

Christian Maurer

Objekt-Ent-  
wick-  
lung  
basierte

**in Lehrprojekten**

v. 7. November 2011

Dr. Christian Maurer  
Keithstr. 16  
10787 Berlin

<http://murus.org/>

## **Abstract Data types**

[ ... ] Being able to use variables of these new,  
user-defined, abstract data types  
in exactly the way as we use variables of built-in data types  
is obviously a good idea.  
Unfortunately, I have never seen a language that achieved this.

## **Object-oriented languages**

A side effect of the application of information hiding  
is the creation of new objects that store data.  
[ ... ] FORTRAN [ ... ] Pascal [ ... ] Simula [ ... ] Smalltalk [ ... ]  
More recent languages have added new types of features  
(known as inheritance) designed to make it possible  
to share representations between objects.  
Often, these features are misused  
and result in a violation of information hiding  
and programs that are hard to change.

The most negative effect of the development of O-O languages  
has been  
to distract programmers from design principles.  
Many seem to believe  
that if they write their program in an O-O language,  
they will write O-O programs.  
Nothing could be further from the truth.

## **Component-Oriented Design**

The old problems and dreams are still with us.  
Only the words are new.

*David L. Parnas*

The Secret History of Information Hiding  
Software Pioneers, Springer 2002

## Vorwort

In diesem Heft wird eine kurze Charakterisierung der Phasen eines *didaktisch reduzierten Programmlebenszyklus* mit dem Schwerpunkt auf einer *an Objekten orientierten Systemarchitektur* gegeben, bei der insbesondere auf die Situation in *Lehrprojekten* eingegangen wird.

Die vorgestellten Grundsätze sind insofern allgemeingültig, als sie weitgehend unabhängig von bestimmten Programmierparadigmen sind. („weitgehend“ nur insofern, als an einigen Stellen das *Zustandskonzept* deutlich durchschimmert, das im deklarativen Paradigma keinen Sinn gibt).

Die Grundsätze, die uns DIJKSTRA, PARNAS, LISKOV, GUTTAG et al. Anfang der siebziger Jahre gelehrt haben, prägten eine Dekade später meinen Informatikunterricht, wurden von mir seit Mitte der achtziger Jahre in der Lehrerfort- und -weiterbildung Informatik an der Freien Universität Berlin unterrichtet und sind seit zwei Jahrzehnten selbstverständlich.

Seit einigen Jahren sind sie nun auch in der Schule die große Mode, wobei allerdings mitunter etwas übertrieben wird (typisch für Modeströmungen, die sich zu *Hypes* entwickeln).

*Christian Maurer*

# Inhaltsverzeichnis

## DER PROGRAMMLEBENSZYKLUS

Vorbemerkung .....	1
Das Phasenmodell .....	1
Typischer Programmlebenszyklus .....	2

## DIE FRÜHEN PHASEN

Systemanalyse .....	4
Anforderungsdefinition .....	6
Ausgewählte ergonomische Fragen .....	8

## SYSTEMARCHITEKTUR

Komponenten einer Systemarchitektur .....	9
Charakterisierung des Komponentenbegriffs .....	10
Objektbasierte Zerlegungen .....	13
Abstrakte Datentypen .....	15
Vorteile einer objektbasierten Systemarchitektur .....	17

## SPEZIFIKATION UND IMPLEMENTIERUNG

Spezifikation .....	20
Implementierung .....	21
Test, Systemintegration, Revision .....	22

## LEHRPROJEKTE

Besonderheiten von Lehrprojekten .....	23
Systemanalyse .....	24
Anforderungsdefinition .....	26
Systemarchitektur, Spezifikation, Implementierung .....	27
Systemintegration, Revision .....	28

## ANHANG

Literatur .....	29
-----------------	----



# DER PROGRAMMLEBENSZYKLUS

## Vorbemerkung

Die Behandlung des Themas *Programmierung im Größeren* dient dem Erwerb grundlegender Techniken der systematischen Entwicklung komplexerer Informatiksysteme.

Zur Einführung werden ausgewählte Teile vorliegender größerer Systeme mit dem Ziel der Sensibilisierung für typische Probleme des Themas vorgestellt. Es schließt sich die Planung der Grundzüge eines neuen Systems oder die Erweiterung eines der untersuchten Systeme an. Die zentrale Leitidee ist, daß sich die Konstruktionen maßgeblich auf die *systematische Entwicklung abstrakter Datentypen* stützen, wie sie einen Kern des Studiums darstellen.

## Das Phasenmodell

Der Kern aller Modelle der Entwicklung eines Informatiksystems umfaßt die Phasen

- *Systemanalyse*,
- *Anforderungsdefinition*,
- *Systemarchitektur* sowie
- *Spezifikation und Implementierung*.

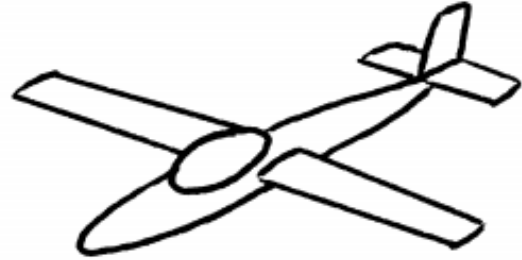
Die Wartungsfreundlichkeit von Systemen wird – selbst bei bescheidenem Umfang – ganz entscheidend durch folgende *grundlegende Prinzipien der Planung, des Entwurfs und der Realisierung* bestimmt:

- die genaue Auseinandersetzung mit dem *sachlichen Hintergrund* der gestellten Aufgabe, ggf. unter Einarbeitung in Aspekte fachfremder Themen,
- die vollständige und widerspruchsfreie Festlegung des *Außenverhaltens* des geplanten Systems,
- eine geeignete *Zerlegung in* weitestgehend unabhängige *Komponenten* und die Beschreibung ihrer wechselseitigen Abhängigkeiten sowie
- die elegante und nachvollziehbare *Beschreibung und Konstruktion der* ermittelten *Komponenten* unter Verwendung einer geeigneten Programmiersprache.

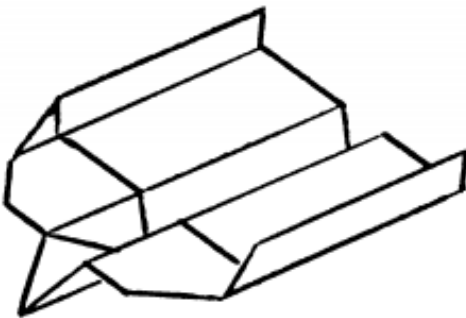
## Typischer Programmlebenszyklus



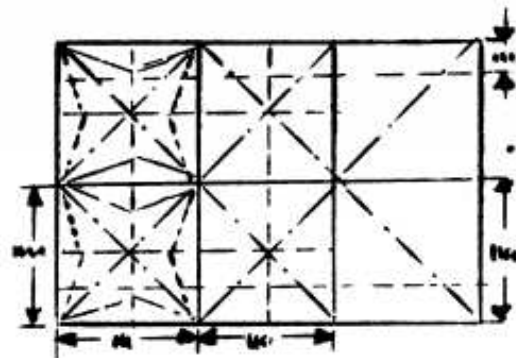
Aufgabe



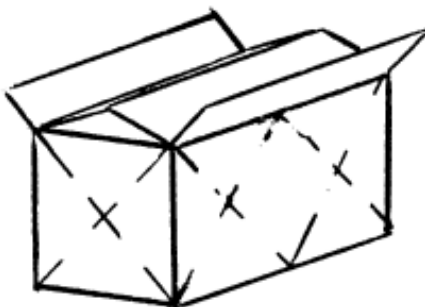
Systemanalyse



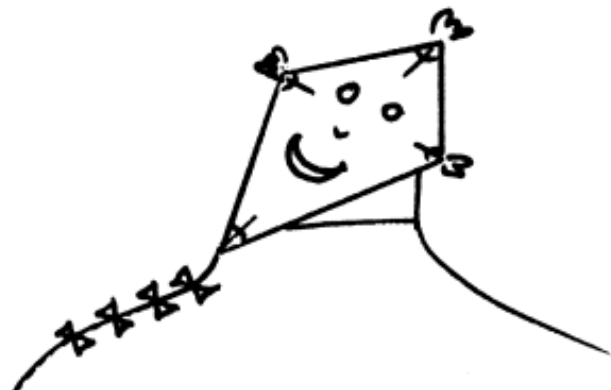
Anforderungsdefinition



Spezifikation



Implementierung



Was *eigentlich* gemeint war ...

Ein Programm macht nicht das, was seine Konstrukteure meinten, sondern genau das, was programmiert ist.

Aus mangelnder Rücksicht auf diese Prinzipien resultieren fehlerträchtige, unbeherrschbare und risikoreiche Systeme, deren

- bestimmungsgemäßes Verhalten,
- Anpaßbarkeit an andere Maschinen, Betriebssysteme, Entwicklungsumgebungen oder Programmiersprachen,
- Entwicklungsfähigkeit und Wartbarkeit bei Änderung oder Fortschreibung der Anforderungen

wegen ihrer inhärenten Instabilität gegen kleine Änderungen grundsätzlich nicht sichergestellt werden kann,

- und deren Teile auch nicht zur Lösung anderer Probleme verwendbar sind.

Im Umkehrschluß sind damit einige Minimalforderungen an die Entwicklung großer Systeme charakterisiert, die in der „Softwarekrise“ vor etwa dreieinhalb Jahrzehnten artikuliert wurden und die dazu geführt haben, daß die Softwaretechnik ein eigenständiges Fachgebiet der Informatik wurde.

Die einzelnen Phasen des Programmlebenszyklus lassen sich nicht beliebig scharf voneinander abgrenzen; in jeder Phase können sich Unvollständigkeiten oder Inkonsistenzen mit den Arbeitsergebnissen vorheriger Phasen herausstellen, die deren Modifikation erzwingen.

Jedes Modell des Programmlebenszyklus geht letztlich von einem verhältnismäßig starren Phasenkonzept aus und wird dem dialektischen Wechselspiel der Phasen untereinander nicht genügend gerecht. Insbesondere ist die Revision eines Systems nicht von der Veränderung oder Erweiterung der Aufgabenstellung zu trennen, die Anlaß zu einer vertieften Systemanalyse gibt; ein erneuter Einstieg in den Programmlebenszyklus mit dem Ziel der Produktion eines im Leistungsumfang erweiterten Systems ist die Folge.

Das genannte Kernmodell ist auch auf die Erstellung von Prototypen anwendbar, wobei ein System unter mehrfachem, sich teilweise überschneidendem Phasendurchlauf systematisch entwickelt wird.

## DIE FRÜHEN PHASEN

### Systemanalyse

Beim Einstieg in ein größeres Programmierprojekt sind gründliche Untersuchungen darüber erforderlich, welche Teile des Bereichs, der automatisiert werden soll, sich mit vertretbarem Aufwand durch einen Rechner erledigen lassen. Dazu gehört neben der Beschaffung der sachlichen Hintergrundinformationen die Analyse der Datenflüsse und der funktionellen oder organisatorischen Abläufe des Systems; ggf. bedeutet das die Einarbeitung in fachfremde Aspekte. Derartige Untersuchungen bilden die notwendigen Voraussetzungen für

- die Festlegung des Leistungsumfanges und
- die Abschätzung des notwendigen Zeitaufwandes

bei der Konstruktion des Informatiksystems. In der industriellen Produktion bilden sie die Grundlage für die Kalkulation der Entwicklungskosten und damit für einen der Faktoren der Preisgestaltung.

Daneben steht der Dialog zwischen den Systemanalytikern und den Auftraggebern (oder späteren Benutzern) über Ausformungen von Details des geplanten Systems; eventuell die Erstellung von Prototypen zur Klärung von Fragen zur Gestaltung der Benutzeroberfläche oder zum Systemverhalten. Daß diese Gespräche gelegentlich mit Verständigungsproblemen behaftet sind, ergibt sich aus den unterschiedlichen Sichtweisen von Systemanalytikern und Benutzern, die sich häufig als Laien gegenüberstehen: Softwaretechniker müssen sich oft erst in die Materie einarbeiten und Fachwissen erwerben, Benutzer haben meist recht unscharfe Vorstellungen über die Möglichkeiten und Grenzen des Rechnereinsatzes.

Die Abstimmung mit den Wünschen der Benutzer bedingt zwar zunächst einen gewissen Mehraufwand bei der Entwicklung; die Folge ist aber eine höhere Akzeptanz des automatisierten Systems durch bessere Einbettung in die vorhandenen Strukturen. Auch bei Entwicklungen für den Massenmarkt entscheidet die Rückkopplung mit den Vorstellungen der Benutzer über den wirtschaftlichen Erfolg.

Im Zuge der vertieften Beschäftigung mit den Sachfragen müssen auch die Rückwirkungen des Rechnereinsatzes einbezogen werden: sie verändern durch die Umstellung auf automatische Datenverarbeitung

eventuell die Struktur des betrachteten Systems, wodurch sich eine Modifikation der Aufgabenstellung herauskristallisieren kann.

Auch die Einsicht, daß sich nicht *jeder* Aspekt eines Systems zur automatischen Bearbeitung eignet, wächst erst mit fortschreitender Analyse: die Realisierung mancher interessanten Idee erweist sich bei näherem Hinsehen als im Rahmen der vorgesehenen Kosten-Nutzen-Relation zu aufwendig oder gegen Gewohnheiten der Benutzer nicht durchsetzbar; mitunter sogar als prinzipiell fragwürdig, weil manche Tätigkeiten von menschlichen Gehirnen, Händen und Stimmen sehr viel besser als von Elektronengehirnen, mechanischen Greifarmen und rechnergestützten Animationen erledigt werden, oder weil es nicht zu verantworten ist, sie an Automaten zu delegieren.

Im Zusammenhang mit solchen Überlegungen wächst auch die Sensibilität für die Risiken blinden Vertrauens in komplexe technische Systeme, die *in rasant wachsendem Maße auf von Menschen (sic!) geschriebenen Texten* (den Quelltexten von Programmen) beruhen.

## Anforderungsdefinition

Nachdem in der Systemanalyse die Aufgabenstellung detailliert festgelegt ist, beginnt die Programmierung im engeren Sinn mit einer exakten Beschreibung des (Außen-)Verhaltens des Systems. Das Leitmotiv in dieser Phase ist die Frage:

*WAS soll das System im einzelnen tun?*

Dabei ist es notwendig, mögliche Sichtweisen eines Entwicklers zunächst zu vernachlässigen und der Benutzersicht unterzuordnen, um nicht zu früh Entscheidungen zu treffen, die die Implementierung später einschränken oder sogar behindern oder im schlimmsten Fall zu schweren Entwurfsfehlern führen, deren Behebung sehr unangenehm, weil extrem zeitraubend werden kann.

Nur auf der Basis solider Kosten-Nutzen-Analysen, Erfahrung in der Entwicklung oder beim Einsatz von Werkzeugen zur Programmkonstruktion und genauer Untersuchungen des Benutzungsprofils sind schon zu diesem Zeitpunkt einschränkende Annahmen zulässig.

Zentrale Aufgabe der Anforderungsdefinition ist die sorgfältige Gestaltung der *Benutzeroberfläche*. Sie ist für alle weiteren Phasen unumgänglich: nur Genauigkeit und Vollständigkeit der Arbeit daran kann sicherstellen, daß unmißverständliche und solide Grundlagen für den Entwurf und die Realisierungsentscheidungen gelegt werden.

Die Benutzeroberfläche ist durch die Gesamtheit der Interaktionen zwischen Benutzern (ggf. auch eingesetzten peripheren Geräten) und dem automatisierten System, das auf einem Rechner läuft, gegeben. Zur Gestaltung der Oberfläche gehört die Beschreibung der *Eingaben* in den Rechner (sowohl durch die Benutzer als auch ggf. die zur Erfassung von Daten eingesetzten Geräte) und der *Ausgaben* des Rechners (auf die für die Datenausgabe vorgesehenen Geräte).

Die Überlegungen dazu lassen sich in zwei Kategorien einteilen:

- die *Darstellungsform der Daten* auf den Ausgabemedien (z. B. dem Bildschirm oder dem – von Druckern oder Plottern bedruckten – Papier) und
- die *Eingaben* von Daten und die *Kommandos zur Bedienung des Systems* (z. B. mittels Tastatur, Maus oder Graphiktablett) durch die Benutzer.

Im Falle der Nutzung oder Ansteuerung weiterer peripherer Geräte (z. B. zur Erfassung oder Aufzeichnung von Audio- oder Videodaten, von Strichcodes oder von Meßwerten unterschiedlichster Art) gehört die Beschreibung ihrer Schnittstellen mit zur Anforderungsdefinition; bei Daten, die auch von anderen Programmen verarbeitet werden sollen, auch die Beschreibung ihrer Datensatzformate.

Die genannten Punkte sind weitgehend unabhängig voneinander zu klären: beispielsweise hängt die Repräsentation der Daten auf dem Bildschirm nicht davon ab, ob das System lediglich durch die Tastatur bedient oder ob auch eine Maus benutzt wird; die Bedienung des Programms hat nichts mit der Darstellung der Daten auf dem Bildschirm, auf dem Papier oder auf peripheren Speichermedien zu tun.

Bei der Gestaltung der Interaktion zwischen Mensch und Maschine müssen ergonomische Anforderungen sorgfältig durchdacht werden, weil sonst die Akzeptanz des Systems nicht gewährleistet ist oder sogar die Gefahr besteht, ein unbrauchbares System zu konstruieren.

Die Arbeitsergebnisse dieser Phase stellen die Grundlage für die Erstellung des Benutzerhandbuchs des Systems dar (bei geschickter Formulierung *sind* sie das Benutzerhandbuch!).

Die Widerspruchsfreiheit und Vollständigkeit der Anforderungsdefinition kann letztlich nur durch das Verhalten einer ersten Version des fertigen Systems entschieden werden. Manche Mängel zeigen sich erst bei dessen Gebrauch, also nach vollständigem Durchlaufen des Programmlebenszyklus. Die Fehler, die dann erst festgestellt werden, lassen sich nur unter Rückgriff auf frühere Phasen beseitigen; im schlimmsten Fall (der katastrophalste Fall – Fehler der Systemanalyse – sei außer Acht gelassen) erweist sich, daß die Anforderungsdefinition nicht in allen Punkten problemadäquat war:

- ungenügend klare oder zu wenig detaillierte Vorstellungen von der Funktionsweise des geplanten Systems,
- mangelnde Genauigkeit bei deren Beschreibung oder
- – häufig noch desaströser – nicht dokumentiertes implizites Wissen in einzelnen Köpfen

führen *nahezu zwangsläufig* zu Inkonsistenzen in der weiteren Arbeit, somit zu fehlerhaften Konstruktionen in den folgenden Phasen; mit der Folge, daß deren Funktionalität nicht den Intentionen entspricht.

Die detaillierten Überlegungen dazu, wie sich das geplante System den Benutzern darstellen soll, liefern einen natürlichen Einstieg in die Entwurfsarbeit, weil sich aus der Sachanalyse in Verbindung mit der Konstruktion der Benutzeroberfläche eine klare Gliederung der im System betrachteten Objekte ableiten läßt.

Was im folgenden Kapitel gezeigt wird, sei hier schon zugesichert:

*Längs dieser Objekte  
ergibt sich die Antwort  
auf die Frage  
nach der Zerlegung des Systems in beherrschbare Teile  
völlig von selbst.*

### **Ausgewählte ergonomische Fragen**

Als Orientierungshilfe zur Gestaltung von Benutzeroberflächen seien einige Fragen von J. Nievergelt und A. Ventura zitiert, die „die meisten Schwierigkeiten der Benutzer interaktiver Programme gut kennzeichnen“:

- Wo bin ich?
- Was kann ich hier tun?
- Wie kam ich hierhin?
- Wo kann ich hin und wie komme ich dorthin?

Dieser Fragenkatalog muß erweitert werden, z. B. durch:

- Wie komme ich hier wieder heraus?
- Wie erfahre ich nach einer Unterbrechung der Arbeit, wo ich bin?
- Was *soll* ich hier tun?
- Mit welchen Tasten, Mausklicks, Befehlen o. ä. erreiche ich, was ich will?
- Kann ich etwas ungeschehen machen? Wenn ja, wie?
- *Welchen* Fehler habe ich gerade gemacht?

## SYSTEMARCHITEKTUR

### Komponenten einer Systemarchitektur

Zwischen der Anforderungsdefinition und der Programmierung im engeren Sinn, d. h. der Spezifikation und Implementierung, liegt die Phase des *globalen Entwurfs* mit dem Ziel einer geeigneten Zerlegung des Gesamtsystems in Komponenten. Das Leitmotiv ist die Frage:

*WAS sind die einzelnen Teile des Systems?*

Das Hauptanliegen in dieser Phase ist die Reduktion der Komplexität des Systems auf ein beherrschbares Maß, die sich möglichst stringent aus den Ergebnissen der Anforderungsdefinition ergibt.

Der Gewinnung sinnvoller Kriterien für eine Zerlegung dienen die folgenden Postulate an die Komponenten:

- ein starker innerer Zusammenhang jeder einzelnen Komponente und
- eine von den anderen Komponenten weitgehend unabhängige
  - *Verständlichkeit*,
  - *Planbarkeit*,
  - *Konstruierbarkeit*,
  - *Prüfbarkeit* und
  - *Wartbarkeit*.

Zur Erfüllung dieser Forderungen ist es sinnvoll, jede Komponente mit zwei Sichtweisen auszustatten:

- der *Spezifikation*, der Aufzählung aller ihrer Leistungen und der exakten – im Idealfall mathematisch formulierten – Beschreibung der *Voraussetzungen* und *Effekte* für jede einzelne Leistung und
- der *Implementierung* dieser Leistungen gemäß der Spezifikation, in der die Entwurfsentscheidung unter Einbeziehung der vorhandenen Ressourcen und Beachtung des Anforderungsprofils an das Systemverhalten (wie z. B. die Optimierung des Laufzeitverhaltens oder der Nutzung des Speichers oder Aspekte der Datensicherheit) getroffen wird.

Als *Klient* einer Komponente K wird jede *andere* Komponente bezeichnet, die Leistungen von K benutzt.

Die Spezifikation einer Komponente kann dann als „Vertrag“ zwischen ihrer Implementierung und ihren Klienten aufgefaßt werden, die die gegenseitigen Rechte und Pflichten regelt:

- für die Implementierung
  - das Recht, davon auszugehen, daß die Voraussetzungen erfüllt sind, und
  - die Pflicht zur Konstruktion entsprechend den Beschreibungen ihrer Effekte,
- für den Klienten
  - die Pflicht, vor dem Aufruf einer Leistung dafür zu sorgen, daß die angegebenen Voraussetzungen erfüllt sind, und
  - das Recht, sich darauf verlassen zu können, daß nach dem Aufruf einer Leistung die angegebenen Effekte bewirkt sind.

Eine strikte Trennung der beiden Teile bietet die Gewähr dafür, daß der Klient einer Komponente nicht implizite Annahmen über ihr Verhalten macht, die er aus der Kenntnis von Implementierungsdetails hat. Nur so ist die Komponente vor unkontrollierten Zugriffen von außen sicher, die ihr Verhalten entgegen der Spezifikation verändern und Nebeneffekte erzeugen können, die sich unvorhersagbar auf das Systemverhalten auswirken.

## Charakterisierung des Komponentenbegriffs

Im folgenden werden die allgemeinen Forderungen an die Komponenten einer Zerlegung des vorigen Abschnittes präzisiert.

*Notwendige* Bedingungen an einen sauberen Komponentenbegriff sind

- die *Einfachheit der Spezifikation* der Komponenten und
- die *Kontextunabhängigkeit* ihrer Implementierungen.

Zur *Einfachheit der Spezifikation* einer Komponente gehören

- eine geeignete Sprachebene zu deren Beschreibung: mindestens eine genaue umgangssprachliche Ausdrucksweise, sofern nicht eine halbformale oder formale Spezifikationssprache,
- Verständlichkeit und Überschaubarkeit, d. h.
  - Minimalität ihres Leistungsangebotes durch die Bereitstellung eines zusammenhängenden, nicht in Teilprobleme zerlegbaren Problemkreises,

- weitestgehende (im Idealfall vollständige) Unabhängigkeit von den Spezifikationen anderer Komponenten,
- die Reduktion von Datentransporten auf ein möglichst geringes Maß – sowohl innerhalb einer Komponente als auch zwischen verschiedenen Komponenten,
- die Wahrung des Geheimnisprinzips „*information hiding*“, d. h.
  - die Beachtung des Grundsatzes, daß die Spezifikation genau die Menge aller Annahmen repräsentiert, die andere Teile des Systems von der betreffenden Komponente machen,
  - folglich die rigide Vermeidung der Offenlegung irgendwelcher Implementierungsdetails,
- ein hoher Allgemeinheitsgrad, d. h.
  - vollständige Abstraktion von den konkreten Gegebenheiten vorgesehener Verwendungszwecke und von implizitem Wissen über die Benutzung durch irgendwelche Klienten,
  - Maximalität des Leistungsangebotes innerhalb des umschriebenen Problemkreises mit dem Ziel der Verwendbarkeit auch für andere Zwecke als dem ursprünglich geplanten,
  - Geschlossenheit im Sinne eines gewissen Reifegrades ihres Entwicklungsstadiums,
  - aber trotzdem Offenheit für Anpassungen oder Erweiterungen ihres Leistungsumfanges.

Der *Kontextunabhängigkeit der Implementierung* einer Komponente lassen sich folgende Punkte zuordnen:

- Beherrschbarkeit der Komplexität, d. h.
  - Beschränkung auf die Erledigung der durch die Spezifikation gegebenen Aufgabe, die durch eine starke innere (logische wie sachliche) Bindung gekennzeichnet ist,
  - Verzicht auf die Konstruktion von Systemteilen, die nicht unmittelbar aus der gegebenen Spezifikation erwachsen,
  - einerseits Begrenzung der Anzahl der benutzten Komponenten auf das Minimum dessen, was für die Erledigung der Aufgabe notwendig ist,
  - andererseits auch ggf. Erhöhung der Übersichtlichkeit durch Auslagerung von Aufgabenteilen in separate Komponenten, die dann ihrerseits all diesen Bedingungen genügen müssen,

- konsequente Ausnutzung des Geheimnisprinzips, d. h.
  - Festlegung auf Datenstrukturen oder Algorithmen erst in der Implementierung, dadurch die Offenhaltung von alternativen Implementierungen, z. B. aufgrund wechselnder Anforderungen an die verfügbaren Ressourcen,
  - Einbeziehung der Möglichkeit des Einsatzes anderer Programmiersprachen oder von Werkzeugen,
  - Isolierung derjenigen Systemteile, die von der zugrundeliegenden Hardware, der Einbettung in ein Betriebssystem, der verwendeten Entwicklungsumgebung oder von benutzten Werkzeugen, die nicht allgemein verfügbar sind, abhängen,
  - als Folge des letzten Punktes die Erleichterung der Portierbarkeit, d. h. der Implementierbarkeit zur Ausführung auf anderen Maschinen, unter anderen Betriebssystemen oder unter Einsatz anderer Werkzeuge oder Entwicklungsumgebungen,
- und damit Interferenzfreiheit, d. h. von der Implementierung von anderen Komponenten unabhängige
  - Realisierbarkeit durch die Auswahl von Datenstrukturen und Algorithmen, die dem Anforderungsprofil angepaßt sind,
  - Entwickelbarkeit durch eine möglichst lose Kopplung an die benutzten Komponenten, also nur über deren Spezifikation, ohne jede Voraussetzung der Kenntnis ihrer internen Daten oder Abläufe,
  - Testbarkeit, d. h. Überprüfbarkeit ihrer einwandfreien Funktion gemäß der Spezifikation – sowohl ohne Einbindung der Implementierung der benutzten Komponenten, als auch ohne Berücksichtigung des Kontextes, in dem sie benutzt wird,
  - Wartbarkeit, d. h. die Lokalisierbarkeit und Behebbarkeit von – auch spät entdeckten – Fehlern, Anpaßbarkeit an andere Bedingungen der Benutzung, insbesondere
  - Weiterentwickelbarkeit, d. h. Erweiterbarkeit ihres Leistungsumfangs.

Konsequenz aus der Unterscheidung dieser beiden Blickwinkel ist die Forderung nach *getrennter Übersetzbarkeit von Spezifikation und Implementierung* der Komponenten.

Daraus ergibt sich eine ganze Reihe von Vorteilen:

- Ein System kann nach der Spezifikation seiner Komponenten vom Übersetzer bereits zumindest auf seine syntaktische Richtigkeit geprüft werden, was einen Ansatz in Richtung auf eine Prüfung auf Widerspruchsfreiheit des geplanten Systems darstellt und Probleme bei der Integration der Komponenten zum ganzen System vermeiden hilft;
- die Spezifikationen können gegen die nachträgliche Veränderung von Implementoren geschützt werden (eine Maßnahme, die einen Schutzmechanismus gegen typische Schwierigkeiten bei der Konstruktion größerer Systeme darstellt);
- bei einer alternativen Implementierung einer Komponente (ohne Veränderung ihrer Spezifikation) muß nicht das ganze System, sondern nur diese Implementierung neu übersetzt werden;
- somit wird die gleichzeitige Konstruierbarkeit verschiedener Komponenten durch verschiedene Personen von Grund auf unterstützt.

Als Folgerung aus diesen Überlegungen ergibt sich, daß in ein größeres Programmsystem in einer Sprache geschrieben werden sollte, in der dieses Konzept als fester Sprachbestandteil enthalten ist.

Auch der Zugriff auf fremdsprachliche Bibliotheken ist mit diesem Ansatz verträglich:

Die Spezifikation wird in der eingesetzten Programmiersprache formuliert, die Implementierung durch die Einbindung der Bibliotheken erledigt.

## Objektbasierte Zerlegungen

Ein – zu seiner Zeit noch unkonventionelles – Zerlegungskriterium hat DAVID L. PARNAS bereits 1972 formuliert: anstatt ein System nach seinen Ablaufschritten zu zerlegen, soll jede Komponente eine *Entwurfsentscheidung* realisieren (zu der es in der Regel Alternativen gibt).

Dieser Anspruch wird durch eine *Systemarchitektur* eingelöst, die sich an den *Objekten* des betrachteten Systems orientiert:

*Die Komponenten eines Systems sind durch seine Objekte und die sie charakterisierenden Eigenschaften und bearbeitenden Operationen definiert.*

Eine derartige *objektbasierte Zerlegung* läßt sich nicht nur auf kanonische Weise aus der Anforderungsdefinition ableiten, sondern liefert, wie im folgenden gezeigt wird, einen stringenten Ansatz für die Architektur eines Systems:

Aus ihr ergeben sich *hinreichende* Bedingungen für einen abgrenzbaren Komponentenbegriff im Sinne des vorigen Abschnitts, d. h. sie erfüllt vollständig alle genannten Forderungen.

Eine *ablauforientierte Zerlegung* entspricht dagegen *in überhaupt keiner Weise* dem von PARNAS vorgezeichneten Weg durch den Programmlebenszyklus.

Ansatz für eine objektbasierte Entwurfsmethodik ist die Aufgabe, aus der Anforderungsdefinition die realen Objekte, die im System manipuliert werden, in einem problemangemessenen Feinheitsgrad zu modellieren und aus ihrem Leistungsspektrum die Zugriffe auf ihre Modelle herauszuarbeiten. Eine derartige Strukturanalyse führt rückwirkend zu einem tieferen Verständnis für das zugrundeliegende reale System und daher auch für die Forderung nach Genauigkeit der Formulierungen in der Anforderungsdefinition. Das führt zur Festlegung

- sowohl der Spezifikationen einzelner Komponenten
- als auch der Wechselbeziehungen verschiedener Komponenten in der Systemarchitektur.

Die Hierarchie der so gefundenen – vorerst ungeordneten – Menge der Komponenten des Systems ergibt sich aus der Aufdeckung der Abhängigkeiten zwischen den betrachteten Objekten:

Eine Komponente, die neue Objekte durch die Zusammenfassung gegebener Objekte auf strukturierte Weise definiert, benutzt genau *diejenigen* Komponenten, die ihrerseits die zusammenzufassenden Objekte definieren.

## Abstrakte Datentypen

Die Forderungen des vorigen Abschnitts nach starkem innerem Zusammenhang und einer von anderen Komponenten weitgehenden Unabhängigkeit der einzelnen Komponenten sind unmittelbar erfüllt, wenn eine Komponente

- entweder eine *Klasse von Objekten des gleichen Typs* oder
- – in Ausnahmefällen – Zugriff auf ein *einzelnes Objekt* behandelt.

Bei Programmiersprachen, die eine Trennung von Spezifikation und Implementierung erlauben, definiert die zugehörige Spezifikation

- einen *Datentyp*, d. h. eine Klasse von Objekten mit ihren Zugriffsoperationen, die in *dem* Sinne *abstrakt* ist, als seine Repräsentation in der Spezifikation zwar kommentarhaft beschrieben sein mag, jedoch syntaktisch nicht sichtbar ist,
- die Zugriffsoperationen auf ein *abstraktes Datenobjekt* (das nur in der Implementierung verwaltet, also nicht explizit zur Verfügung gestellt wird), oder
- eine Menge von *Funktionen* zwischen verschiedenen Datentypen oder -objekten.

Der zweite Fall ist weitgehend obsolet, weil er ein Spezialfall des ersten ist. Eine Ausnahme stellen diejenigen Komponenten der untersten Schicht dar, die das System an die Dienste des Betriebssystems anbinden; einzelne periphere Komponenten des Rechners (wie z. B. Bildschirm, Tastatur, Maus oder Drucker) werden in der Regel nur in *einer* Ausprägung benötigt und deshalb meistens als einzelne Objekte modelliert.

Der letzte Fall ist deswegen weniger bedeutend, weil es ratsam ist, derartige Funktionen in die beteiligten Objektklassen zu integrieren. Auch algorithmisch betonte Systemteile gewinnen strukturell deutlich an Klarheit und lassen sich sauber in die Hierarchie der im System verwendeten Objekte einfügen, wenn sie als Zugriffsoperationen auf gewisse herauszuarbeitende Datenstrukturen erkannt werden.

Damit kann sich eine Zerlegung fast durchgehend auf den erstgenannten Fall stützen.

Die Implementierung eines Datentyps und der Zugriffe darauf setzt sich wiederum aus Datentypen und der Zugriffe auf sie zusammen, die in anderen derartigen Komponenten spezifiziert sind.

Die Zerlegung eines Systems in Komponenten hat nach diesem Prinzip genau *dann* ein Ende erreicht, wenn bei der Implementierung nur noch elementare Datentypen, d. h. Bestandteile der eingesetzten Programmiersprache, verwendet werden, aus denen sich letztlich alle Datentypen zusammensetzen.

Unter der (offensichtlich sinnvollen) Annahme, daß kein Objekt – auch nicht über mehrere Schichten – ein Objekt seines eigenen Typs als Teil enthalten kann, ist diese rekursive Definition fundiert, d. h. sie terminiert.

Somit ergibt sich eine stringente Systemarchitektur in Form einer nach zunehmender Komplexität der Objekte geordneten hierarchisch geschichteten Struktur aus abstrakten Datentypen.

Deren Komponenten können als Schablonen für zur Programmaufzeit gemäß der Implementierung erzeugbaren konkreten Datenobjekten aufgefaßt werden, die durch die spezifizierten Operationen z. B. kopiert, verglichen, mit bestimmten Eigenschaften ausgestattet oder auf sie hin untersucht, verändert, ausgegeben, umgewandelt und wieder vernichtet werden können, oder die – falls sie Behälter für Objekte darstellen, die in anderen Komponenten realisiert werden – daraufhin überprüft werden können, ob sie gewisse konkrete Datenobjekte enthalten, und in die solche Objekte eingefügt oder aus ihnen entfernt, und die gezählt, umgeordnet, zerlegt, zusammengesetzt oder durchlaufen werden können.

Ein derart konstruiertes idealtypisches Programmsystem ist nichts weiter als die *Zusammenbindung wiederverwendbarer Komponenten*, d. h. von „lokalen Systemen“, die – unabhängig vom Kontext, in dem sie verwendet werden – von Interesse sind.

Die Benennung der Komponenten ist abhängig vom verwendeten Programmierparadigma; in Haskell, ML oder Modula-2 heißen sie z. B. „*Moduln*“, in C#, D und Java „*Klassen*“ und in Go „*Pakete*“.

## Vorteile einer objektbasierten Systemarchitektur

Die Behauptung des vorigen Abschnitts, daß bei einer Zerlegung eines Systems nach seinen Objekten sämtliche Postulate an einen sachgerechten Komponentenbegriff auf natürliche Weise erfüllt sind, ist leicht einzusehen.

Eine geeignete Sprachebene für die Spezifikation ist – über die formale Angabe des Datentyps (d. h. der Nennung seines Namens) und der Zugriffsoperationen einschließlich ihrer Parameter hinaus – die Kommentierung des definierten Typs und der Operationen durch Angabe ihrer *Nutzungsvoraussetzungen* und *Effektbeschreibungen*.

Hier tut sich eine entscheidende Schwäche vieler gängiger Programmiersprachen auf: Mangelnde syntaktische Unterstützung für die Absicherungen gegen Nichtbeachtung von Voraussetzungen oder für die Zusicherungen von Effekten.

Ein optimale Sprachebene ist etwa eine algebraische Spezifikation eines jeden exportierten Datentyps durch Operationen (Prozeduren, Funktionen, Methoden) und Gleichungen (Beziehungen zwischen den Operationen).

Die Verständlichkeit und Überschaubarkeit der Spezifikation und die Minimalität ihres Leistungsangebotes sind durch die Behandlung genau eines Datentyps (ggf. Datenobjekts) a priori gewährleistet.

Bei der Verwendung abstrakter Datentypen sichert die Unsichtbarkeit des Aufbaus eines Objekts aus Bestandteilen in der Spezifikation von selbst die Wahrung des Geheimnisprinzips und eine weitestgehende Unabhängigkeit von anderen Komponenten.

Die Reduktion von Datentransporten ist eine einfache Konsequenz aus dem Ansatz, im Idealfall (abgesehen von atomaren Datentypen) als Parameter in Operationen nur den spezifizierten Datentyp zu verwenden.

Allgemeinheitsgrad und Geschlossenheit ergeben sich in Verbindung mit einer anzustrebenden Maximalität des Leistungsangebotes im Rahmen der gesteckten Grenzen: auf jeden Fall soll eine genügend große Vielfalt von Zugriffen auf die Objekte des betrachteten Datentyps vorgesehen werden, um die Komponente möglichst universell verwendbar zu machen.

Das widerspricht keineswegs dem Prinzip der Offenheit:

Eine Komponente läßt sich jederzeit durch die Spezifikation und Implementierung vorerst nicht bedachter, aber später als notwendig erkannter Zugriffe erweitern (was natürlich die Neuübersetzung ihrer Klienten erfordert).

Auch die für die *Implementierung* als notwendig erachteten Postulate werden gewissermaßen von selbst eingelöst:

Die Beherrschbarkeit der auftretenden Komplexität ist durch die Konstruktion von Datentypen aus Bestandteilen gesichert, die vorher definiert sind. Da deren Konstruktionsdetails wiederum in anderen Implementierungen verborgen sind, muß sich die Implementierung der zusammengesetzten Datentypen nicht mehr um Einzelheiten scheren, sondern kann ihre Existenz und die Zugriffe auf sie nur auf der Basis der Kenntnis ihrer Definition, also auf einer recht abstrakten Ebene, voraussetzen.

Wenn sich bei der Implementierung herausstellt, daß weitere, zunächst nicht vorgesehene Teile notwendig sind, gibt das Anlaß zur Konstruktion separater Komponenten, die dann – wiederum nur unter Rückgriff auf ihre abstrakte Beschreibung, d. h. ihre Spezifikation – benutzt werden.

Das Geheimnisprinzip läßt sich hervorragend ausnutzen:

Die Ersetzung von Implementierungen durch Alternativen wird durch das geschilderte Prinzip optimal unterstützt. Typische Beispiele – wohlgemerkt bei gleichen Spezifikationen – gibt es etwa in folgenden Szenarios:

Es können unterschiedliche Implementierungen in Abhängigkeit davon erforderlich sein, ob Datenbestände nur einmal erfaßt und dann vorzugsweise durchsucht, oder ob sie laufend aktualisiert werden und Recherchen vergleichsweise selten sind; die Implementierung des Zugriffs auf Daten in *einem* Rechner unterscheidet sich grundsätzlich vom Zugriff auf verteilte Daten, die auf verschiedenen Rechnern liegen. Häufig gilt es, Alternativen zu prüfen, die zwischen widersprüchlichen Anforderungen an günstiges Laufzeitverhalten einer Komponente und der Forderung nach minimalem Speicherbedarf abwägen. Durchgriffe auf die Basismaschine werden in geeigneten Komponenten isoliert, deren Implementierungen sich für verschiedene Zielsysteme wesentlich voneinander unterscheiden können.

Die Interferenzfreiheit ist durch die Unabhängigkeit von getrennt entwickel- und übersetzbaren Spezifikations- und Implementierungsteilen der Komponenten gesichert.

Zur Vermeidung der Verwendung systemübergreifender Zustandsinformationen, die im ganzen System sichtbar, folglich auch manipulierbar – und daher fast zwangsläufig Quelle höchst fataler, aber schwer auffindbarer Fehler in größeren Systemen – sind, können *globale Variable* – die aus eben diesem Grunde nur als *Teufelszeug* bezeichnet werden können – *derart* in Komponenten (ggf. in lokalen Unterkomponenten) eingekapselt werden, daß ihr Inhalt – im Gegensatz zu lokalen Variablen in Operationen – zwar über die ganze Programmlaufzeit erhalten bleibt, sie jedoch vor unkontrollierbaren Zugriffen von außen sicher sind (was natürlich voraussetzt, daß die verwendete Programmiersprache ein solches Konzept unterstützt).

Andere nützliche Aspekte der geschilderten Methode sind, daß sie

- eine stringente Weiterentwicklung eines Prototyps durch Verfeinerung der bislang auftretenden Strukturen oder durch Zusammenfassung mit anderen Strukturen zu größeren Einheiten ermöglicht,
- eine recht große Gewähr dafür bietet, daß wiederverwendbare Teile von Programmsystemen konstruiert werden (was angesichts begrenzter zeitlicher Möglichkeiten, allgemein der kostenträchtigen Entwicklungsarbeit im Softwarebereich sehr wertvoll ist)
- und sich einer weitgehend dezentralen Programmentwicklung nicht in den Weg stellt.

## SPEZIFIKATION UND IMPLEMENTIERUNG

### Spezifikation

Zu diesem Thema ist im Kapitel zur Systemarchitektur eigentlich alles Wesentliche gesagt, deshalb reichen hier ein paar ergänzende Bemerkungen.

Wenn die Komponente einen *Datentyp* zur Verfügung stellt, wird zuerst dessen Name angegeben (was bei abstrakten *Datenobjekten* natürlich ersatzlos entfällt). Der Name des Datentyps *ist* entweder der Name der Komponente, wenn das die Syntax der verwendeten Programmiersprache vorsieht, mindestens kann er jedoch als *Synonym dafür* aufgefaßt werden.

Darüberhinaus muß zur Vermeidung von Mißverständnissen seine Semantik in einem kurzen Kommentar beschrieben werden, weil selbst bei auf ersten Blick klar erscheinenden Namensgebungen Zweifel entstehen können, z. B. *reelleZahlen* (Genauigkeit?) oder *Kalenderdaten* (Gültigkeitsbereich, Feiertage?).

Falls es erforderlich ist, werden Konstante angegeben, z. B. wenn es darum geht, bestimmte Bereiche zu beschränken.

Variable sollten dagegen – wegen der Gefahren, die von unkontrollierbaren Veränderungen von außen damit verbunden sind – nur in begründbaren Ausnahmefällen Eingang in die Spezifikation finden; wenn das doch einmal notwendig ist, empfiehlt sich die Bereitstellung von Operationen, die die Werte von *Komponentenvariablen*, d. h. von globalen Variablen in der Implementierung der Komponente, die von außen nicht sichtbar sind, liefern bzw. verändern.

Wenn das die benutzte Sprache hergibt, sind unter Umständen zur Parametrisierung von Operationen zusätzliche Aufzähltypen sinnvoll, wie z. B. *Wochentage* (Montag, Dienstag, . . . ) und *Perioden* (täglich, wöchentlich, monatlich, . . . ) in einem Datentyp *Kalenderdaten*.

Der Rest der Spezifikation besteht aus der Auflistung der Zugriffsoperationen auf die Objekte (die „Variablen“ des Datentyps) unter Angabe ihrer *Syntax* und ihrer *Semantik*, wobei die Semantik durch ihre Voraussetzungen und Effekte gegeben ist, die für jede Operation aufgeführt werden müssen.

Dabei darf nicht vergessen werden, Operationen vorzusehen, die es Klienten erlauben, die Voraussetzungen auch zu überprüfen.

## Implementierung

Bei der Implementierung einer Komponente erfolgt zunächst die Festlegung des konkreten Datentyps, der den spezifizierten Datentyp modelliert, oder die konkrete Repräsentation des Datenobjekts, das Träger für die Operationen auf dem abstrakten Datenobjekt ist.

In der Regel handelt es sich dabei um einen der folgenden Fälle:

Auf der untersten Ebene

- durch die Wahl eines elementaren Datentyps (Zeichen, Zeichenketten, Wahrheitswerte, ganze Zahlen, Gleitkommazahlen, selbstdefinierte Aufzähltypen)

auf einer höheren Ebene

- die *Konstruktion neuer Objekte*, deren Attribute gegebene Objekte verschiedenen Typs sind, auffaßbar als „Zusammenbinden“ vermöge eines Typkonstruktors „Tupel“, oder
- das Zusammenfassen gegebener Objekte zu *Mengen von Objekten* (die selbst wiederum Objekte sind) durch eine
  - – bei fester oberer Schranke – *statische* (z. B. Felder, Mengen),
  - andernfalls eine *dynamische* (Geflechte wie z. B. Listen, Bäume oder Graphen) oder
  - *persistente Typkonstruktion* (z. B. sequentielle Dateien, indexsequentielle Dateien oder B-Bäume).

Gängig – und durchaus typisch – dabei ist, die Voraussetzungen in der Implementierung schwächer, dagegen die Effekte leistungsfähiger zu implementieren, als sie in der Spezifikation beschrieben sind.

Bei der Implementierung der Operationen handelt es sich dann in der Regel um in der einschlägigen Literatur wohldokumentierte Algorithmen zur Bearbeitung der jeweiligen Datenstrukturen.

## Test

Neben der – trivialen – Voraussetzung einer rigiden Einhaltung der Spezifikationen bei der Implementierung, für die Integration der Komponenten zum Gesamtsystem ist und ein möglichst umfassender Test der Implementierungen gegen ihre Spezifikation unter systematischer Berücksichtigung der auftretenden Grenzfälle empfehlenswert.

Dazu werden in der Regel geeignete Testumgebungen geschrieben, was natürlich mindestens einfache Prototypen der Implementierungen aller benutzten Komponenten voraussetzt.

Im Programmlebenszyklus ist dafür allerdings *keine eigene Phase* vorgesehen, weil diese Arbeit integraler Bestandteil der Implementierung ist.

## Systemintegration

Ein Programm wird in einfachen Fällen mit einer Eingabeschleife („*Event-Loop*“) gesteuert, in der – abhängig von Tastatureingaben oder Mausklicks – in einzelne Programmbestandteile verzweigt wird; in komplexeren Fällen mittels eines Auswahlmenüs, in der in Programmbestandteile verzweigt wird, die ihrerseits aus Eingabeschleifen bestehen.

In diesen Fällen ist die Systemintegration eine triviale Aufgabe: Implementierung der Eingabeschleife(n) (ggf. nach Rückgriff auf eine Komponente, die Auswahlmenüs realisiert) und Einbindung der entwickelten Komponenten.

## Revision

Ziel einer Revision des Systems ist in der Regel eine Erweiterung der Funktionalität des Systems oder eine Anpassung an veränderte Voraussetzungen für seinen Einsatz. Sie besteht daher grundsätzlich aus einem Neueintritt in die erste Phase des Programmlebenszyklus, von wo aus er erneut zyklisch durchlaufen wird.

## LEHRPROJEKTE

### Besonderheiten von Lehrprojekten

Zwischen der *kommerziellen Entwicklung* von Informatiksystemen und einem *Lehrprojekt* gibt es *grundsätzliche Unterschiede*, die im wesentlichen darin bestehen, daß

- keine reale „Marktsituation“ sondern vielmehr
- der Charakter einer Lehr- und Lernsituation herrscht (daß im Schulunterricht Wünsche für ein bestimmtes „Produkt“ erwachsen, ist zwar ein reizvoller Gedanke, bleibt aber aufgrund der meist hoffnungslosen Überschätzung des Machbaren unrealistisch),
- die Teilnehmer folglich mehr Gestaltungsspielraum haben (oder zumindest zu haben glauben),
- nur Vorerfahrungen im Kleinen – wie zum Bearbeiten von Übungsaufgaben im Grundstudium erforderlich – verfügbar sind,
- die Teilnehmer in problematischen – im Grunde unverträglichen – Mehrfachrollen arbeiten, deren Aufgaben in der Realität durch unterschiedliche Personen wahrgenommen werden: Als
  - Lernende (mit begrenzten und in der Regel in diesem Zeitpunkt noch nicht stabilisierten Kenntnissen),
  - Systemanalytiker,
  - Systemarchitekten,
  - Programmierer – sowohl im Team wie als Alleinentwickler –
  - sowie Endabnehmer oder Benutzer,
- ein widersprüchliches Spannungsfeld besteht zwischen
  - notwendiger Komplexität zum Studium typischer Probleme der Programmierung im Größeren und
  - hinreichender didaktischer Reduktion,
- die Größe eines Projektes in der Lehre wegen begrenzter zeitlicher Ressourcen im Umfang einer – wenn auch längeren – Unterrichtseinheit, die um Größenordnungen unter kommerziellen Projekten liegen, nicht einmal *ansatzweise* damit vergleichbar und
- eine Erweiterung der Entwicklungskapazitäten (z. B. durch Überstunden oder den Einsatz weiterer Mitarbeiter) ausgeschlossen ist.

Für die Leitung eines Lehrprojekts ergeben sich daraus folgende Thesen, deren Beherzigung wärmstens empfohlen wird:

- Softwareprojekte in der Lehre sind weder Projekte im Sinne der Softwaretechnik noch des projektorientierten Unterrichts in der Schule, sondern *Lehrprojekte*.
- Die Themenstellung darf nicht umfassender sein als die detaillierte Untersuchung einer Teilaufgabe in der Systemanalyse.
- Die Anforderungsdefinition darf nicht zur Festlegung der vielen interessanten Ideen ausarten, die nicht zu schaffen sind.
- Die Spezifikation darf nicht weniger rigide sein, als man es beim Programmieren im Kleinen gelernt hat.
- Die Implementierung muß auf der Basis der Vorkenntnisse der Teilnehmer möglich sein.
- Projektleiter müssen die Aufgabenstellung *soweit* eingrenzen, daß sie die Machbarkeit des Vorhabens garantieren können – d. h. daß
  - sie das „Projekt“ vorher schon mindestens als Prototyp, der die wesentlichen Aspekte umfaßt, „am Laufen“ haben und
  - daß sich die geplante Arbeit *in wesentlichen Teilen auf Vorhandenes stützen* kann und
  - dabei sichergestellt ist, daß die Teilnehmer diese Teile in *dem* Maße beherrschen, wie es für die Arbeit notwendig ist.

## Systemanalyse

Der Aufwand für die Systemanalyse sollte in der Regel in engen Grenzen gehalten werden:

Die Zeit für die tiefere Erarbeitung von Spezialwissen ist nicht vorhanden und der Schwerpunkt der Arbeit in einem Lehrprojekt sollte bei *informatischen* Fragestellungen liegen; auch wenn fachübergreifende oder fächerverbindende Aspekte eine Rolle spielen.

Die Systemanalyse kann in einem Lehrprojekt daher nur als Sachanalyse einer didaktisch reduzierten Themenstellung begriffen werden.

Zur Vermeidung langwieriger und nur selten fruchtbarer Diskussionen über in Frage kommende Projektthemen sollte das Thema – ggf. aus einer gut vorbereiteten Auswahl – vorgegeben werden, so daß sich der Gestaltungsspielraum der Teilnehmer auf die Analyse einiger Objektklassen und die Ausarbeitung geeigneter Teilaspekte der Aufgabenstellung konzentriert.

Auf jeden Fall sollte nicht nur die Behandlung eines neuen Themas, sondern immer auch die Weiterentwicklung eines vorhandenen, gut dokumentierten Systems ins Auge gefaßt werden, weil sich viele der angesprochenen Probleme dadurch auf eine recht natürliche Weise erledigen.

Bei der Arbeit in dieser ersten Phase ist bei Anfängern aufgrund ihrer fehlenden Erfahrung immer wieder eine deutliche Tendenz zur Unterschätzung der Komplexität der zu bewältigenden Probleme zu beobachten, was dazu führt, daß ihre Erwartungen an die Größenordnung des Erreichbaren selten realistisch sind.

Die Projektleitung ist daher für die Abschätzung des Volumens der anfallenden Arbeit und damit für die Kalkulation der zeitlichen und personellen Ressourcen und deren Einhaltung verantwortlich.

Daher muß sie – im Grunde modellhaft wie in der kommerziellen Softwareentwicklung – zwingend

- prinzipiell geeignete Themen bereits im Vorfeld der Arbeit auf ihre Brauchbarkeit für ein Lehrprojekt hin gründlich untersuchen,
- die Komplexität des Themas auf die vorauszusetzenden Kenntnisse der Teilnehmer abstimmen,
- letztlich die Auswahl des Themas festsetzen,
- die Details der Aufgabenstellung stark lenkend moderieren,
- in angemessenem Umfang ausgewählte Teile – vollständig implementiert und dokumentiert – zur Verfügung stellen um die Teilnehmer nicht jedesmal „das Rad neu erfinden“ zu lassen,
- die Benutzung dieser Teile systematisch trainieren,
- sich während der Systemanalyse laufend über die Auswirkungen von Vorschlägen der Teilnehmer Klarheit verschaffen – ggf. durch prototypisches Arbeiten an Entwurf und Realisierung.

Neben der Forderung nach angemessener Überschaubarkeit des Themas (kleines Thema, noch kleineres Thema, noch kleiner, noch viel kleiner, noch kleiner) ist eine gewisse Mindestkomplexität unverzichtbar, um für die Softwaretechnik typische Prinzipien und Methoden aufzuzeigen und Einsicht in ihre Notwendigkeit zu vermitteln:

- angemessen tiefe Beschäftigung mit einer Anforderungsdefinition, insbesondere mit Fragen einer ergonomischen Bedienung,
- Einbeziehung einer über mindestens drei Ebenen verschachtelten

Struktur der beteiligten Objekte, die an mindestens einer Stelle verzweigt ist, um eine nichttriviale Tiefe und Verzweigung in der Systemarchitektur zu erreichen,

- beispielhafte Erstellung von wiederverwendbaren Komponenten, die auch an anderen Stellen im Unterricht einsetzbar sind, sowie
- Berücksichtigung der Möglichkeit alternativer Implementierungen bestimmter Komponenten.

### **Anforderungsdefinition**

Die Einplanung unumgänglicher Einschränkungen aus Entwicklersicht gehört zu den Aufgaben der Projektleitung, da der Durchblick auf mögliche Folgeprobleme von Anfängern nicht erwartet werden kann. Projektleiter müssen entsprechende Überlegungen im Vorfeld der Untersuchung geeigneter Themenstellungen – ggf. unter didaktischen Gesichtspunkten – anstellen und dafür sorgen, daß sie bei der Aufgabenstellung berücksichtigt werden.

Die detaillierten Überlegungen dazu, wie sich das geplante System den Benutzern darstellen soll, sind mühselig und zeitaufwendig; sie werden häufig kontrovers diskutiert und ihre Notwendigkeit wird zu Beginn der Arbeit nicht immer eingesehen.

Es ist unabdingbar, daß die Teilnehmer stets über eine gründliche Detailkenntnis der vorliegenden Teilergebnisse verfügen: es zeigt sich bei Lehrprojekten immer wieder, daß in den anschließenden Phasen plötzlich eigene Wege verfolgt werden, die nicht den Festlegungen der Anforderungsdefinition entsprechen.

Sofern es sich nicht um berechtigte Einwände gegen Widersprüche oder Unvollständigkeiten handelt, denen sofort nachgegangen werden muß, sollten Vorstellungen zu inhaltlichen Änderungen in dieser Phase nicht weiter verfolgt, sondern festgehalten werden; die Diskussion wird nach der Fertigstellung einer ersten Version des Systems wieder aufgenommen, um die zwischenzeitlich gewonnenen Erkenntnisse in einem ansatzweise erneuten Durchlaufen des Programmlebenszyklus in die gemeinsame Arbeit einzubringen.

## Systemarchitektur

Zu dieser Phase gibt es keine spezifischen Besonderheiten von Lehrprojekten, die über das hinausgehen, was dazu im entsprechenden Kapitel postuliert ist.

Aus Erfahrung sei lediglich noch einmal auf folgendes hingewiesen:

Der für Anfänger sehr schwierigen Frage nach dem Einstieg in die *eigentliche Programmierstätigkeit* wird mit dem Hinweis *darauf* begegnet, wie *verblüffend einfach* sich die Gliederung des Systems aus einer Anforderungsdefinition ergibt, die sich streng *an den Objekten eines Systems* orientiert.

## Spezifikation

Zwingend ist die für jede verwendete Komponente die Angabe der Semantik der Datenobjekte und die vollständige und widerspruchsfreie Spezifikation aller Zugriffsoperationen unter Angabe aller Voraussetzungen und Effekte. In der Regel reichen dazu

- saubere umgangssprachliche Formulierungen.

Wenn auf entsprechender Vorkenntnisse zurückgegriffen werden kann, kommen natürlich auch formale Methoden in Betracht:

- funktionale Spezifikationen (d. h. in einer funktionalen Programmiersprache),
- algebraische Spezifikationen oder
- die prädikatenlogische Beschreibung mathematischer Modelle.

Der Einsatz darüber hinausgehender formaler Spezifikationsprachen kommt aus Zeitgründen wohl kaum in Frage (und wäre neben den eben angeführten formalen Alternativen methodisch auch fragwürdig).

## Implementierung

Die Implementierungen der Operationen der Komponenten beruht auf den bei der Programmierung im Kleinen erworbenen Kenntnissen, die ggf. durch Studium der einschlägigen Literatur zu ergänzen oder erweitern sind. Das dient nebenbei der Sicherung wie der exemplarischen Vertiefung der entsprechenden Fertigkeiten und Fähigkeiten.

Wenn sich bei der Implementierung einer Komponente Entwurfsfehler zeigen (meist in Form von Unvollständigkeiten oder mangelnder

Eindeutigkeit, hin und wieder wegen Widersprüchlichkeiten), werden die Spezifikation im Einvernehmen mit allen beteiligten Klienten – auf jeden Fall nur nach Rücksprache mit der Projektleitung – korrigiert und deren Implementierungen den Änderungen angepaßt.

Voraussetzung für die Systemintegration ist natürlich der ausgiebige systematische Test der entwickelten Komponenten. In einem didaktisch reduzierten Programmlebenszyklus handelt es sich dabei nicht um eine eigene Phase, sondern um einen selbstverständlichen Bestandteil der Implementierung.

### **Systemintegration**

Die Vollständigkeit und Widerspruchsfreiheit des ganzen Systems zeigt sich nach dem Zusammenbau des Systems durch das Verbinden („*Linken*“) aller Implementierungen.

Bei konsequenter Arbeit in allen vier Phasen des Programmlebenszyklus nach allen in dieser Dokumentation vorgestellten Richtlinien wird einem dabei das Projekt nicht „um die Ohren fliegen“, d. h. es werden keine grundsätzlichen Schwierigkeiten – etwa in Form eines gefürchteten „*big bang*“ – auftreten, sondern nur Fehler, die in aller Regel leicht lokalisierbar und schnell behebbar sind, wobei sich höchstwahrscheinlich herausstellt, daß sie nachweisbar auf klare Verstöße gegen irgendwelche der angeführten Prinzipien zurückzuführen sind.

### **Revision**

Die anschließende Benutzung des Systems gibt meist unmittelbar Anlaß zu Korrektur- und Verbesserungsvorschlägen.

In Verbindung mit möglicherweise schon in der Systemanalyse ins Auge gefaßten Erweiterungen zeichnet sich damit eine Revision des Systems ab, die – beginnend mit einer erweiterten Systemanalyse – ein Durchlaufen einer weiteren Runde des Softwarelebenszyklus darstellt.

Insbesondere bietet es sich dabei natürlich an, einige *derjenigen* Ideen aufzugreifen, die bei der Systemanalyse von der Projektleitung mit dem Hinweis auf die ansonsten zu große Komplexität der Aufgabe „abgewürgt“ werden mußten.

## ANHANG

### Ausgewählte Literatur

- D. L. Parnas:*  
A Technique for Software Module Specification with Examples  
Comm. ACM 15 (1972), 330–336
- D. L. Parnas:*  
On the Criteria To Be Used in Decomposing Systems into Modules  
Comm. ACM 15 (1972), 1053–1058
- B. Liskow, S. N. Zilles:*  
Programming with Abstract Data Types  
SIGPLAN Notices 9 (1975), 50–59
- N. Wirth:*  
Algorithms + Data Structures = Programs  
Prentice Hall 1976
- J. Guttag:*  
Abstract Data Types and the Development of Data Structures  
Comm. ACM 20 (1977), 396–404
- R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch:*  
Einführung in Software-Engineering  
de Gruyter 1979
- E. Denert:*  
Software-Modularisierung  
Informatik-Spektrum 2 (1979), 204–218
- C. Floyd und H. Kopetz (Hrsg.):*  
Software Engineering – Entwurf und Spezifikation  
Teubner 1981
- C. B. Jones:*  
Software Development: A Rigorous Approach  
Prentice Hall 1980
- M. A. Jackson:*  
System Development  
Prentice Hall 1983

- J. Nievergelt und A. Ventura:*  
Die Gestaltung interaktiver Programme  
Teubner 1983
- G. Pomberger:*  
Softwaretechnik und Modula-2  
Carl Hanser 1984
- N. Gehani und A. D. McGettrick:*  
Software Specification Techniques  
Addison-Wesley 1985
- B. Liskov, J. Guttag:*  
Abstraction and Specification in Program Development  
MIT Press and MacGraw Hill 1986
- I. Sommerville:*  
Software Engineering  
Addison-Wesley 1987, 7th ed. 2006
- B. Meyer:*  
Object-oriented Software Construction  
Prentice Hall 1988, 2nd ed. 1997
- K.-P. Löhr:*  
Software-Bausteine  
LOGIN 9 Heft 6 (1989), 15–21
- R. Harrison:*  
Abstract Data Types in Modula-2  
John Wiley & Sons 1989
- E. Denert:*  
Software Engineering  
Springer-Verlag 1991
- G. Pomberger, G. Blaschek / W. Pree:*  
Grundlagen des Software Engineering (2. Aufl.)  
Software Engineering (3. Aufl.)  
Carl Hanser 1996 / 2004
- H. Balzert:*  
Lehrbuch der Softwaretechnik  
Spektrum Akademischer Verlag 1996

- P. Rechenberg, G. Pomberger:*  
Informatik-Handbuch  
Carl Hanser 1997, 2. Aufl. 1999
- E. S. Roberts:*  
Programming Abstractions in C  
Addison Wesley 1998
- L. Nyhoff:*  
ADTs, Data Structures and Problem Solving with C++  
Pearson Education 1999
- B. Liskov, J. Guttag:*  
Program Development in Java – Abstraction, Specification, and  
Object-Oriented Design  
Addison Wesley 2000
- M. Broy, E. Denert (Hrsg.):*  
Software Pioneers  
Springer 2002
- E. Horn, Th. Reinke:*  
Softwarearchitektur und Softwarebauelemente  
Carl Hanser 2002
- B. Brügge, A. Dutoit:*  
Objektorientierte Softwaretechnik  
Addison-Wesley 2003