinerung des Information Hiding Prinzips ist die Forderung, dass beim Entwurf von Modulen darauf geachtet wird, dass **möglichst viele Details der Implementierung hinter der Modulschnittstelle verborgen werden**, um den Benutzer eines Moduls nicht mit unnötigen Details und damit mit Komplexität zu konfrontieren.

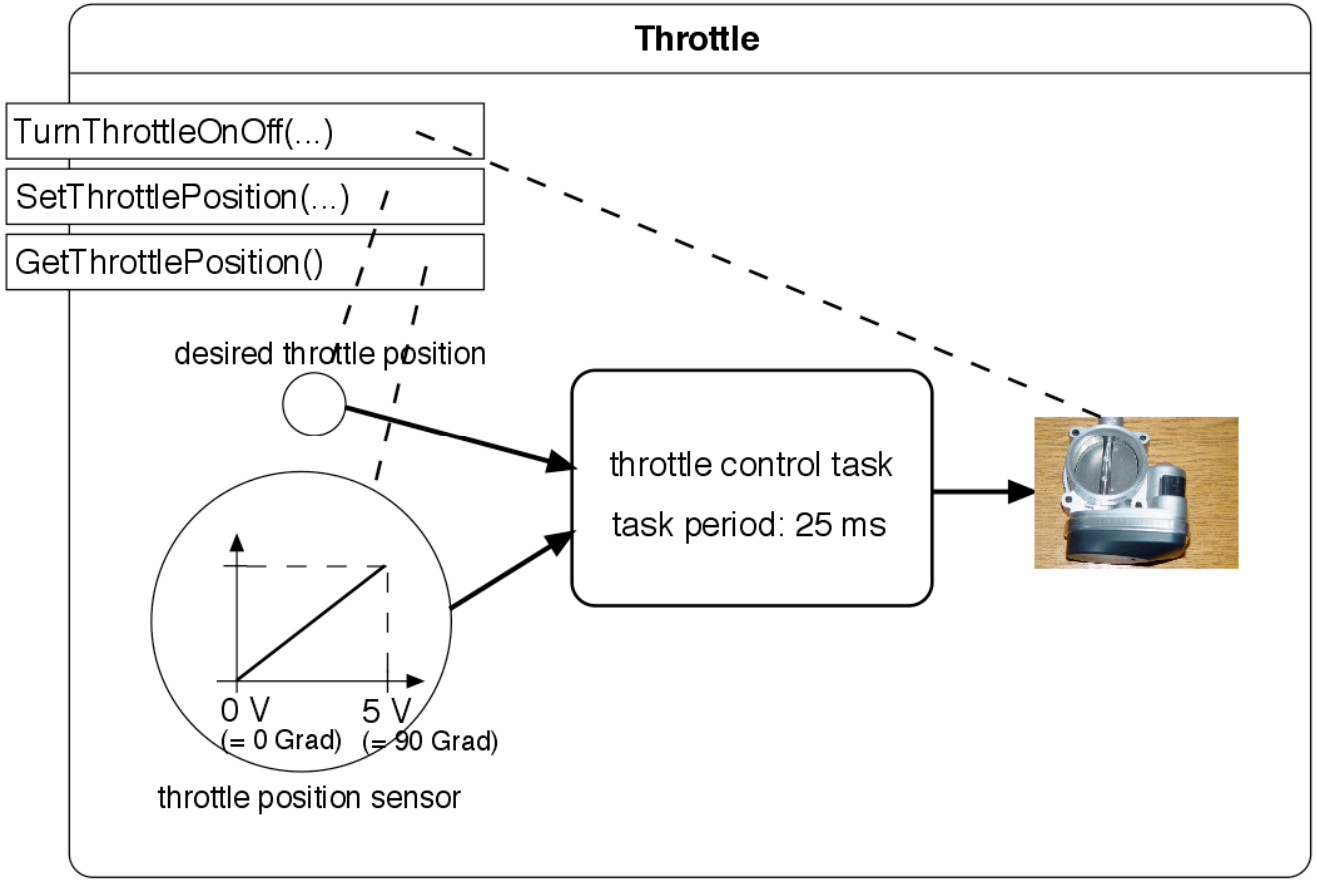
Das kann über das Verbergen der Datenstrukturen hinausgehen.

Beispiel: Modul Throttle zum Ansteuern einer Drosselklappe (I)



```
interface Throttle {
  bool TurnThrottleOnOff(bool onOff);
  bool SetThrottlePosition(float angle);  // 0..90 Grad
  float GetThrottlePosition();
}
```

Beispiel: Modul Throttle zum Ansteuern einer Drosselklappe (II)



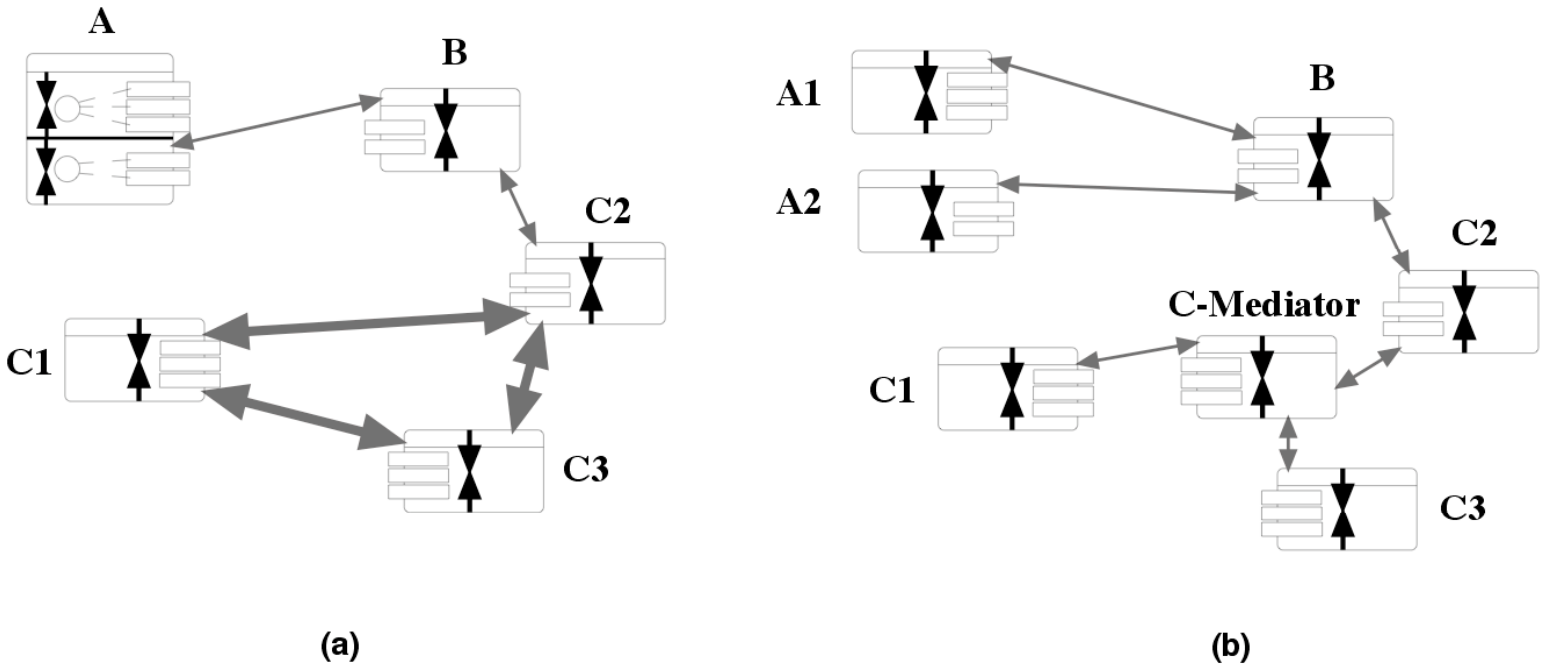
Balance zwischen Kopplung und Kohäsion (I)

- Unter *Modulkopplung* verstehen wir die Abhängigkeit von und Interaktion zwischen Modulen, die einerseits statisch durch die Importschnittstelle festgelegt wird und andererseits dynamisch durch die Aufrufe von Prozeduren und Funktionen beziehungsweise durch den Zugriff auf Daten beeinflusst wird. **Die Modulkopplung soll minimiert werden.**
- Unter *Modulkohäsion* verstehen wir den Zusammenhalt eines Moduls. Der ist dann gegeben, wenn ein Modul **Operationen und Daten zu einer Einheit** zusammenfasst, die logisch zusammengehören. **Die Modulkohäsion soll maximiert werden.**

Balance zwischen Kopplung und Kohäsion (II)

- Man könnte die Kopplung auf ein Minimum bringen, indem man einen einzigen Modul definiert. Wenn nur ein Modul vorhanden ist, ist dieser auch mit keinem anderen Modul gekoppelt, wodurch die Kopplung minimal ist. Allerdings hätte solch ein Modul, außer bei einem sehr einfachen Sachverhalt, auch eine minimale Kohäsion, da sämtliche Systemaspekte bunt zusammengewürfelt in einem Modul zusammengefasst sind.
- Das andere Extrem wäre, dass jede Funktion und Prozedur in einem separaten Modul gekapselt wird. Das würde zu einer nicht erwünschten, engen Kopplung der Module führen und über eine Modulkohäsion im eigentlichen Sinne kann man bei solch einem Modularisierungansatz nicht mehr reden.

Beispiel zur Verbesserung der Modularisierung



Regeln zur Maximierung der Kohäsion (I)

- Die **Datenstrukturen** (Instanzvariablen bei Klassen) und die **Operationen** (= Funktionen/Prozeduren/Methoden) **sollen in enger Beziehung zueinander stehen**. Mehrere Gruppen von Operationen, die jeweils auf verschiedenen Daten arbeiten, sind ein Indikator dafür, dass verschiedene Aspekte, die in keinem oder zu geringem logischen Zusammenhang stehen in einem Modul zusammengefasst worden sind. Ein Aufsplitten des Moduls trägt zur Verbesserung der Kohäsionseigenschaften bei.
- **Die Modulschnittstelle soll keine redundanten Operationen enthalten.** Das ist dann gegeben, wenn für eine Funktionalität mehrere Operationen angeboten werden, die sich nur geringfügig unterscheiden. Außerdem sollen die Operationen so konzipiert werden, dass sie mit einer geringen Anzahl von Parametern das Auslangen finden.

Regeln zur Maximierung der Kohäsion (II)

- Es soll ein **konsistentes und ausdrucksstarkes Namensschema** verwendet werden. Das gilt insbesondere für die Namen von Modulen sowie für die Namen der in den Schnittstellen definierten Operationen. Beispiele für gut gewählte und konsistente Namensschemata finden sich bei modernen objektorientierten Klassenbibliotheken wie den .NET-Bibliotheken und den Java-Bibiliotheken.
- **Globale Datenobjekte sollen vermieden werden.**

Heuristik zur Erreichung einer adäquaten Kopplung

- Die einzelnen Module sollen für sich gut verstehbar sein. Anders formuliert soll es nicht notwendig sein, für das Verstehen eines Moduls weitere Module betrachten zu müssen, um alle Eigenschaften des betrachteten Moduls zu erfassen.
- Eine analoge Aussage gilt für das Testen von Modulen. Je größer das Ensemble von Modulen ist, die benötigt werden, um einen bestimmten Modul testen zu können, desto grösser ist die Kopplung des jeweiligen Moduls mit anderen Modulen.

Beurteilung der Qualität einer Modularisierung (I)

- Die Software-Architektur-Analyse-Methode (SAAM) zielt darauf ab, die beiden Eigenschaften Kopplung und Kohäsion mit geringem Aufwand auf Adäquatheit zu prüfen.
- Es gibt aber keine allgemein gültigen Metriken, um die Ausprägung der beiden Eigenschaften objektiv beurteilen zu können.

Beurteilung der Qualität einer Modularisierung (II)

- Die richtige Balance von Kopplung und Kohäsion bei der Strukturierung von Software zu finden ist daher eine Kunst, die hohe Qualifikation und viel Erfahrung erfordert.
- Ähnlich wie in der Architektur können Beispiele helfen, ein Bewusstsein und ein Gespür für gute Modularisierung zu schaffen.
- Im Gegensatz zur Architektur sind leider nur wenige gute Beispiele von Softwarearchitekturen verfügbar: entweder es gibt sie nicht oder sie sind nicht dokumentiert beziehungsweise nicht veröffentlicht.

Ausprägungen von Modulen

Modul als Abstrakte Datenstruktur (ADS)

- Ein Modul wird definiert, ohne damit einen Typ festzulegen.
- Beispiel: Modul ErrorLogger wie zuvor definiert

```
DEFINITION MODULE ErrorLogger;
  FROM ErrorStream
     IMPORT ErrorMsg, ErrorType; /* Importschnittstelle */
  PROCEDURE AddErrorMsg(↓em: ErrorMs);
  PROCEDURE PrintAllErrorMsgs();
  PROCEDURE PrintErrorMsgsOfCertainType(↓et: ErrorType);
  PROCEDURE ClearAll();
  . . .
END ErrorLogger.
```

Modul als Abstrakter Datentyp (ADT) I

- Ein Modul wird als Typ definiert, von dem beliebig viele Instanzen gebildet werden können.
- In Modula-2: Modul ErrorLogger mit *opaque Type* ErrorLogger

```
DEFINITION MODULE ErrorLoggers;
        FROM ErrorStream IMPORT ErrorMsg, ErrorType;
        TYPE ErrorLogger;
        PROCEDURE NewErrorLogger(↑VAR el: ErrorLogger);
        PROCEDURE AddErrorMsg(↓↑VAR el: ErrorLogger,
                              ↓em: ErrorMsg);
        PROCEDURE PrintAllErrorMsgs(↓el: ErrorLogger);
        PROCEDURE PrintErrorMsgsOfCertainType(↓el: ErrorLogger,
                                              ↓et: ErrorType);
        PROCEDURE ClearAll(↓↑VAR el: ErrorLogger);
        . . .
END ErrorLoggers.
```

Modul als Abstrakter Datentyp (ADT) II

Erzeugen von Instanzen mit NewErrorLogger:

```
calcErrors, inputErrors: ErrorLogger;
NewErrorLogger(↑calcErrors);
NewErrorLogger(↑inputErrors);
AddErrorMsg(↓↑calcErrors, ↓any message);
. . .
PrintAllErrorMsgs(↓calcErrors);
PrintAllErrorMsgs(↓inputErrors);
```

ADT in OO Sprachen (zB in C#) I

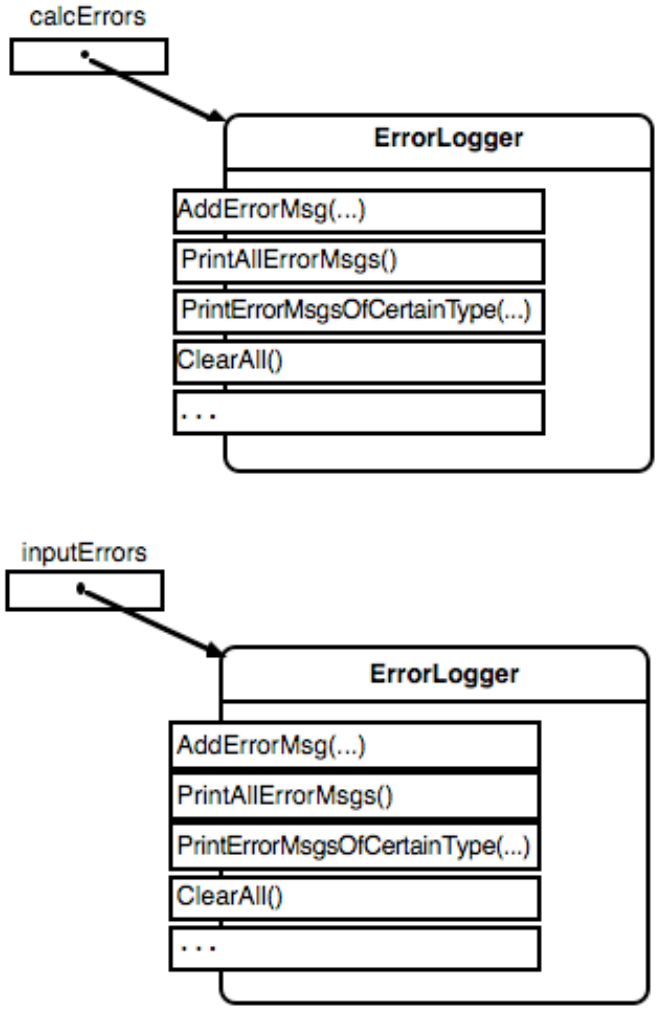
```
using System;
using System.Collections;
namespace ErrorLibrary {
   public class ErrorLogger {
         private IList errorList;

         public ErrorLogger() {
           errorList= new ArrayList();
         }

         public void AddErrorMsg(ErrorMsg em) {
           errorList.Add(em);
         }
         . . .
   }
}
```

Schnittstelle und Implementierung in einer Datei!

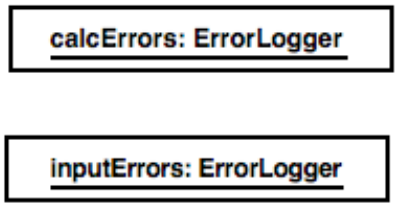
ADT in OO Sprachen (zB in C#) II

```
ErrorLogger calcErrors, inputErrors;
calcErrors = new ErrorLogger();
inputErrors = new ErrorLogger();
. . .
try {
        ...
} catch (FloatingPointException fe) {
        calcErrors.AddErrorMsg(new ErrorMsg("floating point exception“,
                                ...);    // ... bedeutet weitere Informationen zu fe
}
```

ADT in OO Sprachen III

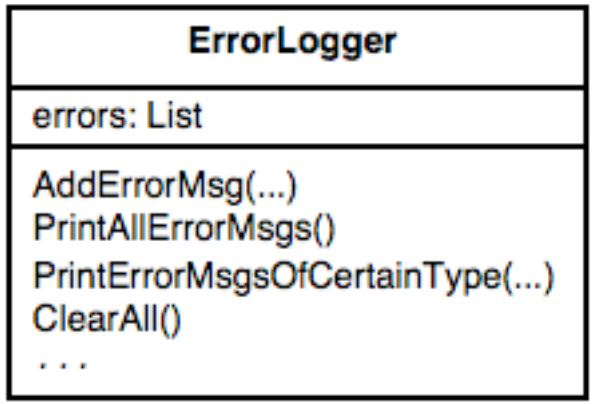


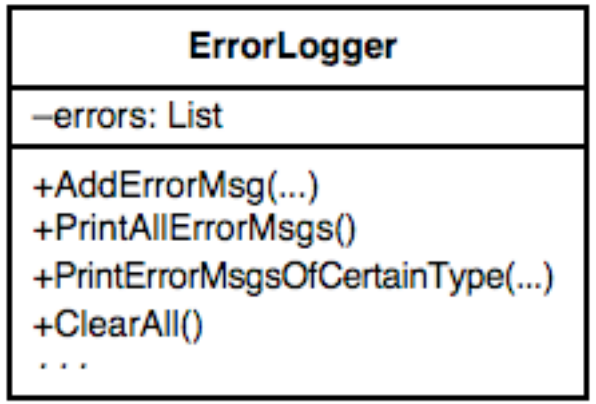
in UML:



ADT in OO Sprachen IV

als UML-Klassendiagramm:





Zugriffsrechte (+/–)

Definition von ADS und ADT in Programmiersprachen

- Oberon(-2): unterstützt beide Konzepte durch Sprachkonstrukte
- Java und C#: Klassen zur Definition von ADT. Syntaktische Unterstützung der Definition von ADS durch statische Instanzvariablen und Methoden.

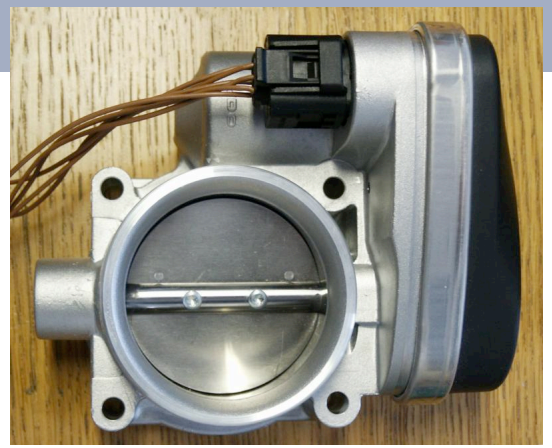
  Packages und Namespaces lassen mehrere Klassen zu einer Einheit zusammenfassen.

Module in gängigen Komponentenstandards

- Komponentenstandards unterscheiden sich unter anderem in der Syntax, wie die Schnittstelle von Komponenten definiert wird:
  - CORBA (Common Object Request Broker Architecture): CORBA-IDL (Interface Description Language); ist eng an C++ angelehnt
  - JavaBeans: Schnittstelle wird in Java definiert
  - Web Services: XML-basierte WSDL (Web Services Description Language)

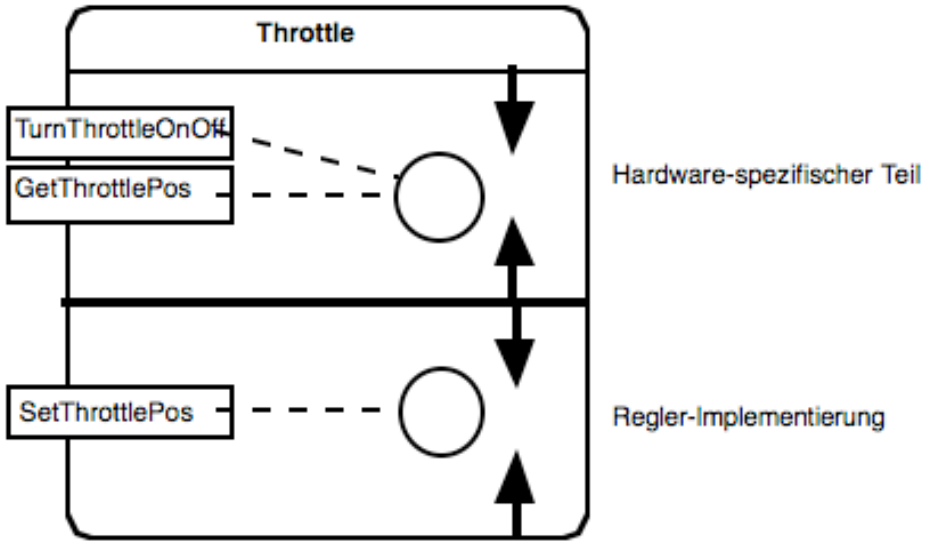
Beispiel: Verbesserung der Kohäsion des Moduls Throttle

Beispiel: Modul Throttle zum Ansteuern einer Drosselklappe

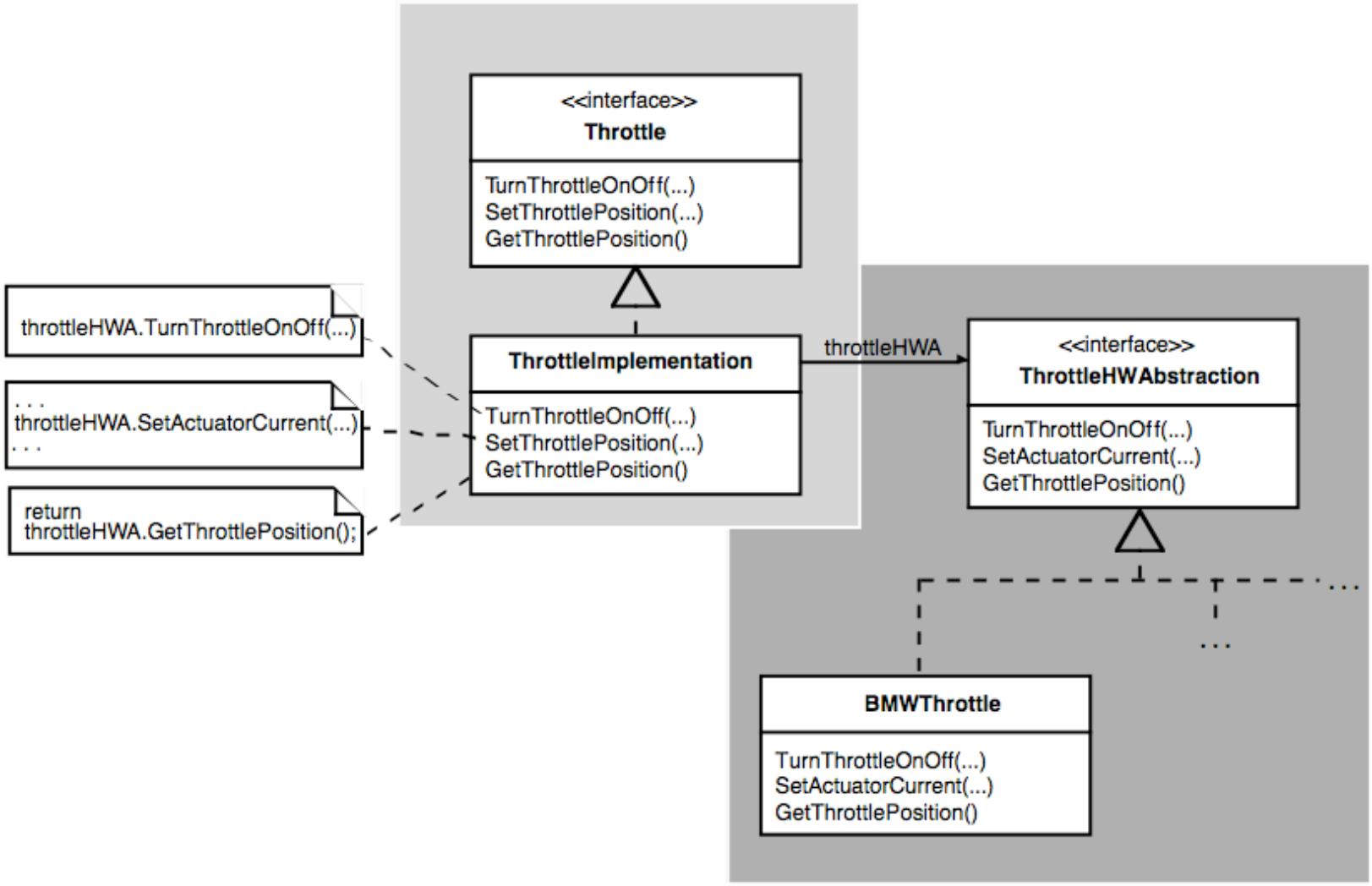


```
interface Throttle {
  bool TurnThrottleOnOff(bool onOff);
  bool SetThrottlePosition(float angle);  // 0..90 Grad
  float GetThrottlePosition();
}
```

geringe Kohäsion:

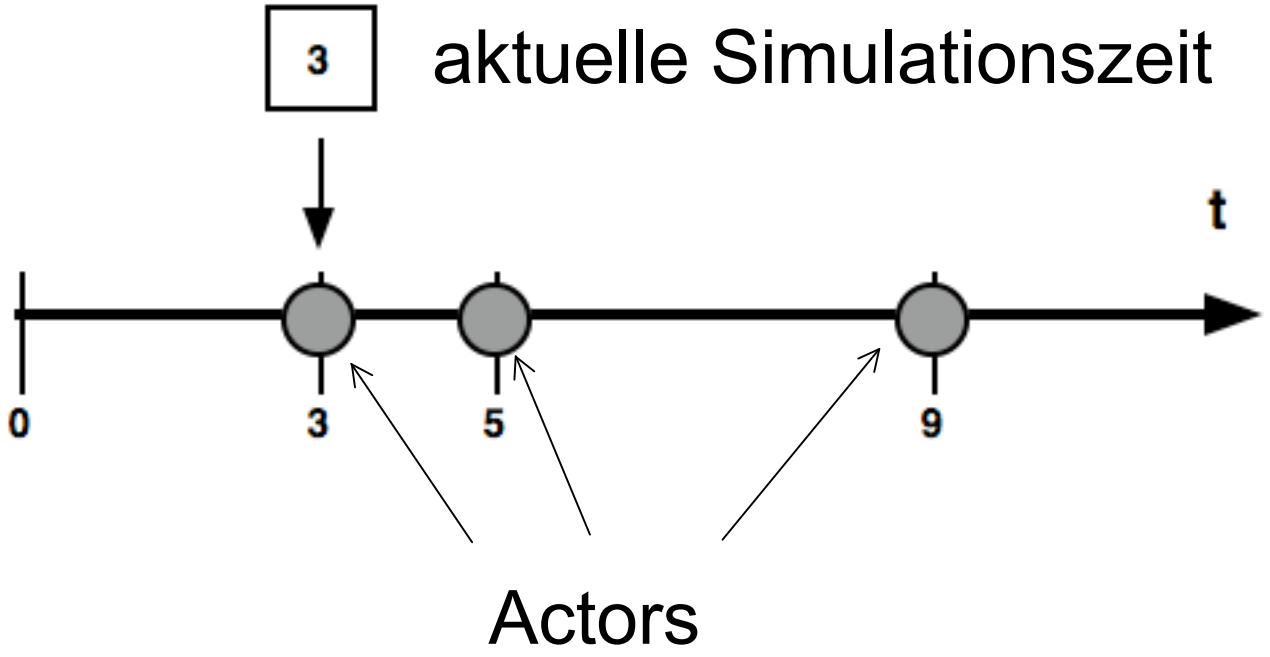


Verbesserung der Kohäsion durch Aufsplittung des Moduls Throttle unter Beibehaltung der Schnittstelle

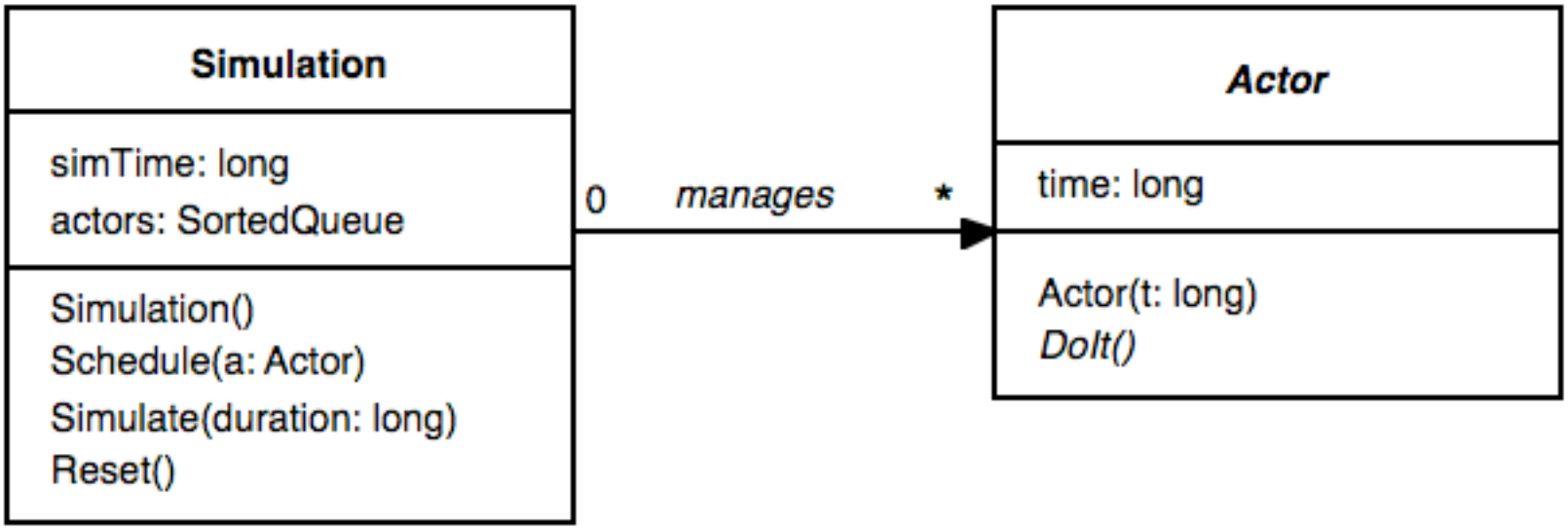


Beispiel: Simulation diskreter Ereignisse

Diskrete Ereignisse auf einer Zeitachse



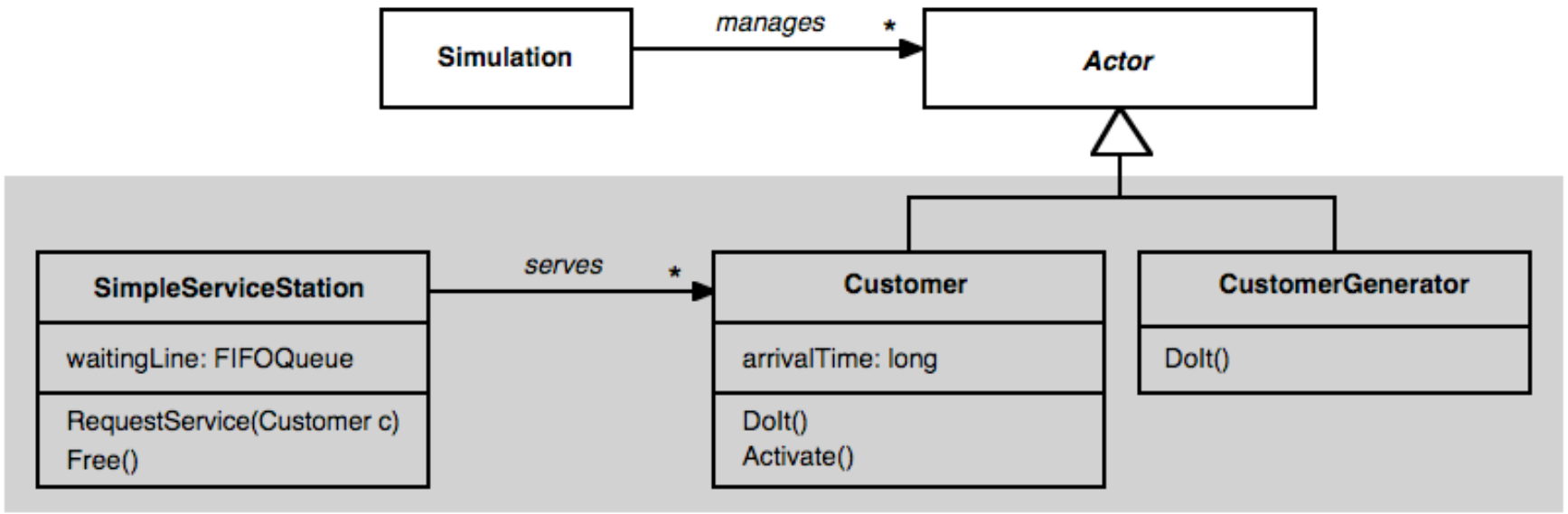
Framework für diskrete Ereignissimulation



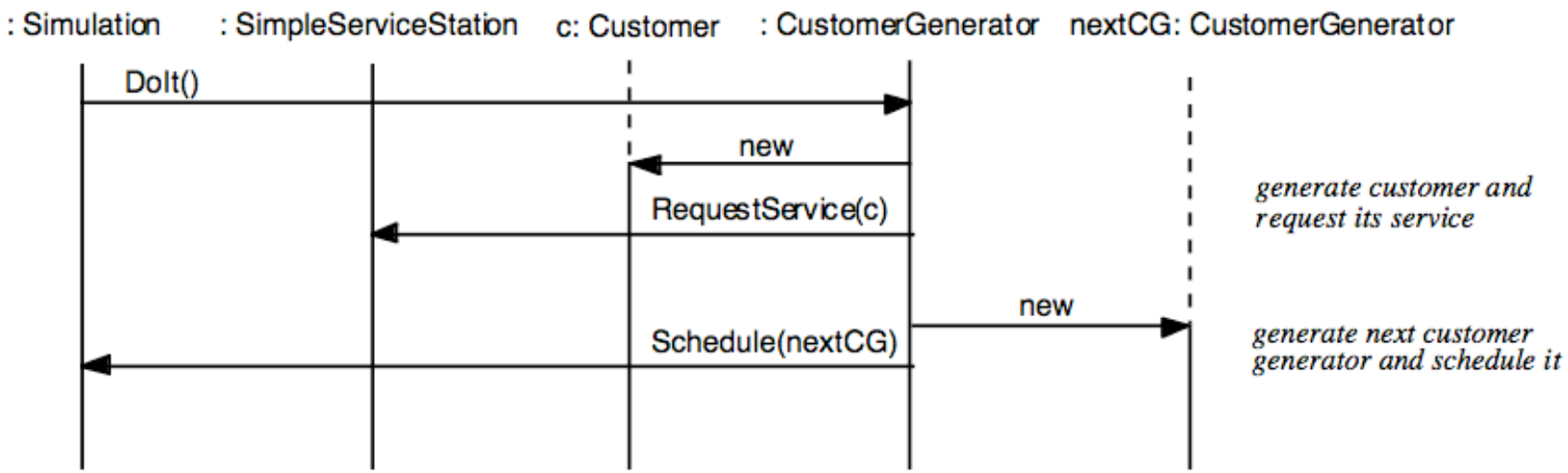
Die C# Implementierung von Simulate()

```
public void Simulate(long duration) {
        long endOfSimulation= simTime + duration;
        do {
          if (actors.Count() != 0) {
                  Actor actor= (Actor) actors.Dequeue();
                   simTime = actor.time;
                  actor.DoIt();
          } else      // no more actors enqueued
                   break;      // exit loop
        } while (simTime <= endOfSimulation);
}
```

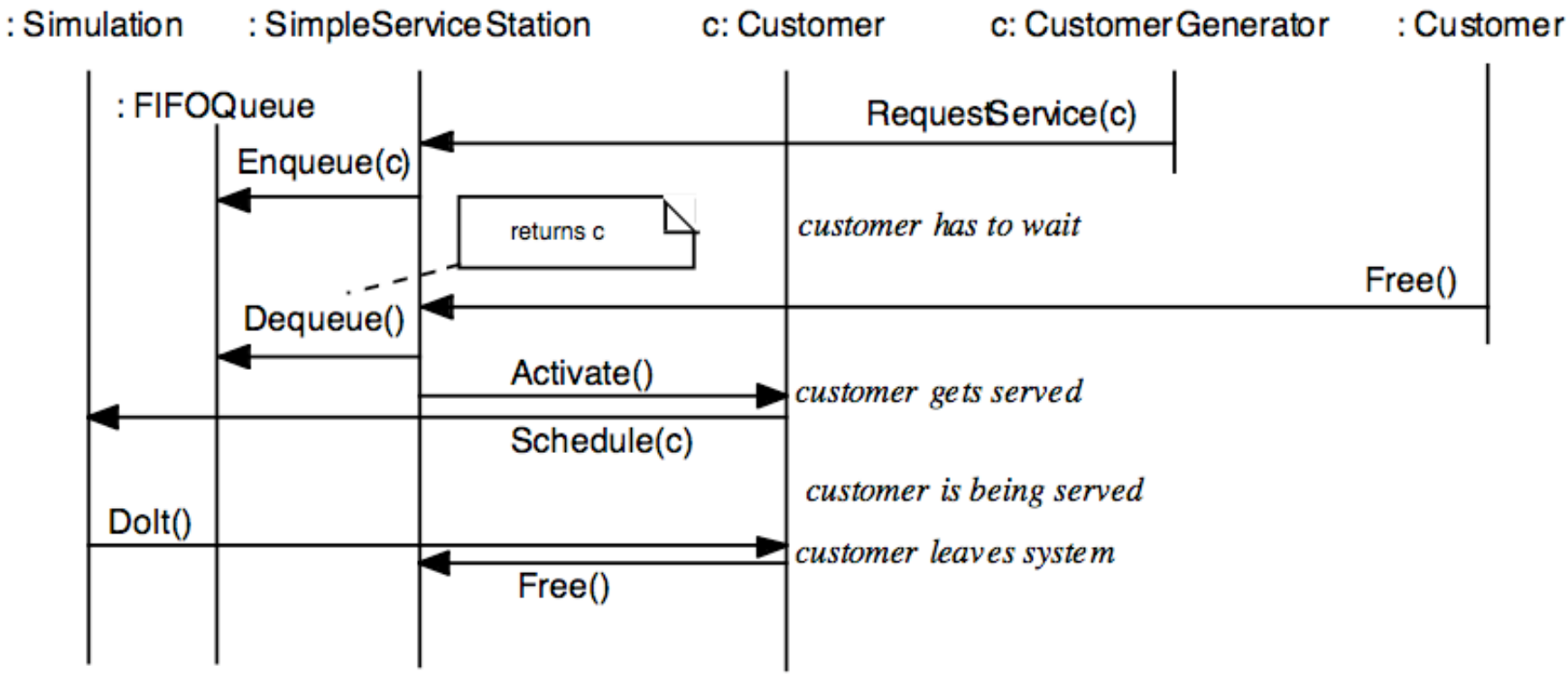
Klassen für die Simulation eines einfachen Bankschalters



DoIt() Methode der Klasse CustomerGenerator



Einreihung eines Kunden in die Warteschlange



Verringerung der Kopplung zwischen ServiceStation und den Aktoren