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.

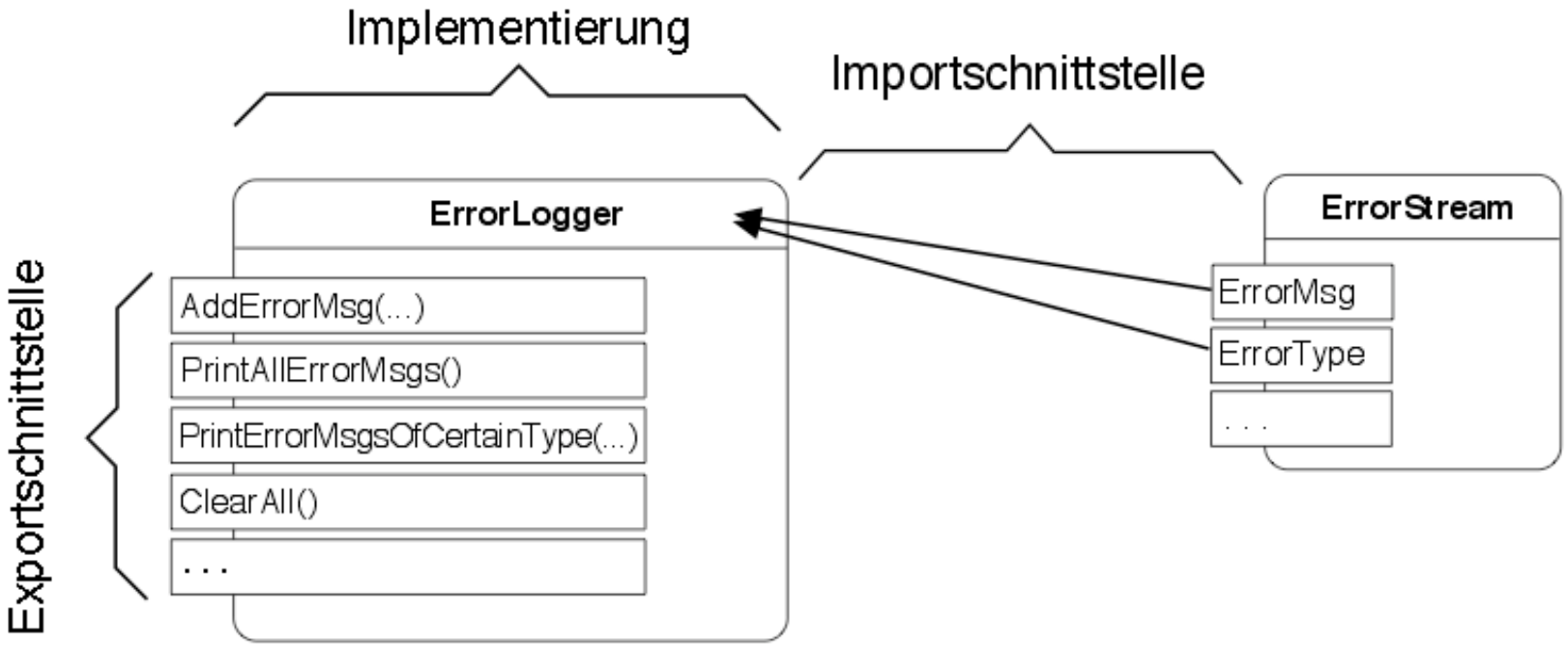
Modul als Ausdrucksmittel zum Modellieren

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.

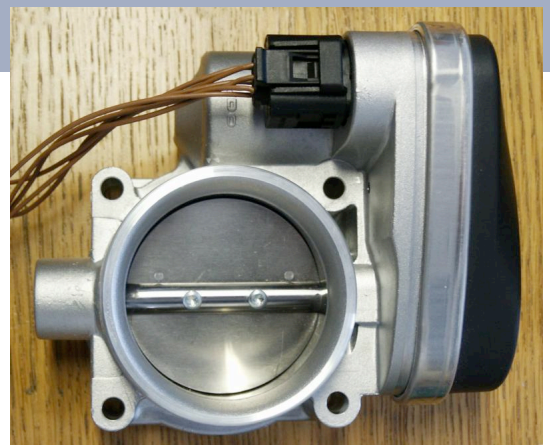
Der Zugriff auf Daten und ihre Manipulation ist nur über Zugriffsprozeduren, die in der Modulschnittstelle angeführt sind, möglich.

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.

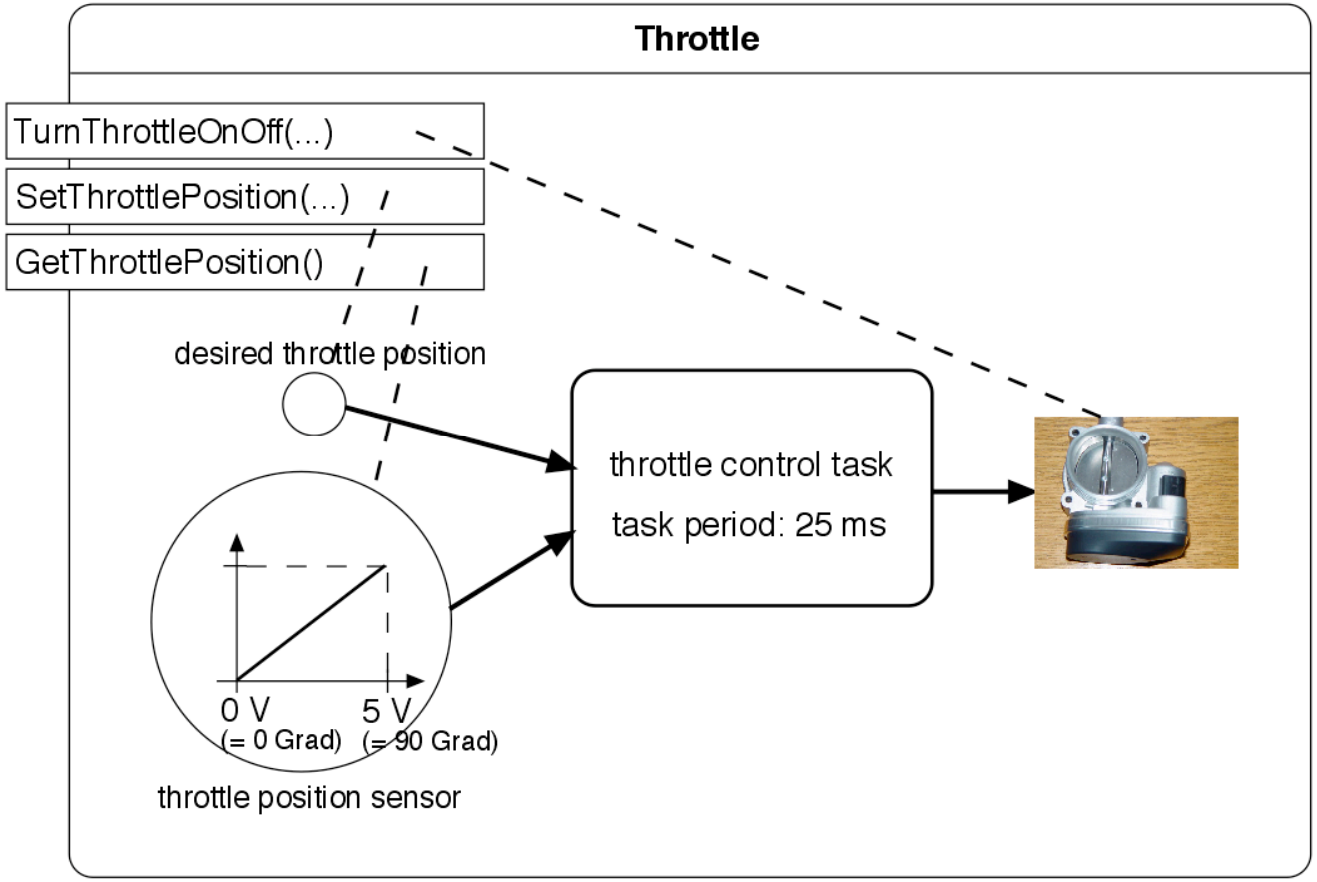
Das kann über das Verbergen der Datenstrukturen hinausgehen.

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)



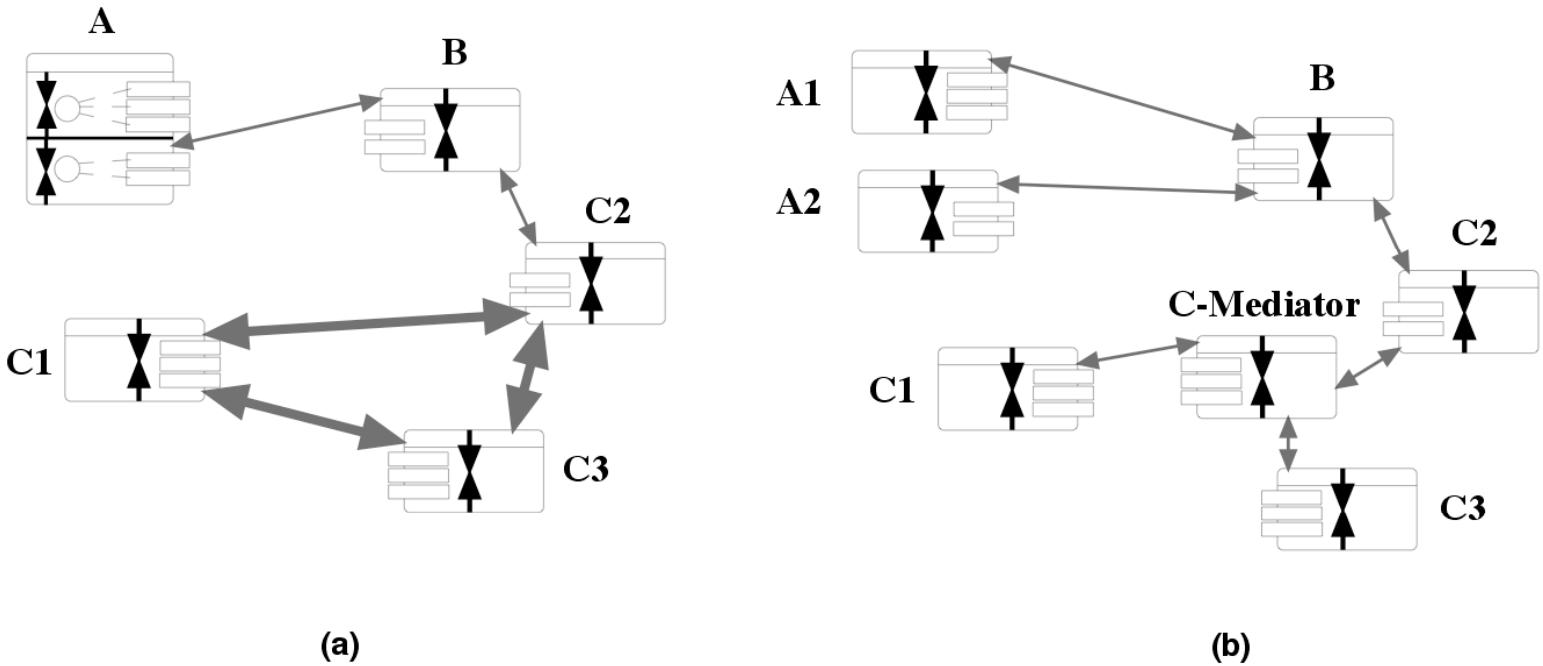
Balance zwischen Kopplung und Kohäsion (I)

- 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

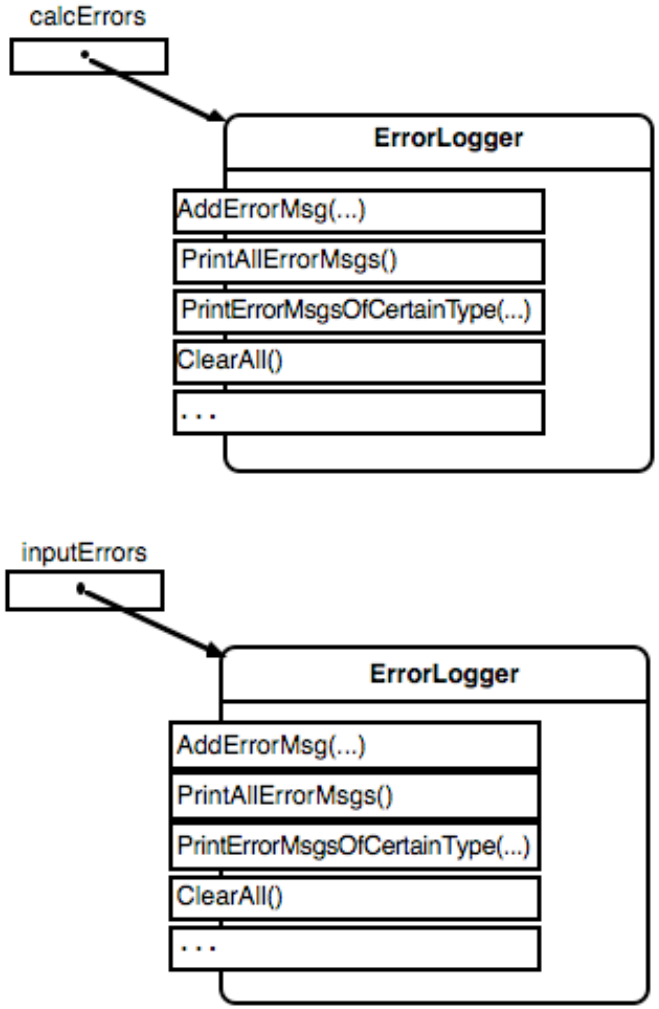
```
using System;
using System.Collections;
namespace ErrorLibrary {
   public class ErrorLogger {
         private IList errorList;

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

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

Schnittstelle und Implementierung in einer Datei!

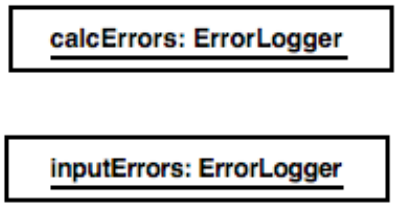
ADT in OO Sprachen (zB in C#) II

```
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

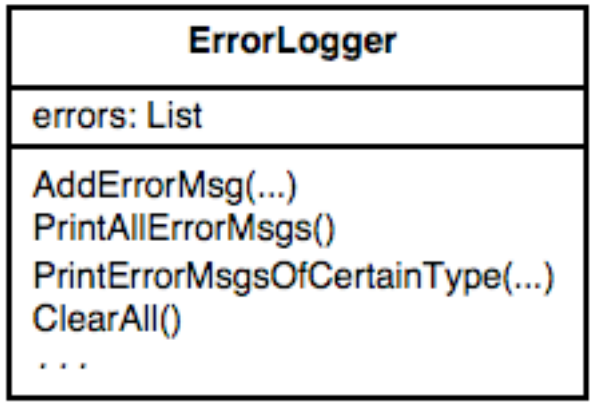


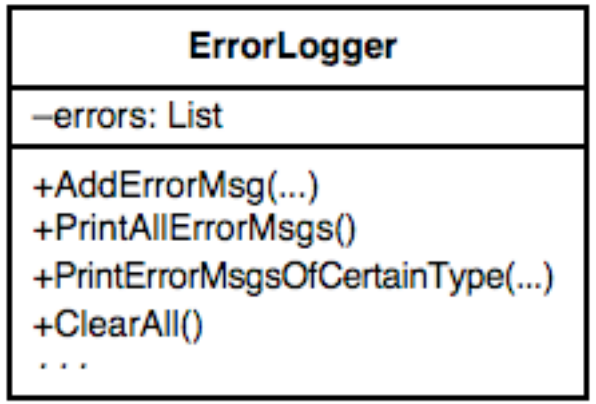
in UML:



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.

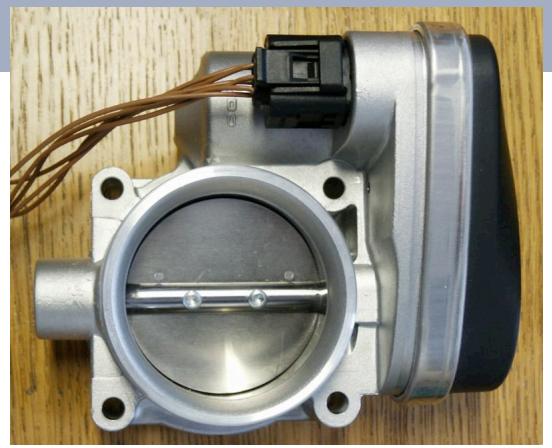
  Packages und Namespaces lassen mehrere Klassen zu einer Einheit zusammenfassen.

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)

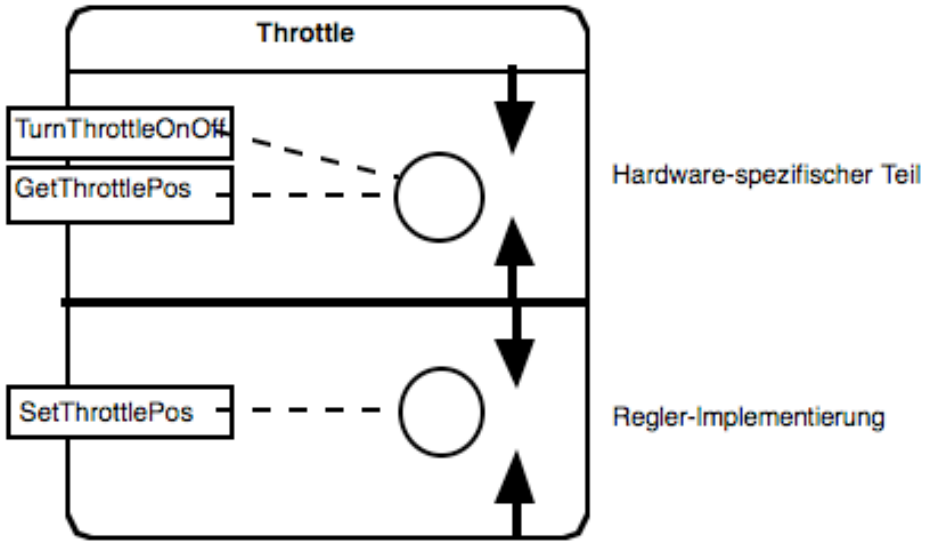
Beispiel: Verbesserung der Kohäsion des Moduls Throttle

Beispiel: Modul Throttle zum Ansteuern einer Drosselklappe

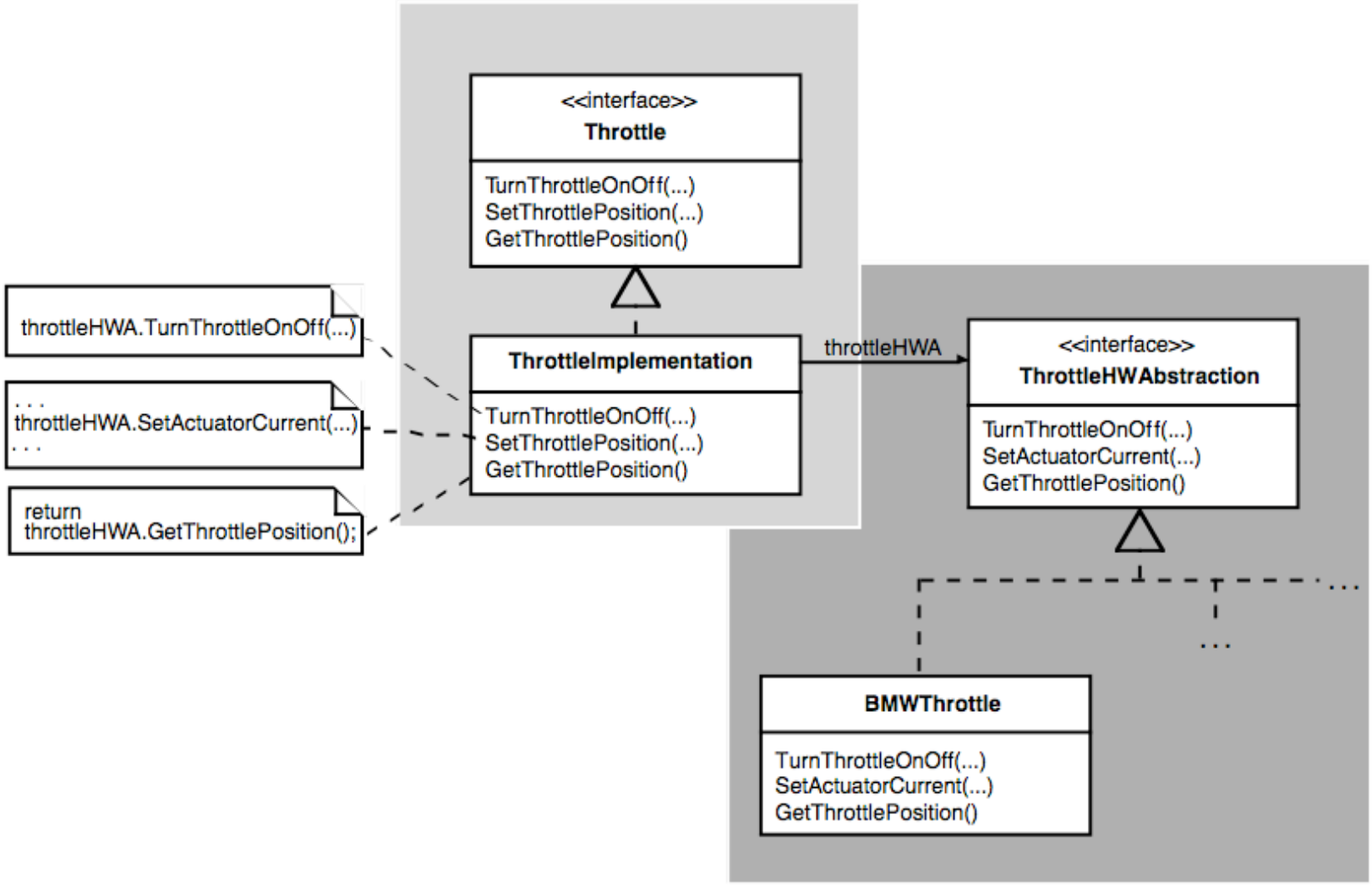


```
interface Throttle {
  bool TurnThrottleOnOff(bool onOff);
  bool SetThrottlePosition(float angle);  // 0..90 Grad
  float GetThrottlePosition();
}
```

geringe Kohäsion:

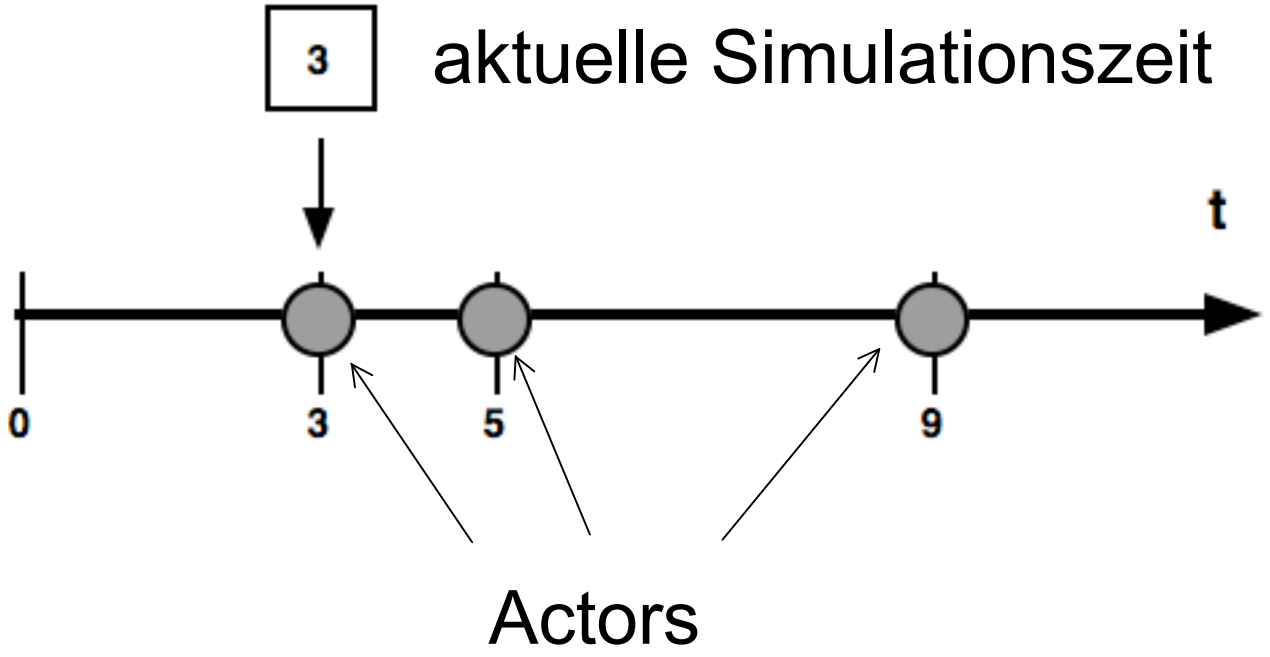


Verbesserung der Kohäsion durch Aufsplittung des Moduls Throttle unter Beibehaltung der Schnittstelle

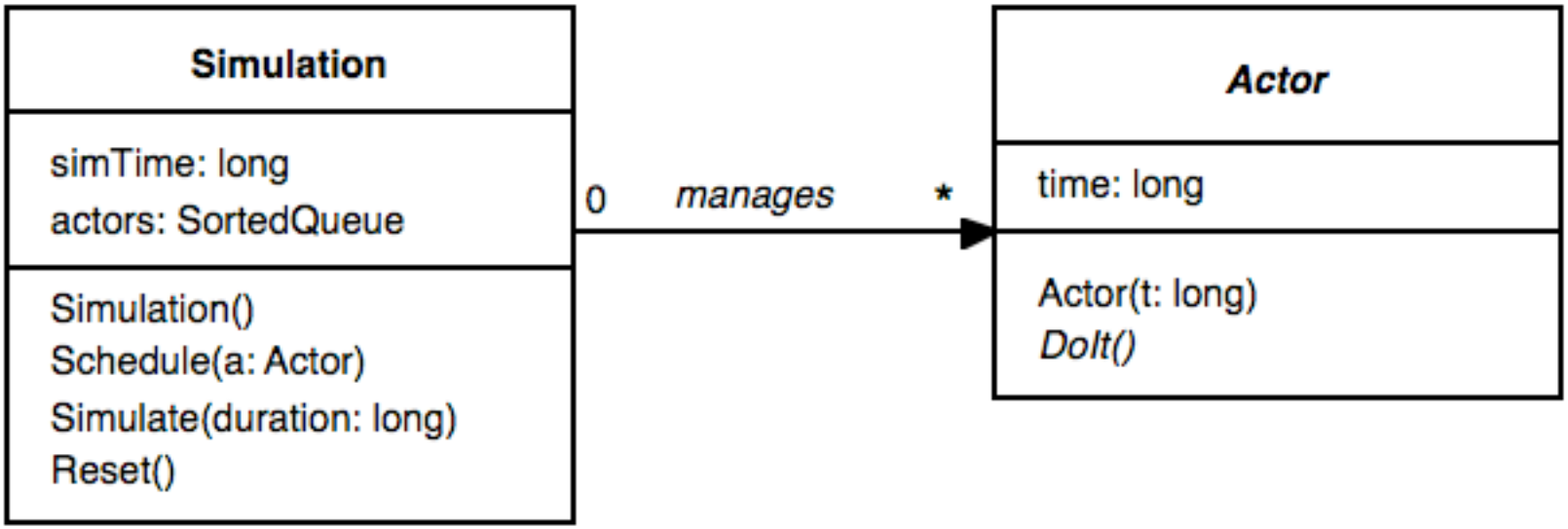


Beispiel: Simulation diskreter Ereignisse

Diskrete Ereignisse auf einer Zeitachse



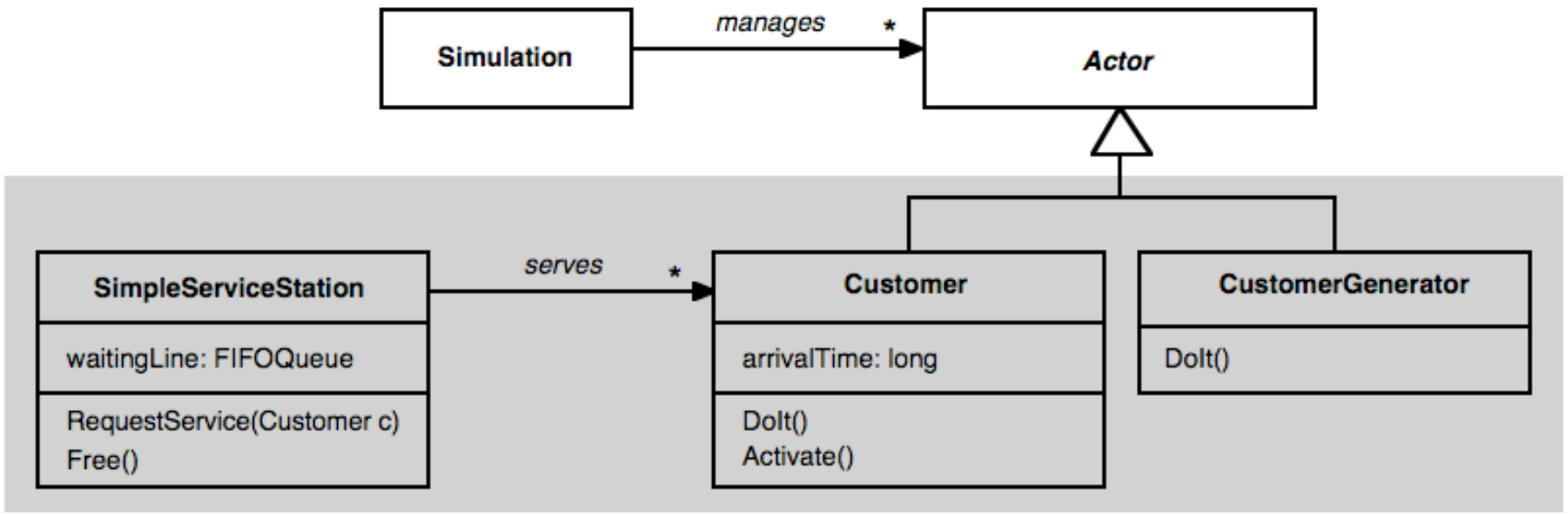
Framework für diskrete Ereignissimulation



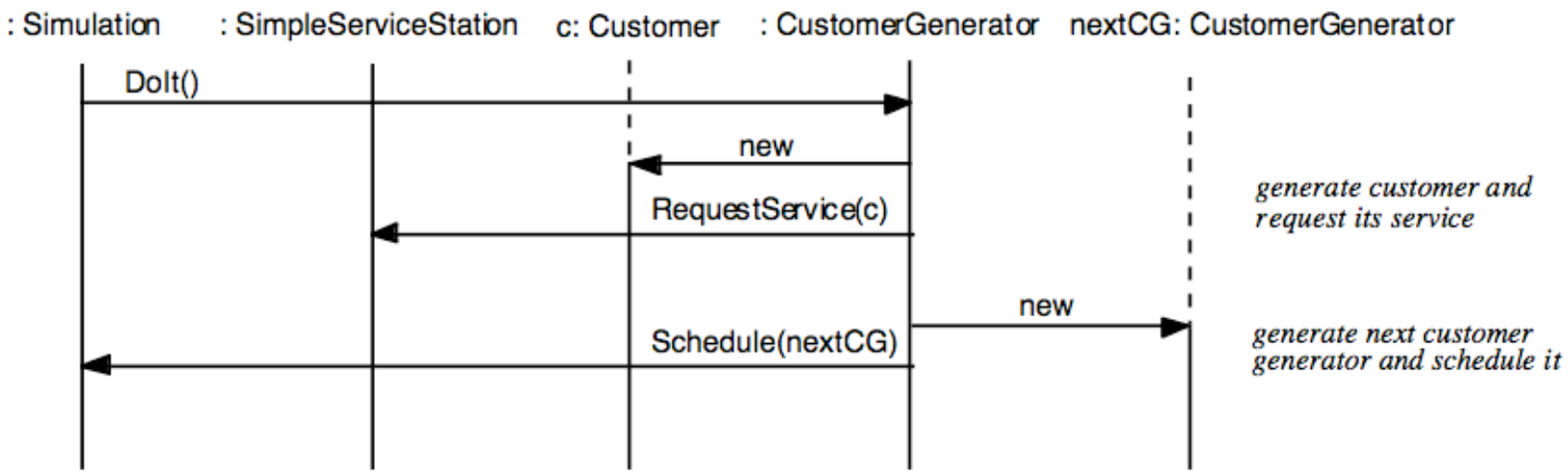
Die C# Implementierung von Simulate()

```
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);
}
```

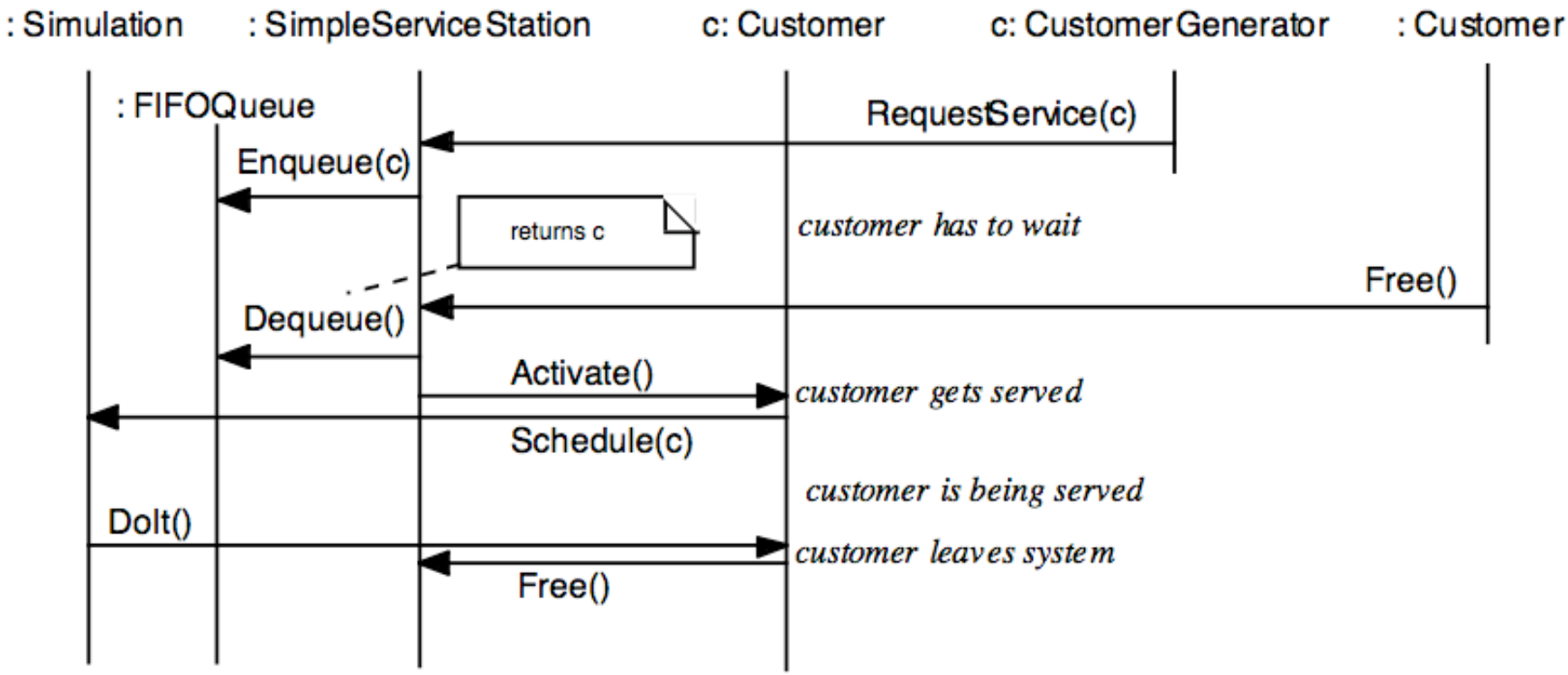
Klassen für die Simulation eines einfachen Bankschalters



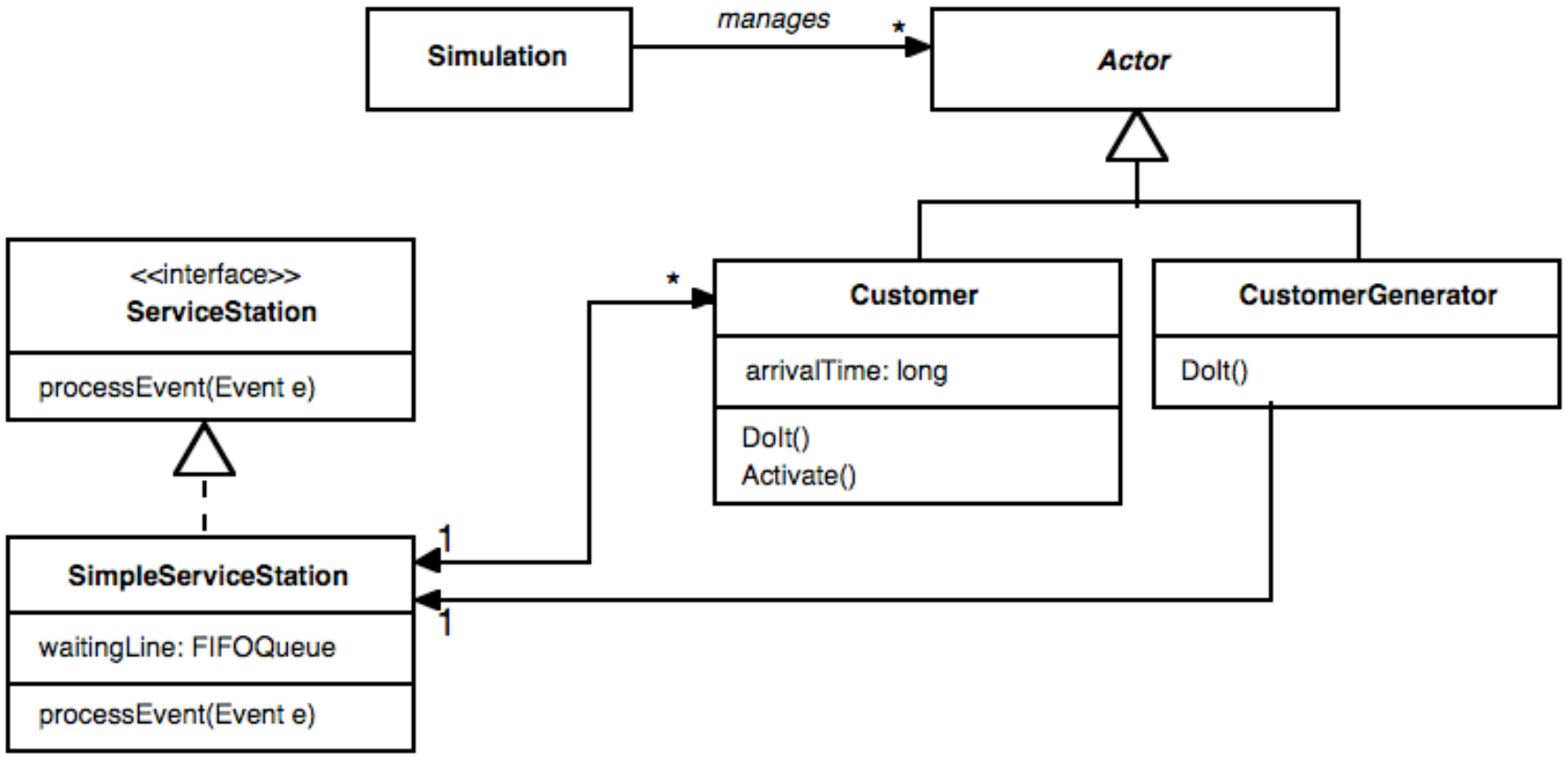
DoIt() Methode der Klasse CustomerGenerator



Einreihung eines Kunden in die Warteschlange



Verringerung der Kopplung zwischen ServiceStation und den Aktoren



Beschreibung von Softwarearchitekturen

Definition Softwarearchitektur

Die Menge aller Komponenten (Module) eines Softwaresystems zusammen mit ihren Wechselwirkungen.

Architekturmuster

- Das Software Engineering Institute (SEI) an der Carnegie-Mellon-University in Pittsburgh, Pennsylvania, hat in den 1990er-Jahren maßgeblich zur Etablierung von Architekturmustern für die Beschreibung von Softwarearchitekturen beigetragen.
- ursprünglich wurde vom SEI eine eigene Notation dafür vorgeschlagen; seit 2003 wird vom SEI auch die UML dafür verwendet

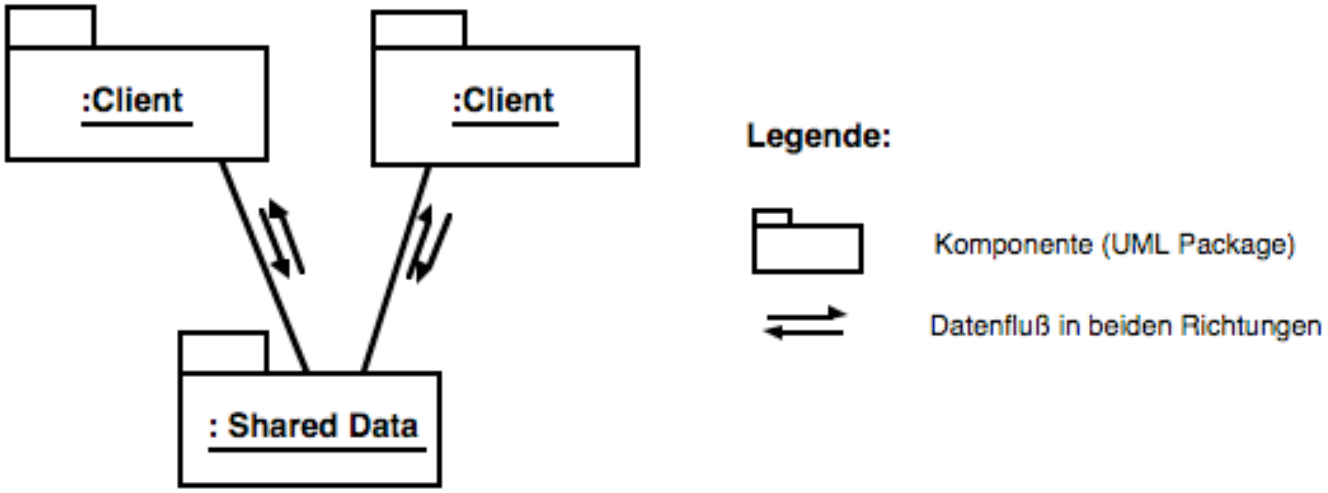
Beispiele für exzellent beschriebene Softwarearchitekturen

- *Project Oberon—The Design of an Operating System and Compiler* von Wirth und Reiser (Addison-Wesley 1992)
  Darin wird die informelle Beschreibung durch schematische Darstellungen, Screen-Shots, und Quelltext ergänzt.
- die Entwurfsmuster von Gamma et al. (Addison-Wesley 1995)

SEI-Architekturmuster im Überblick

Architekturmuster	**Ausprägungen**
Datenzentrierung	Repository-Architektur Blackboard-Architektur
Datenflussorientierung	Batch/Sequential-Architektur Pipes&Filters-Architektur
Call & Return	Top-Down-Architektur Netzwerk-Architektur (Objektorientierung) Schichten-Architektur (layered)
Virtuelle Maschine	Interpreter-Architektur Regelbasierte Architektur
Unabhängige Komponenten	Ereignisgesteuerte Architektur

Datenzentrierung (I)



- In einer Repository-Architektur sind die Daten passiv.
- Eine Blackboard-Architektur hat quasi aktive Daten, die die an Änderungen interessierten Klienten entsprechend informieren. Das Blackboard-Architekturmuster ist ähnlich dem Observer-Entwurfsmuster (Gamma et al., 1995).

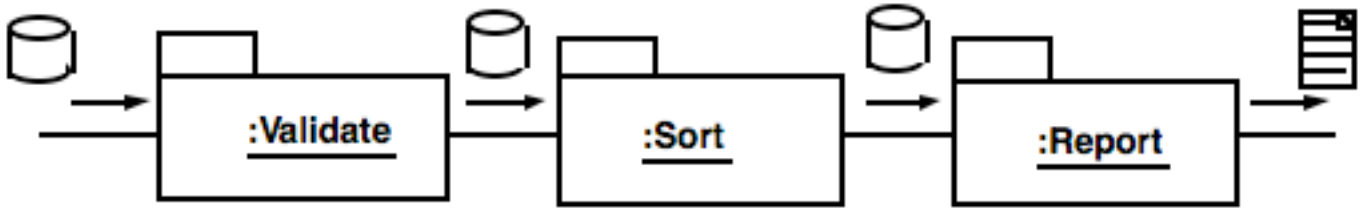
Datenzentrierung (II)

Vorteil:

- Klienten sind voneinander unabhängig.
  Dadurch können Klienten geändert werden, ohne dass andere davon betroffen sind. Es können selbstverständlich auch weitere Klienten hinzugefügt werden.

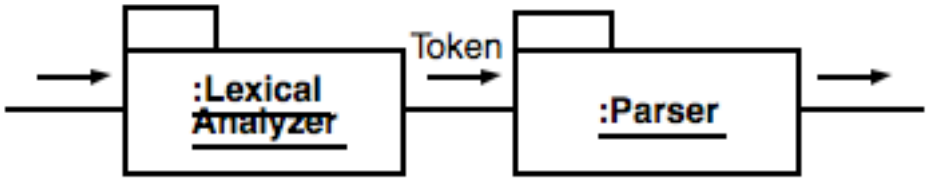
  Der Vorteil der Unabhängigkeit schwindet, wenn die Architektur so geändert wird, dass Klienten eng gekoppelt werden (also vom empfohlenen Architekturmuster abgewichen wird), um zum Beispiel die Performanz des Systems zu verbessern.

Datenflussorientierung – Batch/Sequential



- Das Muster beschreibt eine Abfolge von Transformationen von Eingabedaten.
- Datenflussorientierte Architekturteile zeichnen sich besonders durch Wiederverwendbarkeit und Modifizierbarkeit aus.
- Bei der Batch/Sequential-Ausprägung des Architekturmusters muss jeder Transformationsvorgang beendet sein, bevor der nächste beginnt.

Datenflussorientierung – Pipes&Filters



- Bei der Architekturmusterausprägung Pipes&Filters werden die Daten nicht sequenziell en bloc sondern inkrementell transformiert. Das heißt, dass die Daten in kleinere Einheiten zerlegt werden und diese Einheiten von den Prozessen verarbeitet werden.
- Pipes sind zustandslos und transportieren die Daten von Filter zu Filter und zwar so, dass jeder Filter (autonom) bestimmt, wann er vom vorgelagerten Filter das nächste Element des (Eingabe-) Datenstroms benötigt.
- In der UML-Darstellung ist der Unterschied zwischen Pipes&Filters und Batch/Sequential nicht ersichtlich.

Datenflussorientierung: Vor- und Nachteile

- Der Vorteil der Datenflussorientierung liegt darin, dass es keine komplexen Interaktionen zwischen Komponenten gibt. Die Verarbeitungsprozesse sind Black-Boxes.
- Das datenflussorientierte Architekturmuster ist ungeeignet für die Modellierung interaktiver Anwendungen.
- Ein weiterer Nachteil ist die häufig ungenügende Performanz und Effizienz. Wenn Filter als Kontext den gesamten Eingabestrom benötigen, müssen entsprechende Puffer verwendet werden. Das wirkt sich negativ auf die Speichereffizienz aus.
- Das Datenfluss-Muster eignet sich gut als Grundlage für visuell-interaktive Komposition: Es wird beispielsweise bei der Modellierung von Regelungssystemen im Werkzeug Simulink (www.MathWorks.com) eingesetzt.

Call & Return (I)

- Damit wird der für imperative Programmierung typische Kontrollfluss beschrieben: Prozeduren, Funktionen und Methoden eines Moduls werden aus anderen Modulen aufgerufen und nach ihrer Ausführung wird hinter die Aufrufstelle zurückgesprungen.
- **Top-Down Ausprägung des Call&Return-Architekturmusters:** Bei konventionellen, nicht objektorientierten Implementierungen, führt das zu einer top-down-orientierten Architektur, das heißt eine (Haupt- oder Wurzel-)Prozedur/Funktion/Methode ruft weitere Prozeduren/Funktionen/Methoden auf, etc.

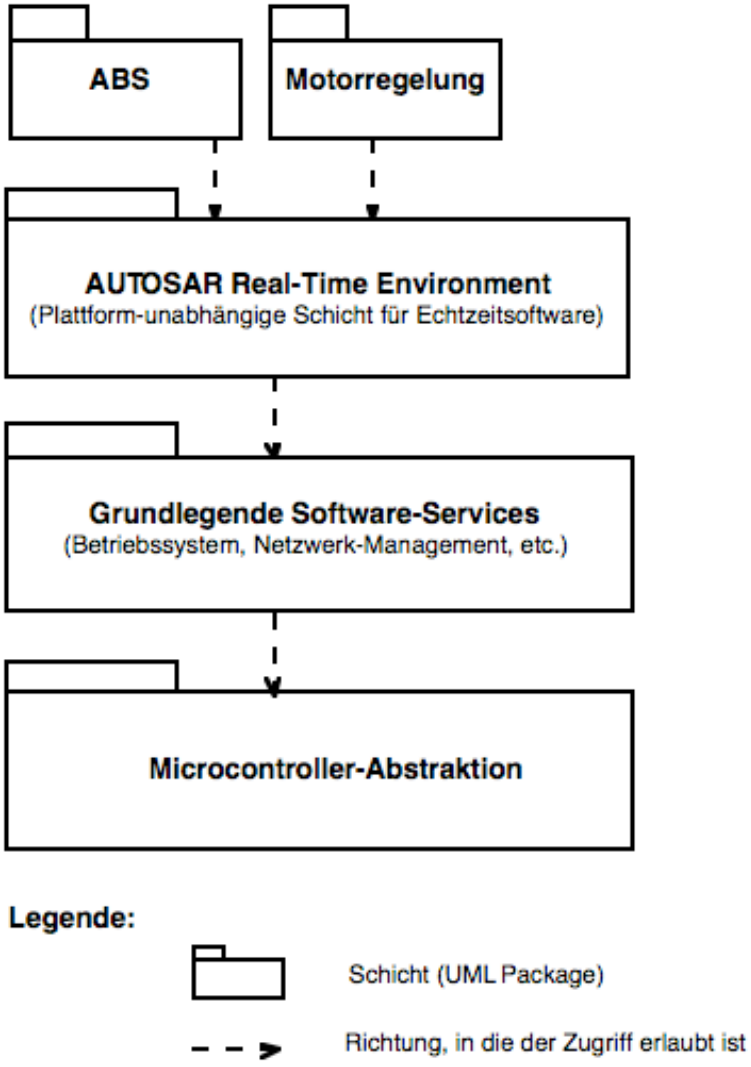
Call & Return (II)

- **Netzwerk beziehungsweise objektorientierte Ausprägung des Call&Return-Architekturmusters:** Die Konstrukte, die objektorientierte Sprachen bereitstellen, erlauben neben der hierarchischen Strukturierung von top-down-orientierten Architekturen auch die Bildung von netzwerkorientierten Architekturen. Die Methodenaufrufe erfolgen in einem Netzwerk von Objekten.

Call & Return (III)

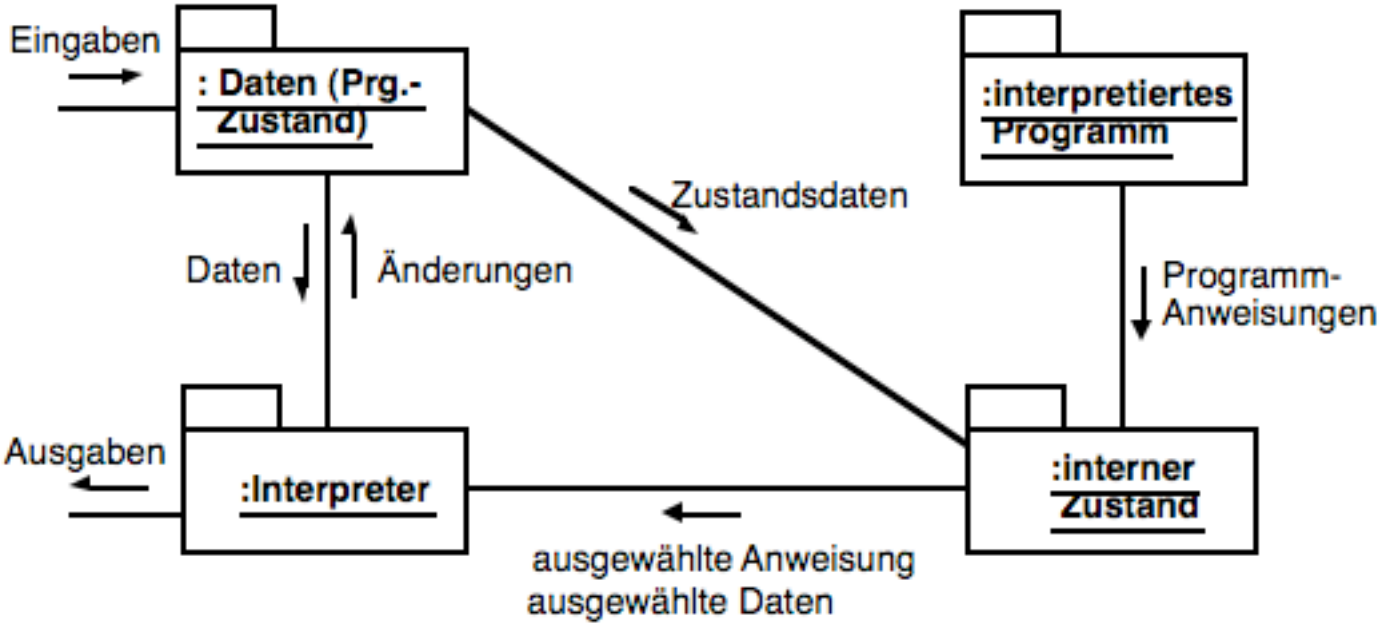
- Eine weitere Ausprägung des Call&Return-Architekturmusters, die sogenannte **Schichtenarchitektur (Layers)**, wird verwendet, um Abstraktionen einzuführen.
- Eine Schicht entspricht einem Modul, der eine Menge weiterer Module enthalten kann und der eine bestimmte Funktionalität anbietet. Mit Ausnahme der untersten Schicht, beruht jede Schicht jeweils auf der Schnittstelle der darunter liegenden Schicht. Das heißt, es soll vermieden werden, dass von einer Schicht aus Aufrufe erfolgen, die Module betreffen, die sich nicht in der unmittelbar darunter liegenden Schicht befinden. Ansonsten ist die Austauschbarkeit der Schichten nicht mehr gegeben.

Beispiel: Schichtenarchitektur von AUTOSAR (Automotive Open Systems Architecture)



weitere Informationen auf www.autosar.org

Virtuelle Maschine (I)



- Eine virtuelle Maschine dient dazu, Funktionalität, die zur Ausführung einer Applikation benötigt wird, aber die durch die benutzte Hardware und/oder Systemsoftware einer bestimmten Plattform, auf der die Applikation ausgeführt werden soll, nicht zur Verfügung gestellt wird, bereitzustellen.
- Mit diesem Architekturmuster wird die Portierbarkeit verbessert.

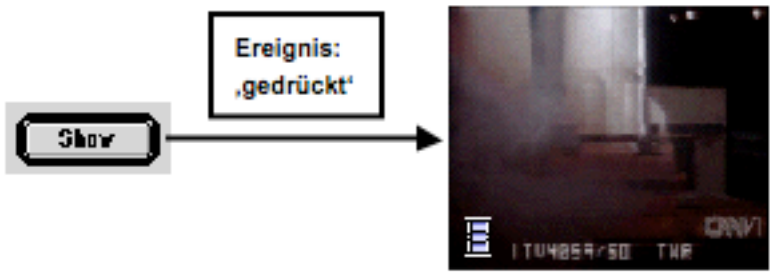
Virtuelle Maschine (II)

- **Interpreter-Architektur:** Das Konzept der virtuellen Maschine wird seit der Einführung von Java verstärkt und dementsprechend verbreitet eingesetzt. Dass mit einer virtuellen Maschine die Portierbarkeit von Software deutlich verbessert werden kann, wurde bereits anfangs der 1970er Jahre durch die Definition des P-Codes (Pascal-Code) gezeigt. Dadurch wurden Pascal-Compiler, die P-Code statt Maschinencode erzeugten, portierbar. Für eine bestimmte Hardwareplattform musste jeweils eine virtuelle Maschine, die den P-Code interpretiert, bereitgestellt werden
- **Regelbasierte Architektur:** Expertensysteme, bei denen Regeln interpretiert werden.

Unabhängige Komponenten (I)

- Dieses Architekturmuster postuliert lose miteinander gekoppelte, voneinander unabhängige Komponenten. Komponenten bezeichnen wir als voneinander unabhängig, wenn die Komponenten die Funktionen/Prozeduren/Methoden von anderen Komponenten nicht direkt aufrufen.
- Eine übliche Form, unabhängige Komponenten lose zu verbinden, sind sogenannte **ereignisorientierte Verknüpfungen**: Eine Komponente registriert sich bei einer anderen Komponente, von der sie über Änderungen informiert werden will. Bei Änderungen, also wenn ein Ereignis eingetreten ist, informiert die Komponente alle Komponenten, die sich bei ihr registriert haben. Die Kopplung ist lose, weil die Komponente lediglich eine Änderung bekannt gibt, nicht aber festlegt, was die anderen Komponenten zu tun haben. Diese können darauf reagieren, oder nicht.

Unabhängige Komponenten (II)



Die Video-Clip-Komponente muss sich bei der Schaltfläche registriert haben, dass sie über das Eintreten von Ereignissen informiert wird. Beim Aktivieren der Schaltfläche wird die Video-Clip-Komponente durch die Nachricht, dass das Ereignises „gedrückt“ eingetreten ist, von der Schaltflächenkomponente informiert. Die Video-Clip-Komponente kann nun darauf reagieren, indem der mit der Komponente assoziierte Video-Clip abgespielt wird.

Die Software-Architektur-Analyse-Methode (SAAM)

Wozu Architekturanalyse?

- Wenn ein Softwaresystem neu entwickelt wird, soll die Analyse verschiedener Architekturoptionen (auch Architekturkandidaten genannt) erlauben, die Erfüllung beziehungsweise Nichterfüllung von Qualitätsattributen einzuschätzen. Verschiedene Architekturkandidaten sollten in der Entwurfsphase evaluiert und auf Basis von verlässlichen Urteilen akzeptiert oder zurückgewiesen werden.
- Eine Architekturanalyse ist außerdem sinnvoll, wenn ein Softwaresystem gekauft oder übernommen werden soll. Nur durch eine ausreichende Architekturanalyse ist es möglich, die Produktqualität, insbesondere im Hinblick auf die Wartbarkeit, Änderbarkeit und Erweiterbarkeit unter den gegebenen Rahmenbedingungen, einzuschätzen.

Auswahl relevanter Kriterien in SAAM (I)

- Betrachten wir als Beispiel das Qualitätsmerkmal Änderbarkeit. Ein Softwaresystem ist bezogen auf bestimmte Aspekte (zum Beispiel beim Aussehen und bei der Konfiguration der Benutzungsschnittstelle) leicht änderbar, hingegen in anderen Bereichen (zum Beispiel bei den unterstützten Datenformaten bei Konversionen) schwer änderbar.
- Die Qualitätsanforderungen bezogen auf ein bestimmtes Qualitätsmerkmal müssen daher kontextbezogen definiert werden.

Auswahl relevanter Kriterien in SAAM (II)

- SAAM führt dazu **Szenarien** ein. Ein Szenario würde beispielsweise beschreiben, dass ein Benutzer der Software die Menus konfigurieren können soll. Der im Szenario erwähnte Benutzer ist ein Beispiel für einen Beteiligten (***Stakeholder*** in der SAAM-Terminologie), also einer Person, die einen Bezug zum Softwaresystem hat. Ein *Szenario* beschreibt stichwortartig die Interaktion eines Beteiligten mit dem Softwaresystem.
- Für die Bewertung einer Softwarearchitektur müssen zuerst die Voraussetzungen durch die Beschreibung von Szenarien geschaffen werden, dass eine zielgerichtete Analyse vorgenommen werden kann.

SAAM im Überblick

(1) **Identifikation und Zusammenstellung der Beteiligten (Stakeholders**)

(2) **Beschreibung und Reihung von Szenarien** (Benutzungs-Szenarien, Änderungs- und Erweiterungs-Szenarien, etc.) nach Wichtigkeit,

(3) Beschreibung von Architekturkandidaten, also von verschiedenen Möglichkeiten einer Strukturierung eines Softwaresystems,

(4) **Klassifikation von Szenarien in direkte und indirekte Szenarien**,

(5) Evaluierung der indirekten Szenarien durch Analyse der Architektur, um eine **Beurteilung der Kopplung der Komponenten** der Softwarearchitektur vornehmen zu können,

(6) Evaluierung der Interaktionen zwischen den indirekten Szenarien durch Analyse der Architektur, um eine **Beurteilung der Kohäsion der Komponenten** der Softwarearchitektur vornehmen zu können, und

(7) zusammenfassende Beurteilung

SAAM – mögliche Stakeholders (I)

Stakeholder	**Interessensschwerpunkte**
Kunde	Zeitplan und Budget Nutzen des Systems Grad der Erwartungserfüllung
Benutzer	Funktionalität Benutzbarkeit Robustheit
Entwickler (developer, maintainer, integrator, application builder)	Modularisierung (Kopplung, Kohäsion) Dokumentation Systemtransparenz/Lesbarkeit (z.B. Aufwand für das Auffinden der Stellen, an denen Systemänderungen/-verbesserungen vorgenommen werden müssen)

SAAM – mögliche Stakeholders (II)

Systemadministrator	Problemursachenidentifikation (z.B. das schnelle Auffinden von Ursachen für Probleme beim Betrieb der Software)
Netzwerkadministrator	Netzwerkdurchsatz Vorhersagbarkeit des Durchsatzes
Tester	Modularisierung (Kopplung, Kohäsion) (konsistente) Fehlerbehandlung Dokumentation Systemtrasparenz/Lesbarkeit
Repräsentant des Anwendungsgebietes	Interoperabilität zu anderen Systemen

Beschreibung und Reihung von Szenarien nach Wichtigkeit

- Die Szenarien werden von den Beteiligten vorgeschlagen und müssen repräsentativ für künftige Anforderungen, Änderungen und Erweiterungen sein.
- Die Szenarien zusammengenommen sollen alle relevanten Benutzungsaspekte des Systems beschreiben und sich insbesondere auf Funktionalitätsaspekte, sowie Entwicklungs- und Änderungsaktivitäten beziehen.
- ca. 10 – 20 Szenarien
- am Ende: Reihung nach Wichtigkeit

Beschreibung der Architekturkandidaten

- In der Praxis hat sich gezeigt, dass eine einfache Beschreibung der Datenverbindungen (welche Komponenten tauschen welche Informationen mit wem aus) und der Steuerungsverbindungen (welche Komponenten veranlassen welche Komponenten – ihrer Spezifikation gemäß – aktiv zu werden) für eine statische Darstellung der Architektur meist ausreicht.
- Als Ergänzung zur statischen Beschreibung wird zum Beispiel durch UML-Interaktionsdiagramme oder durch natürliche Sprache skizziert, wie sich das System zur Laufzeit verhalten soll.
- Für Szenarien, die Änderungen betreffen, ist eventuell die Betrachtung von Quelltext-Fragmenten nötig.

Klassifikation der Szenarien in direkte und indirekte Szenarien

- **Direkte Szenarien** können aufgrund des aktuellen Entwicklungsstandes unmittelbar umgesetzt werden.
- **Indirekte Szenarien** erfordern Änderungen des Systems, die sich auf die Architektur auswirken. Für indirekte Szenarien ist auch der geschätzte Aufwand für die erforderliche Änderung anzugeben (in Personen-Tagen/-Monaten oder -Jahren).
- Bei der SAAM werden in den folgenden Schritten nur die indirekten Szenarien weiter betrachtet. Ein indirektes Szenario kann durch die notwendige Architekturänderung eine oder mehrere Komponenten betreffen.

Evaluierung der indirekten Szenarien

- Das Ergebnis dieses Schrittes ist die Feststellung der Anzahl der von Änderungen betroffenen Komponenten für jedes indirekte Szenario.
- Das ist das Maß für die Kopplung der Komponenten der Softwarearchitektur.

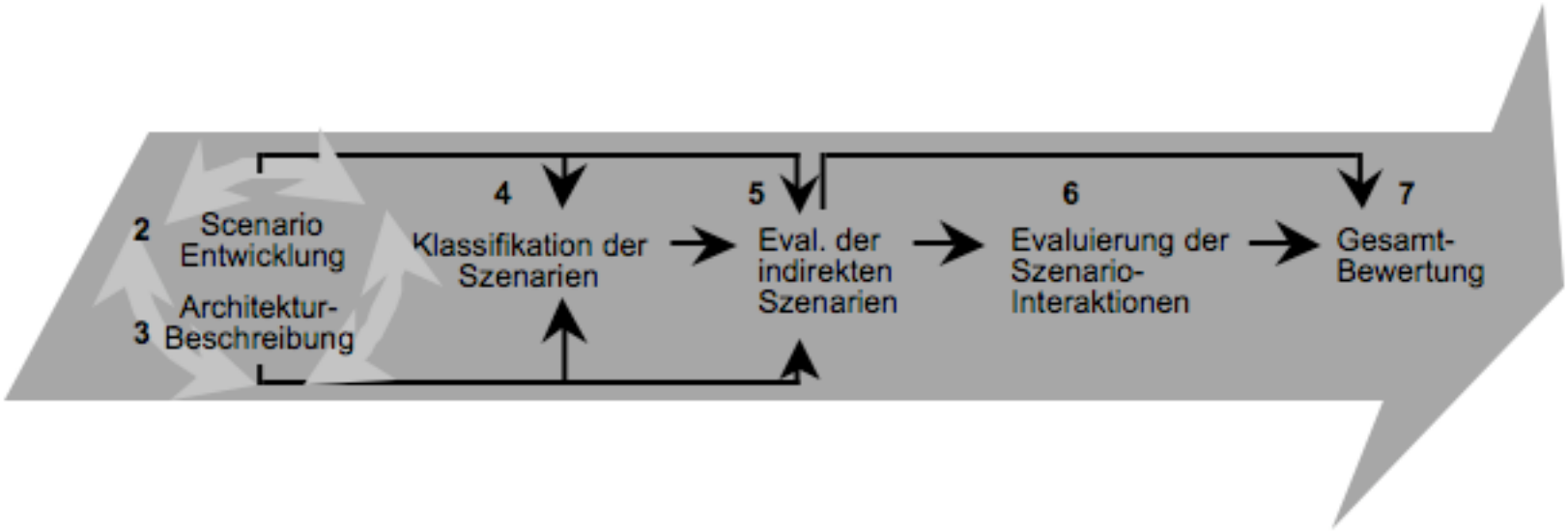
Evaluierung der Interaktionen zwischen den indirekten Szenarien

- In SAAM-Terminologie interagieren zwei indirekte Szenarien, wenn sie eine Änderung derselben Komponente erfordern.
- Die Analyse der Interaktion von Szenarien zeigt auf, welche Komponenten mehrere Aspekte abdecken und somit eine geringe Kohäsion aufweisen. Eine Interaktion von einander ähnlichen Szenarien ist ein Indikator für eine hohe Kohäsion.

Zusammenfassende Bewertung

- Der abschliessende Schritt dient insbesondere dazu, die verschiedenen Architekturkandidaten miteinander zu vergleichen, um eine Reihung der Architekturkandidaten vornehmen zu können. Dafür sollen jene Szenarien ausgewählt werden, die von entscheidender Bedeutung für den Einsatz des Systems sind.
- Das Verständnis der Architekturkandidaten aufgrund der vorhergehenden Evaluierungen bildet die Basis für die zusammenfassende Bewertung, die quantitative und qualitative Aspekte berücksichtigen soll.

Abfolge der SAAM-Schritte



Beispielanwendung der SAAM

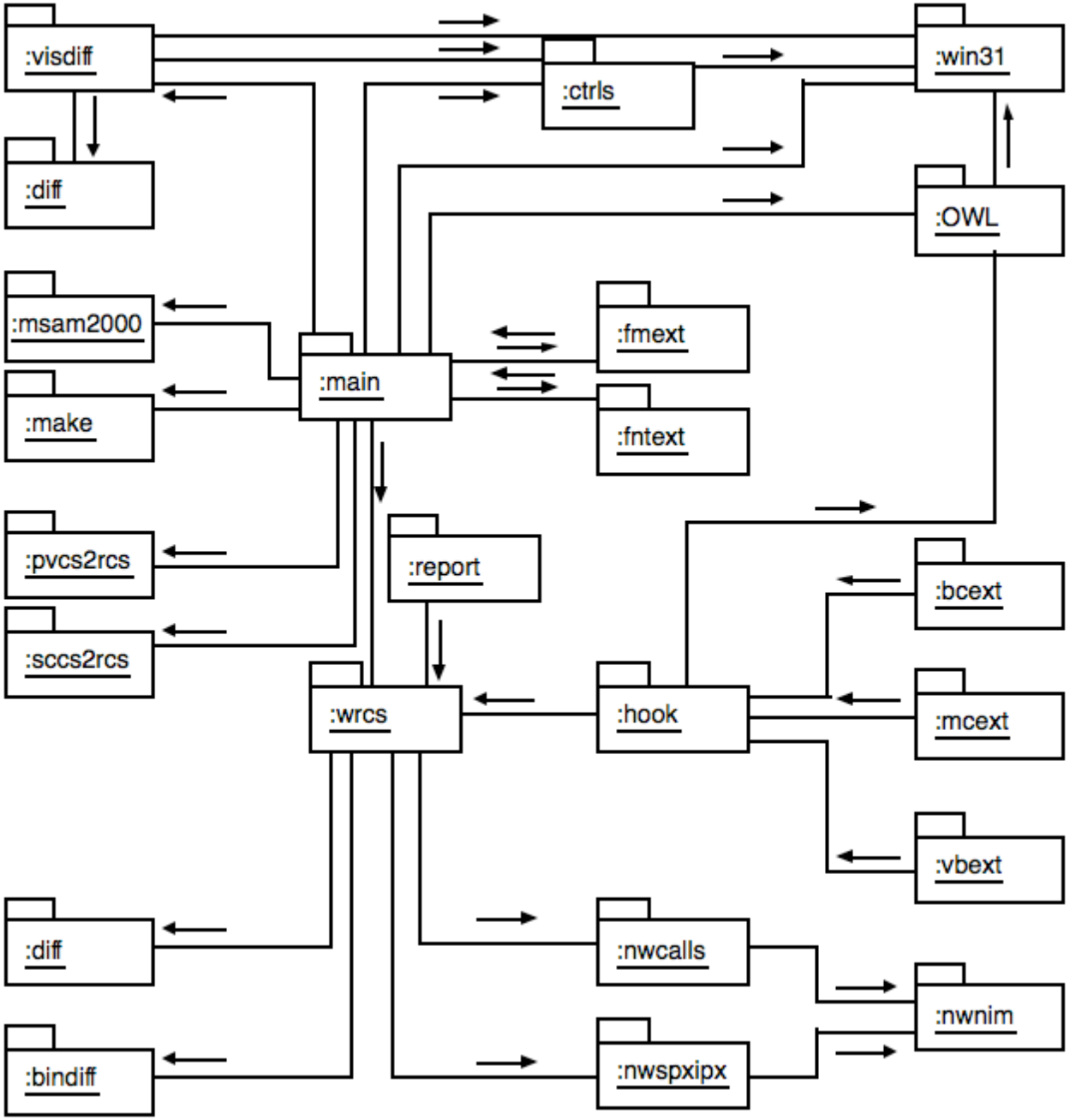
Revision Control System WRCS (I)

- Ein Projekt in WRCS ist eine Gruppe verwandter Dateien, die zusammen ein Produkt ergeben, wenn sie entsprechend verbunden werden:
  - Quelltext-Dateien sein, die zu einem lauffähigen Programm übersetzt werden,
  - Text-Dokumente eines Buches oder
  - digitalisierte Audio- und Video-Daten für einen Werbespot,
  - etc.

Revision Control System WRCS (II)

- Mit WRCS können
  - die Änderungen von Dateien verfolgt werden
  - Archive definiert werden,
  - Dateien ein- und ausgecheckt werden,
  - Releases erzeugt werden,
  - und alte Versionen wiederhergestellt werden.
- Das WRCS wurde mit verschiedenen Entwicklungs-umgebungen integriert. Die verfügbaren Funktionen können von der jeweiligen Entwicklungsumgebung aus oder über die grafische Benutzungsschnittstelle von WRCS ausgeführt werden.

WRCS-Komponenten und deren Interaktionen



WRCS-Architekturmuster

- Call & Return-Architektur
- keine Schichten

Szenarien und deren Wichtigkeit

Stakeholder	Szenario	Wichtigkeit
Endbenutzer	Vergleich von Binärdateien	1
	Konfiguration des Toolbars	3
Entwickler (Maintainer)	Anpassungen der grafischen Benutzungsschnittstelle	6
	Portierung auf ein anderes Betriebssystem	4
Administrator	Änderung der Zugriffsrechte für ein Projekt	5
	Integration in eine neue Entwicklungsumgebung	2

Klassifikation in direkte und indirekte Szenarien

Stakeholder	Szenario	Klassifikation
Endbenutzer	Vergleich von Binärdateien	indirekt
	Konfiguration des Toolbars	direkt
Entwickler (Maintainer)	Anpassungen der grafischen Benutzungsschnittstelle	indirekt
	Portierung auf ein anderes Betriebssystem	indirekt
Administrator	Änderung der Zugriffsrechte für ein Projekt	direkt
	Integration in eine neue Entwicklungsumgebung	indirekt

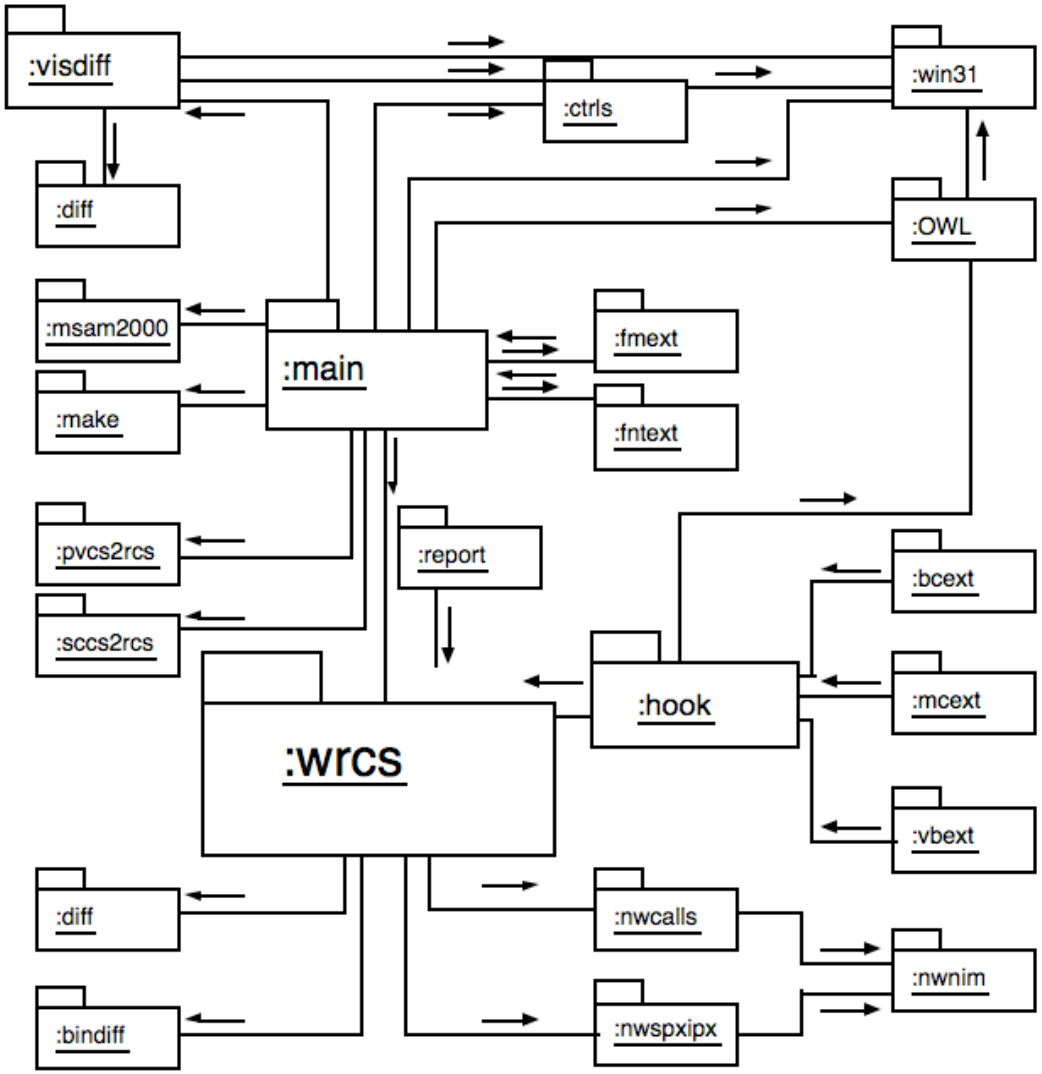
Anzahl der Komponenten, die durch ein indirektes Szenario geändert werden müssen

indirektes Szenario	**Anzahl der zu ändernden Komponenten**
Vergleich von Binärdateien	2
Anpassungen der grafischen Benutzungsschnittstelle	3
Portierung auf ein anderes Betriebssystem	3+
Integration in eine neue Entwicklungsumgebung	4

Evaluierung der Interaktionen zwischen den indirekten Szenarien

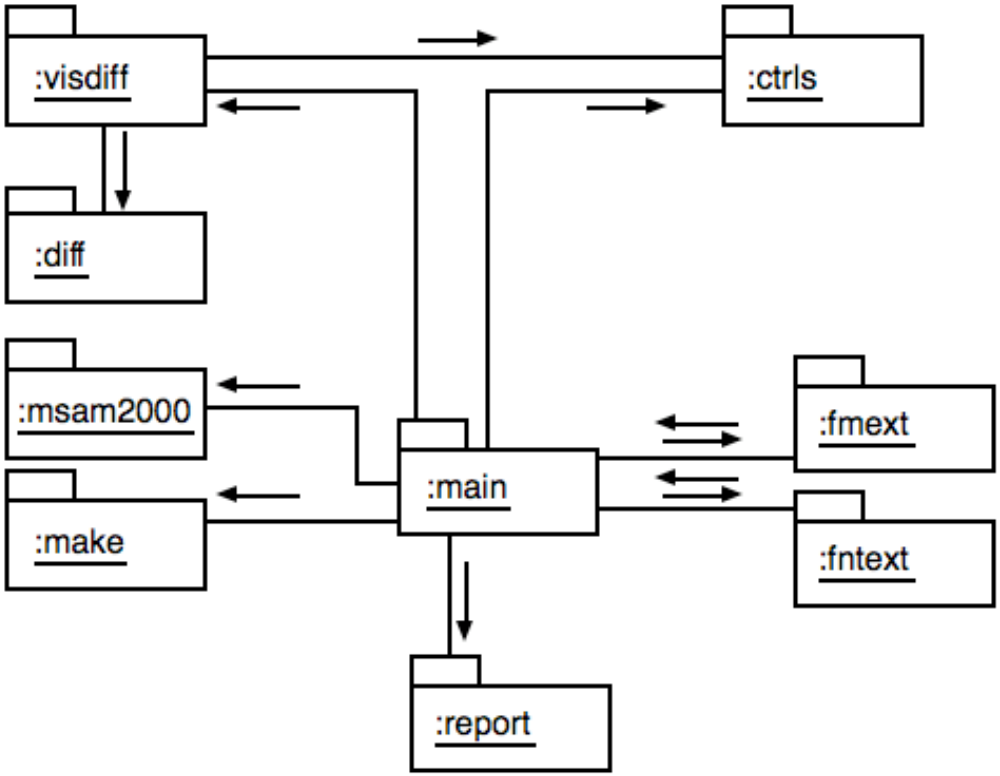
Modul	Anzahl der erforderlichen Änderungen
main	4
wrcs	7
diff	1
bindiff	1
pvcs2rcs	1
sccs2rcs	1
nwcalls	1
nwspxipx	1
nwnlm	1
hook	4
report	1
visdiff	3
ctrls	2

Visualisierung der Szenario-Interaktionen



Vorteile und Risken bei Anwendung der SAAM (I)

- Die Brauchbarkeit der Ergebnisse hängt entscheidend von der richtigen Wahl des Granularitätsgrades der Architekturbeschreibung ab.
  ZB wäre folgende Granularität bei WRCS problematisch:



Vorteile und Risken bei Anwendung der SAAM (II)

- Ermöglicht rasche und zielsichere Bewertung der Qualität einer Softwarearchitektur, insbesondere was Änderungen und Erweiterungen betrifft.
- Keine aufwändigen und detaillierten Code-Inspektionen und dergleichen sind nötig.
- Ein Aufwand von wenigen Personentagen (2 bis 5 Tage je nach Sytemkomplexität) ist erforderlich.
- Dennoch sind viel Erfahrung und fachliches Wissen von den Beteiligten die Voraussetzung für eine erfolgreiche SAAM-Anwendung.
- Stakeholder-Beteiligung ist die sozioökonomische Komponente der SAAM und sichert die Effizienz des Analyseprozesses.
- Die SAAM bietet eine pragmatische Möglichkeit, Kopplung und Kohäsion zu „messen“ und auf Grundlage der Messergebnisse eine Bewertung der Balance von Kopplung und Kohäsion einer gewählten Modularisierung vorzunehmen.

Mehrdimensionale Modularisierung durch Aspektorientierte Programmierung (AOP)

AOP als Weiterentwicklung von Metaprogrammierung

- AOP geht auf Gregor Kiczales zurück und wurde Mitte der 1990er Jahre bekannt
- Die grundlegende Idee der AOP ist, dass für ein Softwaresystem nicht bloß *eine* (statisch festgelegte) Modularisierung existiert, sondern dass unterschiedliche Sichten auf die Modularisierung eines Softwaresystem gelegt werden können. Deswegen bezeichnen wir AOP auch als ein Mittel zur „*mehrdimensionalen Modularisierung*“.
- Die zusätzlichen, die statische Modularisierung überlagernden Modularisierungen sind an Bedingungen geknüpft, die zur Laufzeit evaluiert werden.
- Die Metaprogrammierung, mit der die Semantik einer Programmiersprache erweitert werden kann, wurde bei AOP angewendet, um die Modularisierung von Software zu verbessern.

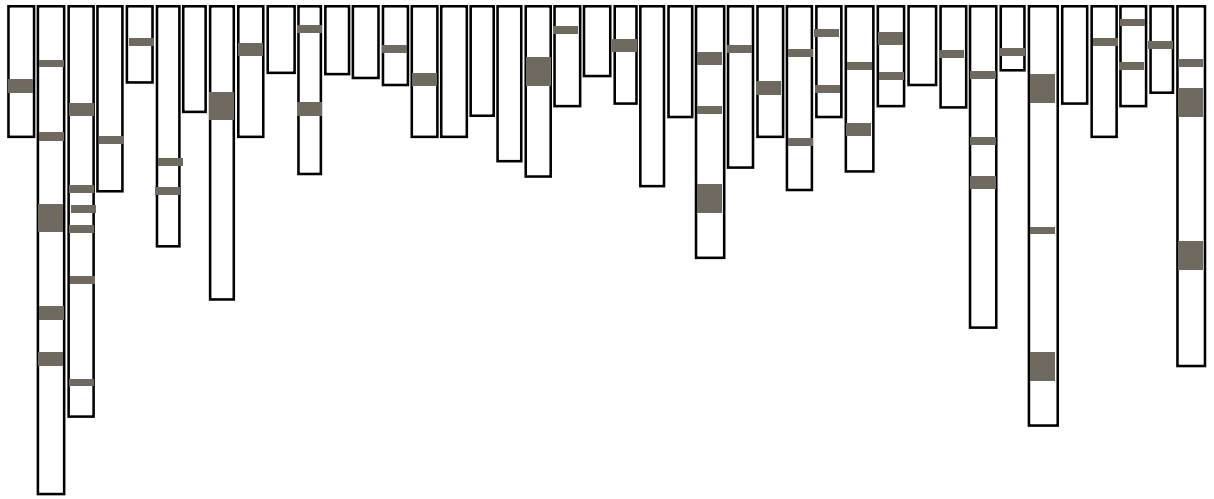
Ausgangspunkt von AOP: das Problem einer einzigen statischen Modularisierung (I)

Beispiel Apache Tomcat Web-Server (laut aspect.org):

gute Modularisierung des Aspekts
“URL-Mustererkennung”

Ausgangspunkt von AOP: das Problem einer einzigen statischen Modularisierung (II)

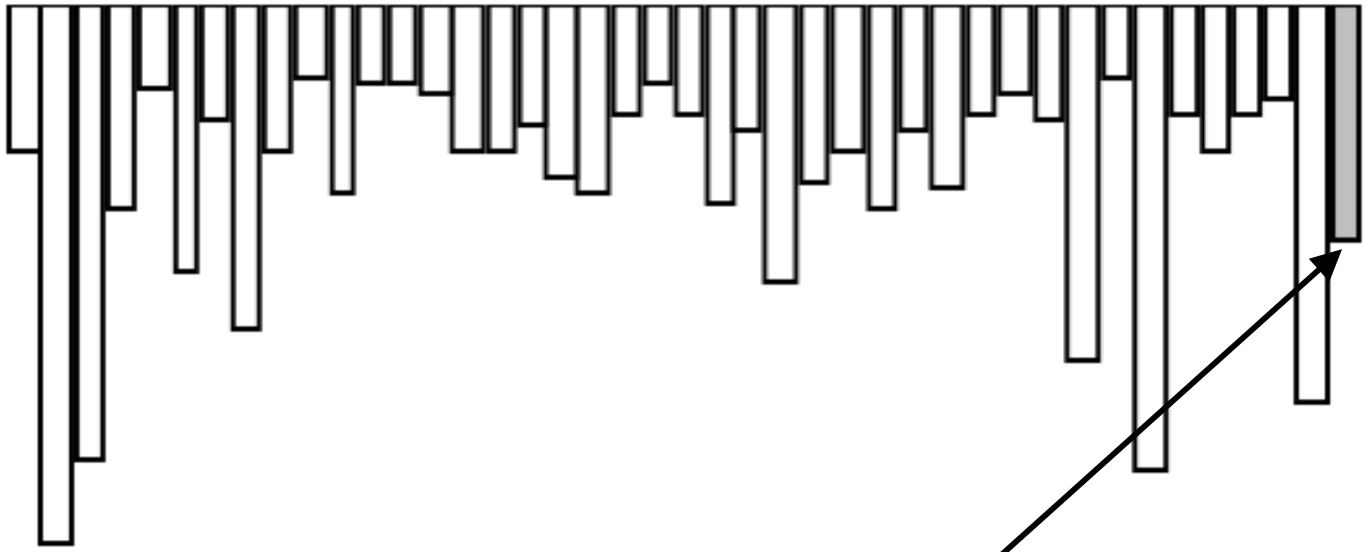
Beispiel Apache Tomcat Web-Server (laut aspect.org):



schlechte Modularisierung des Aspekts “Logging”

AOP-Lösung: Cross-Cutting

Beispiel Apache Tomcat Web-Server (laut aspect.org):



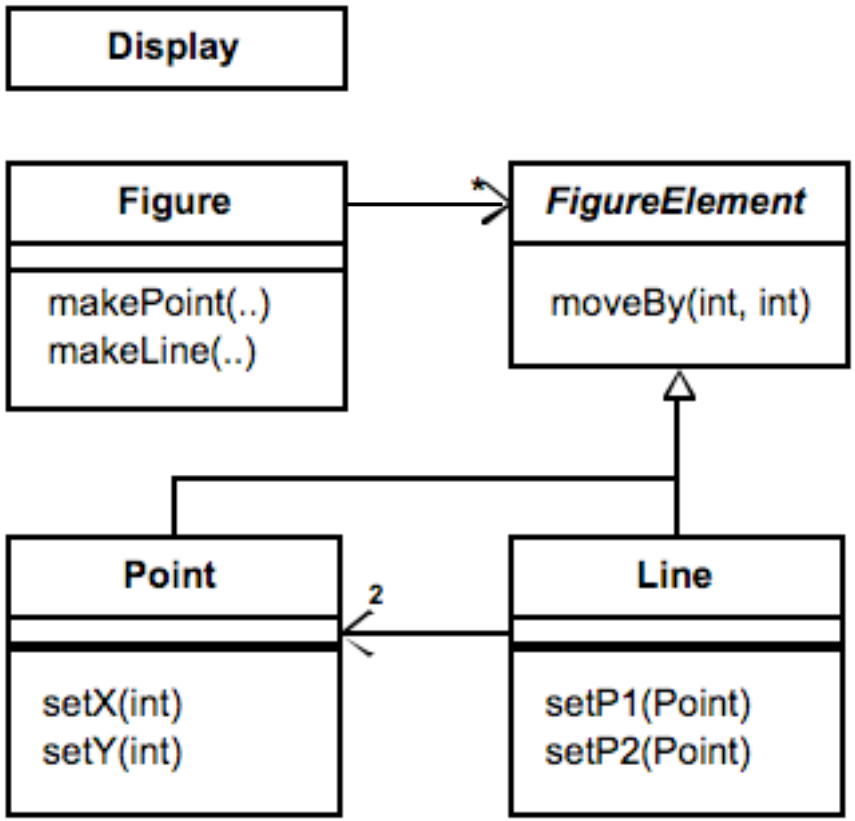
“Cross-Cutting” des über viele Module verteilten Logging-Codes zu einem neuen AOP-Modul Logging

AOP-Konzepte in AspectJ (I)
Beispiel: Grafikeditor

- AspectJ ist die AOP-Erweiterung von Java
- AspectJ erweitert mit den sogenannten Dynamic Join Points die Semantik der Sprache Java.
- Die anderen vier Konstrukte von AspectJ Point Cuts, Advices, Intra Class Declarations (auch Open Classes genannt), und Aspects ergänzen lediglich Java, ohne direkt die Semantik der Sprache zu verändern.

AOP-Konzepte in AspectJ (II)
Beispiel: Grafikeditor

Ausgangspunkt (nicht AOP):



AOP-Konzepte in AspectJ (III)
Beispiel: Grafikeditor

Ausgangspunkt (nicht AOP):

```
class Line implements FigureElement {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) { this.p1 = p1; Display.update(); }
  void setP2(Point p2) { this.p2 = p2; Display.update(); }
  void moveBy(int dx, int dy) { ... Display.update(); }
}
```

AOP-Konzepte in AspectJ (IV)
Beispiel: Grafikeditor

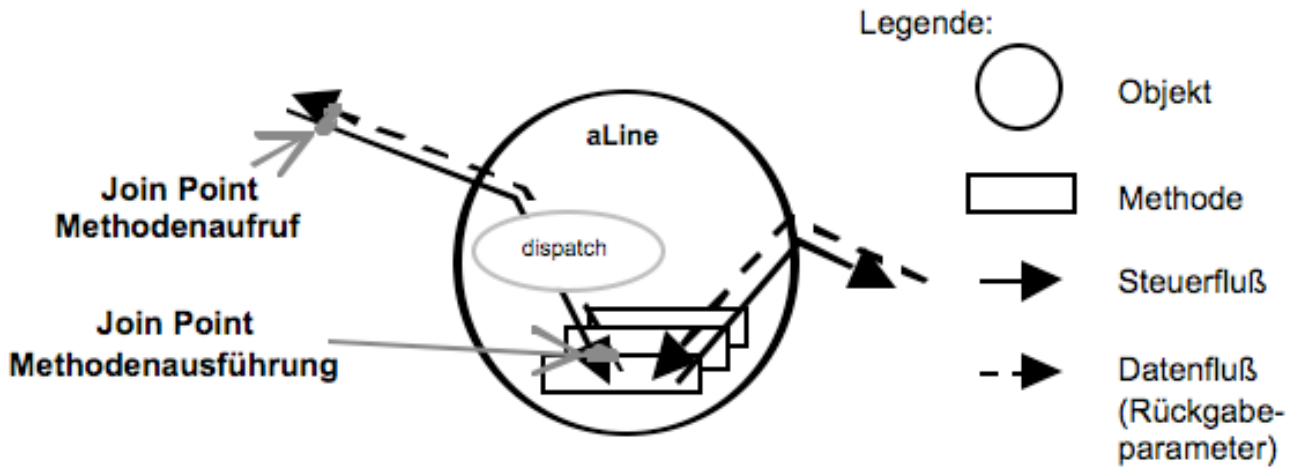
Ausgangspunkt (nicht AOP):

```
class Point implements FigureElement {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) { this.x = x; Display.update(); }
  void setY(int y) { this.y = y; Display.update(); }
  void moveBy(int dx, int dy) { ... Display.update(); }
}
```

AOP-Konzepte in AspectJ (V)
Beispiel: Grafikeditor

AOP-Join-Point-Konzept:

Der AOP-Entwickler bekommt dadurch die **Möglichkeit, in Methoden auf den Aufruf einer Methode beziehungsweise auf die Ausführung einer Methode zu reagieren.**
In AspectJ werden folgende Arten von Join-Points unterschieden: Methoden- und Konstruktor-Aufrufe; Methoden- und Konstruktor-Ausführungen, Zugriff auf Instanzvariablen (Setter- und Getter-Methoden), Ausführung von Ausnahmebehandlungen, statische und dynamische Initialisierung.



AOP-Konzepte in AspectJ (VI)
Beispiel: Grafikeditor

Ein **Point Cut** ist in AOP als logisches Prädikat auf einem Join Point definiert. ZB trifft der Point Cut

```
call(void Line.setP1(Point))
```

zu, wenn ein Methodenaufruf mit der angebenen Signatur void Line.setP1(Point) erfolgt.

Das **AspectJ-Schlüsselwort call** wird als **Designator** bezeichnet, der für den Join Point Methodenaufruf definiert worden ist. Point Cuts können mit logischen Operatoren (gemäß Java-Syntax) verknüpft werden, wie zum Beispiel:

```
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point))
```

AOP-Konzepte in AspectJ (VII)
Beispiel: Grafikeditor

Ein Point Cut kann auch benannt werden. In unserem Beispiel nennen wir den Point Cut move:

```
pointcut move():
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point))
```

Ein Point Cut kann auch Parameter haben. In diesem einfachen Beispiel ist das nicht erforderlich.

AOP-Konzepte in AspectJ (VIII)
Beispiel: Grafikeditor

Außer call bietet AspectJ weitere Designatoren, wie zum Beispiel execute, get, set, und handler, die den weiteren Join Point-Arten

- Methoden-Ausführung,
- Zugriff auf Instanzvariablen (Setter- und Getter-Methoden), und
- Ausführung von Ausnahmebehandlungen

entsprechen.

Darauf gehen wir in der überblicksartigen Darstellung von AOP nicht ein.

AOP-Konzepte in AspectJ (IX)
Beispiel: Grafikeditor

Unter einem **Aspect** versteht man eine spezifische Klasse, die quer durch andere Klassen „schneidet“ (**cross cutting**), um den dort verstreuten, aspektbezogenen Code, der logisch zusammengehört, in einem Modul zusammenzufassen.

AOP-Konzepte in AspectJ (X)
Beispiel: Grafikeditor

```
aspect DisplayUpdating {
  pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point))              ||
    call(void Line.setP2(Point))              ||
    call(void Point.setX(int))                ||
    call(void Point.setY(int));

  after() returning: move() {        // after Advice
    Display.update();
  }
}
```

AOP-Konzepte in AspectJ (XI)
Beispiel: Grafikeditor

- Unter einem **Advice** versteht man in AOP **den Code, der ausgeführt wird, wenn ein Point Cut zutrifft.** Der AOP after Advice beschreibt, was zu tun ist, wenn der Point Cut move zutrifft, also mindestens eines der call-Prädikate zutrifft: Die statisch definierte Methode update() der Klasse Display wird aufgerufen.
- Damit werden die Display.update() Anweisungen überflüssig, wie sie in der konventionellen, objektorientierten Implementierung in den Methoden der Klassen Point und Line enthalten waren.

Weitere AOP-Konzepte werden in dieser Einführung nicht erläutert

Die Erläuterung der zahlreichen weiteren Facetten von AOP, wie zum Beispiel

- Parameter,
- Vor- und Nachbedingungen,
- Wildcards,
- Reflection,
- abstrakte Aspekte, sowie die
- Einbettung von AOP in Programmierumgebungen,

würden den Rahmen dieses als Überblick konzipierten Abschnittes sprengen.

Kritische Betrachtung von AOP

- Ein Nachteil von AOP ist, dass die Lokalität bei Erweiterungen verloren geht: Wenn ein neues FigureElement, zum Beispiel ein Circle, hinzugefügt wird, muss zusätzlich die „globale Klasse“, also aspect DisplayUpdating, geändert werden.
- Je mehr Point Cuts vorgesehen werden, umso schwieriger ist es nachvollziehbar, wann welche Prädikate zutreffen. Man verliert schnell den Überblick.

Zusammenfassung wichtiger Modularisierungsprinzipien

Wichtige Modularisierungsprinzipien

- Die wichtigste Eigenschaft eines **Moduls** ist, **ein Mittel für Abstraktionen** zur Verfügung zu stellen. Es sollen unwichtige (Implementierungs-)Details hinter einer Schnittstelle verborgen werden.
- Die Festlegung einer Softwarearchitektur durch die Modularisierung eines Softwaresystems erfordert eine gute Kenntnis des Anwendungsbereichs. Die Komponenten sollen den Entitäten des Anwendungsbereichs entsprechen.
- Die Module sollen eine Balance von Kopplung und Kohäsion aufweisen.

Der Bezug zu Architektur...

- Die Modularisierung eines Softwaresystems ist eine schwierige, **zum Teil künstlerische Aufgabe**, wo es ähnlich wie bei der Gebäudearchitektur keine Methoden gibt, die verlässlich zu einem guten Ergebnis führen.
- Christopher Alexander, ein Architekturprofessor an der Universität von Kalifornien in Berkeley, der die Muster von seiner Meinung nach guten Gebäudearchitekturen beschrieben hat (Alexander, 1977), prägte den Begriff „quality without a name“. **Eine gute Architektur hat eine „quality without a name“, die nicht explizit beschrieben werden kann.**
- Was hilft, ein guter Architekt zu werden, ist, **viele gute Architekturen** zu **studieren**, die diese „quality without a name“ haben. **Leider gibt es bisher nur wenige Softwarearchitekturen, die diese Qualität haben und einzusehen sind.**