

# LARS-Projektbericht 1998

## **Anpassung der Informatik-Grundausbildung für Elektroniker an neue technologische Entwicklungen**

Bericht über die erste Phase eines innovativen Lehrprojekts  
im Rahmen des Förderprogramms  
„Leistungsanreizsysteme in der Lehre“ (LARS)  
der Studienkommission für Hochschuldidaktik  
an Fachhochschulen in Baden-Württemberg

**Projektleiter:** Prof. Dr. Karlheinz Hug  
**Studentische Mitarbeiter:** Olav Augustin, Michael Jörger (je 60 Stunden)  
**Fachhochschule:** FH Reutlingen - Hochschule für Technik und Wirtschaft  
**Fachbereich:** FB Elektronik  
**Postanschrift:** Federnseestr. 4, 72764 Reutlingen  
**E-Mail:** Karlheinz.Hug@FH-Reutlingen.de  
**Projektzeitraum:** März 1998 bis Januar 1999

## Inhaltsverzeichnis

1	Einführung .....	5
2	Ziele und Rahmen .....	6
2.1	Gesamtziele des Projekts .....	6
2.2	Allgemeine Anforderungen an das Gerüst .....	6
2.3	Vorerfahrungen .....	7
2.4	Lehrsprache und Entwicklungsumgebung .....	8
2.5	Terminologie.....	8
2.5.1	Graphische Notation .....	8
2.5.2	Komponenten.....	8
2.5.3	Module.....	11
2.5.4	Klassen.....	12
2.5.5	Verträge.....	13
2.5.6	Rohe und gare Module .....	13
3	Anforderungen und Konzeption .....	15
3.1	Teilziele der ersten Projektphase .....	15
3.2	Simulation einer Telefonvermittlungsanlage.....	15
3.3	Modellierung elektronischer Bauelemente.....	16
4	Analyse und Entwurf .....	19
4.1	Entwicklungsrichtlinien.....	19
4.2	Entwicklungsschritte .....	19
4.3	Testverfahren .....	20
4.3.1	Testverfahren für Module .....	20
4.3.2	Testverfahren für Klassen.....	20
4.4	Benutzungshierarchie der Module.....	21
4.5	Die einzelnen Module.....	22
4.5.1	ElectricsPhysConstants.....	22
4.5.2	ElectricsTypes.....	23
4.5.3	ElectricsResistor .....	23
4.5.4	TestResistor.....	24
4.5.5	ElectricsResistorsA.....	24
4.5.6	TestResistorsA .....	24
4.5.7	ElectricsResistorsV.....	25
4.5.7.1	Entwurf .....	25
4.5.7.2	Zur Spezifikation .....	27
4.5.7.3	Bemerkungen.....	28
4.5.8	TestResistorsV .....	28
4.5.9	ElectricsResistorsR0.....	28
4.5.10	ElectricsResistorsR.....	28
4.5.10.1	Entwurf .....	29

	4.5.10.2	Aliasintechnik .....	30
	4.5.10.3	Abstraktionstechnik .....	30
	4.5.10.4	Polymorphe Rekursion .....	31
	4.5.10.5	Zur Spezifikation .....	31
	4.5.11	TestResistorsR .....	32
5		Spezifikation und Implementierung .....	33
	5.1	Statistische Auswertung von ElectricResistorsR .....	33
	5.1.1	Anzahl der Prozeduren .....	36
	5.1.2	Komplexität der Prozeduren .....	36
6		Fazit und Ausblick .....	38
	6.1	Zu den Mitteln und Methoden .....	38
	6.2	Zu den Ergebnissen .....	38
	6.3	Weitere Entwicklungsschritte .....	39
	6.4	Weitere Ziele .....	39
A		Dokumentation .....	40
	A.1	Resistors .....	40
B		Schnittstellen .....	41
	B.1	ElectricResistorsV flach .....	41
	B.2	ElectricResistorsR .....	41
C		Quellprogramme .....	42
	C.1	ElectricPhysConstants .....	42
	C.2	ElectricTypes .....	42
	C.3	ElectricResistor .....	42
	C.4	TestResistor .....	42
	C.5	ElectricResistorsA .....	42
	C.6	TestResistorsA .....	42
	C.7	ElectricResistorsV .....	42
	C.8	TestResistorsV .....	42
	C.9	ElectricResistorsR0 .....	42
	C.10	ElectricResistorsR .....	42
	C.11	TestResistorsR .....	42
D		Testprotokolle .....	43
	D.1	TestResistorsR testet ElectricResistorsR .....	43
E		Literatur .....	44
F		Danksagung .....	45

## Abbildungsverzeichnis

Bild 1	Graphische Modellierungselemente .....	8
Bild 2	Beziehungen zwischen Grundbegriffen .....	9
Bild 3	Benutzung.....	9
Bild 4	Klassifikation, Bestandteil- und allgemeine Beziehungen.....	10
Bild 5	Bestandteile von Komponenten .....	10
Bild 6	Modularten .....	12
Bild 7	Klassenarten .....	13
Bild 8	Klassenstrukturmodell für elektronische Bauelemente.....	17
Bild 9	Modul-Benutzungshierarchie mit der Wurzel TestResistorsR.....	22
Bild 10	Ohmscher Widerstand .....	24
Bild 11	Reihenschaltung von Widerständen .....	25
Bild 12	Parallelschaltung von Widerständen .....	25
Bild 13	Ansatz eines Entwurfs mit Kompositum.....	26
Bild 14	Klassendiagramm für statische Konfiguration von Widerständen.....	27
Bild 15	Schaltung von Widerständen.....	29
Bild 16	Klassendiagramm für dynamische Konfiguration von Widerständen .....	29
Bild 17	Objektdiagramm zur Aliasingtechnik .....	30
Bild 18	Klassendiagramm für Erweiterung allgemeiner Konzeptklassen .....	31
Bild 19	Statistik über ElectricsResistorsR .....	34

# 1 Einführung

Gegenstand des hier vorgestellten Lehrprojekts ist, die Grundlagenausbildung in Informatik für Elektroniker an neue technologische Entwicklungen anzupassen, um die Studierenden besser als bisher auf professionelles Arbeiten als Software-Entwickler und wachsende Qualitätsanforderungen an Software vorzubereiten. Neue Aspekte für die Informatik-Grundausbildung sind:

- Programmieretechnik basierend auf der Objekt- und Komponententechnologie entwickeln, weil diese günstigere Voraussetzungen für arbeitsteiliges Entwickeln von Software hoher Qualität bietet als konventionelle prozedurale Ansätze [11].
- Die Vertragsmethode didaktisch durcharbeiten und von Beginn an lehren, weil sie wesentlich dazu beiträgt, Software zuverlässiger zu gestalten als es z.B. mit dem Ansatz des „defensiven Programmierens“ möglich ist [9].
- Begriffe wie Baukasten, Modul- und Klassenbibliothek, Entwurfsmuster (design pattern) und Gerüst (framework) exemplarisch veranschaulichen, weil diese der Wiederverwendbarkeit und Erweiterbarkeit von Software dienen [2].

In der Software-Praxis gewinnt die Objekt- und Komponententechnologie an Bedeutung. Mit ihr befaßte Informatiker haben neue Entwicklungsmethoden geschaffen, informatische Bereiche wie Betriebssysteme, Benutzungsoberflächen, Datenbanken, Workflow-Management beeinflußt und dazu beigetragen, früher divergierende Ansätze zu vereinheitlichen. Diese Tendenz wird anhalten, da das Potential dieser Technologie längst nicht ausgeschöpft ist. In der industriellen Praxis sammelt man Erfahrungen mit Bibliotheken, Entwurfsmustern und Gerüsten, und man erkennt zunehmend, wie wichtig arbeitsteiliges Entwickeln von Software und die dazu erforderliche kommunikative Kompetenz sind. Dagegen schätzt und nutzt die Industrie die Vertragsmethode noch viel zu wenig, obwohl keine andere in letzter Zeit vorgeschlagene Methode mehr dazu beitragen könnte, die Zuverlässigkeit von Software zu erhöhen.

In der Informatikausbildung für Ingenieure sind objekt- und komponentenorientierte Konzepte, Methoden und Techniken noch ungenügend vertreten. Uns scheint sinnvoll, diese fachlichen Neuerungen - insbesondere die genannten Aspekte - in die Grundausbildung zu integrieren. Dieser neue Lehransatz bedarf aber gründlicher Aufbereitung. Es genügt nicht, ein populäres professionell genutztes komplexes Werkzeug, ein Gerüst oder eine Bibliothek unbesehen in der Lehre einzusetzen und darauf zu hoffen, daß sich damit methodische Fähigkeiten bei den Lernenden automatisch einstellen.

## 2 Ziele und Rahmen

### 2.1 Gesamtziele des Projekts

Die Studierenden sollen in der Informatik-Grundlagenausbildung besser als bisher auf zukunftssträchtige Technologien und professionelle Arbeitsweisen vorbereitet werden. Auch die Informatikausbildung für Elektroniker soll neue informatische Lehrinhalte antizipieren:

- Mehr fachliche Kompetenz durch Vermittlung vertiefter Grundlagen und methodischer Fähigkeiten mit langer Halbwertszeit statt kurzlebiger Detailinformation über Produkte.
- Mehr soziale und kommunikative Kompetenz durch kooperative Lehr- und Lernformen.
- Mehr Transferkompetenz durch Kenntnis neuartiger Technologien.

### 2.2 Allgemeine Anforderungen an das Gerüst

Die Projektidee ist, ein exemplarisches objekt- und komponentenorientiertes Gerüst für Lehrzwecke zu entwickeln, das speziell folgende Anforderungen erfüllt:

- (1) Es ist auf Kenntnisse und Erwartungen von Studierenden der Elektrotechnik zugeschnitten. Dazu sollen die modellierten fachlichen Inhalte aus elektrotechnischen Gebieten stammen und mit anderen Lehrveranstaltungen des FBs synchronisiert sein (Synergieeffekte beim Lernen).
- (2) Es ist auf Bedürfnisse und erwartete Fähigkeiten von Elektronik-Ingenieuren mit Informatik-Kenntnissen ausgerichtet (in Abgrenzung von informatisch breiter einsetzbaren Voll-Informatikern).
- (3) Es eignet sich für den Einsatz in der Grundlagenausbildung. Dies bedingt Beschränkung auf Wesentliches und Reduktion unnötiger Kompliziertheit. Insbesondere ist es so aufgebaut, daß es für kooperative Übungen in einem für Anfänger überschaubaren Kontext gut einsetzbar ist.
- (4) Kooperative Arbeit mit dem Gerüst impliziert kooperative Arbeit am Rechnernetz, d.h. wir proben gleichzeitig die Nutzung des Netzes in der Lehre.
- (5) Es eignet sich zur anschaulichen Erklärung wichtiger objekt- und komponentenorientierter Techniken und Entwurfsmuster anhand konkreter, exemplarischer Anwendungen.
- (6) Das Gerüst wird evolutionär - nach Auswertung der mit ihm gewonnenen Erfahrungen - und partizipativ - unter Mitwirkung von Studierenden - weiterentwickelt.

Wir kennen kein Gerüst, das diese Ideen vereint, insbesondere kein kommerziell verfügbares. Deshalb haben wir mit einer Eigenentwicklung im Rahmen der Möglichkeiten einer FH begonnen. Das Gerüst soll kein vollständiges, abgeschlossenes, komplexes Anwendungspaket werden, sondern ein offener, erweiterungsfähiger Baukasten, bestehend aus einzelnen, überschaubaren, verständlichen Bausteinen, die in verschiedenen Zuständen sein können: entworfen, spezifiziert, teilimplementiert, implementiert, gete-

stet, reimplementiert usw. Die Bausteine können zu komplexeren Bausteinen komponiert werden - dies kann im Rahmen der Weiterentwicklung des Gerüsts geschehen, vor allem aber im Rahmen von Übungen, die die Studierenden einzeln oder in Gruppenarbeit bewältigen. Dabei können auch neue Studien- und Arbeitsformen erprobt werden; Praktika und Tutorien werden eine zunehmend wichtige Rolle spielen.

## 2.3 Vorerfahrungen

Wir haben im SS 1996 damit begonnen, die Informatik-Grundausbildung auf der Basis der Objekt- und Komponententechnologie neu zu gestalten. Ähnliche Maßnahmen gibt es seit WS 1994/95 am FB Automatisierungstechnik der FH Reutlingen. Erfahrungen aus dieser Aktivität sind in [3] bis [8] veröffentlicht. In diesem Zeitraum haben wir eine exemplarische Modul- und Klassenbibliothek für Lehrzwecke entwickelt, die als Basis für weitere Arbeiten dient.

Über Erfahrungen mit Entwurfsmustern und Gerüsten in der Lehre an der FH Konstanz berichtet Schmid [10]. Im Unterschied dazu wollen wir am FB Elektronik der FH Reutlingen ein speziell für die Informatik-Grundausbildung von Elektronikern geeignetes Gerüst entwickeln.

Wir haben im Informatikunterricht des Grundstudiums gelegentlich elektrotechnische Probleme und ihre algorithmischen Lösungen als Beispiele gewählt, um daran Programmier-techniken darzustellen und zu üben. Unsere Absicht, den Studierenden über Inhalte ihres Hauptfachs den Zugang zu informatischen Themen zu erleichtern, hat sich jedoch kaum erfüllt. Ein Grund dafür ist, daß die parallel laufenden Lehrveranstaltungen über Informatik und elektrotechnische Grundlagen schwer zu synchronisieren sind. Kurz gesagt: Die Studierenden wissen noch zu wenig über die Elektrotechnik.

- Wählt man ein elektrotechnisches Problem, das die Studierenden kennen, so ist der programmiertechnische Gehalt der Lösung oft ziemlich gering. Beispielsweise lassen sich physikalische Naturkonstanten als Konstanten eines Programms darstellen, elektrotechnische Formeln kann man direkt als Funktionsprozeduren formulieren.
- Wählt man ein elektrotechnisches Problem, zu dessen Lösung man die Programmier-technik braucht, die man dem Lehrplan entsprechend gerade vermitteln will, so kennen die Studierenden das elektrotechnische Problem oder die zugrundeliegende Mathematik oft noch nicht. Da z.B. komplexe Zahlen erst im 2. Semester in Mathematik erscheinen, kann man sie nicht im 1. Semester in Informatik als Beispiel für eine Klasse heranziehen.

Ein typisches Beispiel ist eine Programmieraufgabe, die wir mehrfach im 2. Semester gestellt haben: Die Ortskurve eines doppelten RC-Glieds berechnen und tabellarisch und graphisch ausgeben. Die Studierenden kennen den Begriff der Ortskurve vor dem 4. Semester nicht und sind wohl daher nicht gewillt, die vorgegebene Formel zu akzeptieren, von ihrer Bedeutung, ihrem Zweck abzusehen und sie in einfache Funktionsprozedur umzusetzen. Es fällt ihnen schwer, vom elektrotechnischen Problem „Interpretation einer Ortskurve“ zu abstrahieren und sich dem programmiertechnischen Problem „graphische Ausgabe einer durch eine parametrisierte Formel gegebenen Kurve“ zuzuwenden. Auf sie wirkt es eher abschreckend, ein Problem einer anderen Lehrveranstaltung auch noch in das Programmierpraktikum zu „verschleppen“.

## 2.4 Lehrsprache und Entwicklungsumgebung

Wir setzen in der Informatik-Grundausbildung **Component Pascal** als Lehrsprache ein, eine modulare, objekt- und komponentenorientierte Programmiersprache in der Entwicklungslinie Pascal - Modula - Oberon (für Literaturhinweise siehe [3], [4], [5]). Die zugehörige Entwicklungsumgebung namens **BlackBox** Component Builder von Oberon microsystems ist ein plattformunabhängiges komponentenorientiertes Gerüst mit Varianten für die Plattformen Apple und Microsoft (Windows, OLE, COM, DCOM, ActiveX). Von BlackBox gibt es eine kostenlose Ausbildungsversion. Nähere Informationen findet man unter <http://www.oberon.ch>.

Auf allen Rechnern, die am FB Elektronik für die Informatikausbildung in Praktikumsräumen und Software-Labors bereitstehen, ist BlackBox in einer Server-Version verfügbar. Für das Projekt stehen folgende Komponenten zur Benutzung bereit:

- Alle Standardsubsysteme von BlackBox.
- Die am FB Elektronik entwickelten Subsysteme Basis, Containers, Generals, Graph, Math, Test, Utilities.

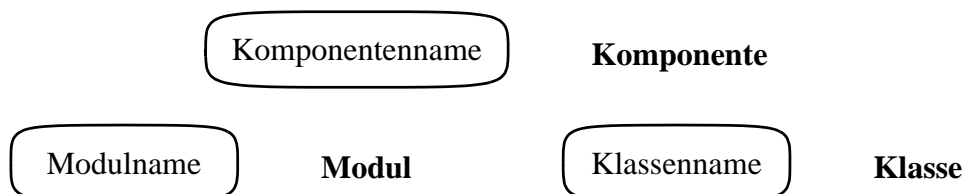
## 2.5 Terminologie

Wir definieren hier projektweit benutzte Begriffe und Notationen, wobei wir Grundkenntnisse der Objekttechnologie voraussetzen. Wir orientieren uns an Begriffen und Bezeichnungen, über die sich die Fachwelt weitgehend einig ist. Darüber hinaus versuchen wir, Begriffe aus verschiedenen Kulturen (objektorientierte Analyse, Eiffel, Oberon) in eine multikulturelle Terminologie zu integrieren. Dabei müssen wir auftretende Namenskonflikte lösen (z.B. Kommando in Eiffel, Oberon).

### 2.5.1 Graphische Notation

Zur graphischen Darstellung von Analyse- und Entwurfsmodellen verwenden wir eine Notation, die sich an die Unified Modeling Language (UML) anlehnt [1]. Wir erlauben uns aber manchmal Abweichungen und Erweiterungen. (Die UML weist einige Mängel auf, insbesondere unterstützt sie die Vertragsmethode nicht.) Die folgenden Bilder zeigen die wichtigsten graphischen Modellierungselemente zusammen mit Begriffsdefinitionen. Die Umrahmung von Komponenten lassen wir meist weg.

**Bild 1 Graphische Modellierungselemente**

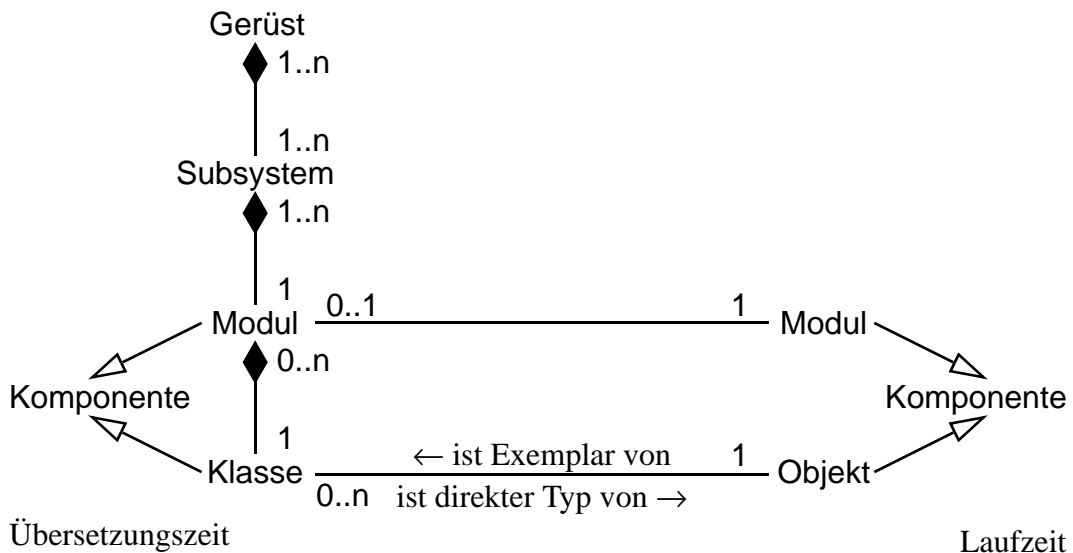


### 2.5.2 Komponenten

Ein **System** oder **Gerüst** besteht aus einer Menge von Subsystemen. Ein **Subsystem** besteht aus einer Menge von Modulen. Klassen sind Bestandteile von Modulen. Ein

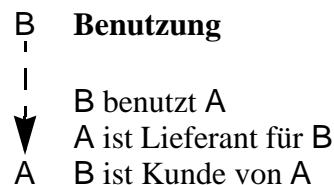
**Modul** ist ein einzelnes Exemplar einer abstrakten Datenstruktur. Eine **Klasse** ist ein erweiterbarer abstrakter Datentyp. Ein Exemplar einer Klasse heißt **Objekt**. Eine **Komponente** zur Übersetzungszeit ist ein Modul oder eine Klasse, zur Laufzeit ein Modul oder ein Objekt. (Dieser Komponentenbegriff unterscheidet sich von dem in [11].)

**Bild 2 Beziehungen zwischen Grundbegriffen**



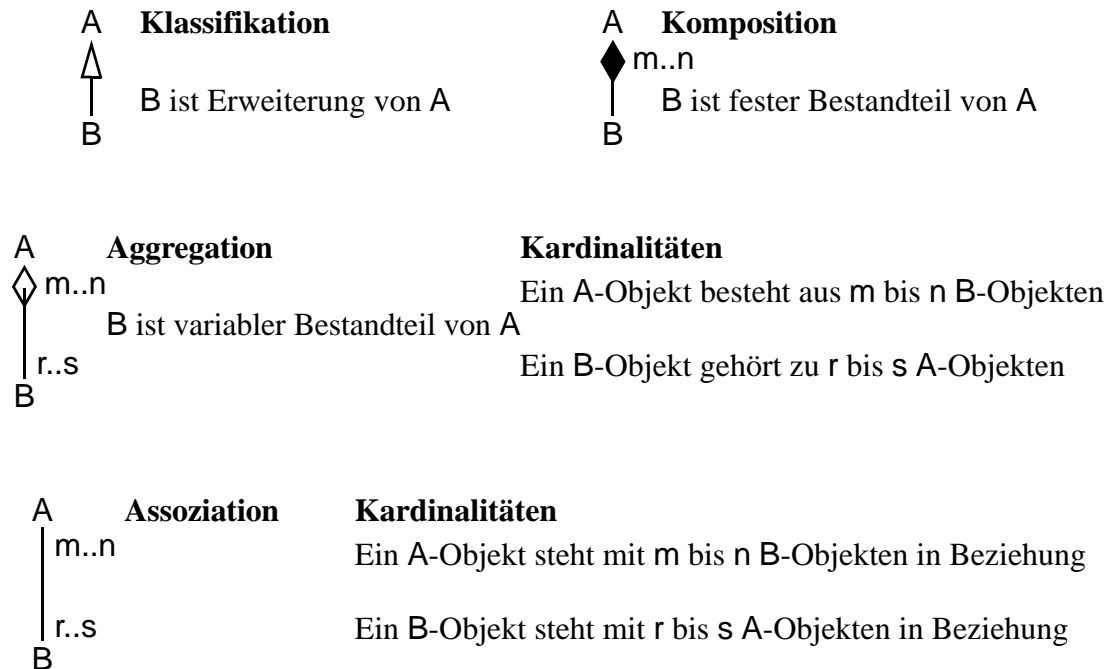
Welche Beziehungen zwischen Komponenten existieren? Allgemein ist die **Benutzung**: Eine nutzende Komponente heißt **Kunde**, eine benutzte **Lieferant**.

**Bild 3 Benutzung**

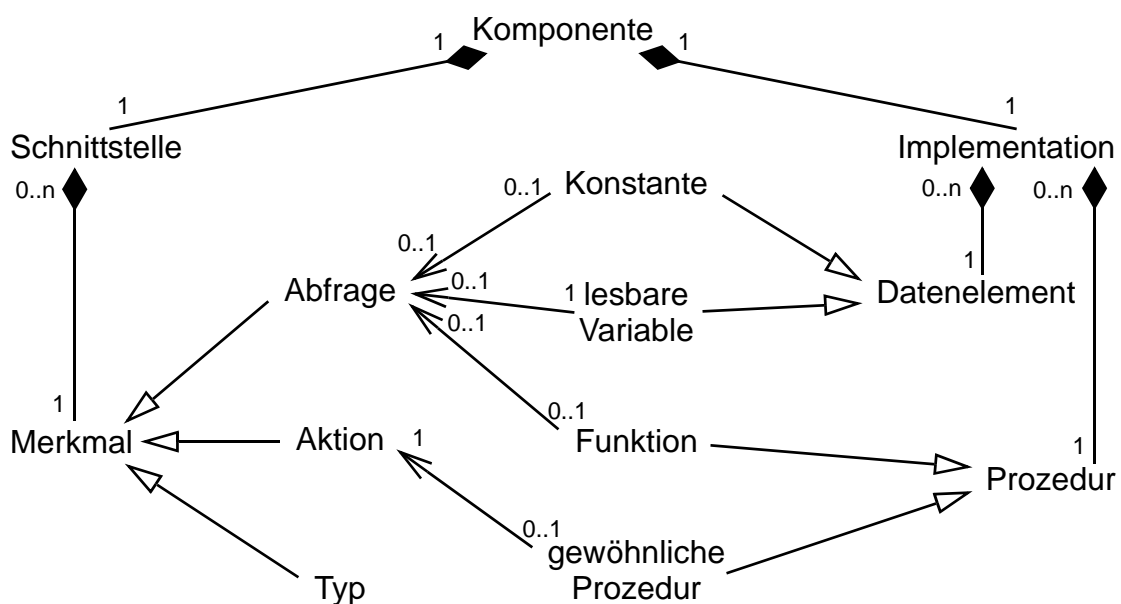


Bei Klassen gibt es die **Klassifikation** (Generalisierung, Spezialisierung, Vererbung): Eine Klasse B kann als **Erweiterung** einer Klasse A definiert sein; A heißt dann **Basis-klasse** von B. Auf der Ebene der Analyse gibt es zwischen Modulen und Klassen sowie Klassen und Klassen außerdem **Bestandteil-** und **allgemeinere Beziehungen** wie **Komposition**, **Aggregation** und **Assoziation**, die bei der Implementierung alle auf die Benutzung abgebildet werden. Ein Objekt einer Klasse kann sich aus Objekten anderer Klassen zusammensetzen:

- ein **Ganzes** und seine **Teile**,
- eine **Menge** und ihre **Elemente**,
- ein **Behälter** und sein **Inhalt**.

**Bild 4 Klassifikation, Bestandteil- und allgemeine Beziehungen**

Jede Komponente hat eine Schnittstelle und eine Implementation. Die **Schnittstelle** ist für Kunden der Komponente nutzbar, sie besteht aus exportierten **Merkmalen**, bei denen wir Abfragen, Typen und Aktionen unterscheiden. Abfragen und Aktionen beschreiben das **Verhalten** einer Komponente. **Typen** dienen der Spezifikation von Parametern und Ergebnissen von Abfragen und Aktionen. Die **Implementation** ist vor Zugriffen von außerhalb der Komponente geschützt, sie besteht aus Daten und Algorithmen, die den **Zustand** und die **Struktur** der Komponente festlegen.

**Bild 5 Bestandteile von Komponenten**

Eine **Abfrage** liefert Informationen über ihre Komponente oder ihre Eingabeparameter. Eine **Aktion** verändert den Zustand ihrer Komponente oder ihrer Ausgabeparameter. Eine Abfrage kann durch eine Konstante, eine schreibgeschützte Variable oder eine seiteneffektfreie Funktion implementiert sein. Eine Aktion ist durch eine gewöhnliche Prozedur implementiert. Eine **Prozedur** ist eine Funktionsprozedur oder eine gewöhnliche Prozedur. (Die letztgenannten Bezeichnungen entstammen der Oberon-Kultur.)

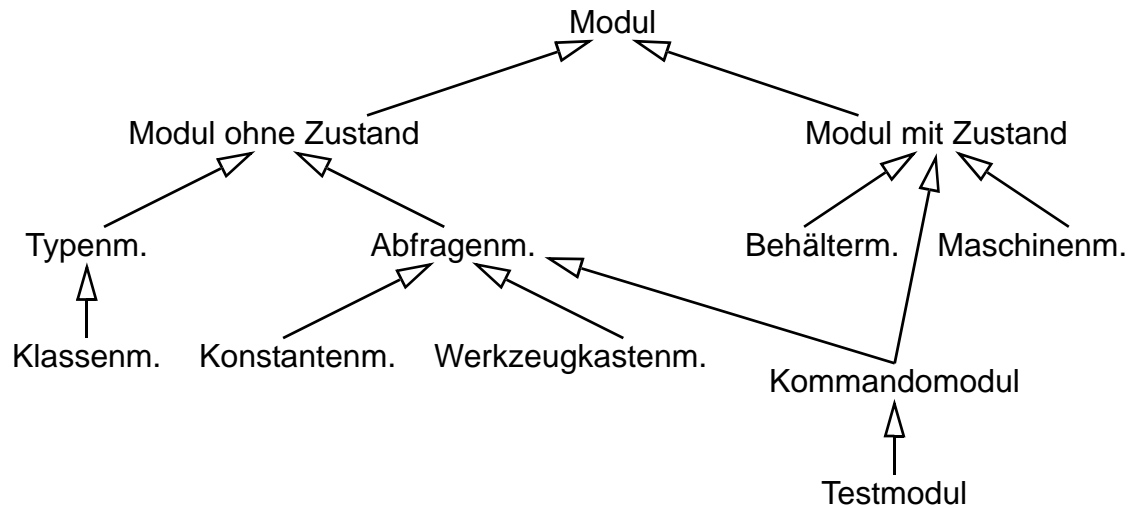
### 2.5.3 Module

Wir untersuchen Module nach drei Kriterien:

- Hat das Modul einen Zustand?
- Enthält die Modulschnittstelle Merkmale unterschiedlicher Art oder nur von einer bestimmten Art?
- Bietet das Modul eine Schnittstelle zur Benutzungsoberfläche? Ist es von dort aus aufrufbar (Eingabe) oder produziert es eine dort sichtbare Ausgabe?

Daraus ergeben sich verschiedene Arten von Modulen. Die folgenden **Modularten** sind „rein“; in der Praxis kommen oft Mischungen von zwei oder mehr Arten vor.

- **Module mit Zustand** müssen Aktionen bieten, die den Modulzustand ändern. Wir unterscheiden zwei Arten:
  - Die allgemeinste Art ist das **Maschinenmodul**: Seine Abfragen und Aktionen stellen eine abstrakte Maschine dar. Kunden können den Maschinenzustand mit Aktionen verändern und mit Abfragen Auskunft über den Zustand erhalten.
  - Ein **Behältermodul** kann Elemente eines Typs aufnehmen, speichern und wieder abgeben. Sein Zustand entspricht den gespeicherten Elementen.
- Bei **Modulen ohne Zustand** unterscheiden wir auch zwei Arten:
  - Ein **Abfragenmodul** bietet an seiner Schnittstelle nur Abfragen, keine Aktion. (Deshalb kann es keinen veränderbaren Zustand haben.) Es gibt zwei Unterarten:
    - ◆ Ein **Konstantenmodul** stellt eine Schnittstelle aus Konstanten zur Verfügung.
    - ◆ Ein **Werkzeugkastenmodul** bietet eine Schnittstelle mit parametrisierten Prozeduren. Kunden können diese Prozeduren nutzen, um übergebene Materialien prüfen oder bearbeiten zu lassen.
  - Ein **Typenmodul** stellt eine Schnittstelle aus Typen zur Verfügung. Kunden können diese Typen nutzen, um Größen zu vereinbaren. Ein Spezialfall ist das **Klassenmodul**: Es definiert eine oder mehrere Klassen.
- Ein **Kommandomodul** bietet an seiner Schnittstelle Kommandos. Ein **Kommando** ist eine parameterlose Aktion, die von der Benutzungsoberfläche aus aufrufbar ist. Es gibt Kommandomodule mit und ohne Zustand. Kommandos können eine Ein-/Ausgabeschnittstelle bieten. Kommandos mit Eingabe verhalten sich wie Prozeduren mit eingeschränkter Signatur. Kommandos mit reiner Ausgabe, d.h. sonst zustandserhaltend, verhalten sich wie Abfragen.
  - Ein **Testmodul** ist ein spezielles Kommandomodul zum Testen eines anderen Moduls.

**Bild 6 Modulararten**

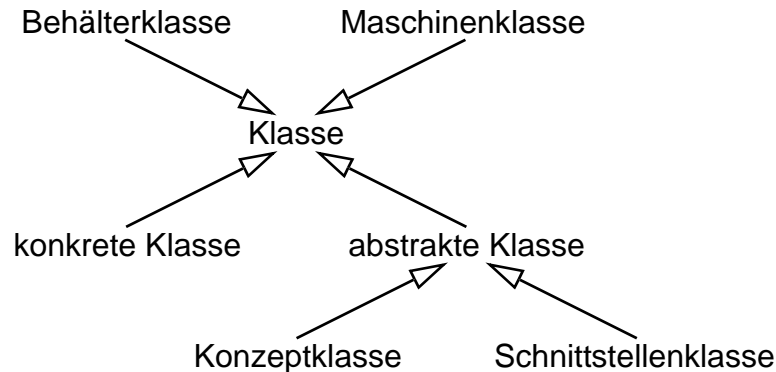
### 2.5.4 Klassen

Bei Klassen sind die Kriterien zur Unterscheidung der Modulararten nicht sinnvoll anwendbar, da eine Klasse stets Objekte mit eigenen Zuständen beschreibt. Sonst wären die Objekte nicht unterscheidbar und besser durch ein Modul zu realisieren. Mit einem Zustand muß es Abfragen und Aktionen geben. Klassen können weder Konstanten noch Typen noch Kommandos exportieren. Daher bleiben folgende **Klassenarten**, analog zu obigen Modulararten:

- Eine **Maschinenklasse** beschreibt Objekte, die abstrakte Maschinen darstellen.
- Eine **Behälterklasse** beschreibt Objekte, die Elemente aufnehmen, speichern und wieder abgeben können.

Die Erweiterbarkeit von Klassen bietet einen mächtigen Abstraktionsmechanismus. Daraus ergeben sich weitere Klassenarten, hier wieder in „reiner“ Form, praktisch oft gemischt:

- Eine **abstrakte Klasse** bietet eine für Erweiterungen nutzbare Schnittstelle, hat aber keine vollständige Implementation.
  - Eine **Konzeptklasse** ist eine abstrakte Klasse, die ein bestimmtes Konzept, eine Idee darstellt.
  - Eine **Schnittstellenklasse** ist eine abstrakte Klasse, die eine vollständige Schnittstelle bietet.
- Eine **konkrete Klasse** ist vollständig implementiert, von ihr können Objekte erzeugt werden.

**Bild 7 Klassenarten**

### 2.5.5 Verträge

Wir ergänzen die Kunden-Lieferanten-Metapher um Verträge zwischen Kunden und Lieferanten. Eine **Zusicherung** ist eine Bedingung an einer bestimmten Programmstelle, die bei jedem Ablauf des Programms erfüllt sein muß, damit das Programm als korrekt gilt. Wir unterscheiden folgende Arten von Zusicherungen:

- **Vorbedingungen** einer Prozedur müssen beim Aufruf der Prozedur erfüllt sein. Dafür zu sorgen ist die Aufgabe des Kunden.
- **Nachbedingungen** einer Prozedur müssen bei der Rückkehr aus einem Prozeduraufruf erfüllt sein. Dafür zu sorgen ist die Aufgabe des Lieferanten.
- **Invarianten** einer Komponente gelten immer, wenn die Komponente nicht aktiv ist, also vor und nach jedem Aufruf. Invarianten sind implizite Vor- und Nachbedingungen aller Abfragen und Aktionen einer Komponente.

Eine Komponente läßt sich mit solchen Zusicherungen formal spezifizieren und testen. Die Gesamtheit der Invarianten und Vor- und Nachbedingungen einer Komponente stellt einen **Vertrag** dar, den die Komponente ihren Kunden bietet. **Spezifikation, Entwurf und Programmieren durch Vertrag** ist eine Methode, die Zusicherungen in diesem Sinne systematisch einsetzt [9].

### 2.5.6 Rohe und gare Module

Das Modul ist nicht nur Einheit des Zerlegens, Speicherns, Übersetzens und Ladens, sondern auch **Lehr- und Lerneinheit**. Unter didaktischen Aspekten unterscheiden wir zwei Arten:

- Ein **rohes** Modul dient nur dem didaktischen Zweck, eine bestimmte Programmier-technik zu demonstrieren. Es kann ein Zwischenergebnis, eine Lösung in einer Folge von immer besseren Lösungen für dasselbe Problem sein. Es dient kaum als Lieferant.
- Ein **gares** Modul hat sich über seinen didaktischen Zweck hinaus einem professionellen Zustand genähert. Es stellt die beste erreichte Lösung eines Problems dar, dient als Lieferantenmodul für Lösungen anderer Probleme und ist somit wiederverwendbar.

Zwischen rohen und garen Modulen ist natürlich Platz für halbrohe, halb-gare Module. Roh heißt nicht unfertig! Alle Module sind vollständig spezifiziert, dokumentiert, aus-

führbar und getestet. Die Studierenden brauchen sie als Anschauungsmaterial. Freilich werden in der professionellen Praxis rohe Module nicht konserviert, sondern zu garen Modulen weiterentwickelt; aufbewahrt wird dort nur der beste Stand, nicht der zweitbeste. In der Lehre können wir uns nicht darauf beschränken, eine komplexe Lösung eines komplexen Problems zu demonstrieren. Wir müssen auch die Schritte dorthin vermitteln, und dazu dienen rohe Module.

## 3 Anforderungen und Konzeption

### 3.1 Teilziele der ersten Projektphase

Hier in der ersten Phase des Projekts ist die Gestaltung des exemplarischen Gerüsts für Lehrzwecke zu konzipieren. Dazu sind wesentliche Techniken auszuwählen und Komponenten zu definieren, die dann schrittweise zu realisieren und in der Lehre zu erproben sind.

Die erste Aufgabe lautet: Finde für Studierende der Elektronik im 1. und 2. Semester ein geeignetes Thema, mit dem sich folgende Konzepte und Techniken gut veranschaulichen, erklären und üben lassen. Dabei ist der Zweck die Software-Technik, das Mittel die Elektronik. Wir wollen nicht Lehrveranstaltungen über elektrotechnische Grundlagen ersetzen, sondern die Elektronik als Vehikel benutzen, um die Software-Technik in den Vordergrund zu schieben.

- (1) Modularisierung und Datenabstraktion, Module und abstrakte Datenstrukturen, verschiedene Modularten.
- (2) Kunden-Lieferanten-Modell, Spezifikation, Entwurf und Programmieren durch Vertrag.
- (3) Statische Modulstrukturen, Benutzungshierarchien.
- (4) Typisierung und Klassifikation, abstrakte Datentypen und Klassen, verschiedene Klassenarten.
- (5) Komposition, Aggregation, Assoziation, Entwurfsmuster.
- (6) Statische Klassenstrukturen, Klassifikations- und Kompositionshierarchien.
- (7) Polymorphie und dynamisches Binden.
- (8) Wert- und Referenzsemantik.
- (9) Dynamische Objektstrukturen, Aggregations- und Assoziationsstrukturen.

Welcher Themenbereich kann diese Anforderungen erfüllen? Wir haben eine Reihe von Themen wie Simulationen von Fahrstühlen und Autoverleihen diskutiert. Die Themen

- Simulation einer Telefonvermittlungsanlage,
- Modellierung elektronischer Bauelemente

haben einen starken Bezug zur Elektronik und den im Studiengang gelehrteten Inhalten und kommen daher in die engere Auswahl.

### 3.2 Simulation einer Telefonvermittlungsanlage

Eine naheliegende Zerlegung dieses Problembereichs in Komponenten ist, Telefone als Maschinenklasse zu modellieren, eine Vermittlungsanlage als Maschinenmodul oder Klasse (falls mehrere Exemplare gefordert), und den Simulator als Kommandomodul. Eine Analyse ergibt folgende Argumente:

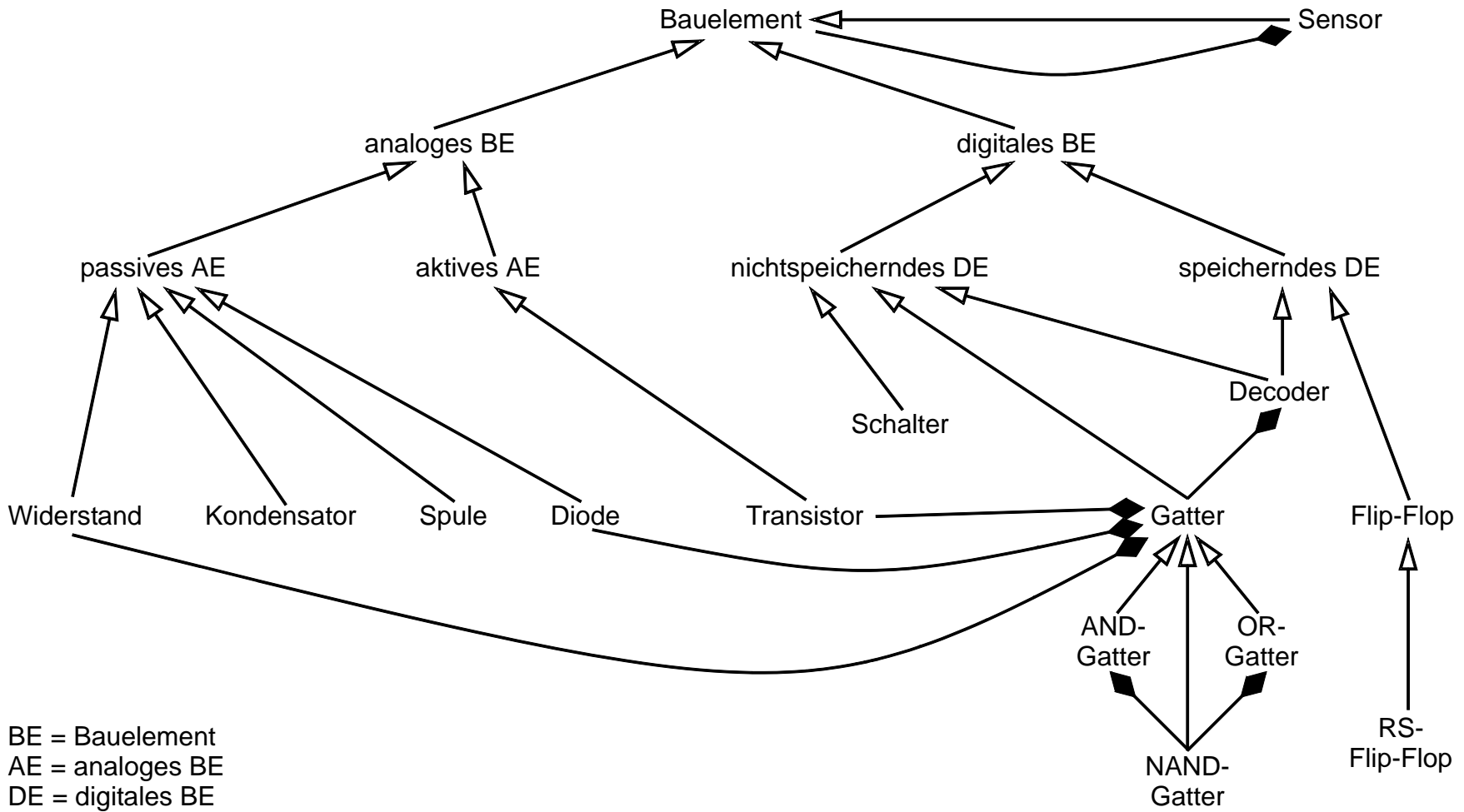
- + Das Verhalten der Komponenten ist mit Zustandsmodellen darstellbar. Dies ist günstig, da Zustandsübergangsdiagramme in Grundvorlesungen im 1. Semester behandelt werden.
- + Das informatische Thema Daten- bzw. Objektstrukturen läßt sich ausgiebig üben. Beispielsweise kann eine Vermittlungsanlage Telefone mittels Behälterklassen verwalten. Diese Behälter können speicherplatz- oder suchzeitoptimierte Listen oder Bäume sein. Jedoch setzen dynamische Objektstrukturen einige Programmierkenntnisse voraus, die wir erst im 2. Semester vermitteln.
- + Im Simulator können verschiedene Ablaufsteuerungsstrategien realisiert und untersucht werden. Sie können auf Monte-Carlo-Methoden basieren. Ein Zufallszahlengenerator wird ein wichtiges Element des Simulators sein. Dem stehen geringe wahrscheinlichkeitstheoretische Grundkenntnisse der Studierenden entgegen.
- + Die Kommunikation zwischen Telefonen und Vermittlungsanlage läßt sich mittels Call-Back-Mechanismen realisieren. Die entsprechenden Entwurfsmuster setzen allerdings wieder einige Programmiererfahrung voraus.
- + Das Thema hat enge Beziehungen zu dem für die Elektronik wichtigen Gebiet der Kommunikationsprotokolle. Dieses Fachgebiet wird allerdings erst ab dem 4. Semester behandelt, so daß wir wieder keine Kenntnisse voraussetzen können.
- Das Thema bietet über die angedeutete Zerlegung in Grundkomponenten hinaus wenig Anhaltspunkte, Klassifikations-, Aggregations- und Assoziationsstrukturen zu studieren.
- Das Thema ist trotz eines relativ einfachen Strukturmodells relativ komplex. Es skaliert nicht beliebig nach unten. So ist z.B. ein einzelnes Telefon praktisch nicht benutzbar.

Bewerten wir die Argumente zusammenfassend: Die Simulation einer Telefonvermittlungsanlage ist zwar ein interessantes und lohnendes Thema im Rahmen der Ausbildung von Software-Entwicklern. Aufgrund seiner Komplexität scheint es uns jedoch nicht optimal für die Grundausbildung geeignet.

### **3.3 Modellierung elektronischer Bauelemente**

Bei diesem Thema gehen wir von einem vorliegenden Klassenstrukturmodell aus, das in Bild 8 dargestellt ist. Das Entwurfsmuster Kompositum kommt mehrfach darin vor.

**Bild 8 Klassenstrukturmodell für elektronische Bauelemente**



Aus diesem umfangreichen Strukturmodell ist ein Teil auszuwählen, bei dem die Projektarbeit beginnen soll. Für den Bereich der **passiven analogen Bauelemente** sprechen folgende Gründe:

- + Passive analoge Bauelemente sind einfache, aber wichtige Grundelemente der Elektronik.
- + Sie werden in den elektrotechnischen Grundvorlesungen im 1. und 2. Semester ausführlich behandelt und sind so den Studierenden hinreichend bekannt.
- + Es sind **Zweipole**, d.h. sie besitzen einen Eingang und einen Ausgang.
- + Sie sind durch wenige Größen leicht zu beschreiben (z.B. der Ohmsche Widerstand durch seinen Widerstandswert).
- + Die beschreibenden Größen stehen in einfachem Zusammenhang mit Strom und Spannung (z.B.  $U = R \cdot I$ ). So lassen sich bei Festhalten einer Größe und Ändern einer zweiten die Auswirkungen auf die dritte Größe leicht darstellen.
- + Mehrere Zweipole können einfach miteinander verbunden werden. Dabei können wieder Zweipole entstehen.
- + Es existieren hervorragende Klassifikations- und Aggregationsstrukturen.
- + Entwurfsmuster wie Kompositum sind vorhanden, gut erkennbar und so leicht zu veranschaulichen und erklären.
- + Es gibt gute Erweiterungsmöglichkeiten; das Thema skaliert nach unten und oben.

Mögliche Schwachstellen sind:

- Da es keine direkte Kommunikation zwischen den Klassen (z.B. Widerstand und Kondensator) gibt, können Kommunikationsbeziehungen nicht so gut studiert werden.
- Probleme beim Übergang vom Zweipol zum Vierpol:
  - Der Aufwand dafür ist unbekannt und schwer zu schätzen.
  - Den Studierenden fehlt im Grundstudium der elektrotechnische Hintergrund; z.B. wird der RC-Tiefpaß (Vierpol) erst im 4. Semester vollständig behandelt.

Die Argumente abwägend entscheiden wir uns dafür, das Thema **Modellierung von Zweipolen** weiter zu bearbeiten und die Simulation der Telefonvermittlungsanlage zurückzustellen.

## 4 Analyse und Entwurf

Im folgenden setzen wir Grundkenntnisse der Konventionen von BlackBox voraus.

### 4.1 Entwicklungsrichtlinien

- (1) Alle Module sind nach einem einheitlichen Muster dokumentiert (siehe `Generals\Module`). Dokumentationssprache ist Englisch.
- (2) Alle Komponenten sind nach der Vertragsmethode spezifiziert (siehe Anhang A Dokumentation und `Electrics\Docu\ModuleName`).
- (3) Jedem Anwendungsmodul `ElectricsModuleName` mit einer gewissen Komplexität ist ein Testmodul `TestModuleName` zugeordnet.
- (4) Alle Komponenten sind nach einheitlichen Testverfahren getestet (siehe 4.3).

### 4.2 Entwicklungsschritte

- (1) Definiere ein Subsystem `Electrics`, das alle projektspezifischen Module umfaßt.
- (2) Definiere ein gares Konstantenmodul `PhysConstants` für physikalische und elektrotechnische Konstanten.
- (3) Definiere ein gares Typenmodul `Types` für elektrotechnische Größen.

In folgenden Schritten modellieren wir das passive analoge Bauelement **Widerstand** als Einzelstück und als Teil zweipoliger Schaltungen. Anhand von drei Kriterien sind verschiedene Entwurfsentscheidungen zu treffen.

- Anzahl der Exemplare 1 oder n? Abstrakte Datenstruktur oder abstrakter Datentyp? Modul oder Klasse?
  - Benutzung mittels Wertsemantik oder Referenzsemantik?
  - Konfiguration von Schaltungen statisch oder dynamisch?
- (4) Definiere ein rohes Maschinenmodul `Resistor` für einen einzelnen Widerstand. Definiere dazu ein Testmodul.
  - (5) Definiere ein rohes Klassenmodul `ResistorsA` für Einzelwiderstände als konkrete Maschinenklasse. Definiere dazu ein Testmodul.
  - (6) Definiere ein rohes Klassenmodul `ResistorsV` für Einzelwiderstände und einfache, statisch konfigurierte Reihen- und Parallelschaltungen als Maschinenklassen. Faktorisiere aus diesen geeignete Konzept- und Schnittstellenklassen heraus. Demonstriere daran das Entwurfsmuster Kompositum mit Wertobjekten. Definiere dazu ein Testmodul.
  - (7) Definiere ein rohes Klassenmodul `ResistorsR0`, das die Wertobjekte von `ResistorsV` durch Referenzobjekte ersetzt.
  - (8) Definiere ein gares Klassenmodul `ResistorsR` für Einzelwiderstände und dynamisch konfigurierbare Reihen- und Parallelschaltungen beliebiger Schachtelungstiefe als Maschinenklassen. Demonstriere daran das Entwurfsmuster Kompositum

mit Referenzobjekten. Demonstriere mittels geeigneter Konzept- und Schnittstellenklassen die Bearbeitung einer rekursiven polymorphen Objektstruktur. Definiere dazu ein Testmodul.

### 4.3 Testverfahren

Die Testverfahren basieren auf der Vertragsmethode, mit der wir alle Komponenten entwickeln. Jedes Modul und jede Klasse ist mit Invarianten und Vor- und Nachbedingungen von Prozeduren spezifiziert. Dadurch kann sich jedes Modul und jedes Objekt selbst testen, die Testanweisungen sind von vornherein als permanente Spezifikation eingebaut. Module und Objekte müssen zum Testen nur benutzt werden. Zur Laufzeit werden die Zusicherungen geprüft; verletzte Zusicherungen führen zu einem Trap.

Die zugrundegelegte Testmethode ist der Funktions- oder Black-Box-Test, d.h. es wird getestet, ob sich der Prüfling gemäß seiner Spezifikation verhält. Ein Strukturtest scheint nicht erforderlich. Testfälle werden i.a. nach Zufallsverfahren ausgewählt und erzeugt. Für spezielle Situationen können Testfälle aufgrund einer Analyse der Fehlererwartung, der Unstetigkeitsstellen oder von Ursachen-Wirkungen ausgewählt werden.

#### 4.3.1 Testverfahren für Module

Gegeben sei ein beliebiges Modul `ElectricsM` (kein Typenmodul).

- (1) Programmiere in `ElectricsM` eine Prozedur `CheckInvariants`, die die Modul invarianten enthält.
- (2) Programmiere zu jeder Prozedur von `ElectricsM` die Vor- und Nachbedingungen.
- (3) Programmiere in `TestM` ein Kommando `TestOnce`, das jede Aktion von `ElectricsM` in jeder relevanten Situation einmal aufruft. Ist die Aktion parametrisiert, so versorge die Parameter mit Zufallswerten. Die Abfragen von `ElectricsM` brauchen meist nicht extra aufgerufen werden, da sie in Zusicherungen benutzt und dadurch mitgetestet werden. `TestOnce` stellt also einen randomisierten Einzeltest zur Verfügung.
- (4) Programmiere in `TestM` ein Kommando `TestUntilBreak`, das `BasisTestUtilities.UntilBreakDo` mit `TestOnce` als Parameter aufruft. `UntilBreakDo` führt einen randomisierten Dauertest aus, bis ein Abbruch gefordert wird.
- (5) Rufe `TestM.TestUntilBreak` auf. Tritt ein Trap auf, so ist der Test erfolgreich, da ein Fehler erkannt wurde. Über das Trapfenster erhält man alle Laufzeitinformationen über die Trapursache (globale und Kellervariablen, Aufrufkette) mittels Hyperlinks. Meist kann man daraus schnell auf die Fehlerursache schließen. Solange das Kommando schweigsam läuft, ist kein Fehler erkannt.

Bei diesem Verfahren benötigen wir keine „Testausgabe“. Das Quellprogramm wird nicht durch zusätzliche Testanweisungen „verschmutzt“. Ein- und Auskommentieren bzw. bedingtes Übersetzen von Testanweisungen entfällt. Ebenso sparen wir uns die Mühe, Testausgabe zu dechiffrieren.

#### 4.3.2 Testverfahren für Klassen

Zum Testen von Klassen modifizieren wir das Verfahren für Module.

Gegeben sei ein Klassenmodul `ElectricsKs`, das eine Klasse `K` definiert. Enthält das Modul mehrere Klassendefinitionen, so ist das Verfahren auf jede einzelne Klasse anzuwenden.

- (1) Programmiere zu `K` eine Prozedur `CheckInvariants`, die die Klasseninvarianten enthält.
- (2) Programmiere zu jeder Prozedur von `K` die Vor- und Nachbedingungen.
- (3) Gibt es Invarianten, die für eine Menge von Objekten von `K` gelten (nicht nur für ein einzelnes Objekt), so programmiere diese Invarianten in einer Modulprozedur `ElectricsKs.CheckInvariants`.
- (4) Programmiere in `TestKs` ein Kommando `TestKOnce`, das ein Objekt `k` vom Typ `K` erzeugt und jede Aktion von `k` in jeder relevanten Situation einmal aufruft. Ist die Aktion parametrisiert, so versorge die Parameter mit Zufallswerten. Die Abfragen von `k` brauchen wieder meist nicht extra aufgerufen werden. `TestKOnce` bietet wieder einen randomisierten Einzeltest. Gegebenenfalls ruft `TestKOnce` auch `ElectricsKs.CheckInvariants` auf.
- (5) Programmiere in `TestKs` ein Kommando `TestKUntilBreak`, das `BasisTestUtilities.UntilBreakDo` mit `TestKOnce` als Parameter aufruft.
- (6) Rufe `TestKs.TestUntilBreak` auf.

#### 4.4 Benutzungshierarchie der Module

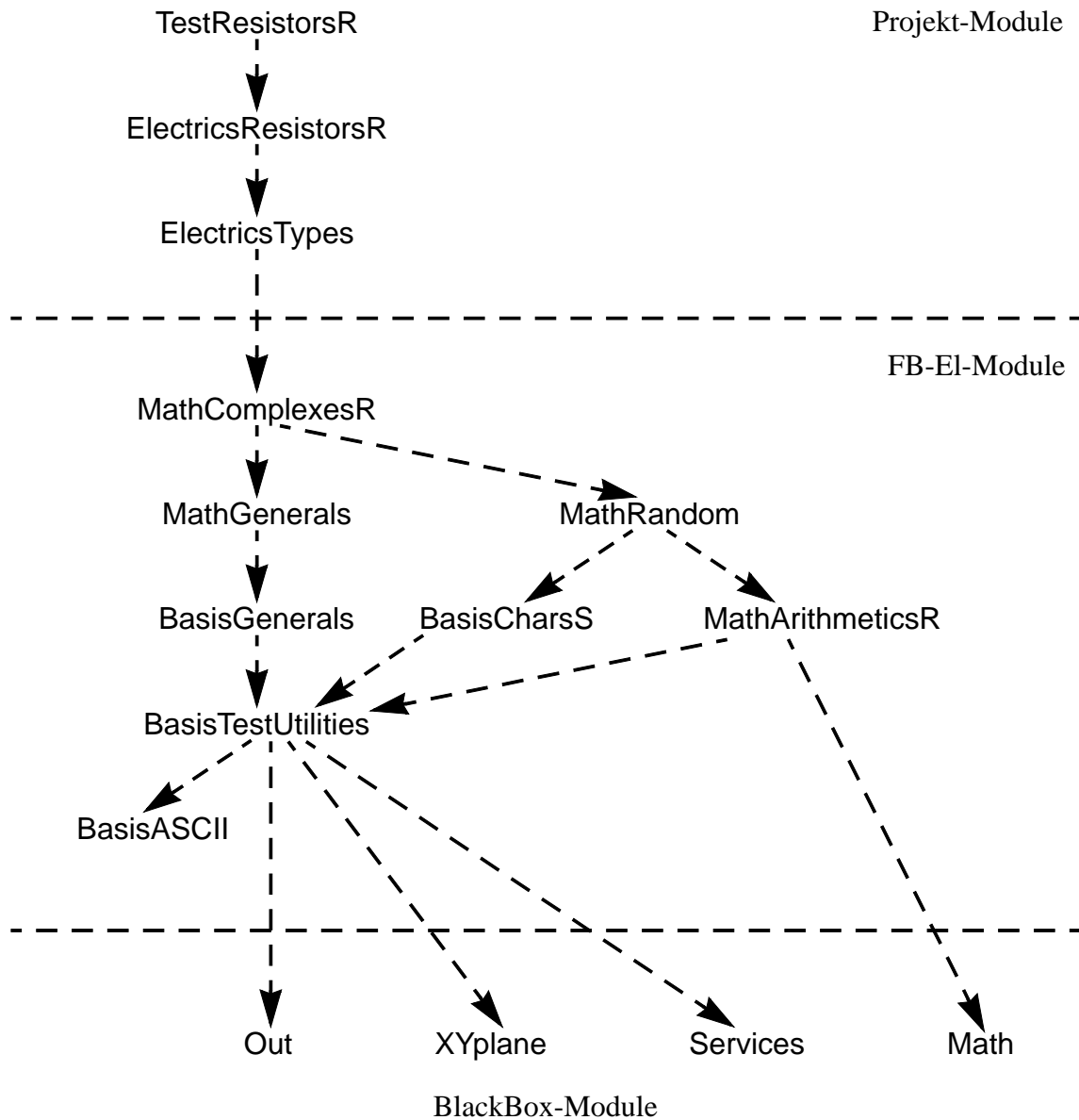
Die zu realisierenden Module des Projekts können vorhandene Module der Entwicklungsplattform benutzen. Es ergeben sich viele Benutzungsbeziehungen, die man nicht alle in einem Diagramm darstellen kann. Wir beschränken uns darauf, exemplarisch vom Testmodul `TestResistorsR` und seinem Prüfling `ElectricsResistorsR` ausgehend die Benutzungsbeziehungen, die naturgemäß eine Hierarchie bilden, in Bild 9 darzustellen.

Um das Diagramm übersichtlich zu halten, lassen wir direkte Benutzungsbeziehungen weg, wenn es schon einen indirekten Pfad vom Kunden zum Lieferanten gibt. Beispielsweise ist `BasisTestUtilities` Lieferant für fast alle anderen Module, wäre also Ziel vieler Kanten, die das Diagramm überladen würden.

Andererseits strukturieren wir die Benutzungshierarchie durch drei Schichten von oben nach unten:

- Module des LARS-Projekts,
- Module des FBs Elektronik,
- Module der BlackBox-Entwicklungsumgebung.

Die BlackBox-Module `Out`, `XYplane` und `Services` benutzen viele weitere Module des BlackBox-Gerüsts. Diese Beziehungen sind hier nicht darzustellen.

**Bild 9 Modul-Benutzungshierarchie mit der Wurzel TestResistorsR**

## 4.5 Die einzelnen Module

### 4.5.1 ElectricsPhysConstants

Das Modul versammelt physikalische Konstanten, die in der Elektrotechnik gebraucht werden. Es dient als exemplarische Lehrinheit für ein Konstantenmodul.

Die Konzentration solcher Konstanten in einem Modul ist aus folgenden Gründen sinnvoll:

- Systemweit einheitliche Namen, angelehnt an die in der Physik verwendeten Standardnamen. (Griechische Buchstaben und Indizes müssen abgebildet werden.)

- Systemweit einheitliche Werte, möglichst genau gewählt (aus seriöser Literaturquelle).
- Zuverlässigkeit: Beim Tippen literaler Werte schleichen sich leicht Fehler ein, deshalb sollen solche Werte nur an einem Platz stehen.
- Wartbarkeit und Erweiterbarkeit.

Zwischen manchen Konstanten bestehen Beziehungen, die sich durch arithmetische Ausdrücke formulieren lassen. Für die Implementation einer abhängigen Konstanten ergeben sich daraus zwei Möglichkeiten: symbolische Konstante oder parameterlose Funktion. Das Modul bietet exemplarisch beide Implementationen.

Beziehungen zwischen Konstanten sind physikalische Gesetze. Sie sind als Modulvarianten zu formulieren. Es genügt, diese Invarianten einmal - beim Laden des Moduls - zu prüfen.

Das Modul bietet außerdem einige Konversionsfunktionen zum Umrechnen physikalischer Größen zwischen SI-Einheiten und anderen noch gebräuchlichen Einheiten. Damit ist das Modul kein reines Konstantenmodul mehr, es bleibt jedoch ein Abfragenmodul mit einem Konstanten- und einem Werkzeugkastenteil. Ein eigenes Modul für Konversionsfunktionen scheint nicht angemessen.

#### 4.5.2 ElectricsTypes

Das Modul versammelt Typdefinitionen für einfache variable physikalische Größen, die in der Elektrotechnik gebraucht werden. Es dient als exemplarische Lehrinheit für ein Typenmodul.

Die Konzentration solcher Typdefinitionen in einem Modul ist aus folgenden Gründen sinnvoll:

- Systemweit einheitliche Namen, die den in der Physik verwendeten entsprechen.
- Dokumentationswert und Lesbarkeit: Der Name des Typs einer Größe drückt ihren Zweck aus, das spart einen Kommentar. Beispiel:

r : ET.Resistance

statt

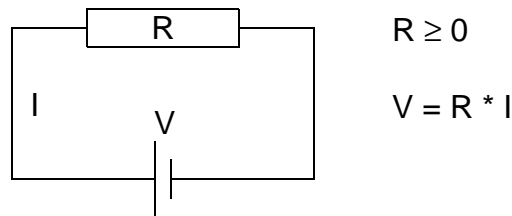
r : REAL (\* Resistance \*)

- Wartbarkeit und Erweiterbarkeit.

Zwischen manchen Größen bestehen in bestimmten Situationen Beziehungen, die sich durch Formeln ausdrücken lassen: physikalische Gesetze. Die Situationen modellieren wir durch Klassen, die in anderen Modulen definiert sind. Die Beziehungen erscheinen dort als Klasseninvarianten.

#### 4.5.3 ElectricsResistor

Das Modul modelliert einen einzelnen Ohmschen Widerstand mit den Größen Widerstand R, Spannung V und Stromstärke I, für die das Ohmsche Gesetz  $V = R * I$  gilt. Es dient als rohe Lehrinheit dazu, sich mit der Spezifikation eines Widerstands zu befassen.

**Bild 10 Ohmscher Widerstand**

Alle Größen sind reellwertig. Jede Größe kann einzeln abgefragt und gesetzt werden.

Der Widerstand kann nicht negativ sein. Diese Wertebereichseinschränkung für  $R$  und das Ohmsche Gesetz erscheinen als Invarianten. Die Aktionen zum Setzen der Größen  $R$  und  $V$  erhalten Vorbedingungen, unter denen die Invariante erhalten werden kann.

Ändert eine der drei Größen ihren Wert, so muß sich i.a. der Wert einer zweiten Größe ändern. Da es drei Größen sind, haben wir die Wahl zwischen zweien.

Wir betrachten  $R$  als statische,  $V$  und  $I$  als dynamische Größen. Ändert sich eine dynamische Größe, so ändert sich auch die andere dynamische Größe, aber nicht die statische.

Es bleibt der Fall: Was passiert, wenn sich  $R$  ändert? Es kann sich  $V$  oder  $I$  ändern (oder beide). Um das Modul flexibel zu halten, d.h. nicht durch eine vielleicht falsche Entscheidung einzuschränken, bieten wir das **Konzept der fixierten Größe**. Genau eine der Größen  $V$  und  $I$  ist fixiert. Mit  $R$  ändert sich dann auch die nicht fixierte Größe. Zu  $V$  und  $I$  gibt es je eine Aktion zum Fixieren und eine Abfrage, ob es fixiert ist.

#### 4.5.4 TestResistor

Es sind die Aktionen zum Setzen der Größen  $R$ ,  $V$ ,  $I$  zu testen. Bei  $R$  sind beide Fixierungszustände zu testen, dabei werden auch die Aktionen zum Fixieren getestet. Die Abfragen werden implizit über die Zusicherungen getestet.

#### 4.5.5 ElectricsResistorsA

Das Modul modelliert einen Ohmschen Widerstand mit den Größen  $R$ ,  $V$  und  $I$ , für die das Ohmsche Gesetz  $V = R * I$  gilt, als abstrakten Datentyp. Es dient als rohe Lehreinheit dazu, den Übergang von einer abstrakten Datenstruktur zu einem abstrakten Datentyp zu veranschaulichen (daher das Suffix *A* für abstract data type im Modulnamen). Das Modul geht durch einige Änderungen, die schematisch auszuführen sind, aus dem Modul `ElectricsResistor` hervor.

Der abstrakte Datentyp `Resistor` erhält natürlich eine Initialisierungsprozedur.

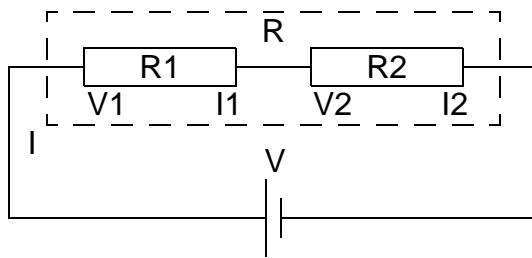
#### 4.5.6 TestResistorsA

Das Modul geht durch einige schematisch ausführbare Änderungen aus dem Modul `TestResistor` hervor.

### 4.5.7 ElectricsResistorsV

Das Modul modelliert verschiedene Arten Ohmscher Widerstände - Einzelwiderstände und Reihen- und Parallelschaltungen aus je zwei Einzelwiderständen - als Klassen. Es dient als rohe Lehrinheit dazu, den Übergang von einem abstrakten Datentyp zu erweiterbaren abstrakten Datentypen - also Klassen - zu veranschaulichen. Außerdem demonstriert es statische Konfigurationen mit Wertsemantik. Als Ausgangspunkt dient das Modul ElectricsResistorsA.

**Bild 11 Reihenschaltung von Widerständen**



$$R \geq 0$$

$$V = R * I$$

$$R = R1 + R2$$

$$V = V1 + V2$$

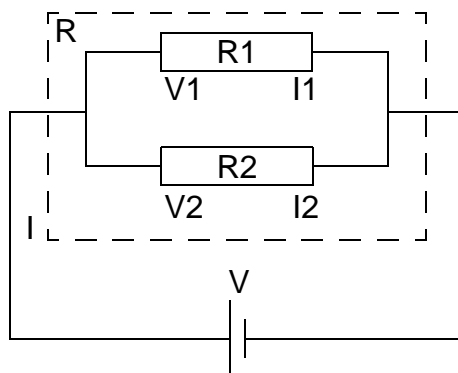
$$I = I1 = I2$$

$$Ri \geq 0$$

$$Vi = Ri * Ii$$

Die Aufgabe erfordert über schematisches Vorgehen hinaus eine objektorientierte Analyse und einen Entwurf. Das Modul dient damit auch dem Zweck, Abstraktions- und Kompositionstechniken zu demonstrieren.

**Bild 12 Parallelschaltung von Widerständen**



$$R \geq 0$$

$$V = R * I$$

$$R = (R1 * R2) / (R1 + R2)$$

$$V = V1 = V2$$

$$I = I1 + I2$$

$$Ri \geq 0$$

$$Vi = Ri * Ii$$

#### 4.5.7.1 Entwurf

Aus den drei Arten von Widerständen sind gemeinsame Merkmale herauszufaktorisieren, um eine abstrakte Klasse für das Konzept Widerstand zu bilden.

Einzelwiderstände können feste Bestandteile von Reihen- und Parallelschaltungen sein. In UML-Begriffen handelt es sich um eine Komposition; in Begriffen der Implementation um Wertsemantik (daher das Suffix V für value im Modulnamen). Da sowohl die Teile als auch das Ganze Spezialfälle von Widerständen sind, liegt das Entwurfsmuster Kompositum vor [2].

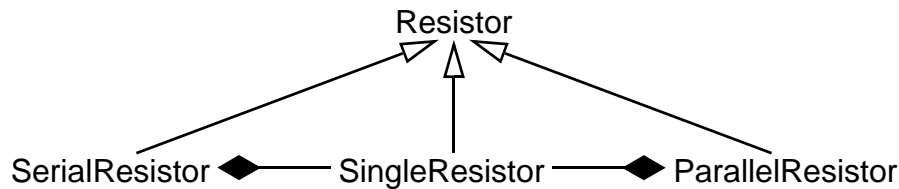
**Bild 13 Ansatz eines Entwurfs mit Kompositum**

Bild 13 zeigt eine naheliegende Klassenstruktur mit diesem Entwurfsmuster. Der Ansatz greift jedoch zu kurz. Der Grund liegt darin: Bei zusammengesetzten Widerständen gilt das Ohmsche Gesetz sowohl für das Ganze als auch die Teile, und es gelten weitere Regeln wie das Kirchhoffsche Gesetz bei Parallelschaltungen. Zusammengesetzte Klassen müssen daher mehr Invarianten garantieren. Für den Zugriff auf die Teile haben wir zwei Entwurfsalternativen:

- (1) Die Teile verbergen und Zugriffe darauf indirekt durch eine erweiterte Schnittstelle des Ganzen ermöglichen.

Die abstrakte Klasse `Resistor` kann damit keine Schnittstellenklasse für die zusammengesetzten Klassen sein. Das Erweitern der Schnittstelle bei konkreten Klassen widerspricht dem Zweck des Abstrahierens. Außerdem muß die erweiterte Schnittstelle die interne Struktur der Klasse reflektieren. Sie kann also nicht allgemein genug sein. Insbesondere ist die Lösung nicht erweiterbar hinsichtlich dynamischer Konfigurationen.

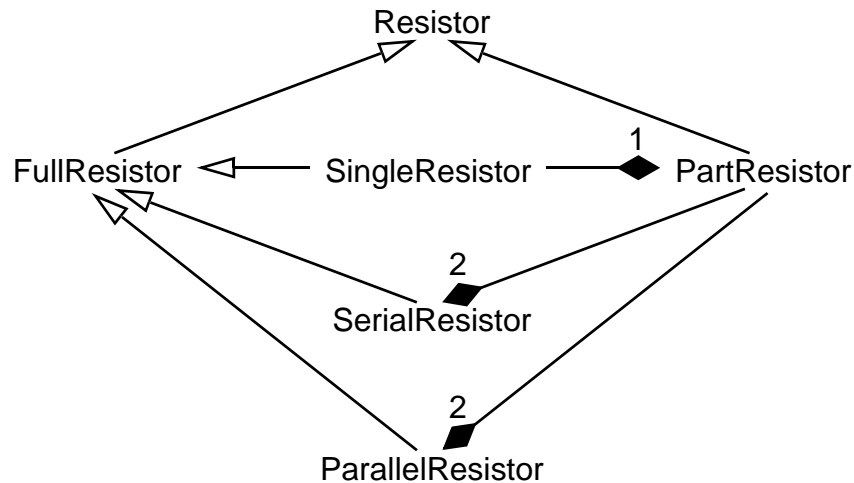
Diese Lösung ist daher nicht attraktiv.

- (2) Die Teile öffentlich halten.

Diese Entwurfsalternative hat nicht die Nachteile von (1). Sie erlaubt jedoch unbeschränkte Zugriffe auf Teile. Da (bzw. solange) die Teile nichts voneinander wissen und nicht, ob und wovon sie Teil sind, kann eine Aktion an einem Teil die Invariante des Ganzen oder des anderen Teils ungültig machen.

Die Entwurfsalternative ist daher nicht korrekt. Sie widerspiegelt übrigens auch nicht den physikalischen Sachverhalt: Man kann z.B. in einer Reihenschaltung nicht einfach den Strom an einem Teilwiderstand ändern.

Um das Problem zu lösen brauchen wir unterschiedliche Rechte für Zugriffe auf das Ganze und seine Teile. Das Ganze braucht selbst vollen Zugriff auf seine Teile, kann seinen Kunden aber nur eingeschränkte Zugriffe auf die Teile gewähren. Da die Lehrsprache dazu keine fertigen Mechanismen bietet, modellieren wir eine Lösung mittels zusätzlicher Klassen mit entsprechenden Schnittstellen.

**Bild 14 Klassendiagramm für statische Konfiguration von Widerständen**

Die abstrakte Klasse **Resistor** bietet eine eingeschränkte Schnittstelle, die für das Ganze und seine Teile paßt. Dazu gehören alle Abfragen, eine Initialisierungsprozedur sowie die Aktion zum Setzen von  $R$ . Von **Resistor** gibt es eine erweiterte Schnittstellenklasse **FullResistor** mit der vollständigen Schnittstelle für das Ganze. Hinzu kommen nur noch die Aktionen zum Setzen und Fixieren von  $V$  und  $I$ . Die bisherigen konkreten Klassen **SingleResistor**, **SerialResistor** und **ParallelResistor** übernehmen diese Schnittstelle.

Weiter gibt es eine konkrete Klasse **PartResistor** mit eingeschränkter Schnittstelle, die ein **SingleResistor**-Objekt enthält und ihrerseits nur zur Komposition zu nutzen ist.

#### 4.5.7.2 Zur Spezifikation

Diese Lösung erlaubt, den Widerstand eines Teils zu ändern; man stelle sich die Teile als Drehwiderstände vor. Wir stellen an das Setzen eines Teilwiderstands allerdings die Vorbedingung

$$I = 0,$$

da es sonst unpraktikabel ist, alle Invarianten zu erhalten. Dagegen genügt beim Setzen des Gesamtwiderstands die Vorbedingung

$$\text{newR} > 0 \text{ OR } V = 0 \text{ OR NOT FixedV.}$$

Diese Bedingung ist schwächer, denn aus  $I = 0$  folgt  $V = R * I = 0$  und daraus  $\text{newR} > 0 \text{ OR } V = 0 \text{ OR NOT FixedV}$ .

Erweiterungsklassen übernehmen die Verträge ihrer Basisklassen, wobei Vorbedingungen schwächer, Nachbedingungen und Invarianten stärker werden dürfen. Die Lehrsprache bietet zwar kein automatisches Erben von Zusicherungen, im Sinne korrekter Software-Entwicklung sind diese Regeln aber diszipliniert einzuhalten. Das Setzen des Widerstands in der Basisklasse **Resistor** ist daher mit der stärkeren Vorbedingung  $I = 0$  zu spezifizieren.

### 4.5.7.3 Bemerkungen

Wir haben versucht, das Konzept der fixierten Größe beim Ändern eines Widerstands von 4.5.3 zu übernehmen. Das führt zu einer weiteren Invarianten, denn das Ganze und alle seine Teile müssen denselben Fixierungszustand haben. Beim Setzen eines Teilwiderstands brauchen wir aber die stärkere Vorbedingung  $l = 0$ , d.h. der Fixierungszustand des Teils spielt gar keine Rolle. Dies zeigt, daß das Konzept nicht leicht auf Konfigurationen übertragbar ist.

Der vorliegende Entwurf ist inflexibel wegen der statischen Konfiguration. Für jede Konfiguration von Widerständen ist eine eigene Klasse zu definieren. Dabei sind nur einstufige Zusammensetzungen möglich; die Teile einer Konfiguration müssen Einzelwiderstände sein. Man kann zwar vorhandene Schnittstellenklassen erweitern, trotzdem bleibt einiges zu programmieren. Diese Nachteile beseitigt der Entwurf 4.5.10.

### 4.5.8 TestResistorsV

Das Modul geht durch Erweiterungen aus dem Modul `TestResistorsA` hervor. Es wird je ein Objekt der konkreten Klassen getestet. Bei zusammengesetzten Objekten sind Zugriffe auf das Ganze und die Teile zu testen.

### 4.5.9 ElectricResistorsR0

Das Modul modelliert dieselbe Situation wie 4.5.7. Der Unterschied liegt in der Implementierung mit Referenz- statt Wertsemantik. Das Modul entsteht aus `ElectricResistorsA` durch minimale schematische Änderungen.

Für dieses Rohmodul ist kein eigenes Testmodul vorgesehen.

### 4.5.10 ElectricResistorsR

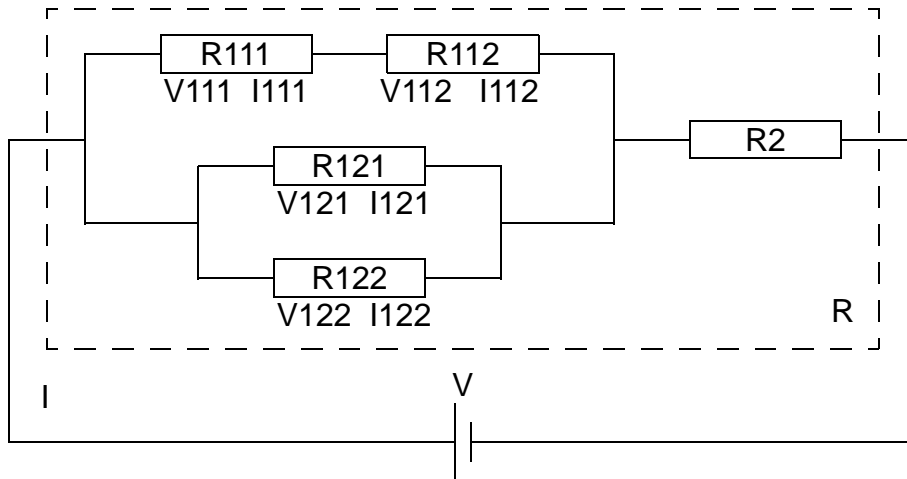
Das Modul modelliert wie `ElectricResistorsV` verschiedene Arten Ohmscher Widerstände - Einzelwiderstände und Reihen- und Parallelschaltungen - als Klassen. Es soll aber als gute Lehrereinheit wiederverwendbar sein. Ausgehend vom Modul `ElectricResistorsV` bzw. `ElectricResistorsR0` sind folgende Änderungen und Erweiterungen am Entwurf zu realisieren:

- (1) Referenzsemantik statt Wertsemantik (daher das Suffix `R` für reference im Modulnamen). Konfiguration durch Aggregation statt Komposition.
- (2) Dynamische Konfigurationen statt statische. Konfigurationen mit beliebig vielen Stufen statt nur einer.
- (3) Neue Schnittstellenklasse durch Faktorisierung zusammengesetzter Klassen.
- (4) Vereinfachung der Klassenstruktur durch Nutzung von Referenzen für Aliasing.
- (5) Weitergehende Abstraktion, indem die Konzeptklasse `Resistor` zu einer Erweiterung der Konzeptklasse `Comparable` gemacht wird.
- (6) Nutzung der Merkmale von `Comparable` und seiner Basisklassen.
- (7) Bereitstellung von Operationen zum Kopieren dynamischer Objektstrukturen.
- (8) Vereinfachung durch Verzicht auf das Konzept der fixierten Größe.

Die Änderung (1) ist bereits mit `ElectricsResistorsR0` realisiert.

Änderung (2) bedeutet, daß man beliebige zweipolige Schaltungen aus Widerständen modellieren kann, und zwar durch dynamisches Erzeugen der entsprechenden Objektstruktur. Bild 15 zeigt ein Beispiel einer solchen Schaltung.

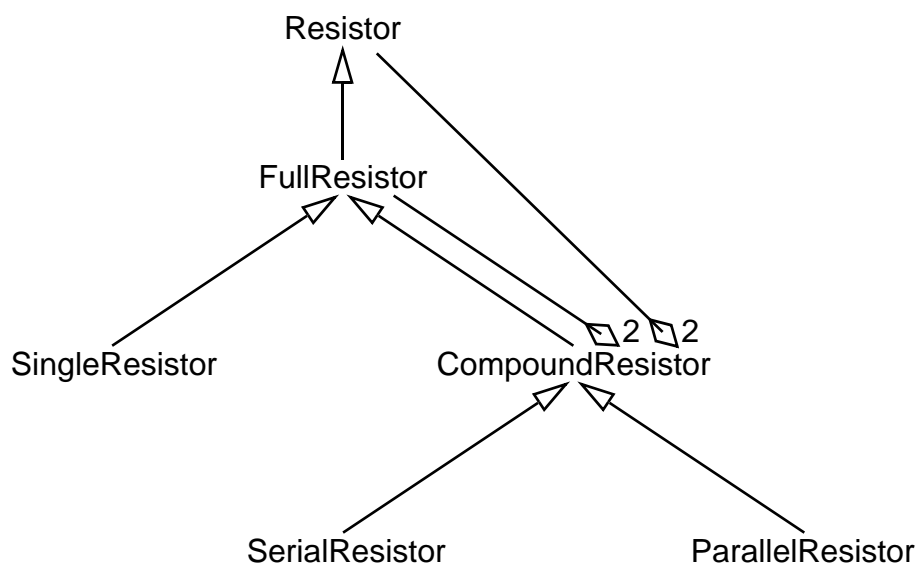
**Bild 15 Schaltung von Widerständen**



#### 4.5.10.1 Entwurf

Auch dieses Modul dient dazu, Abstraktions- und Kompositionstechniken zu demonstrieren. Wir gehen vom Klassendiagramm Bild 14 aus. Gemäß (1) ersetzen wir die Komposition durch Aggregation. Wegen (2) kann das Teil kein Einzelwiderstand mehr sein, sondern wir müssen dazu eine abstrakte Klasse nehmen, die polymorphe Ersetzungen erlaubt. Gemäß (3) bilden wir zu `SerialResistor` und `ParallelResistor` eine abstrakte Schnittstellenklasse `CompoundResistor`.

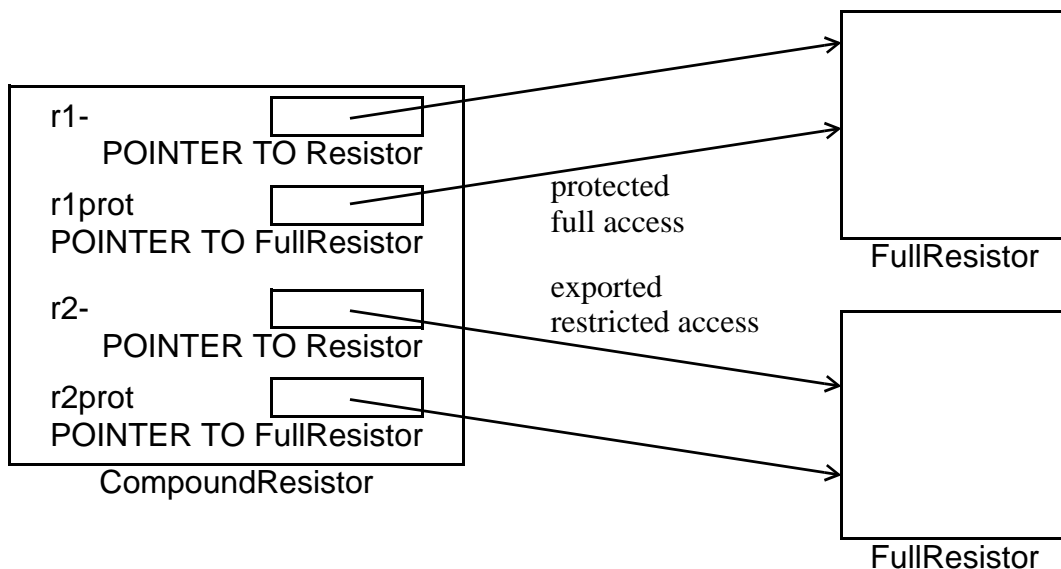
**Bild 16 Klassendiagramm für dynamische Konfiguration von Widerständen**



#### 4.5.10.2 Aliasingtechnik

Wir lassen `CompoundResistor` die bisherige Rolle von `PartResistor` übernehmen; dadurch entfällt `PartResistor`. Die Technik dazu ist Aliasing: zwei Referenzen mit unterschiedlichem Typ und Exportstatus zeigen auf dasselbe polymorphe Objekt. Ein `Resistor`-Zeiger wird exportiert, erlaubt aber nur eingeschränkte Zugriffe auf das Objekt. Der zweite Zeiger auf dasselbe Objekt ist ein `FullResistor`-Zeiger; er erlaubt alle Zugriffe, wird aber nicht exportiert. Bild 17 veranschaulicht diese Technik.

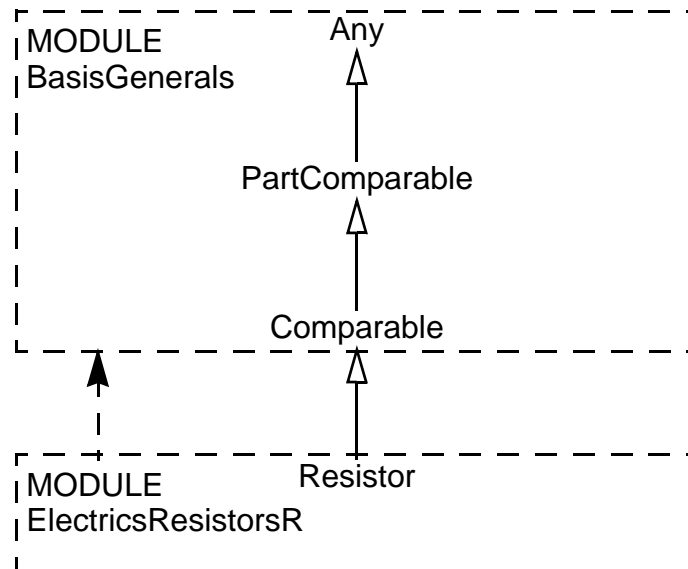
**Bild 17 Objektdiagramm zur Aliasingtechnik**



#### 4.5.10.3 Abstraktionstechnik

Widerstände sind vergleichbare Dinge: Man kann ihren Widerstandswert vergleichen. Um diese Eigenschaft gemäß (5) zu modellieren, nutzen wir eine vorhandene Klassenbibliothek und definieren `Resistor` als Erweiterung der Konzeptklasse `Comparable`, die Vergleichsoperationen und davon abhängige Dienste bereitstellt.

`Comparable` liegt in der Erweiterungslinie `Any` - `PartComparable`, so daß `Resistor` eine Reihe weiterer nützlicher Operationen von `Any` erbt (6). Bild 18 zeigt auch die Verteilung der Klassen auf verschiedene Module.

**Bild 18 Klassendiagramm für Erweiterung allgemeiner Konzeptklassen**

Von Any erbt Resistor die Operationen Copy und Clone zum Kopieren von Objekten. Es handelt sich um die flachen Versionen. Da wir es hier mit dynamischen Objektstrukturen zu tun haben, führen wir auch tiefe Versionen DeepCopy und DeepClone ein, die nicht nur Referenzen auf Objekte, sondern stets die referenzierten Objekte selbst kopieren, wie von (7) gefordert. Die beiden Versionen von Copy und Clone bieten viele Möglichkeiten, darunter:

- Kopiere das Verhalten einer Widerstandsschaltung auf eine andere Schaltung mit gegebener Struktur. Beispielsweise: Wandle eine Reihenschaltung in eine verhaltensgleiche Parallelschaltung um.
- Erstelle eine strukturgleiche Kopie einer gegebenen Schaltung.

#### 4.5.10.4 Polymorphe Rekursion

Die dynamische Objektstruktur zu einer Widerstandsschaltung ist ein binärer Baum. Bekanntlich sind Bäume rekursive Datenstrukturen, die sehr gut mit rekursiven Algorithmen zu bearbeiten sind.

DeepCopy und DeepClone arbeiten direkt und indirekt rekursiv zusammen, aber nicht auf herkömmliche Art, sondern mittels polymorpher Aufrufe. So ruft das DeepCopy einer konkreten zusammengesetzten Klasse das DeepCopy der Basisklasse CompoundResistor, dieses ruft das DeepClone der beiden Teile, welche wiederum DeepCopy aufrufen. Die Rekursion bricht ab bei Einzelwiderständen, also Objekten vom Typ SingleResistor. Man beachte, daß keine Fallunterscheidungsanweisung zum Abbruch der Rekursion benötigt wird. Jeder Fall wird nur durch die Klasse behandelt, zu der er gehört.

#### 4.5.10.5 Zur Spezifikation

Inzwischen haben wir eine Reihe von Erweiterungen gegenüber 4.5.7 entworfen. Um die Komplexität der Klassen etwas zu reduzieren, verzichten wir in diesem Entwurf auf das Konzept der fixierten Größe (8). Dadurch fallen einige Prozeduren und Zusicherungen

weg. Der Preis dafür ist, daß wir die Vorbedingung für das Setzen eines Widerstandswerts generell auf  $I = 0$  verschärfen müssen. Praktisch ist dies aber nicht gravierend: Man kann ja den Stromwert vor dem Ändern des Widerstandswerts speichern und danach restaurieren.

#### **4.5.11 TestResistorsR**

Das Modul geht durch Erweiterungen aus dem Modul TestResistorsV hervor. Zusätzlich sind die Kopieroperationen zu testen. Dynamische Objektstrukturen werden nach Zufallsverfahren erzeugt.

Die zu testenden Klassen bieten von Any geerbte Ausgabeprozeduren. Diese können für Testausgabe genutzt werden. Es ist nützlich, die zufälligen dynamischen Objektstrukturen mittels Testausgabe nachvollziehbar darzustellen; siehe dazu eine exemplarische Testausgabe in Anhang D Testprotokolle.

## 5 Spezifikation und Implementierung

Die Software erstellen wir nach dem Vorgehensmodell der nahtlosen Software-Entwicklung [9]. Dabei sind folgende Aspekte wichtig:

- Die Spezifikationsphase braucht nicht vollständig abgeschlossen sein, bevor mit der Implementierung begonnen wird. Man wechselt zwischen Spezifizieren und Implementieren, bis das Ergebnis gut genug ist. Dies bedeutet jedoch keinen Rückfall in ein Drauflosprogrammieren ohne Spezifikation! Das iterative Vorgehen ist der Software-Praxis besser angemessen.
- Die Spezifikationen aller Komponenten existieren nicht als selbständige Dokumente, sondern sind in die Quellprogrammtexte integriert. Man arbeitet an einem Dokument, beginnend mit der Spezifikation und endend mit Implementierungsdetails. Trotzdem sind die Spezifikations- und Implementationsteile klar zu unterscheiden. Ein Werkzeug könnte die Spezifikation aus einem Quelldokument extrahieren und in einem separaten Dokument speichern.

Details zur Spezifikation und Implementation sind in den Quellprogrammtexten dokumentiert, so daß wir uns hier mit Hinweisen begnügen können. Syntaktische Spezifikationen von Modulen sind exemplarisch in Anhang B Schnittstellen, Quelltexte im Anhang C Quellprogramme zusammengestellt.

Wir bringen hier aber eine statistische Auswertung des Moduls `ElectricsResistorsR`, die Erkenntnisse über den objektorientierten Programmierstil liefern kann.

### 5.1 Statistische Auswertung von `ElectricsResistorsR`

Bild 19 zeigt eine Tabelle mit Maßzahlen des Moduls `ElectricsResistorsR`. Die Spalten verzeichnen die 6 Klassen des Moduls, die Zeilen die insgesamt 23 prozeduralen Merkmale dieser Klassen. Datenmerkmale sind weggelassen.

Ein Tabelleneintrag ist nichtleer, wenn die Klasse die Prozedur **definiert**, d.h. neu definiert oder redefiniert. Ein Eintrag hat die Form

[a,] b + c + d

mit der Bedeutung

- a Anzahl der Parameter,
- b Anzahl der einfachen Anweisungen,
- c Anzahl der strukturierten Anweisungen,
- b Anzahl der Zusicherungen

der Prozedur. Die Zeilen enthalten die Angaben der verschiedenen Versionen einer Prozedur. Die Parameteranzahl erscheint nur bei der Neudefinition.

Die letzte Spalte **Summe** enthält zu jeder Prozedur die Summe der b + c + d über die Klassen.

Die drittletzte Zeile **Summe** enthält zu jeder Klasse die Summe der [a,] b + c + d über die Prozeduren.

**Bild 19 Statistik über ElectricResistorsR**

Klasse	Resistor	Full Resistor	Single Resistor	Compound Resistor	Serial Resistor	Parallel Resistor	Summe
Prozedur							
R	0, 0 + 0 + 0		1 + 0 + 0		1 + 0 + 0	2 + 1 + 0	4 + 1 + 0
V	0, 0 + 0 + 0		1 + 0 + 0		1 + 0 + 0	1 + 0 + 0	3 + 0 + 0
I	0, 0 + 0 + 0		1 + 0 + 0		1 + 0 + 0	1 + 0 + 0	3 + 0 + 0
Level	0, 0 + 0 + 0		1 + 0 + 0	1 + 0 + 0			2 + 0 + 0
NbrOfParts	0, 0 + 0 + 0		1 + 0 + 0	1 + 0 + 0			2 + 0 + 0
Equal	1, 1 + 1 + 1						1 + 1 + 1
LessEqual	1, 1 + 1 + 1						1 + 1 + 1
Clone	0, 0 + 0 + 0	0 + 0 + 0	3 + 0 + 0	0 + 0 + 0	3 + 0 + 0	3 + 0 + 0	9 + 0 + 0
DeepClone		0, 0 + 0 + 0	1 + 0 + 0	0 + 0 + 0	3 + 0 + 0	3 + 0 + 0	7 + 0 + 0
Random		0, 8 + 1 + 0					8 + 1 + 0
Initialized				0, 1 + 0 + 0			1 + 0 + 0
CheckInvariants	0, 0 + 0 + 6			0 + 0 + 4	0 + 0 + 4	0 + 0 + 5	0 + 0 + 19
InitDefault	0, 0 + 0 + 4		3 + 0 + 0	9 + 0 + 0			12 + 0 + 4
InitRandom	0, 0 + 0 + 3		3 + 0 + 0	5 + 0 + 0			8 + 0 + 3
SetR	1, 0 + 0 + 0		1 + 0 + 4		8 + 1 + 4	6 + 1 + 4	16 + 2 + 12

Klasse	Resistor	Full Resistor	Single Resistor	Compound Resistor	Serial Resistor	Parallel Resistor	Summe
Prozedur							
WriteContent	1, 11 + 0 + 0		2 + 0 + 0	3 + 0 + 0	2 + 0 + 0	2 + 0 + 0	20 + 0 + 0
WriteWrapping	1, 7 + 3 + 0						7 + 3 + 0
Write	0, 2 + 0 + 0						2 + 0 + 0
SetV		1, 0 + 0 + 0	2 + 2 + 3		5 + 2 + 3	2 + 0 + 3	9 + 4 + 9
SetI		1, 0 + 0 + 0	1 + 0 + 3		2 + 0 + 1	10 + 1 + 2	13 + 1 + 6
Copy		1, 3 + 1 + 3	2 + 1 + 3	2 + 1 + 0			7 + 3 + 6
Init				2, 4 + 0 + 5			4 + 0 + 5
DeepCopy				1, 4 + 0 + 2	1 + 1 + 1	1 + 1 + 1	6 + 2 + 4
<b>Summe</b>	5, 23 + 5 + 15	3, 23 + 5 + 15	23 + 3 + 13	3, 30 + 1 + 14	27 + 4 + 13	31 + 4 + 15	11, 145 + 19 + 70
<b>Prozeduren</b>	9 + 6	4 + 2	0 + 14	3 + 10	0 + 11	0 + 11	16 + 54
<b>Mittelwerte pro Prozedur</b>	0,33, 1,53 + 0,33 + 1	0,6, 3,83 + 0,09 + 2,5	1,64 + 0,21 + 0,93	1, 2,31 + 0,08 + 1,08	2,45 + 0,36 + 1,18	2,82 + 0,36 + 1,36	0,48, 2,07 + 0,27 + 1

Die zweitletzte Zeile **Prozeduren** enthält Einträge der Form

e + f

mit der Bedeutung

e Anzahl der neu definierten Prozeduren,

f Anzahl der redefinierten Prozeduren

der Klasse.

Die letzte Zeile **Mittelwerte pro Prozedur** enthält Einträge der Form

[g,] h + i + k

mit der Bedeutung

g mittlere Anzahl der Parameter pro neu definierter Prozedur (bei **Resistor** gehen redefinierte Prozeduren mit ein),

h mittlere Anzahl einfacher Anweisungen pro definierter Prozedur,

c mittlere Anzahl strukturierter Anweisungen pro definierter Prozedur,

b mittlere Anzahl von Zusicherungen pro definierter Prozedur.

### 5.1.1 Anzahl der Prozeduren

Die Klasse **Resistor** erbt von **Comparable** einige Prozeduren, die nicht redefiniert werden. Diese sind in der Statistik nicht berücksichtigt.

Die Konzeptklasse **Resistor** ist die Klasse, die die meisten neuen Prozeduren einführt (9 bzw. 56%) und auch definiert (15 bzw. 65%). Die konkreten Klassen **SingleResistor**, **SerialResistor** und **ParallelResistor** führen dagegen gar keine neuen Prozeduren ein, sondern redefinieren nur welche (11..14). Von den 6 Klassen redefinieren 5 weniger als 61% aller Prozeduren. Das heißt, sie übernehmen mindestens 39% der geerbten Prozeduren unverändert.

- Nur abstrakte Klassen führen neue Prozeduren ein, konkrete Klassen redefinieren sie. Dies entspricht den Rollen, die die Klassen bei der Generalisierung/Spezialisierung spielen.

Die insgesamt 124 Prozeduren aller Klassen (100%) teilen sich so auf: 70 (56%) definiert, 54 (44%) geerbt. Von den 70 definierten Prozeduren sind 16 (13%) neu definiert, 54 redefiniert (44%).

- Im Schnitt muß eine Klasse nur wenig mehr als die Hälfte ihrer Funktionalität selbst definieren, die andere Hälfte erbt sie.
- Im Schnitt muß eine Klasse weniger als ein Viertel der zu definierenden Prozeduren neu definieren, die anderen drei Viertel muß sie nur redefinieren.

Es zeigt sich, wie die Objektorientierung zur Wiederverwenbarkeit beiträgt.

### 5.1.2 Komplexität der Prozeduren

Die mittlere Anzahl der Parameter pro Prozedur liegt unter 1/2. Über die Hälfte der Prozeduren ist parameterlos. Die maximale Parameterzahl 2 tritt nur bei einer Prozedur auf.

- Die Prozeduren haben eine einfache Schnittstelle (Signatur).

Die mittlere Anzahl der Anweisungen pro definierter Prozedur beträgt 3,34. Dies ist der Bruttowert, da Zusicherungen mitgezählt sind. Der Nettowert ohne Zusicherungen ist 2,24, was 69% entspricht. 31% des Aufwands dienen der Spezifikation und ihrer Überprüfung.

- Der Aufwand für Spezifikation und Implementation verhält sich wie 1:2. Dies ist ein akzeptables Verhältnis.

Der Aufwand für die Spezifikation läßt sich freilich reduzieren, indem man Zusicherungen wegläßt. Ziel ist jedoch, eine möglichst vollständige formale überprüfbare Spezifikation zu haben, nicht auf die Spezifikation zu verzichten.

Die 70 definierten Prozeduren enthalten nur 19 strukturierte Anweisungen. Es kommen nur Fallunterscheidungen, keine Schleifen vor. Das Verhältnis von einfachen zu strukturierten Anweisungen ist etwa 8:1. Die Prozedur mit den meisten Anweisungen ist *WriteContent*, sie enthält nur einfache Ausgabeanweisungen.

- Die Prozeduren sind sehr einfach implementiert, im Schnitt mit zwei einfachen Anweisungen.
- Tief strukturierte Algorithmen kommen praktisch nicht vor. Die Funktionalität der Klassen wird stattdessen durch polymorphe Rekursion erzielt.

Die Ergebnisse dieser metrischen Auswertung sind nicht singular, sie stimmen tendenziell mit Ergebnissen aus anderen Projekten überein. Sie zeigen deutlich, daß sich mit der Objektorientierung der Programmierstil wandelt. Was beim strukturierten Programmieren als wesentlich erschien (z.B. schrittweises Verfeinern von Algorithmen), könnte an Bedeutung verlieren. Die Komplexität eines objektorientierten Programms verlagert sich von der Komplexität der Algorithmen hin zur Komplexität der Klassenstrukturen. Wer objektorientiert programmieren will, muß daher nicht nur mit Struktogrammen umgehen können, sondern auch mit Klassendiagrammen.

## 6 Fazit und Ausblick

Wir haben in der ersten Projektphase

- Ideen entwickelt, wie ein exemplarisches Gerüst für Lehrzwecke zu gestalten ist;
- uns für den Themenbereich Modellierung von Zweipolen entschieden;
- wesentliche software-technische Konzepte, Methoden, Verfahren und Entwurfsmuster gewählt;
- mit diesen erste Komponenten des Gerüsts realisiert;
- einige Komponenten ansatzweise in der Lehre erprobt.

Das Projekt hat sich gelohnt. Wir haben dabei unsere Kenntnisse der Objekttechnologie vertieft, uns intensiv mit didaktischen Fragen beschäftigt und neue Erfahrungen gesammelt.

### 6.1 Zu den Mitteln und Methoden

Modul-, Klassen- und Objektdiagramme haben sich als sehr hilfreich für die Analyse und den Entwurf erwiesen. In der objektorientierten Software-Entwicklung sollte man auf Klassendiagramme nicht verzichten, denn sie eignen sich gut, um Entwürfe zu studieren. Die graphischen Notationen von Bild 1 bis Bild 4 sind anschaulich, die zugrundeliegenden Konzepte relativ einfach. Daher sollte man beides im Rahmen der Informatik-Grundlagenausbildung vermitteln.

Die Vertragsmethode halten wir für äußerst nützlich für die Spezifikation, für den Test scheint sie uns unverzichtbar. Die Nützlichkeit von Invarianten hat sich etwa schon beim Testen des Konstantenmoduls `ElectricsPhysConstants` erwiesen: Wir haben Tippfehler und fehlerhafte Formeln sowohl in dem Modul als auch in der benutzten Literatur gefunden. Hier könnte jemand entgegensetzen, daß man diese Fehler auch mit anderen Methoden gefunden hätte. Dagegen könnten wir ein Klassenmodul von der Komplexität von `ElectricsResistorsR` ohne die Vertragsmethode nicht mit vergleichbarem Aufwand erstellen. Ohne exakte formale Spezifikation würde man beim Implementieren orientierungslos hin- und herändern. Ohne Laufzeitunterstützung zum Prüfen, ob die Implementation der Spezifikation entspricht, würde man Fehler leicht übersehen.

Die Vertragsmethode unterstützt systematisches Testen weitaus besser als konventionelle Debugger, mit denen man Haltepunkte setzt und ein Programm schrittweise laufen läßt. Die zur Vertragsmethode passende Methode zur Fehlersuche ist das Rückwärtsrechnen, die Verifikation in umgekehrter Richtung. Nach unserer Erfahrung lernen Studierende relativ gut, diese Methoden kombiniert anzuwenden.

### 6.2 Zu den Ergebnissen

Wann und wie kann man die erstellten Komponenten im Unterricht einsetzen?

Die Module `ElectricsPhysConstants`, `ElectricsTypes`, `ElectricsResistor` und `TestResistor` kann man problemlos im 1. Semester verwenden. Es lassen sich leicht Aufgaben formulieren, zu deren Lösung man diese Module benutzen kann.

Die Klassenmodule `ElectricsResistorsA` und `ElectricsResistorsV` nutzen objektorientierte Grundkonzepte; man kann sie für eine Einführung in die Objektorientierung verwenden. Als Praktikumsaufgabe bietet sich etwa an, ein entsprechendes Modul `ElectricsCapacitorsV` für Schaltungen mit Kondensatoren zu programmieren.

Das Klassenmodul `ElectricsResistorsR` ist von seinen programmiertechnischen Merkmalen am interessantesten: ausgearbeitete Klassenstruktur mit mehreren Konzept- und Schnittstellenklassen, Referenzsemantik, polymorphe dynamische Objektstrukturen mit polymorph-rekursiven Algorithmen. Das Modul hat auch den höchsten Grad von Wiederverwendbarkeit. Weiß man jedoch, wie schwierig die Themen Rekursion und Zeiger für Programmieranfänger sind, dann muß man in Betracht ziehen, daß dieses Modul möglicherweise zu komplex für das 2. Semester ist. Andererseits: Polymorphe Rekursion ist eine der mächtigsten und elegantesten Techniken der Objekttechnologie. Es wäre schade, wenn Studierende sie nicht kennenlernen könnten.

### 6.3 Weitere Entwicklungsschritte

Die nächsten konkret geplanten Entwicklungsschritte sind:

- (1) Führe für Module `Capacitor*` für Kondensatoren, `Inductor*` für Spulen und `Diode*` für Dioden die Entwicklungsschritte 4.2 (4) bis (8) analog zu den Widerstandsmodulen durch. Einige Schritte lassen sich als Praktikumsaufgaben formulieren.
- (2) Führe eine Konzeptklasse `TwoTerminalNetwork` für Zweipole als Abstraktion von Widerständen, Kondensatoren, Spulen und Dioden ein. Gehe von reellen zu komplexen Größen über.
- (3) Definiere eine kontextfreie Sprache, die Reihen- und Parallelschaltungen von Zweipolen beschreiben kann. Entwickle einen Interpreter, der aus einer mit dieser Sprache spezifizierten Schaltung die entsprechende Objektstruktur dynamisch erzeugt. Damit kann man beliebige Zweipolkonfigurationen interaktiv eingeben und bearbeiten.
- (4) Entwickle Komponenten zum Visualisieren der realisierten Zweipolklassen.

### 6.4 Weitere Ziele

Als nächstes Ziel wollen wir die Tauglichkeit des Lehransatzes anhand eines prototypischen Einsatzes der realisierten Komponenten prüfen. Das Konzept ist dann etappenweise ggf. anzupassen und weiterzuentwickeln, das Gerüst ist weiter auszubauen. Nach jeder Etappe sind die erzielten Ergebnisse zu evaluieren, bevor mit der nächsten Etappe begonnen wird.

Nach und nach wollen wir das Gerüst verstärkt in Vorlesungen und Praktika einsetzen und den Lehransatz in das Curriculum integrieren. Weitere Ziele sind die Ausgestaltung kooperativer Übungen mit dem Gerüst und die Optimierung des Lehransatzes und der erarbeiteten Methoden und Materialien, so daß sie auf andere FBe übertragbar sind.

## **A Dokumentation**

### **A.1 Resistors**

## **B Schnittstellen**

Schnittstellendateien erzeugt BlackBox automatisch aus Quellmoduldateien. Es gibt eine normale und eine flache Form von Schnittstelle; bei der flachen Form erscheinen bei Erweiterungsklassen alle Basisversionen ihrer Prozeduren. Wir stellen hier nur je ein Beispiel vor, stellvertretend für alle anderen Module.

### **B.1 ElectricsResistorsV flach**

### **B.2 ElectricsResistorsR**

## **C Quellprogramme**

### **C.1 ElectricsPhysConstants**

### **C.2 ElectricsTypes**

### **C.3 ElectricsResistor**

### **C.4 TestResistor**

### **C.5 ElectricsResistorsA**

### **C.6 TestResistorsA**

### **C.7 ElectricsResistorsV**

### **C.8 TestResistorsV**

### **C.9 ElectricsResistorsR0**

### **C.10 ElectricsResistorsR**

### **C.11 TestResistorsR**

## **D Testprotokolle**

Wir stellen hier nur ein Beispiel vor, stellvertretend für alle anderen Module.

### **D.1 TestResistorsR testet ElectricResistorsR**

## E Literatur

- [1] Fowler, Martin; Scott, Kendall: *UML konzentriert. Die neue Standard-Objektmodellierungssprache anwenden*. Bonn: Addison-Wesley (1998)
- [2] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Bonn: Addison-Wesley (1996)
- [3] Hug, Karlheinz; Ketz, Helmut: *Objektorientierung mit Oberon-2 in der Ingenieur-Grundausbildung*. Informatik-Spektrum, Band 20, Heft 6 (Dezember 1997) S. 350-356
- [4] Hug, Karlheinz; Ketz, Helmut: *Oberon-2 als erste Lehrsprache für Ingenieure*. Positionspapier zur Diskussionsrunde „Auswahl einer geeigneten Programmiersprache“ des Workshop 2 „Objektorientierung in der Ausbildung“ auf der GI-Fachtagung 98 „Informatik und Ausbildung“, Stuttgart, 30.3.1998, 4 Seiten; auf: CD-ROM „GI-Tagung Informatik und Ausbildung ‘98“; auch: <http://www.informatik.uni-stuttgart.de/fakultaet/ausbildung98>
- [5] Hug, Karlheinz; Ketz, Helmut: *Informatik-Grundausbildung für Ingenieure mit Oberon-2*. TEX, Die Zeitschrift der Fachhochschule Reutlingen, Heft 61 (Mai 1998) S. 53-59
- [6] Ketz, Helmut; Hug, Karlheinz: *Informatik-Grundausbildung für Ingenieure - Hochschuldidaktische Betrachtung und Erfahrungsbericht*. In: Claus, Volker (Hg): „Informatik und Ausbildung“, GI-Fachtagung 98, Stuttgart, 30.3.-1.4.1998, Berlin: Springer (1998) S. 52-62; auch: <http://www.informatik.uni-stuttgart.de/fakultaet/ausbildung98>
- [7] Ketz, Helmut; Hug, Karlheinz: *Computer Science as a Core Subject in Engineering Education*. In: Peschges, Klaus-Jürgen (Hg): „Sharing Experience to Increase Internationalization and Globalization in Engineering Education“, International Conference, Fachhochschule Mannheim - University of Applied Sciences, 17.-19.9.1998, Conference Proceedings, S. 429-433
- [8] Ketz, Helmut; Hug, Karlheinz: *Informatik für Ingenieure - Ergebnisse eines zielgerichteten, umfassenden und objektorientierten Ansatzes der Lehre*. In: Klauck, Ulrich (Hg): Katalog zur Veranstaltung „Tag der Lehre“, Fachhochschule Aalen - Hochschule für Technik und Wirtschaft, 27.11.1998, S. 49-50
- [9] Meyer, Bertrand: *Object-oriented Software Construction*. Englewood Cliffs, New Jersey: Prentice Hall (1997) 2. ed.
- [10] Schmid, Hans A.: *Objektorientierte Entwurfsmuster und Frameworks in der Informatik-Ausbildung an der Fachhochschule Konstanz*. Informatik-Spektrum, Band 20, Heft 6 (Dezember 1997) S. 364-371.
- [11] Szyperski, Clemens: *Component Software. Beyond Object-Oriented Programming*. Harlow: Addison-Wesley (1998)

## **F Danksagung**

Mein besonderer Dank gilt Olav Augustin und Michael Jörger für ihre kreative Mitarbeit an dem Projekt. Ohne ihr Engagement wäre diese Arbeit nicht zustande gekommen. Meinem Kollegen Prof. Helmut Ketz danke ich für seine nützlichen Kommentare und Anregungen.

Karlheinz Hug

