WHU-Forschungspapier Nr. 19/ Juni 1993

Wartung objektorientierter Softwaresysteme

Prof. Dr. Franz Lehner, Dipl.Ing. Dr. Hermann Sikora, WHU Koblenz, Lehrstuhl für Wirtschaftsinformatik und Informationsmanagement

Wissenschaftliche Hochschule für Unternehmensführung Koblenz
Burgplatz 2
56179 Vallendar
Telefon 02 61 / 65 09 - 0
Telefax 02 61 / 65 09 - 1 11

Wartung objektorientierter Softwaresysteme

Stand des Wissens und Ergebnisse einer Expertenbefragung

Franz Lehner, Hermann Sikora

Lehrstuhl für Wirtschaftsinformatik und Informationsmanagement
WHU Koblenz
Burgplatz 2
D-5414 Vallendar

Inhalt

1 Einführung

Die objektorientierte Softwareentwicklung kann zunächst als Hilfsmittel zur modellmäßigen und softwaretechnischen Strukturierung von Aufgaben verstanden werden. Eine allgemein akzeptierte Definition gibt es bisher nicht und die gängigen objektorientierten Programmiersprachen, Werkzeuge und Analysemethoden unterscheiden sich in vielen Details. Mit der zunehmenden Verbreitung der objektorientierten Programmierung (OOP) wurden auch einige herkömmliche Programmiersprachen in diese Richtung weiterentwickelt. Beispiele dafür sind C++ und ObjectPascal.

Mit dem objektorientierten Ansatz werden gerade in bezug auf die Wartung viele Hoffnungen verbunden bzw. Erwartungen geweckt. Begründet wird dies häufig damit, daß objektorientierte Programmiersprachen einige Eigenschaften aufweisen, welche u.a. die Modularität, Änderbarkeit und Wiederverwendbarkeit von Programmen unterstützen, und daß sie daher die Wartbarkeit generell positiv beeinflussen. Es gibt jedoch bisher nur wenige Studien, in welchen diese Aussagen empirisch untersucht werden [z.B. MATT87, LIHE93, MANC90, HUMP90]. Ziel des Beitrags ist es, einen Beitrag zur Untersuchung der Auswirkung objektorientierter Softwareentwicklung auf die Wartung zu leisten. Den Ausgangspunkt bilden grundsätzliche Überlegungen auf der Basis der verfügbaren Literatur. Im Anschluß daran werden die Ergebnisse einer Expertenbefragung zur objektorientierten Softwareentwicklung unter der besonderen Berücksichtigung der Auswirkungen auf die Wartung dargestellt.

1.1 Wartungsbegriff und Wartungsarten

In der Umgangssprache wird unter Wartung die Reinigung, die Pflege, die permanente Instandhaltung (Reparaturen) und die vorbeugende Instandhaltung (Verschleißvorbeugung) von Geräten und Maschinen verstanden. In der Kostenrechnung kennt man die Begriffe Reinigung und Instandhaltung. Die Auffassung von Wartung im Sinn des Austausches alter, abgenutzter Teile gilt jedoch nicht für Software. Hier besteht die Wartung in der Beseitigung von Fehlern, in der Anpassung an Änderungen der Umwelt oder der Benutzeranforderungen sowie in der Weiterentwicklung und in Unterstützungsleistungen. Die einzelnen Wartungsarten können wie folgt systematisiert werden [LEHN91]:

- **korrigierende Wartung**: hier handelt es sich um die eigentliche Notfallwartung, d.h. um die Korrekturen zunächst unerkannter Fehler, die beim Einsatz des Softwaresystems auftreten.

- **adaptierende Wartung**: Anpassung des Softwaresystems an eine veränderte Umwelt wie z.B. Betriebssystemänderungen, neue Hardware oder geänderte gesetzliche Bestimmungen.
- **perfektionierende Wartung**: Verbesserung der Leistungen im Sinne einer Performanceverbesserung, aber auch Restrukturierung des Systems mit dem Ziel, den zukünftigen Wartungsaufwand zu reduzieren.
- Die **Weiterentwicklung** umfaßt den Einbau zusätzlicher Funktionen, die ursprünglich nicht vorgesehen waren sowie die Integration mit benachbarten, neu hinzugekommenen oder geänderten Systemen (Funktionserweiterung).
- Mit **Unterstützung** sind Schulungsmaßnahmen für Benutzer, Hilfestellung bei Bedienungsproblemen, Klärung von Fehlersituationen, Performancemessungen, Abstimmung der Wartungsaktivitäten usw. gemeint.

Als Zeitpunkt der Abgrenzung zwischen Entwicklung und Wartung gilt die Übernahme des gesamten Softwaresystems oder eines Teilsystems in den produktiven Betrieb. Art oder Umfang der Änderung, Weiterentwicklung usw. spielen für die Einordnung in die Kategorie Wartung keine Rolle. Voraussetzung ist allerdings, daß trotz der geplanten Änderung die ursprüngliche Zielsetzung und der unterstützte Aufgabenbereich des Softwaresystems im wesentlichen unverändert bleiben.

1.2 Wartung und Wartbarkeit objektorientierter Softwaresysteme

Die Begriffe **Wartung** und **Wartbarkeit** stehen in einer engen Verbindung miteinander. Wartung bezeichnet jene Maßnahmen, mit denen ein betriebsbereiter Zustand erhalten oder hergestellt wird. Wartbarkeit bezieht sich auf alle Eigenschaften, die eine effektive und effiziente Wartung unterstützen. Ein System wird also, bezogen auf Wartung und Wartbarkeit, aus einem unterschiedlichen Blickwinkel betrachtet. Der Zusammenhang zwischen Wartung und Wartbarkeit kann auch über **Qualitätseigenschaften** hergestellt werden. Unter dem Qualitätsmerkmal "Wartbarkeit" werden jene Eigenschaften zusammengefaßt, welche für die Änderbarkeit, für die Weiterentwicklung, für den wirtschaftlichen Einsatz und für die Lebensdauer eines Systems von Bedeutung sind. Software-Qualitätssicherung zielt - mehr oder minder direkt - immer auch auf die Sicherstellung der Wartbarkeit ab. Ein allgemein anerkanntes System solcher Qualitätsmerkmale für die Wartbarkeit existiert allerdings bisher weder für Software generell noch für spezielle Ausprägungen [LEHN91]. Die einschlägige Literatur beinhaltet lediglich eine mehr oder minder genau erläuterte Sammlung qualitativer Merkmale (z.B. Erweiterbarkeit, Anpaßbarkeit, Korrigierbarkeit).

Im Zusammenhang mit der Wartbarkeit sind die Auswirkungen von **Programmiersprachen und Programmiertechniken** besonders hervorzuheben. Die Ergebnisse empirischer Untersuchungen dazu sind allerdings widersprüchlich und die Auswirkungen nicht endgültig geklärt. Einigkeit besteht lediglich in der Auffassung, daß bestimmte Strukturen und Anforderungen an ein fertiges Softwareprodukt beachtet werden sollten. Die Wartung bzw. Wartbarkeit von Software wird demnach durch die Wahl der Sprachkonstrukte und die Gestaltung der Interaktion zwischen den Komponenten des Systems stark beeinflußt. Empfohlen wird u.a. die Beachtung folgender Punkte: Modularität, Programmstruktur, Datenstruktur, Softwarearchitektur, Portabilität, Dokumentation und Parametrisierung. Meist besteht aber nur begrenzter Spielraum, da sich die Auswahlmöglichkeit auf die in der Programmiersprache verfügbaren Sprachkonstrukte beschränkt. Qualitative Aussagen wie "niedrige Komplexität", "verständliche Struktur", "klarer Programmierstil" u.ä. drücken aus, daß Menschen transparente, lineare und einfache Programmabläufe vorziehen bzw. besser verstehen können. Die genannten Eigenschaften gelten generell für die Strukturierung von Software-Systemen. In bezug auf objektorientierte Programmiersprachen wird häufig die Ansicht vertreten, daß sie gegenüber Prozeduralen Programmiersprachen in diesen Punkten Vorteile aufweisen. Die Eigenschaften selbst versucht man mit Hilfe von Software-Metriken zu erfassen bzw. zu messen.

Die Erfahrung im Bereich der Software-Metrik zeigt allerdings, daß es derzeit nicht möglich ist, eine umfassende Liste aller Qualitätsmerkmale (und damit auch der Qualitätsmaße) zu erstellen. Man vermutet, daß jede Organisation bzw. jede Entwicklungsumgebung ihre eigenen, spezifischen Charakteristika besitzt. Sie müssen daher im Kontext einer konkreten Programmiersprache jeweils neu interpretiert werden. Dies führt dazu, daß für eine Metrik oft lediglich Richtlinien vorgegeben werden können. Oft wird auch übersehen, daß die eigentliche Schwierigkeit in der Interpretation der Meßergebnisse besteht; hier existieren im Augenblick noch keine sehr fundierten Erfahrungen und Ergebnisse. Dennoch kann heute davon ausgegangen werden, daß Metriken für diese Zwecke verstärkt eingesetzt werden. Sie sind z.T. bereits fester Bestandteil objektorientierter Entwurfsumgebungen (z.B. Booch-Methode).

Der Versuch, herkömmliche Metriken, die primär für prozedurale Sprachen entwickelt wurden, unverändert auf objektorientierte und logische Sprachen zu übertragen führte bisher zu wenig befriedigenden Ergebnissen [siehe z.B. BUTH91, ZUSE91, DUMK92]. So ist z.B. zu bezweifeln, daß die Meßgrößen selbst in verwandten Sprachen (Komplexität eines Programmes in C und in C++) überhaupt in gleicher Weise definiert werden können. Auch der Einfluß bestimmter Qualitätsmerkmale ist bei unterschiedlichen Programmierparadigmen im allgemeinen unterschiedlich zu bewerten. Zwar ist allen Sprachen gemeinsam, daß Daten gespeichert und manipuliert werden, die Art, wie dies geschieht unterscheidet sich jedoch

fundamental zwischen prozeduralen, logischen und objekt-orientierten Sprachen. Auch die Kriterien zur Strukturierung von Funktionen und Daten sind anders. Das bedeutet, daß zwar die gleichen Prinzipien (z.B. Strukturierung, Hierarchisierung usw.) angewendet werden, daß diese Prinzipien situationsabhängig zu interpretieren sind. Aus diesem Grund wurden für objektorientierte Systeme spezielle Metriken vorgeschlagen oder entwickelt [vgl. z.B. BUTH91, DUMK92[. Abbildung 1 zeigt ein solches System von Meßgrößen, das bei der GMD entwickelt wurde. Konkrete Eigenschaften einzelner Meßgrößen und die empirische Validität solcher Metriken wurden für objektorientierte Programmiersprachen erst ansatzweise untersucht [vgl. LIHE93, BUTH91].

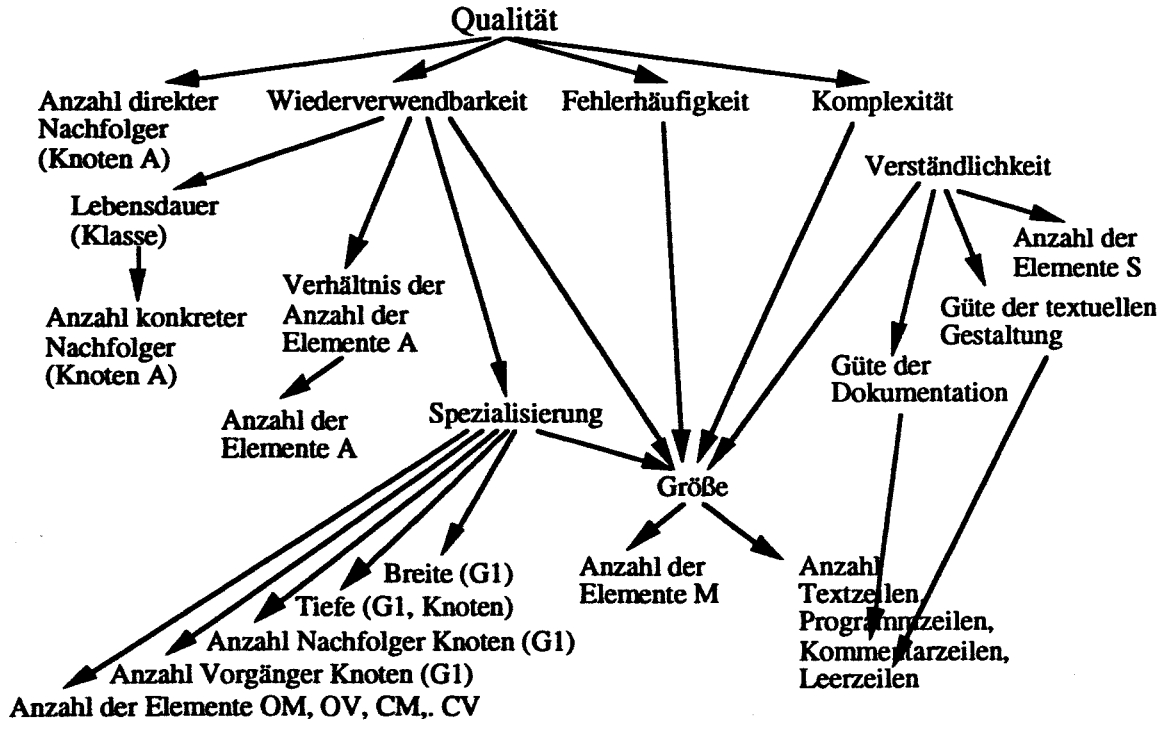

Abb. 1: Zusammenhang zwischen Meßgrößen zur Qualitätsmessung objekt-orientierter Systeme [Quelle: BUTH91, 65]

Speziell für objektorientierte Sprachen wird u.a. die Beachtung folgender Punkte empfohlen [MATT87]:

- Vermeidung von direkten Zugriffen auf Instanzierungsvariable;
- Nutzung der Vererbung, um Gemeinsamkeiten auszudrücken und die Wiederverwendbarkeit zu verbessern;
- Vermeidung unnötiger Objektdefinitionen (z.B. Klasse mit nur einem Child);
- Schaffung neuer Objektklassen, wenn wesentliche Unterschiede betont werden sollen.

In diesem Zusammenhang ist auch auf Versuche hinzuweisen, Programmierkonventionen für einzelne Sprachen festzulegen. Im Unterschied zu Normen besteht die Aufgabe von Konventionen u.a. darin, die Nachteile von heterogenen Programmierstilen auszugleichen und auf diese Weise die spätere Wartbarkeit des Systems sicherzustellen. Als Beispiel können die Programmierkonventionen für C++ genannt werden [MEEG92].

2 Grundlagen der objektorientierten Software-Entwicklung

2.1 Geschichtliche Entwicklung

Der Begriff "Objektorientierung" sagt aus, daß man sich an "Objekten" orientiert und nicht etwa an Funktionen oder Datenflüssen. Die erste Programmiersprache, mit der man Objektorientierung auf Code-Ebene umsetzen konnte, war Simula [DAHL66, DAHL70]. Simula wurde Ende der 60er Jahre entworfen, um Simulationsprobleme adäquater als mit den damals bekannten Programmiersprachen lösen zu können.

Simula unterstützte die Nachbildung von Objekten der realen Welt auf Code-Ebene (z.B. Objekte "Zapfsäule" und "Auto" in einer Tankstellensimulation, Objekte "Kassa" und "Kunde" in einer Supermarktsimulation u.dgl.), sodaß sich Simulationen wesentlich effizienter und natürlicher als mit anderen Sprachen formulieren ließen. Simula umfaßte bereits alle wesentlichen Merkmale der objektorientierten Programmierung (OOP), die in Punkt 2.2 näher erläutert werden.

Die wichtigste Innovation für die OOP brachte Smalltalk [GOLD76, GOLD83, GOLD85]. Smalltalk ist eine "reine" objektorientierte Programmiersprache, d.h. ein Entwickler wird mehr oder weniger gezwungen, objektorientiert zu entwickeln. In Smalltalk ist die Grenze zwischen Entwicklungsumgebung und Sprache fließend.

Die größte Marktbedeutung hat C++ [STRO86]. C++ ist eine hybride Programmiersprache, d.h. daß C++ sowohl objektorientierte als auch traditionell prozedurale Programmierung erlaubt. C++ entstand auf der Basis von C, indem neue Sprachmechanismen hinzugefügt wurden, welche die objektorientierte Programmierung unterstützen.

Weitere Beispiele für objektorientierte Programmiersprachen sind Objective-C [COX86], ebenfalls ein C-Hybrid, Eiffel [MEYE88], eine von der Syntax her an Modula und Ada erinnernde Sprache, die viele interessante softwaretechnische Konzepte aufweist, sowie diverse objektorientierte Pascal-Dialekte. Darüberhinaus gibt es Sprachen/Entwicklungsumgebungen,

die Brücken zur funktionalen Welt bilden (z.B. CLOS – Common Lisp Object System), neue Konzepte umsetzen (z.B. SELF [UNGA91] und Ω-2 [BLAS91] die "klassenlose OOP") oder in ihrer Struktur an "Viertgenerationsumgebungen mit OOP-Elementen" erinnern.

2.2 Eigenschaften objektorientierter Sprachen ("klassische Objektorientierung")

Damit eine Programmiersprache als objektorientiert bezeichnet werden kann, muß sie folgende Eigenschaften aufweisen, die nachfolgend näher erläutert werden:

- Darstellung von Objekten und deren Definition durch Klassen,
- Datenabstraktion durch Datenkapselung und Geheimnisprinzip,
- Vererbung sowie
- Polymorphismus und dynamisches Binden.

Darstellung von Objekten und deren Definition durch Klassen

Ein Objekt im Sinne der OOP ist eine Zusammenfassung von Daten und logisch zusammengehörigen Operationen. Die Daten bestimmen den Status eines Objekts, die Operationen verändern den Status. Die Menge der Operationen legt das mögliche Verhalten eines Objekts fest, d.h. die Aktivitäten, die ein Objekt ausführen kann.

Die Definition von Objekten auf Code-Ebene erfolgt in fast allen heute bekannten OO Programmiersprachen durch das Konzept der Klasse *(class)*. In einer Klasse wird der Aufbau eines Objekts beschrieben, bestehend aus Instanzvariablen (Daten, *instance variables)* und Methoden (Operationen, *methods)*. Von einer Klasse können beliebig viele Objekte (Instanzen, *instances)* angelegt (instanziert) werden. Eine Klasse ist gleichsam eine "Schablone für Objekte", aus der Objekte "herausgestanzt" werden können. Eine Klasse ist somit einer Typdefinition vergleichbar.

Eine Methode ist einer Prozedur in prozeduralen Sprachen vergleichbar und stellt einen Algorithmus dar, der einer Nachricht *(message)* zugeordnet ist. Die Methode gelangt zur Ausführung, wenn das Objekt, zu dem die Methode gehört, die entsprechende Nachricht erhält. Eine Methode modifiziert in der Regel den Zustand "ihres" Objekts, der durch die Instanzvariablen repräsentiert wird. Eine Methode kann ihrerseits Nachrichten an andere Objekte oder das eigene Objekt senden und somit die Ausführung anderer Methoden veranlassen.

Eine Nachricht ist einem Prozeduraufruf in prozeduralen Sprachen vergleichbar. Wird eine Nachricht an ein Objekt geschickt und gibt es in diesem Objekt eine zugehörige Methode, ge-

langt diese zur Ausführung. Gibt es keine zugehörige Methode, reicht das Objekt die Nachricht an seine Oberklasse weiter. Dies wird solange ausgeführt, bis eine zugehörige Methode gefunden und ausgeführt oder die Wurzelklasse erreicht wird. In letzterem Fall liegt ein Laufzeitfehler vor. Abhängig von der Klasse eines Objektes kann dieselbe Nachricht bei verschiedenen Objekten unterschiedliche Aktionen auslösen (dynamische Bindung, siehe später).

Neben dem Klassenkonzept gibt es auch das (noch wenig verbreitete) Konzept der *Objektprototypen*, bei dem Objekte durch Cloning von Prototypen erzeugt werden (nicht mit Prototyping zu verwechseln!).

Datenabstraktion (data abstraction) durch Datenkapselung (data encapsulation) und Geheimnisprinzip (information hiding)

Eine Klasse kann als abstrakter Datentyp betrachtet werden. Datenabstraktion drückt aus, daß von der konkreten Realisierung des Datentyps abstrahiert wird: Klassen präsentieren eine abstrakte Sicht der Daten durch Datenkapselung, d.h. einem Nutzer einer Klasse werden nur Zugriffsoperationen zur Verfügung gestellt, die konkrete Datenstruktur bleibt verborgen. Dieses Prinzip, die konkrete Implementierung einer Datenstruktur zu verbergen, wird als Geheimnisprinzip oder Information Hiding bezeichnet [PARN72].

Vererbung (inheritance)

Neben der Benutzung durch Instanzierung können Klassen auch als Basisklassen *(base classes)* für die Definition von Unterklassen *(subclasses)* dienen. Von einer Klasse können durch Anwendung eines Vererbungsmechanismus beliebig viele Unterklassen gebildet werden, die wiederum als Basisklassen dienen können.

Vererbung bedeutet die Weitergabe der Eigenschaften einer (Basis-) Klasse an alle ihre Unterklassen, ohne daß sie dort erneut notiert werden müssen. Eine Unterklasse erbt definitionsgemäß alle Methoden und Instanzvariablen der zugehörigen Basisklasse.

In den Unterklassen können Instanzvariablen und Methoden neu hinzugefügt und Methoden modifiziert oder ersetzt werden. Durch die fortgesetzte Anwendung der Unterklassenbildung entsteht eine Hierarchie von Klassen (Vererbungs- oder Klassenhierarchie), die die Beziehungen der Klassen zueinander (Ober- und Unterklassen) wiedergibt. Man kann Vererbung somit als Mechanismus zur Definition neuer Typen betrachten, die auf bestehenden Typen aufbauen (Typerweiterung).

Im Falle der Einfachvererbung *(single inheritance)* hat jede Klasse einer Klassenhierarchie mit Ausnahme der Wurzelklasse(n) genau eine zugehörige Basisklasse. Die Klassenhierarchie weist eine Baumstruktur auf. Bei der Mehrfachvererbung *(multiple inheritance)* kann eine Klasse einer Klassenhierarchie mehr als eine zugehörige Basisklasse haben. Die Klassenhierarchie weist in diesem Fall eine Netzstruktur auf.

Die Vererbung unterstützt die komfortable Wiederverwendung bestehender Komponenten in einem Ausmaß, wie dies in traditionellen prozeduralen Sprachen nicht möglich ist.

Abstrakte Klassen *(abstract classes*, hat nichts mit Datenabstraktion im obigen Sinne zu tun!) werden nicht für den Zweck geschaffen, Instanzen (Objekte) davon anzulegen, sondern dienen der flexiblen und effizienten Organisation einer Klassenhierarchie. Beispielsweise könnte eine abstrakte Klasse alle gemeinsamen Instanzvariablen und Methoden einer logisch zusammengehörigen Gruppe von Klassen enthalten, die dann als Unterklassen dieser abstrakten Klasse definiert werden.

Durch dieses "Faktorisieren" von Gemeinsamkeiten können Klassenhierarchien oftmals übersichtlicher und in ihrer Anwendung effizienter gestaltet werden. Die Nachbildung realer Objekte auf Code-Ebene alleine reicht nicht aus, ein effizientes objektorientiertes System zu entwickeln. Nur in Verbindung mit der Bildung abstrakter Klassen (die keine Korrespondenzen in der realen Welt haben) können die Potentiale der OOP voll ausgeschöpft werden.

Polymorphismus und dynamisches Binden

In statisch typisierten Sprachen bezeichnet Polymorphismus *(polymorphism*, "Vielgestaltigkeit") die Eigenschaft einer "Objektvariablen" (eine Variable, mit der ein Objekt referenziert wird), zur Laufzeit Referenzen auf unterschiedliche Objekte zu unterstützen, deren Klassen aber in einer Vererbungsbeziehung stehen müssen.

Eine Objektvariable wird beispielsweise als Referenzgröße für Objekte der Klasse X definiert. Durch Polymorphismus ist es nun möglich, daß mit dieser Objektvariablen auch solche Objekte referenziert werden können, deren Klassen Unterklassen von X sind. Der umgekehrte Weg (Referenzierung von Oberklassen von X) ist nicht möglich.

Die Mächtigkeit dieses Konzeptes entsteht in Kombination mit der dynamischen Bindung: Erhält ein Objekt eine Nachricht, so ist der auszuführende Methoden-Code nicht nur vom Namen der Methode, sondern auch vom Laufzeit-Typ (Klasse) des Objekts abhängig. Bei dy-

namischer Bindung wird also erst zur Laufzeit festgestellt, welcher Code tatsächlich auszuführen ist.

Polymorphismus erlaubt es also, den selben Methodennamen als Nachricht an mehrere Objekte zu schicken, deren Klassen in einer Vererbungsbeziehung stehen. Welche konkrete (objektspezifische) Methode sich hinter diesem Namen verbirgt, stellt die dynamische Bindung fest.

2.3 "Objektorientierung" versus "Objektbasierung"

Fehlt einer Programmiersprache eines der in Punkt 2.2 beschriebenen Merkmale, so kann sie auch nicht als objektorientiert bezeichnet werden. Beispielsweise unterstützt Ada zwar die Datenabstraktion, nicht jedoch die Anwendung von Vererbung, Polymorphismus oder dynamischer Bindung. Für Sprachen dieser Kategorie hat sich die Bezeichnung "objektbasiert" eingebürgert.

2.4 Anwendungsentwicklung mit Klassenbibliotheken und Application Frameworks

Eine objektorientierte Programmiersprache alleine ist nur der halbe Weg. Erst die Verbindung der Sprache mit einer Klassenbibliothek (Klassenhierarchie), die vorgefertigte Bausteine für die Entwicklung von Anwendungen zur Verfügung stellt, und Werkzeugen, die das Arbeiten mit der Klassenbibliothek unterstützen (z.B. Browsing-Werkzeuge, grafische Hierarchie-Editoren), ergibt eine Entwicklungsumgebung.

Eines der fortgeschrittensten Konzepte der objektorientierten Systementwicklung sind Application Frameworks (AFs). AFs sind Klassenbibliotheken, die nicht nur aus einer losen (wenn auch hierarchisch geordneten) Anordnung von Klassen bestehen, sondern auch den nicht-applikationsspezifischen Kontrollfluß und alle notwendigen Grundfunktionen (z.B. Verwaltung der Fenster, Pop-Up und Pull-Down Menüs) einer leeren Musterapplikation enthalten.

AFs besitzen z.Zt. vor allem im Zusammenhang mit grafischen Benutzeroberflächen große Bedeutung, da sie den Software-Entwickler, der Software für eine grafische Oberfläche erstellt, von der sehr aufwendigen Programmierung immer gleichartig wiederkehrender applikationsneutraler Teile einer Endanwendung entlastet. Der Entwickler erweitert durch konsequente Anwendung des objektorientierten Paradigmas diesen Applikationsrahmen zu dem gewünschten Endanwendungsprogramm. Um zu einer Endanwendung zu gelangen, müssen

vom Entwickler an definierten Stellen ("Hooks") applikationsspezifische Modifikationen vorgenommen werden. Abbildung 2 verdeutlicht diesen Sachverhalt.

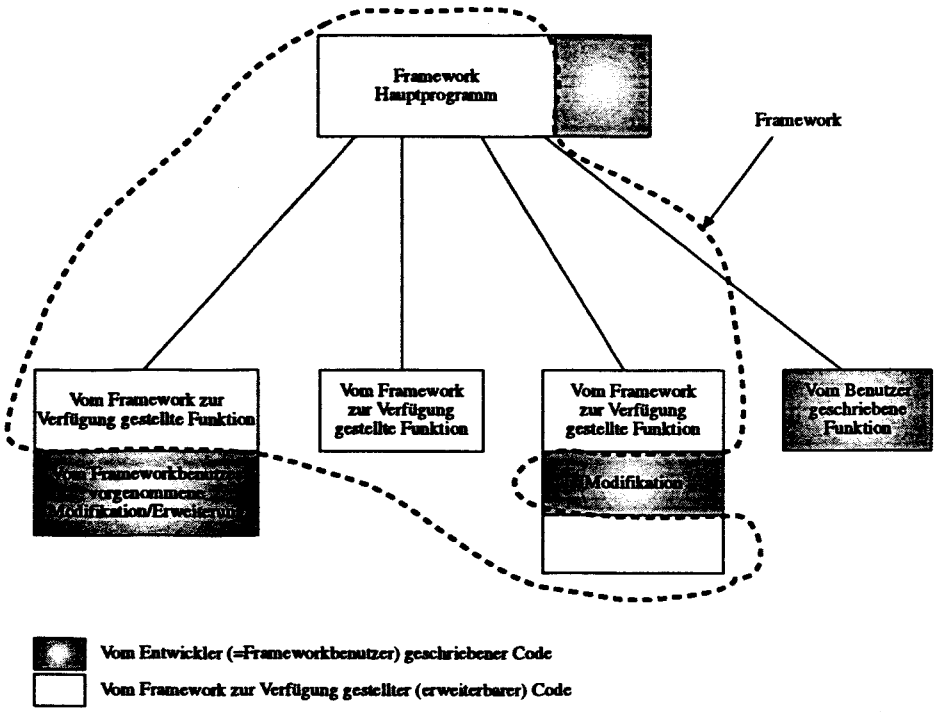

Abb. 2: Software-Entwicklung mit einem OO Application Framework (vgl. [SCHM86])

Abbildung 3 zeigt die grobe Architektur von ET++, einem AF für die Applikationsentwicklung mit C++ auf Unix-Workstations. Abbildung 4 zeigt einen Teil von ET++ (die "Visual Objects" Klassenhierarchie).

Klassenbibliotheken, die keine AFs sind (somit keine Kontrollfluß-Mechanismen enthalten), bieten also "nur" eine Sammlung von Klassen, die miteinander in einer Vererbungsbeziehung stehen. Dies kann für viele Zwecke ausreichend sein. Beispielsweise kann eine einfache Klassenbibliothek Basisdatenstrukturen zur Verfügung stellen, die als "Basic Building Blocks" verwendet werden können, z.B. verschiedene Collection-Klassen (Listen, Mengen u.dgl.).

2.5 Anwendungsentwicklung mit "nicht-klassischer Objektorientierung"

Für die bisher beschriebenen Eigenschaften (Datenabstraktion, Vererbung, Polymorphismus, dynamische Bindung, Klassenbibliotheken) kann man das Adjektiv "klassische Objektorientierung" vergeben: man findet sie in Sprachen, deren Wurzeln (mehr oder weniger gut er-

kennbar) in der Welt der prozeduralen Sprachen liegen (z.B. Simula, Smalltalk, C++, Eiffel, objektorientierte Pascal-Dialekte, usw.).

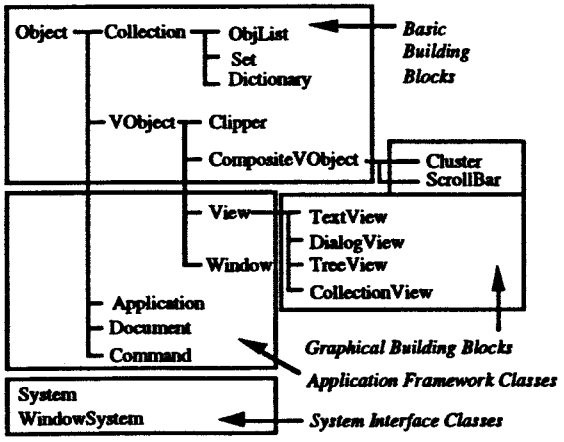

Abb. 3: Grobarchitektur von ET++ [GAMM91, WEIN88, WEIN89, WEIN91]

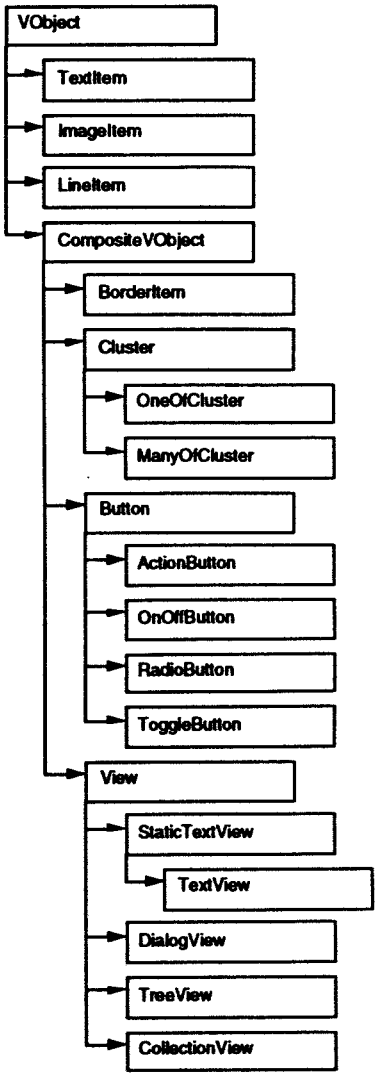

Abb. 4: Teil von ET++ in größerem Detail

Objektorientierung ist aber kein Monopol dieser Welt. Es gibt Anwendungsentwicklungssysteme, die nicht aus den Teilen *objektorientierte Programmiersprache + Klassenbibliothek + Werkzeuge* bestehen, dennoch aber als objektorientiert bezeichnet werden können. Es handelt sich dabei um Systeme, bei denen der Werkzeugcharakter im Vordergrund steht, eine isolierte objektorientierte Programmiersprache ist nicht erkennbar. Typischerweise werden die objektorientierten Eigenschaften (vor allem die Objektdefinition und -erzeugung sowie die Vererbung) nicht durch Codierung, sondern durch die Anwendung von integrierten Werkzeugen zugänglich gemacht. Mitunter bieten diese Systeme auch Möglichkeiten, Regelwissen expertensystem-ähnlich zu spezifizieren.

Zunächst stellt eine solche Umgebung (ähnlich Application Frameworks) einen leeren Applikationsrahmen und ein nicht (nur) durch Codierung, sondern durch Werkzeuge zugängliches Objektsystem zur Verfügung. Beispielsweise wird die Bildung einer (endapplikationsspezifischen) Unterklasse durch die Anwendung eines grafischen Hierarchie-Editors erreicht. Für die Gestaltung der Endbenutzeroberfläche stehen generierende GUI-Editoren zur Verfügung. Für die Verbindung zu Datenbanken werden entsprechende Front-End Werkzeuge angeboten.

Der "applikatorische Kern" (die Ablauflogik) wird mit Sourcecode-Editoren in das um die applikationsspezischen Unterklassen erweiterte Objektsystem eingebracht. Dafür steht meist eine proprietäre, in der Regel an 4GL-Sprachen erinnernde Syntax zur Verfügung. Deshalb werden diese Anwendungsentwicklungssysteme gelegentlich auch als "objektorientierte 4GLs" bezeichnet. Eine Kategorisierung wird dadurch aber nicht erleichtert: ist schon der Begriff "4GL" als schwer greifbar anzusehen, so ist dies bei "objektorientierte 4GL" noch viel stärker der Fall.

Als charakteristisches Merkmal dieser Umgebungen kann zusammenfassend die Schwierigkeit der Trennung zwischen Werkzeugen, Objektsystem und Sprache angesehen werden. Aber auch hier gibt es "klassische Vorbilder": Smalltalk war das erste objektorientierte Entwicklungs*system* mit fließender Grenze zwischen Sprache und Umgebung.

2.6 Objektorientierung in Analyse und Design

Parallel zur endgültigen Etablierung der OOP Ende der 80er Jahre wurde damit begonnen, die Objektorientierung auch in die Analyse und das Design von Informationssystemen einzubringen. Objektorientierung sollte kein Monopol der Programmierung sein. Die zentrale "Vision", die der Entwicklung jeder objektorientierten Analyse- und Designmethode zu Grunde gelegt wurde und wird, besteht darin, einen bruchlosen Software-Life-Cycle zu erreichen: objektorientiertes Denken und Vorgehen von der Analyse über das Design bis zur Programmierung.

Heute existieren einige Dutzend Methoden der objektorientierten Analyse (OOA) und des objektorientierten Designs (OOD). Diese Methoden fußen auf verschiedenen Ansätzen: einige setzen auf traditionellen Methoden auf und erweitern diese um OO Prinzipien, andere sind "rein OO". Es gibt auch einige mit dem Attribut "objektorientiert" versehene Methoden, die bei näherer Betrachtung bestenfalls das Attribut "objekthaft" verdienen.

Ein de-facto Standard, wie er etwa durch Structured Analysis (SA) [DeMA78, GANE79, McME84, YOUR89] in der traditionellen Welt gegeben ist, ist in der OO Welt (noch) nicht erkennbar. Dennoch gibt es einige "key players" (Reihung ohne Wertung): Shlaer&Mellor

[SHLA88, SHLA89, SHLA92] und Rumbaugh et al. [RUMB91] im Bereich der datenmodellorientierten Methoden, Booch [BOOC91] und Coad&Yourdon [COAD91] bei strukturorientierten Methoden, Page-Jones et al. [PAGE89, PAGE90] bei datenflußorientierten Methoden, Wirfs-Brock et al. [WIRF90] bei verhaltensorientierten Methoden sowie Odell&Martin [ODEL92] bei ereignisorientierten Methoden. Schaschinger gibt in [SCHA92] einen Vergleich objektorientierter Analyse- und Design-Methoden.

Eine der Hauptschwächen heutiger OOA/OOD-Methoden ist darin zu sehen, daß sie einen der zentralen Vorteile der OOP, nämlich die Wiederverwendung von Software-Bausteinen im Rahmen von Klassenbibliotheken, nicht oder nur ungenügend berücksichtigen. So gut die Methoden geeignet sind, die Analyse- und Designaktivitäten von Projekten zu unterstützen, die "auf der grünen Wiese beginnen", so problematisch sind sie, wenn es darum geht, umfangreiche, bewährte Klassenbibliotheken einzubinden. Vor allem die Praktiker betonen, daß durch dieses Manko erst recht wieder ein Bruch entsteht: OOA und OOD-Methoden seien zuwenig auf bottom-up Entwurfsschritte ausgerichtet, die in der objektorientierten Systementwicklung aber eine wichtige Rolle spielen.

3 Ergebnisse einer empirischen Untersuchung

3.1 Untersuchungsmethode und Vorgehen

Bisher gibt es nur relativ wenige Untersuchungen, die sich mit den Auswirkungen der objektorientierten Softwareentwicklung auf die Wartung empirisch beschäftigen [z.B. MATT87, LIHE93, MANC90, HUMP90]. Dies wurde zum Anlaß genommen, die Ergebnisse einer einschlägigen Untersuchung auch im Hinblick auf die Wartungsproblematik auszuwerten. Der eigentliche Zweck der Untersuchung bestand darin, Problembereiche, Erfahrungen und Trends zur objektorientierten Programmierung zu erheben. Dazu wurden unstrukturierte Fach-Interviews mit 35 OOP-Entwicklern durchgeführt, die zur Hälfte aus einem betrieblichen Umfeld, zur Hälfte aus dem Hochschulbereich stammten. Jedes Interview (Vier-Augen-Gespräche) dauerte durchschnittlich 90 Minuten und umfaßte 35 Fragen, die in folgende Gruppen gegliedert waren: Merkmale und Konzepte objektorientierter Programmiersprachen, Klassenbibliotheken/Application Frameworks, Projektmanagement, Trends. Die Befragungsergebnisse wurden qualitativ/taxativ ausgewertet.

Nachfolgend werden jene Auswertungsergebnisse, die sich auf Wartung von objektorientierten Softwaresystemen beziehen, gekürzt und verdichtet präsentiert. Die Gliederung

entspricht den Fragen im Interviewleitfaden. Bezüglich einer vollständigen Darstellung der Untersuchungsergebnisse wird auf die angeführte Literatur verwiesen [SIKO92].

Jede Frage wird kursiv geschrieben und umrandet präsentiert, z.B.:

Wie sehen Sie ... ?

Bemerkungen der Verfasser zu einer Frage (z.B. nähere Erläuterungen, die nicht zur Auswertung selbst gehören) werden durch ein ">" eingeleitet und in einem eigenen Zeichensatz notiert, z.B.:

> Dies ist eine Bemerkung.

Die Aussagen zu einer Frage (die Fragenauswertung) werden in einzelne Absätze gegliedert wiedergegeben. Zu Beginn eines Absatzes oder einer logisch zusammengehörigen Gruppe von Absätzen steht ein "◊" mit einer entsprechenden Überschrift, z.B.:

◊ *Dies ist eine Überschrift zu einem Auswertungsabsatz*

3.2 Ergebnisse

Welche Probleme gibt es für einen Entwickler, der in die OOP einsteigt?

◊ *Objektorientiertes Denken*
Das Hauptproblem zu Beginn ist die Notwendigkeit, das herkömmliche, rein algorithmische Denken aufzugeben und das Denken in objektorientierten Zusammenhängen zu beginnen (oder zu entdecken, daß dieses Denken schon latent vorhanden war und nur mehr aktiviert und geschult werden muß). Die Fähigkeit der strukturierten Erfassung des Problembereichs durch *abstraktes* Denken ist für die OOP wichtiger als für die konventionelle Programmierung.

Ein Problem stellt der Wunsch dar, den Kontrollfluß in einem AF 100%ig transparent wie in herkömmlichen Systemen verfolgen zu wollen. Rein sequentielles Denken ist dafür ein Hindernis. Die Notwendigkeit, vertraute Denkmuster aufzugeben, stellt für manche Entwickler ein großes Problem dar, vor allem, wenn sie auch in der nicht-OOP Welt noch nie mit ereignis-gesteuerter Programmierung konfrontiert waren.

◊ *Fragestellungen eines Anfängers*
KBs müssen als Sammlung von Algorithmen und Datenstrukturen begriffen werden. Trotz vorhandener Werkzeuge ist die Komplexität einer KB nicht einfach zu meistern. Typische auftretende Fragen sind:

- Wo beginne ich? Welche ist meine erste Klasse? Welche Klassen sind zu berücksichtigen, um das gegebene Problem zu lösen? Was sind die "Schlüsselklassen"?
- Welche Klasse liefert welche Funktionalität (welche Semantik liegt hinter einer spezifischen Klassenschnittstelle)? Welche Beziehungen existieren zwischen den Klassen?
- Ist es notwendig, eine Unterklasse zu definieren oder reicht eine Instanz einer bestehenden Klasse?
- Welche Methoden müssen oder dürfen nicht überschrieben werden, um eine Unterklasse korrekt zu spezifizieren? Wie werden Methoden zur Ausführung gebracht?

(Beispiel: Welche Aufrufe in den View-Klassen einer KB sind notwendig, damit etwas in einem Fenster dargestellt wird?)

- Was kann an das AF delegiert werden und was muß "manuell" gelöst werden? Warum wurden die gegenständlichen Entwurfsentscheidungen getroffen?
- Wo sollte mit dem Debugging begonnen werden?

◊ *Bewältigen der Anfangsprobleme*
Um sich in einer KB zurechtzufinden, ist es wichtig, fremden Code lesen zu lernen und verteilte Strukturen zu verstehen. Deshalb ist es wichtig, allgemeine (idealerweise auch sprach- bzw. systemunabhängige) "Entwurfsmuster" zu dokumentieren, z.B. Verteilung des allgemeinen Kontrollflusses auf Klassen oder Prinzipien der Verwaltung von Dokumenten, die zu einer gerade aktiven Anwendung gehören.

Bücher mit diesen Inhalten als Anfänger-Basisliteratur sind selten. Sie müssen die zugrundeliegende Philosophie und die Konzepte hinter der KB an Hand konstruktiver Beispiele *("Programming Pearls")* erläutern, z.B. "Was ist ein Command-Objekt und wie wird es korrekt verwendet?". Bestehende Programmierkonventionen müssen transparent erklärt werden: Warum existieren sie? Was muß getan werden, um sie einzuhalten?

◊ *Vom objektorientierten Denken zur objektorientierten Applikationsentwicklung*
Objektorientiertes Denken muß zur objektorientierten Applikationsentwicklung führen, ohne in konventionelles Denken zurückzufallen.

Hier helfen natürlich AFs, da die grundlegende Applikationsstruktur zur Verfügung gestellt wird. Es muß nicht von Null begonnen werden, Entwurfsrahmen und Richtlinien sind vorhanden. Das Einarbeiten in AFs wird deshalb anders aussehen als das in KBs: Ein AF besteht in der Regel *nicht* aus (relativ) unabhängigen Blöcken, die entsprechend benutzt werden können. Das AF ist für den Einstieg in die OOP zwar hilfreich, zu Beginn jedoch hat der Anfänger Schwierigkeiten, die Fülle an Beziehungen (z.B. zwischen Dokumenten, Views, Fenstern, Kommandos usw.) zu erfassen. Es gibt keine Filtermechanismen, die den Anfänger davor bewahren, ggf. von der Detail-Komplexität erschlagen zu werden.

Besonders schwierig ist das Erkennen von Beziehungen zwischen Klassen, die nicht auf Vererbung beruhen, z.B. im Falle des MVC-Modells in Smalltalk [GOLD83, GOLD85], da jede Klasse potentiell mit jeder anderen Klasse korreliert und dies nicht evident ist. Das Fehlen von Typen verstärkt diese Problematik weiter.

Dieser schwierige Prozeß des Einarbeitens und Verstehens kann naturgemäß als wertvolle Investition betrachtet werden, wenn es z.B. darum geht, eine Entscheidung für einen Kontrollmechanismus zu treffen und mehrere vorhandene Mechanismen das gegenständliche Problem zu lösen scheinen. Die Lernkurve ist unterschiedlich zu der in der konventionellen Programmierung. Nach einem ersten normalen Anstieg folgt eine relativ lange Durststrecke: Es wird nichts wirklich Neues gelernt, allerdings wird Erfahrung gewonnen.

◊ *Qualität der KBs/AFs aus der Sicht des Anfängers*
Die Einarbeitungshürden hängen naturgemäß auch sehr von der Qualität der KB/des AFs und der zugehörigen Dokumentation ab. Werden beispielsweise spezielle Merkmale einer Sprache (z.B. das Friend-Konzept von C++) nicht im Verhältnis zu ihrer Bedeutung, sondern über Gebühr verwendet, wird es (nicht nur) für den Anfänger besonders schwierig, die eigentlich implementierten Konzepte zu erkennen.

Viele Probleme heutiger KBs/AFs haben ihre Ursache in der Fülle ihrer Konzepte. Es gibt KBs, die z.B. 20 verschiedene Sets und 30 Stacks anbieten. Solche flachen KBs erzeugen

mehr Probleme als sie lösen. Statt sich primär mit der eigentlichen Problemlösung beschäftigen zu können, stehen Fragen wie "Was ist der Unterschied zwischen den einzelnen Sets?" im Vordergrund. Der bessere Weg wäre z.B. ein abstrakter Stack mit einer vollständigen Beschreibung von Ableitungsmöglichkeiten.

Welche Probleme gibt es mit der (einfachen[1]) Vererbung? Was muß beachtet werden?

◊ *Vermischung von Konzepten*
Mit der Vererbung lassen sich drei verschiedene Konzepte umsetzen: Spezialisierung, Erweiterung und Generizität (im Sinne von Parametrisierung). Da es auf der Ebene der Codierung keine explizite, syntaktisch erkennbare Unterscheidung zwischen diesen Anwendungen der Vererbung gibt, ist entsprechende Entwurfs- und Implementierungsdisziplin notwendig.

◊ *Auffinden von Klassen*
Der Prozeß des Findens von geeigneten Basisklassen für die Lösung eines gegenständlichen Problems ist nicht trivial. Wenn man glaubt, die entsprechende Klasse gefunden zu haben, steht man vor dem Problem, deren Funktionalität örtlich und logisch genau zu erfassen. Die wichtigsten Fragen dabei: Wie ist die Klasse zu benutzen? Welche *"Hooks"*[2] gibt es? Die zeitraubende Tätigkeit des Durchsuchens von Dutzenden von Dateien mit Klassendefinitionen sollte durch ein Klassen-Retrieval-System mit Erklärungskomponente ersetzt werden.

◊ *Verstehen einer Klassenbibliothek*
Code und Funktionalität sind über die Klassenhierarchie verteilt und deshalb nicht unmittelbar verständlich. Um das Verhalten einer bestimmten (Unter-) Klasse zu verstehen, ist das Verstehen der Funktionalität aller zugehörigen Oberklassen notwendig. Ein Benutzer verfügt über einen hohen Freiheitsgrad in der Benutzung einer KB. Der Aufruf einer überschriebenen Methode verlangt Kenntnis der geerbten Funktionalität.

Ist das Wissen nicht ausreichend, so ist die Wahrscheinlichkeit hoch, daß sogar routinemäßige Änderungen von Oberklassen im Rahmen der Wartung große Probleme in Unterklassen mit sich bringen. Oftmals hat ein Benutzer nur Spekulationen oder Vermutungen über die Funktionalität und korrekte Verwendung einer Klasse, was aber zuwenig ist, wenn ein Projekt unter industriellen Bedingungen durchgeführt wird.

◊ *Werkzeuge für das Einarbeiten*
Werkzeuge, die den Prozeß des Einarbeitens in obigem Sinne unterstützen, sind noch sehr selten, in letzter Zeit sind aber verstärkte Forschungs- und Entwicklungsanstrengungen in diese Richtung zu beobachten. Ein Werkzeug sollte beispielsweise folgende Fragen beantworten:

- Was wird wo verwendet?
- Was wird wo definiert?
- Wie wird diese Klasse korrekt bzw. am effizientesten verwendet?
- Welche Beziehungen existieren zwischen den Klassen?
- Gibt es irgendwelchen Aufwand ohne Nutzen, z.B. geerbte Instanzvariablen und Methoden, die nicht benötigt werden?

---

1 Diese Frage bezieht sich allgemein auf die Vererbung und die einfache Vererbung im speziellen. Mehrfache Vererbung wird zunächst ausgeschlossen, da es dafür eine eigene Frage gibt.

2 *Hooks* sind jene Stellen in generischen Applikationsgerüsten (z.B. AFs), an denen vom Entwickler die applikationsspezifischen Modifikationen vorgenommen werden müssen.

◊ *Entwurfsentscheidungen*
Es ist schwierig, eine KB ohne Werkzeuge vollständig zu verstehen. Praktisch unmöglich ist es, dies mit einer schlecht entworfenen Bibliothek zu vollbringen. Aber noch um eine Dimension schwieriger ist es, eine gute KB zu entwerfen[3]. Der Designer sollte im Idealfall alle zukünftigen Anwendungen und Benutzerwünsche hinsichtlich Vererbung voraussehen können, was aber nicht möglich ist. Um aber dennoch zu einer gut entworfenen KB zu gelangen, muß neben der Anwendung bewährter Entwurfsprinzipien die (aktive) Bereitschaft vorhanden sein, die KB ständig zu verbessern. Mehrere Re-Designs der Vererbungshierarchie sind die Regel, nicht die Ausnahme.

Dies hat natürlich auch Auswirkungen auf bereits bestehende Applikationen: Wenn die Vorteile einer neuen KB/AF-Version in bestehenden Applikationen nutzbar gemacht werden sollen, so ist eine erneute Übersetzung notwendig. Es ist dann auch von der Qualität der Applikations-Software abhängig, wie umfangreich und aufwendig die Wartungsarbeiten ausfallen, die sich durch das Re-Design der Vererbungshierarchie ergeben.

Oftmals sind nur Teilaspekte einer (Basis-) Klasse von Interesse, aber alle Eigenschaften dieser werden vererbt, wenn eine neue Unterklasse definiert wird. Es stellt sich nun die Frage, ob eine Unterklasse gebildet und der Overhead des "unnütz Geerbten" in Kauf genommen werden soll, oder ob stattdessen die Unterklasse nicht besser unterhalb der der Basisklasse zugehörigen Oberklasse angesiedelt werden sollte (Bildung einer "Schwesterklasse"); oder sollte nicht überhaupt auf die Bildung einer neuen Unterklasse verzichtet und versucht werden, das Problem mit der ursprünglichen Basisklasse zu lösen?

Nach diesen Ausführungen ist evident, daß der Grad an Wiederverwendbarkeit von KBs von der Qualität des Entwurfs abhängt. Schlechte Entwurfsentscheidungen können auch durch das objektorientierte Paradigma nicht gemildert werden. Sie resultieren in jedem Paradigma in schlechten Ergebnissen.

◊ *Programmierkonventionen*
Bei der Implementierung einer Klassenhierarchie müssen Programmierkonventionen konsequent eingehalten werden, z.B. bezüglich Namensgebung: Der Designer sollte versuchen, durch geschickte und konsistente Namenswahl (z.B. Klassennamen) ggf. auch Hierarchie-Informationen in Bezeichner zu packen (zumindest die "lokale Vererbungsumgebung" betreffend). Die in der Bibliothek verwendeten Konventionen sollten ein Modell für den Anwendungsprogrammierer darstellen.

Konventionen sollten auch Regeln für die Dokumentation der benutzten Konzepte umfassen. Hier besteht ein wesentliches Problemfeld, da lineare Dokumentation nicht ausreicht bzw. nur sehr eingeschränkt nutzbringend möglich ist. In diesem Bereich sind Forschungsaktivitäten im Gange, vor allem im Bereich der Hypertext- und Literate-Programming-Systeme. Ein Beispiel für ein solches Werkzeug Art ist DOgMA [SAME90, SAME91].[4]

Fragenkomplex "Mehrfachvererbung"

> Es werden drei Fragen zum Thema Mehrfachvererbung auswertungsmäßig zusammengefaßt. Zunächst werden die Fragen mit einer kurzen quantitativen Auswertung präsentiert, dann erfolgt die qualitative Auswertung. Ein *"Unentschieden"* in den quantitativen Auswertungen steht für Antworten

---

3 Gamma und Weinand behandeln in [GAMM91] und [WEIN91] ausführlich den Entwurf und die Implementierung des objektorientierten AFs ET++.

4 Weitere Ausführungen zu diesem Thema finden sich unter *"Welche OOP-spezifischen Probleme bezüglich Lesbarkeit von Programmen sehen Sie?"* und *"Welche OOP-spezifischen Anforderungen an die Code-Dokumentation sehen Sie?"*.

der Art *"Kann so nicht beantwortet werden, das kommt darauf an."* Die zugehörigen Begründungen sind in der qualitativen Auswertung enthalten.

Halten Sie Mehrfachvererbung für problematisch?

Von 35 Befragten:

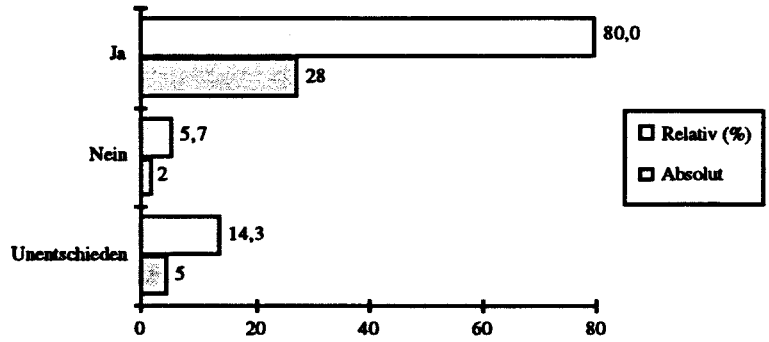

Abb. 5: "Mehrfachvererbung problematisch?"

Gibt es Probleme, die mit Mehrfachvererbung eleganter als mit Einfachvererbung gelöst werden können?

Von 35 Befragten:

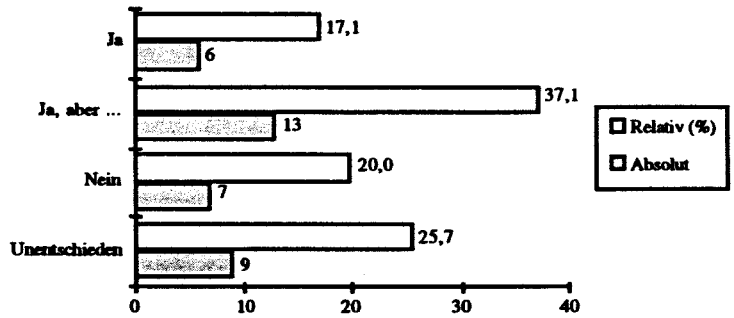

Abb. 6: "Probleme eleganter lösbar mit Mehrfachvererbung?"

Besteht die Gefahr, durch Mehrfachvererbung zu schlechtem Design verleitet zu werden?

Von 35 Befragten:

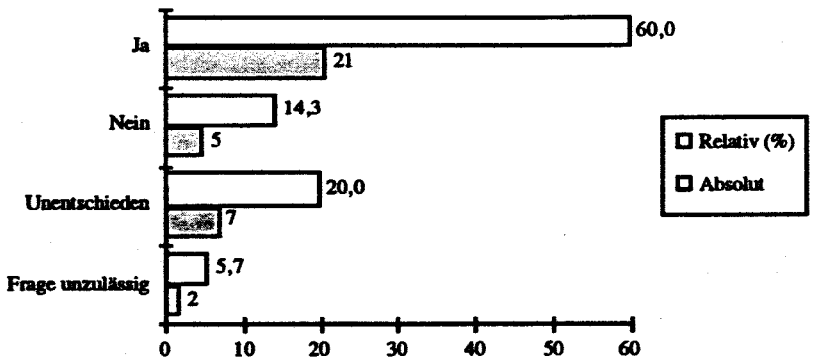

Abb. 7: "Verleitet Mehrfachvererbung zu schlechtem Design?"

> Im folgenden werden im Rahmen der qualitativen Auswertung die Pro- und Contra-Argumente bezüglich Mehrfachvererbung (*Multiple Inheritance*, kurz *MI*) gegenübergestellt. Zuvor jedoch werden allgemein gehaltene Aussagen angeführt.

Allgemeine Aussagen zur MI:

◊ *Verschiedene Sichten und deren Vermischung*
Bei der Betrachtung einer MI-Anwendung muß man von zwei Perspektiven ausgehen, der *Ausdrucks-* und der *Code-Ebene*. Bei ersterer geht es um die Formulierung sinnvoller Semantik (Modellierung), d.h. die MI-Konstruktion soll nicht nur um ihrer selbst erstellt werden, sondern eine "Vererbung mit Sinn" repräsentieren. Hier hat sich die MI aber eher als Hemmschuh herausgestellt.

Die Anwendung der MI auf der Code-Ebene an sich (Implementierung, losgelöst von der Semantik betrachtet) bringt meist keine direkten Formulierungsprobleme mit sich. Sie umfaßt den *Spezialisierungs-* und den *Erweiterungsaspekt.* Der *Spezialisierungsaspekt* reflektiert die Sicht der Typenvererbung, bei der es darum geht, ausgehend von einem Basistyp durch fortgesetzte Untertypenbildung immer spezialisiertere Typen zu definieren. Beim *Erweiterungsaspekt* geht es lediglich darum, neue Eigenschaften (z.B. Instanzvariablen, Methoden) in eine Klasse zu packen. Die fortgesetzte Anwendung wird in einer schwer durchschaubaren Hierarchie resultieren; alternative Ansätze (z.B. *Delegation*[5]) sind zu prüfen.

Von einer disziplinierten Anwendung der MI kann dann gesprochen werden, wenn eine MI-Klasse eine semantische Beziehung im Sinne der Datenmodellierung ausdrückt, also die Umsetzung eines Modellierungsergebnisses repräsentiert. Viele MI-Anwendungen haben aber keine klare Semantik, es geht oftmals nur darum, verschiedenartig im Vererbungsbaum gelagerte Klassen bequem an einer Stelle verarbeiten zu können.

Pro MI:

◊ *Voraussetzung für den nutzbringenden Einsatz von MI*
MI ist ein mächtiges Konzept, das unter der Voraussetzung, daß der Entwickler über einen großen Erfahrungsschatz verfügt und äußerst diszipliniert implementiert, nutzbringend ist. Dem Entwickler muß bewußt sein, daß die Eleganz einer MI-Lösung meist eine höhere strukturelle Komplexität mit sich bringt.

Ist diese Voraussetzung erfüllt, bietet MI mehr Vor- als Nachteile, besonders dann, wenn eine 1:1 Abbildung aus der Analyse wichtig ist und dort mit MI-Prinzipien gearbeitet wurde. MI liefert zwar nicht notwendigerweise die elegantere Lösung, aber die logischere in Verbindung mit dem "geistigen Modell", wenn dieses Modell mit Einfachvererbung *(Single Inheritance*, kurz *SI)* nicht natürlich ausgedrückt werden kann.

◊ *Anwendungsbeispiele*
Plausible MI-Anwendung sind selten, aber sie existieren:

---

5 Gamma [GAMM91] nennt als Charakteristika einer *Delegationsklasse*: Sie ist meist Wurzelklasse einer Klassenfamilie und ermöglicht es ihren Erben, deren Objekte untereinander zu verketten und Methodenaufrufe entlang einer solchen Kette weiterzuleiten.

- *Verbinden von Klassenbibliotheken:* Steht man vor der Aufgabe, zwei oder mehr KBs (z.B. eine für grafische Benutzeroberflächen und eine für Datenbankanschluß) verbinden zu müssen, stellt MI meist den einzig gangbaren (wenn auch schwierigen) Weg dar.
- *Unterstützung mehrfacher Sichtweisen:* Wenn man mit grafischen Objekten arbeitet, ist es oft notwendig, unterschiedliche Sichten auf diese zu unterstützen, z.B. den Standpunkt der *Farbhierarchien* versus den der *Verarbeitungshierarchien.* Obwohl solche Problemstellungen auch mit SI gelöst werden können, gibt es Situationen, in denen MI-Konstruktionen die elegantere Entwurfsentscheidung repräsentieren.
- *Unterstützung portabler Anwendungen:* Eine Hardware-unabhängige KB beispielsweise verfügt über Protokolle, die Hardware-Abhängigkeiten kapseln. Diese werden über MI mit Hardware-unabhängigen Klassen kombiniert, um eine Hardware-spezifische Version zu erzeugen. Dadurch entfällt die Notwendigkeit, mehrere, sich nur marginal in wenigen Hardware-Spezifika unterscheidende Versionen der KB zu verwalten.

Contra MI:

◊ *Komplexität*
Auf den ersten Blick erscheint MI als sehr nützlich, da man unbegrenzte Kombinationsmöglichkeiten hat. MI ist aber nur syntaktisch einfach, semantisch ist sie sehr komplex. Eine KB mit Mehrfachvererbung repräsentiert eine Netzstruktur, die bei weitem komplexer zu verstehen ist als eine Baumstruktur, die sich durch Einfachvererbung ergibt. Mehrfachvererbung multipliziert die (schon nicht-triviale) Komplexität der Einfachvererbung. Je mehr eine KB einem regulären Baum ähnelt, desto weniger Probleme sind zu erwarten.

Wenn MI sehr häufig eingesetzt wird, entsteht einerseits eine Reihe neuer, schwieriger Entwurfsprobleme, andererseits wird Wiederverwendung und Wartung schwieriger, da man mehr tiefer gehende Kenntnisse über das Klassennetzwerk als Ganzes benötigt. Nahezu unmöglich ist das Nachvollziehen von "Verhaltenspfaden" in AFs, die sich primär auf MI stützen. Es ist äußerst schwierig, eine Unterklasse korrekt zu verwenden, die dieselben Eigenschaften mehrfach auf unterschiedlichen Pfaden von einer Basisklasse erbt. Ob die Vorteile der MI die zusätzliche Komplexität (neue Semantik und Regeln) rechtfertigen, ist fraglich.

◊ *Wenig plausible Anwendungsfälle*
Die meisten in der Literatur angegebenen Beispiele für MI sind primitiv, falsch oder weit hergeholt, z.B. eine Klasse *"stringList"*, die von den Klassen *"string"* und *"list"* erbt. Diese Beispiele basieren meist auf disjunkten Mengen, die mit MI einfach zu realisieren sind, aber von MI nicht wirklich profitieren. Andere Beispiele, die nicht auf disjunkten Mengen basieren, sind schon schwieriger mit MI zu realisieren, aber immer noch primitiv. MI-Hierarchien sind einfacher als SI-Hierarchien zu bauen, in vielen Fällen kann man aber eine Vermischung von *"haben-"* und *"sein-"* (*"Has-a"* und *"Is-a"*) Relationen beobachten. Dies weist auf einen Mangel an Problemverständnis hin.

◊ *Leichtfertige Anwendung*
Wenn man sich auf SI beschränkt, ist man gezwungen, ein gegenständliches Problem gründlicher zu durchdenken, da man eine zentrale Klasse finden muß, von der man ausgeht. Dieser längere Nachdenkprozeß wird in der Regel zu einem besseren Design führen als eine verlockende, weil "kleine und schnelle MI-Lösung", die ungeahnte negative Auswirkungen mit sich bringen kann. Die Erkenntnis, daß man mit MI neue Sachverhalte einfacher in eine

KB einbauen kann als mit SI, darf nicht das Entscheidungskriterium sein. Durch die sorglose Anwendung von MI erzeugt man z.B. das Problem, immer umfangreichere Klassen zu erhalten, die immer mehr überflüssige (weil nicht benötigte) Eigenschaften enthalten.

Eines der wenigen Beispiele, in denen die Anwendung der MI das Verständnis erleichtert, ist der Spezialfall des einfachen Mischens von Protokollen, die durch Klassen repräsentiert werden, die in der Vererbungsstruktur weit voneinander entfernt liegen.

◊ *Konfliktlösung*
Konflikte bei wiederholter bzw. mehrdeutiger Vererbung müssen manuell behoben werden, es gibt keinen Automatismus. Die Mechanismen zur Konfliktlösung stellen naturgemäß eine weitere Dimension der Komplexität dar. Die in den heutigen Sprachen angebotenen Mechanismen sind aber meist nicht transparent genug.

◊ *Eingeschränkte Flexibilität*
Meist ist eine MI-Lösung zu statisch: Es ist ein statischer Prozeß, verschiedene Protokolle zu kombinieren. Die Flexibilität ist beschränkt, viele Sachverhalte müssen – im Gegensatz zu einer SI-Lösung mit Wrapping Konzept – zur Übersetzungszeit bekannt sein.

Welche OOP-spezifischen Probleme bezüglich Lesbarkeit von Programmen sehen Sie?

◊ *Überblicken der Schnittstellen*
Grundsätzlich muß gesagt werden, daß OOP gute Lesbarkeit unterstützt, wenn sie korrekt angewendet wird, was aber eine Binsenweisheit darstellt. In keinem Fall ist es trivial, die Klassenschnittstellen einer KB zu überblicken, auch wenn entsprechende Browsing-Werkzeuge vorhanden sind. Dies ist teilweise im objektorientierten Paradigma selbst begründet (Typhierarchien und dynamische Bindung), teilweise in den Sprachmechanismen, die diese Konzepte umsetzen.

Das Hauptproblem ist die versteckte Koppelung zwischen Klassen. Man unterscheidet *Vererbungskoppelung* und *Benutzungskoppelung*. Letztere drückt aus, daß eine Klasse etwas einer anderen Klasse benutzt. Schnittstellen sind Schlüssel zu Mengen von Operationen. Um die Übersichtlichkeit zu erhöhen, sollte es möglich sein, Schnittstellen auf mehreren Ebenen zu definieren, z.B. eine Ebene, die das gesamte Angebot der Schnittstelle umfaßt, und eine andere Ebene, die (auf Methodenebene) nur das zeigt, was man zur korrekten Verwendung benötigt. Ein solcher Mechanismus für das Explizitmachen von Koppelungen würde die Lesbarkeit in bezug auf die Übersichtlichkeit von Schnittstellen erhöhen.

◊ *Verstreutheit des Codes*
Rein sequentielles Lesen von OOP-Code ist nicht sinnvoll und auch nicht möglich, da der auszuführende Code hochgradig verteilt vorliegt. Viele verschiedene Informationsquellen müssen in die Betrachtung einbezogen werden, wenn z.B. ein Fehler in einem Stück Code aufgespürt werden soll (z.B. ein Objekt erhält eine Nachricht, für die es keine zugehörige Methode hat). Trotz entsprechender Werkzeuge ist es nicht einfach, die Anzahl an (meist kleinen) Methoden zu überblicken, die (z.B. im Falle von C++) über Dutzende von Dateien verstreut sein können.

◊ *Dynamische Bindung und Polymorphismus*
Programme müssen in einem viel größeren Kontext als in der traditionellen Software-Entwicklung betrachtet werden, da durch die dynamische Bindung eine neue Art des Kontrollflusses entsteht. Einfaches statisches Nachvollziehen ist nicht möglich. Durch Polymorphismus ist die Funktionalität eines Codestückes nicht unmittelbar erkennbar, besonders in Sprachen ohne strenge Typenprüfung.

◊ *Statische Hierarchie versus dynamischer Laufzeit-Graph*
Die Schwierigkeiten, den Kontrollfluß zu überblicken, liegen im Paradigma begründet. Statisch gesehen geht es darum, einen Vererbungsbaum (im Falle der Einfachvererbung) zu verstehen. Gelangt ein Programm zur Ausführung, entsteht ein Graph. Diese "Kluft" ist der Grund für Probleme in der Lesbarkeit. Einerseits wird durch die Benutzung einer Klassenhierarchie die Lesbarkeit erhöht, da es sich dabei um die Anwendung eines Strukturierungselements handelt, andererseits hat ein Entwickler davon nur dann einen Nutzen, wenn er sich den zugehörigen Laufzeit-Graphen vergegenwärtigen kann. Viele mögliche Wege müssen verfolgt werden, um die zur Verfügung gestellte Funktionalität erfassen zu können.

◊ *Werkzeuge*

- Navigations-Werkzeuge sind nur dann von Vorteil, wenn sie - im Gegensatz zur traditionellen Systementwicklung - die strukturelle Komponente stärker als die algorithmische berücksichtigen. Werkzeuge müssen zwei "Code-Richtungen" abdecken: Vom Standpunkt der Datenabstraktion und -kapselung und von dem der Vererbung, die Code vertikal dazu existieren läßt.
- Parallel dazu müssen Werkzeuge vorhanden sein, die die Dateiverwaltung übernehmen, existiert doch in vielen objektorientierten Systemen eine große Anzahl relativ kleiner Dateien, auf die die Gesamtfunktionalität verteilt ist. Die Abbildung einer Projektstruktur auf Dateien ist wesentlich aufwendiger als bei herkömmlichen Systemen.

◊ *Programmierkonventionen*
Das Einhalten von Programmierkonventionen ist ein noch stärkerer Erfolgsfaktor für die Lesbarkeit als in herkömmlichen Systemen. Neben den (sehr wichtigen) Konventionen für die Namensgebung sollten Konventionen für Verhaltensmuster als Reaktion auf bestimmte Ereignisse definiert werden.

Welche OOP-spezifischen Anforderungen an die Code-Dokumentation sehen Sie?

◊ *Nicht-lineare Dokumentation*
Dokumentation von objektorientiertem Code ist ebenso wichtig wie auch in konventionellen Systemen, aber sie ist unterschiedlich. Sie sollte nicht nur Sachverhalte erklären, sondern auch die Struktur des gegenständlichen objektorientierten Systems (z.B. AF) widerspiegeln und unterstützen, z.B. durch verschachtelte Dokumentation, automatisch erzeugte Verbindungen und Querverweise, die durch eine On-line Hypertext-Komponente aufbereitet werden. Lineare Dokumentation ist nicht ausreichend, da der Kontext eines interessierenden Bereiches größer ist und sich häufiger ändert als in konventionellen Systemen.

◊ *Dokumentation von Klassen*
Klassen sind schwieriger zu beschreiben als Module, die in sich beschreibbar sind. Wenn ein Modul (oder eine Funktion) zu groß ist, wird es einfach weiter zerlegt. In der OOP ist es schwierig, die verschiedenen gültigen Sichten für eine (große) Klasse zu beschreiben, besonders in Sprachen ohne strenge Typenprüfung. Bei diesen ist es wichtig, Typeninformationen in Kommentaren oder als Teil des Namens zu notieren. In Smalltalk beispielsweise sind die *"Categories"* in Verbindung mit entsprechenden Konventionen hilfreich für diesen Zweck.

Auch auf Methodenebene ist die Dokumentation schwieriger, da Methoden in der Regel wesentlich kürzer als konventionelle Prozeduren sind und meist keine in sich geschlossene

Aufgabe erfüllen. Wichtig vielmehr ist die Beschreibung der Beziehungen zwischen Klassen und deren Methoden. Die zentrale Frage dabei: Wo soll diese Information plaziert werden? Jede Klasse sollte auch mit einer Dokumentation der Entwurfsentscheidungen versehen werden, um Wiederverwendung besser unterstützen zu können[6]. Dafür sind Documentation-Retrieval-Systeme notwendig, die das "Anbringen" und Wiederfinden von Objekt-Dokumentation unterstützen.

◊ *Dokumentation von KBs*
KBs (insbesondere AFs) sollten in einem kochbuchartigen Stil dokumentiert werden, d.h. Beispiele, die rezeptartig die korrekte Verwendung der KB zeigen, stehen im Mittelpunkt eines "Cookbooks". Ein Cookbook soll den Zweck einer oder mehrerer Klassen erklären, z.B. "wird nur aufgerufen", "muß überschrieben werden", "wird nur in speziellen Situationen wie ... überschrieben" usw. Die wichtigste Aufgabe dabei ist die Beschreibung der großen, übergeordneten Beziehungen innerhalb der KB.

Keinesfalls sollte sich die Dokumentation in einer Erläuterung einer Klasse nach der anderen erschöpfen. Die Dokumentation der Instanzvariablen, Methoden und der Vererbungshierarchie ist zwar wichtig, kann aber nur die Rolle einer ergänzenden Abrundung spielen. Es ist bei weitem wichtiger, die zugrundeliegenden Konzepte (z.B. Event-Handling, Refresh-Mechanismen, Undo-Mechanismen, Change-Propagation u.dgl.) durch Beispiele aufzuzeigen, die das Zusammenspiel mehrerer Klassen zeigen ("Programming by Contract").

◊ *Dokumentation von Applikationen*
Neben den problemspezifischen Teilen muß die softwaretechnische Dokumentation von objektorientiert implementierten Applikationen mindestens folgende Bereiche erklären:

- Verhalten und Interaktion der beteiligten Klassen
- Beschreibung des Hauptkontrollflusses
- Möglichkeiten von Erweiterungen durch Aufzeigen des zugrundeliegenden Klientenmodells, sodaß die Notwendigkeit für "Try and Error" auf ein Minimum reduziert wird.

  Folgende Fragen müssen in diesem Zusammenhang beantwortet werden:
  - Welche Methoden müssen oder dürfen auf welche Art und Weise überschrieben werden?
  - Welche Methoden dürfen nicht überschrieben werden?
  - Welche Methoden müssen in welcher Reihenfolge zur Ausführung gelangen, damit Ziel x erreicht wird?

Welche Werkzeuge sollten in einer OOP-Umgebung enthalten sein, die speziell das Arbeiten mit KBs/AFs unterstützen?

◊ *Kern der Entwicklungsumgebung*

---

[6] In diesem Zusammenhang schlägt Gamma [GAMM91] vor, daß erfahrene Designer versuchen sollten, Design-Strukturen zu Design-Mustern zu abstrahieren. Auf diese Weise entwickelt sich im Laufe der Zeit eine Reihe von Design-Mustern *("Design Patterns")*, die anderen Entwicklern zur Verfügung gestellt werden können, z.B. im Rahmen der Dokumentation. Gamma stellt das Werkzeug *ET++DE (ET++ Design Environment)* vor, bei dem es sich um einen um Design-Annotationen erweiterten Klassenbrowser handelt. Eine Design-Annotation speichert Design-Information des Klassenproduzenten für die späteren Klienten einer Klasse. Ein ET++DE Benutzer kann dann auf die zur jeweiligen Selektion im Klassen- oder Quellcode-Browser gehörigen Design-Annotationen zugreifen.

Die Entwicklungsumgebung sollte (die im folgenden genannten) Werkzeuge nahtlos integrieren, wobei hier die Smalltalk-Umgebung *(Workspace)* als richtungsweisender Pionier betrachtet werden kann. Das Herzstück sollte ein OOPL-Interpreter mit intelligentem, inkrementellem Linker bilden, um kurze Turn-Around-Zeiten zu erreichen und langwierige Compile-Link-Go Zyklen zu vermeiden. Dies ist ein kritischer Faktor, da experimentelle Programmierung nur bei kurzen Turn-Around-Zeiten sinnvoll möglich ist.

◊ *Browser und Inspektor (Inspector)*
Das wichtigste Werkzeug ist ein grafischer, mit Hypertext-Eigenschaften versehener (Klassen-) Browser als allgemeines, zentrales Navigationswerkzeug, wobei unterschiedliche Granularitäten unterstützt werden sollten (von der Ebene der Instanzvariablen bis hinauf zur KB als Ganzes). Der (oder die) Browser sollte(n) das Verfolgen einer Vielzahl von Querverbindungen unterstützen und auf diese Art unterschiedliche Sichten auf die KB ermöglichen. Folgende Merkmale sind dabei von zentralem Interesse, die das Einarbeiten in bzw. das Erweitern und Warten von KBs unterstützen:

- Aufzeigen der hierarchischen Beziehungen
- Zugriff auf
  - die (verteilte) Schnittstelle einer Klasse mit allen geerbten Eigenschaften ("flache Sicht" einer Klasse)
  - kontextsensitive Erklärungskomponenten
  - die Hooks (selektiv mit Ausblendung von im Moment nicht relevanten Details)
  - Objekt-Meta-Information zur Laufzeit (z.B. Abfrage der Klasse eines Objekts, Feststellen, welches Objekt eine bestimmte Nachricht gesendet hat u.dgl.)
  - alle statisch ermittelbaren Informationen (z.B. Metriken)
- Beantwortung von Fragen wie z.B.
  - "Was ist die Funktionalität dieser Klasse?"
  - "Was sind die typischen Anwendungsbereiche dieser Klasse?"
  - "Wer erbt was von wem?"
  - "Wo wird ... definiert und verwendet?"
  - "Wo wird ... überschrieben?"
  - "Welche Beziehungen existieren?"
  - "Was muß getan werden, um diese Klasse korrekt zu verwenden?"
  - "Wer implementiert die abstrakte Klasse ...?"

  usw.
- Auffinden von Information in einem speziellen Kontext (z.B. Suche einer bestimmten Instanzvariablen und Aufzeigen der Zeile oder des Source-Code-Blocks, wo sie verwendet wird)
- Unterstützung für komfortables Quellcode-Lesen (z.B. Betrachten von Methoden)

Ein den Browser ideal ergänzender Inspektor[7] sollte nicht nur auf ein einzelnes Objekt, sondern auf mehrere, zusammenhängende Objekte angewendet werden können.

◊ *Help-System*
Ein Help-System sollte on-line auf der Ebene von Klassen verfügbar sein und über Wissen aus repräsentativen Beispielen verfügen. Die Grenze zur eigentlichen Dokumentation ist dabei fließend. Um die Navigation möglichst komfortabel und effizient zu gestalten, sollten Hypertext-Werkzeuge Verwendung finden, die ihrerseits hierarchisch gegliedert und schrittweise von "Erfahrung" zu "Referenz" strukturiert sind. Zur Laufzeit wären etwa "Manual Pages" für Objekte, die alles Wissenswerte zugänglich machen, äußerst hilfreich.

◊ *Debugging und Instrumentierung*
Ein dynamischer, möglichst hoch angesiedelter Quellcode-Debugger sollte während der Laufzeit und post mortem verfügbar sein und die Option der Einzelschrittausführung auf Methodenebene anbieten, um die zwischen Objekten bestehenden Protokolle komfortabel kennenlernen zu können.

◊ *Prototyping-Werkzeuge*
Um den Prozeß der inkrementellen Software-Entwicklung, wie er für die objektorientierte Systementwicklung typisch ist, über die eigentliche Umgebung hinaus zu unterstützen, sollten Prototyping-Werkzeuge verfügbar sein, und zwar auf unterschiedlichen Ebenen. Auf höchster Ebene könnte beispielsweise eine visuelle Komponente für das "Application-Painting" verwendet werden.

◊ *Source-Code-Management Komponente*
Eine Komponente für die Verwaltung des erzeugten Source-Codes muß die Besonderheiten objektorientierter Systeme berücksichtigen, z.B. hoher "Verstreutheitsgrad" des Codes. Dabei sollten nicht nur rein administrative Tätigkeiten wahrgenommen werden, sondern beispielsweise auch ein Klassen-Archivierungs- und -Retrieval-Werkzeug enthalten sein, das bei Bedarf mehrere mögliche Klassen mit zugehöriger Erklärung für die Lösung eines gegenständlichen Problems anbietet.

Welche Möglichkeiten sehen Sie, schlechtes objektorientiertes Design zu verhindern?

◊ *Talent, gutes Training und Erfahrung*
Die drei (paradigmenunabhängigen) Säulen für gutes Design sind Talent, gutes Training und Erfahrung. Technisches Know-How alleine reicht nicht, Intuition und Offenheit für neue Ideen sind nötig. Gutes Design kann im vorhinein nicht geplant werden, da es ein kreativer Prozeß ist. Ein Benutzer kann aber durch Verwendung einer gut entworfenen KB/AF bis zu einem gewissen Grad gezwungen werden, seinerseits gut zu entwerfen.

◊ *Studieren von gutem Design*
Das Studieren von gutem Design kann ein wertvoller Katalysator sein. Subtile Probleme können naturgemäß nicht vorhergesehen werden, hier hilft nur ein *Review* (siehe später). Die Qualität der Einarbeitung in die Problemstellung und der Unterstützung des Einarbeitens in die verwendete KB/AF wird großen Einfluß auf die Qualität des entstehenden Designs haben.

---

[7] Ein Inspektor (Inspector) ermöglicht die "Inspektion" von Objekten zur Laufzeit, z.B. indem die Werte der Instanzvariablen eines Objekts angezeigt werden. Darüberhinaus wird die Navigation in den dynamisch existierenden Objektstrukturen ermöglicht.

◊ *Gute Richtlinien*
Krasse Design-Fehler können durch entsprechende Richtlinien vermieden werden, ähnlich zu denen der konventionellen, modularen Programmierung, z.B. "Wie sind Module zu entkoppeln?" oder "Was sollte in ein Modul aufgenommen werden?". Richtlinien können nicht nur für bestimmte KBs/AFs angegeben werden, sondern auch für bestimmte Problemklassen.

◊ *Design-Entscheidungen in Teams*
Wichtige Design-Entscheidungen sollten nicht alleine, sondern in Teams getroffen werden, die erfahrene Entwickler enthalten.

◊ *Reviews und Redesigns*
Design-Reviews sollten institutionalisiert und im Team durchgeführt werden, z.B. von einer unabhängigen Gruppe im Rahmen eines "Defect-Prevention"-Prozesses, der Diagnose und Behandlung umfaßt. Selbstkritik muß als essentiell erkannt werden. Als Ergebnis eines Reviews darf vor einem iterativem Prozeß des permanenten Redesigns und der Umstrukturierungen auf großer Basis und erneuten Reviews nicht zurückgeschreckt werden.

◊ *Training*
Folgende Bereiche sollten regelmäßig durch entsprechendes Training behandelt werden:

- Allgemeine Entwurfsprinzipien (z.B. Abstraktion, Generalisierung, Faktorisierung), um Flexibilität zu erreichen
- Bewährte Entwurfsphilospohien (z.B. Single Rooting, Narrow Inheritance Interface), deren Integration und Anwendung einiger weniger, mächtiger Konzepte
- Meisterung der Gratwanderung zwischen "Keep It Simple" und "Keep It General" und zwischen "Vermeide eine zu große Abstraktionslücke zwischen Unter- und Oberklasse" und "Vermeide eine zu große Hierarchietiefe"
- Verwendung von (indirekter) Rekursion, um Flexibilität zu erreichen und um zu spezialisierte "Endklassen" zu vermeiden.
- Gute und schlechte Code-Beispiele (Code-Verdopplung, degenerierte Hierarchien, zu viele Instanzvariablen, Unverständlichkeiten im allgemeinen)
- Unterschiede zu konventionellen Lösungen
- Unterschiede zwischen *"Has-a"* und *"Is-a"* Beziehungen
- Behutsame Verwendung spezieller Sprachkonzepte (z.B. Friend-Konzept in C++)

◊ *Anwendung des Vererbungsmechanismus*
Es muß klar gemacht werden, daß Vererbung ein Mechanismus ist, um Typattribute zu vererben. Eine *"Is-a"* Beziehung muß gegeben sein, z.B. ist ein Typ *Button* ein sinnvoller Untertyp des Typs *View*. Andererseits ist es zweifelhaft, ob ein Typ *RestrictedTextView* als Untertyp von *TextView* implementiert werden sollte. Unterklassenbildung sollte nicht angewendet werden, nur um einen anderen Typ zu erhalten. Dies wird zu einer großen Zahl von Unterklassen führen, die wiederum die (vermeintliche) Notwendigkeit für Mehrfachvererbung hervorruft. Vererbung ist kein Vehikel, um jedes Auftreten von Codeverdopplung zu vermeiden.

Welche Unterschiede sehen Sie im Management zwischen konventionellen und objektorientierten Projekten?

◊ *Phasenorientierung anders*
Traditionelle Life-Cycle-Modelle sind in der OOP nicht sinnvoll anwendbar. OOP bringt eine herkömmlich nicht realisierbare Flexibilität in der Entwicklung mit sich. Der Prozeß der objektorientierten Software-Entwicklung umfaßt ebenfalls Phasen, ist aber mehr durch Phasen der Wiederverwendung und des (Rapid) Prototypings charakterisiert, was ein inkrementelles Arbeitsmodell mit mehr Iterationen als in der konventionellen Entwicklung mit sich bringt.

◊ *Prototyping stärker im Vordergrund*
In der OOP dominieren die Redesign-Aktivitäten, nicht die Design-Aktivitäten. Das kann charakterisiert werden durch *"Analyze a little, design a little, code a little, test a little, and again from the beginning"*. Die Phasen werden kürzer, überlappen sich stark und etablieren dadurch kleine Zyklen. Die Turn-Around-Zeiten müssen naturgemäß sehr kurz sein. Prototyping-Ansätze passen daher sehr gut zur objektorientierten Software-Entwicklung.

◊ *Design unterschiedlich*
Die Design-Phase ist fundamental unterschiedlich, da die verfügbaren Software-Bausteine im Mittelpunkt stehen, wodurch ein eher bottom-up orientierter Entwicklungsansatz entsteht. Wenn keine KB/AF vorhanden ist und von Null begonnen werden muß, ist der Aufwand höher als bei der konventionellen Systementwicklung, da die allgemeine Basis für eine gesamte Problemklasse in Form einer KB/AF erstellt werden muß (bzw. sollte).

◊ *Implementierung kürzer*
Die Implementierungsphase ist viel kürzer als in der konventionellen Programmierung, wenn gebrauchsfertige Bausteine verwendet werden können. Diese Phase ist durch das Ziel gekennzeichnet, die vorhandenen Bausteine korrekt und effizient wiederzuverwenden. Die Entwicklungsfortschrittskurve ist nicht linear: sie startet flach mit einem in der Folge schnellen Anstieg. Design und Implementierung wachsen zusammen, die Grenzen sind fließend. Gute OO-Designer sind in der Regel gute OO-Programmierer und umgekehrt.

◊ *Redesign häufiger und einfacher*
In der OOP tritt Redesign häufiger auf als in der konventionellen Programmierung, in der Schnittstellen im Vergleich relativ schnell definiert werden. Andererseits ist Redesign in der OOP einfacher durchzuführen als in der konventionellen Programmierung. Modifikationen können einfacher durchgeführt und wieder, falls nötig, entfernt werden. Auf diese Art wird experimentelles Vorgehen unterstützt. Einige klassische Entwurfsaktivitäten im Rahmen der Applikationsentwicklung verschwinden, da viele Teile bereits im AF vorliegen (z.B. die Grundstruktur des Aufbaus und des Ablaufes einer Applikation).

◊ *Wartung: einfacher*
Wartungsaktivitäten im klassischen Verständnis (im Gegensatz zum erwähnten Redesign) sind einfacher durchzuführen, da alle auf einem AF basierenden Applikationen die gleiche Grundstruktur aufweisen. Dies resultiert in vereinfachter Adaptierung und Erweiterung, was einen zentralen Vorteil der OOP darstellt.

◊ *Wartung: Erweiterungen leichter*
Zukünftige Erweiterungen im Rahmen der Wartung sind nie exakt vorherzusehen. Die OOP bietet hier den flexibleren Ansatz, indem der Entwickler viele Sachverhalte für eine spätere Modifikation offen lassen kann, ohne daß die Zielapplikation darunter leidet. Die Modifikati-

onen können dann auch von einem anderen Autor ohne große Reibungsverluste vorgenommen werden.

Wartungsarbeiten in der konventionellen Entwicklung umfassen mehr Tätigkeiten im Hinblick auf die Stabilisierung des Systems (Beseitigung von Fehlern) als dies in der OOP der Fall ist. Dort wird durch häufiges Wiederverwenden eine zugrundeliegende KB/AF schnell reifen, was sich positiv auf die Stabilität der davon abgeleiteten Applikations-Software auswirkt. Die Qualität der KB/AF vererbt sich als Basisqualität in die Zielanwendungen.

◊ *Dokumentenstruktur durchgängig*
Ein Unterschied zur herkömmlichen Programmierung ist die Möglichkeit einer konsistenten, objektorientierten Struktur durch alle Dokumente, die während eines Projektes erzeugt werden, von Analyse- über Design- zu Code-Dokumenten.

◊ *Archivierung komplexer*
Ein weiterer Unterschied besteht in der Archivierung aller entstehenden Dokumente, vor allem der Code-Dateien. Die Gesamtfunktionalität ist (meist) über wesentlich mehr Dateien verstreut als in der konventionellen Programmierung, wodurch sich bei der OOP die Verwaltung und Zugriffskontrolle der beteiligten Dateien komplexer gestaltet.

◊ *Objektsystem- und Applikationsentwicklung verflochten*
OOP-Projektmanagement umfaßt zwei Bereiche: Erstens die Wartung des Objektsystems (z.B. KB/AF) und zweitens die Entwicklung und Wartung von Zielapplikationen. KB/AF- und Applikationsentwickler müssen miteinander in engem Kontakt stehen, beide Seiten profitieren von Hinweisen der jeweils anderen Gruppe. Es gibt mehr Abhängigkeiten zwischen den Teams bzw. Teilen eines Projektes als in der konventionellen Programmierung und weniger voneinander unabhängige Teile. In der konventionellen Programmierung werden nur die Schnittstellen und die Funktionalität beschrieben, in der OOP gibt es darüberhinausgehende Verflechtungen.

◊ *Auswirkungen auf die arbeitsteilige Software-Entwicklung*
Die erwähnten Sachverhalte haben starke Auswirkungen auf die Arbeitsteilung. Es gibt noch wenig Erfahrung und Forschungsergebnisse über Entkopplung, um die arbeitsteilige Entwicklung aktiv zu unterstützen. Ziel ist immer die Konstruktion einer Klassenhierarchie, was viele Vereinbarungen und bedeutende Informationsflüsse zwischen den Teams verlangt, und zwar in einem stärkeren Ausmaß, als dies in der konventionellen Programmierung der Fall ist. Übereinkünfte über Schnittstellen sind schwieriger zu erreichen, da mit der Hierarchie eine zusätzliche Dimension berücksichtigt werden muß.

Andererseits unterstützt die OOP Datenkapselung besser als die konventionelle (auch modulorientierte) Programmierung. Auf diese Art wird die "Voraus-Konstruktion" unabhängiger Objekte möglich. Es muß eine eigene Phase und eigenes Personal dafür geben, neu erzeugte Klassen auf ihre allgemeine Anwendbarkeit im Hinblick auf eine Integration in die KB/AF zu überprüfen und entsprechend zu archivieren.

Es sollte ein neues Anreiz/Beitragssystem geben, sodaß nicht nur spezifische, kurz- und mittelfristige Projektziele, sondern auch langfristige Ziele im Sinne der Weiterentwicklung der KB/AF aktiv verfolgt werden. Projekte kommen und gehen, KBs/AFs leben meist für eine längere Zeit, weshalb entsprechende Wartung essentiell ist. Auf lange Sicht ist die KB/AF die Quelle der Wertschöpfung. Rein projektorientiertes Denken muß durch unternehmensorientiertes Denken abgelöst werden. Software-Wiederverwendung muß im Vordergrund stehen und als Hauptziel erkannt werden.

◊ *Auswirkungen auf die Kostenrechnung*

Die Schlüsselfrage ist: "Was ist der monetäre Wert einer Klasse?". Die Auswirkung auf die Kostenrechnung muß berücksichtigt werden, da klassische Kostenrechnung nicht mehr anwendbar ist. Aus all diesen Gründen sind Projektmanager traditioneller Prägung "unglücklich", da die klassische Projektplanung hinfällig ist, die Planungsaufgabe wird schwieriger.

Ist objektorientierte Software anders zu testen als konventionelle?

◊ *Allgemeine Unterschiede zwischen konventionellem und objektorientiertem Testen*
In der konventionellen Programmierung wird durch Testen überprüft, ob die Zielfunktionalität ohne Fehler erreicht wurde. In der modularen Programmierung ist es relativ einfach, (Schnittstellen-) Tests mit einer hohen Granularität durchzuführen. In der OOP können zwei Fälle unterschieden werden: Erstens das Testen von Klassen/Objekten und KBs/AFs als solche und zweitens deren Verwendung im Rahmen einer Endbenutzer-Anwendung.

◊ *Charakteristika objektorientierten Testens*
Es wird viele gleiche Testmethoden in der objektorientierten und in der konventionellen Programmierung geben, aber die alles übergreifenden Teststrategien sind unterschiedlich. In der OOP sind ausführbare Ergebnisse (z.B. Prototypen) in der Regel früher als in der konventionellen Programmierung verfügbar, speziell wenn interaktive, interpretative Programmierumgebungen verwendet werden, die die inkrementelle Software-Entwicklung unterstützen.

Wegen dieses explorativen Charakters der OOP treten Testprozesse früher und häufiger auf (z.B. mit jeder Klassenveränderung), sind aber weniger intensiv und erstrecken sich im Normalfall auf kleinere Einheiten als in der konventionellen Programmierung, was aber nicht bedeutet, daß immer nur einzelne (kleine) Klassen für sich getestet werden (siehe nächster Punkt). Ein anderer Grund für die häufigeren Testaktivitäten liegt in der Natur des Paradigmas: Die Einführung neuer (abstrakter) Oberklassen verlangt unmittelbares Testen. Viele Testschritte können von den Abstraktionsschritten abgeleitet werden. Es gibt deshalb keine klaren Phasengrenzen zwischen Implementierung und Test, eine explizite Testphase gibt es nicht.

Andererseits werden Testprozesse auch kürzer, wenn eine qualitativ hochwertige, ausgereifte KB/AF verwendet wird, es sei denn, die zugrundeliegende Sprache unterstützt strenge Typenprüfung nicht (z.B. Smalltalk). In diesem Fall wird sich der Testprozeß verlängern, die Wahrscheinlichkeit, daß Fehler erst sehr spät auftreten (z.B. während des Einsatzes) und entdeckt werden, ist höher.

◊ *Testen der Beziehungen zwischen Klassen*
Vergleicht man beispielsweise Methoden und Prozeduren, so sind Methoden kompakter (kürzer) und die zugrundeliegenden Algorithmen weniger komplex. Fehler treten seltener in den Methoden als solchen auf, die Komplexität (und damit auch die Testkomplexität) verlagert sich auf die Kommunikation, die zwischen den Objekten abläuft. Testen einzelner Komponenten (z.B. einzelne Klassen) bzw. kleiner Teile, die aus ein paar Klassen bestehen, ist nur selten möglich. Normalerweise werden größere (in der Bedeutung "keine isolierte Klasse") Einheiten getestet, was kein Widerspruch dazu ist, daß Testaktivitäten früher und häufiger auftreten.

◊ *Testen des Kontrollflusses*
Im Falle von AFs muß ein ereignisorientierter Kontrollfluß getestet werden. Viel für das Testen notwendige Wissen ist zum Testzeitpunkt allerdings nicht verfügbar, deshalb werden viele Testaktivitäten zum Ausführungszeitpunkt hin verlagert. Einfache statische Code-Analysen werden weniger häufig auftreten. Es ist bei weitem komplexer, formalisierte Sche-

mata für den Test des Kontrollflusses in allen seinen möglichen Ausprägungen zu erstellen, da es wesentlich mehr "Einstiegspunkte" gibt als in herkömmlichen, nicht-objekt- und ereignisorientierten Systemen. Die dynamische Bindung erhöht die Komplexität des Testens zusätzlich. Es gibt neue Fehlerklassen und es ist schwieriger, einen bestimmten Testabdeckungsgrad zu erreichen.

Welche Qualitätssicherungsmaßnahmen sehen Sie?

◊ *Unterschiede zur konventionellen Programmierung*
Es gibt keine dramatischen Unterschiede zur konventionellen Programmierung, im wesentlichen gibt es die selben organisatorischen Notwendigkeiten, wie z.B. eine QA (Quality-Assurance) Strategie für Endbenutzer-Applikationen. Neben OOP-spezifischen Maßnahmen (Stabilisierung der KB/AF, Untersuchung von Endprodukten hinsichtlich allgemein wiederverwendbarer Klassen, Untersuchung hinsichtlich neuer möglicher Abstraktionen durch Faktorisierung u. dgl.) sind alle klassischen analytischen und konstruktiven Methoden relevant.

◊ *QA auf verschiedenen Ebenen*
Die Durchführung von QA-Methoden findet auf verschiedenen Ebenen statt. In der Spezifikationsphase unterstützt die Verwendung von Prototyping-Werkzeugen die Qualitätssicherung für die Spezifikationsresultate. Auf der Ebene des Entwurfes und der Implementierung sind Design- und Code-Reviews, die von erfahrenen Entwicklern geleitet werden, essentiell. Diese Sitzungen bringen auch einen Trainingseffekt für die Teilnehmer mit sich, indem gezeigt wird, wie das Ziel des Strebens nach mehr Allgemeinheit als notwendig und nach Effizienz erreicht werden kann, um langfristig den Erfolg zu sichern.

Aber gerade Code-Reviews sind wegen der übergreifenden Komplexität schwierig: Aufgrund der dynamischen Bindung kann durch Lesen von (statischem) Code nicht festgestellt werden, welche Methoden zur Ausführung gelangen. Im (theoretisch) schlechtesten Fall müssen alle kombinatorisch denkbaren Möglichkeiten statisch durchgespielt werden.

◊ *Wiederverwendung als Qualitätssicherungsmaßnahme*
Wiederverwendung an sich ist eine Qualitätssicherungsmaßnahme: Je öfter eine Komponente wiederverwendet wird, desto stabiler und ausgereifter wird diese werden, was sich wiederum positiv auf die Qualität der Endprodukte auswirkt, die auf diese Komponente zurückgreifen.

◊ *Beachtung von Konventionen*[8]
Die Einhaltung von Programmierrichtlinien muß von Beginn an sichergestellt werden, z.B. durch Überprüfung einzelner Komponenten in Code-Reviews oder automatisch durch entsprechende Werkzeuge. Beispiele dafür sind Style-Checker oder Werkzeuge, die die Größen von Methoden überprüfen (große Methoden sind Indikatoren für Design-Mängel). Richtlinien sollten beispielsweise Konventionen für folgende Bereiche umfassen:

- Namensgebung
- Optische Strukturierung des Codes
- Strukturierte Aufteilung von Klassen auf Dateien
- Header-Kommentare in Methoden
- Verwendung von Assertionen auf Design- und Code-Ebene, z.B. Formulierung von Vor- und Nachbedingungen, entweder unterstützt durch Sprachmerkmale (wie in

[8] Vgl. dazu den Absatz "Programmierkonventionen" der Frage *"Welche Probleme gibt es mit der (einfachen) Vererbung?"* sowie den gleichnamigen Absatz der Frage *"Welche OOP-spezifischen Probleme bezüglich Lesbarkeit von Programmen sehen Sie?"* .

Eiffel), durch die KB/AF, durch die Programmierumgebung oder im einfachsten Fall im Rahmen von Kommentaren (Assertionen sollen auch helfen, Entwurfsentscheidungen einfacher nachvollziehen zu können)

- Dokumentation
- Sprachspezifische Regelungen, unter welchen Bedingungen welche Sprachmerkmale verwendet werden dürfen, deren Anwendung gegen bestimmte Prinzipien verstößt (z.B. Verwendung des Friend-Konzepts in C++, um das Geheimnisprinzip zu durchbrechen)

◊ *Metriken für Klassen*
Metriken können definiert werden, um automatisch einen ersten Eindruck über die Reife von Klassen zu erlangen. Beispiele für Parameter, die in eine Reife-Metrik eingehen könnten, sind beispielsweise:

- Wie lange gibt es eine bestimmte Klasse bereits?
- Wie oft wurde eine bestimmte Klasse (durch Instanzierung und Unterklassenbildung) verwendet?
- Wie viele Fehler wurden entdeckt und behoben?

Welche objektorientierten Programmiersprachen werden in der Zukunft von Bedeutung sein?

> Bei dieser Frage wurde den Interview-Partnern bewußt keine Liste mit Sprachen zur Auswahl vorgelegt, vielmehr sollten sie ihr eigenes "Bewußtsein" über vorhandene Sprachen unbeeinflußt reflektieren. Von der Tatsache, daß einige "Sprachen" im Gegensatz zu anderen auch vollständige Programmierumgebungen umfassen (z.B. Smalltalk) wurde abstrahiert, d.h. bei der Beurteilung gingen die Befragten davon aus, daß es für jede Sprache eine entsprechend komfortable Umgebung gibt.

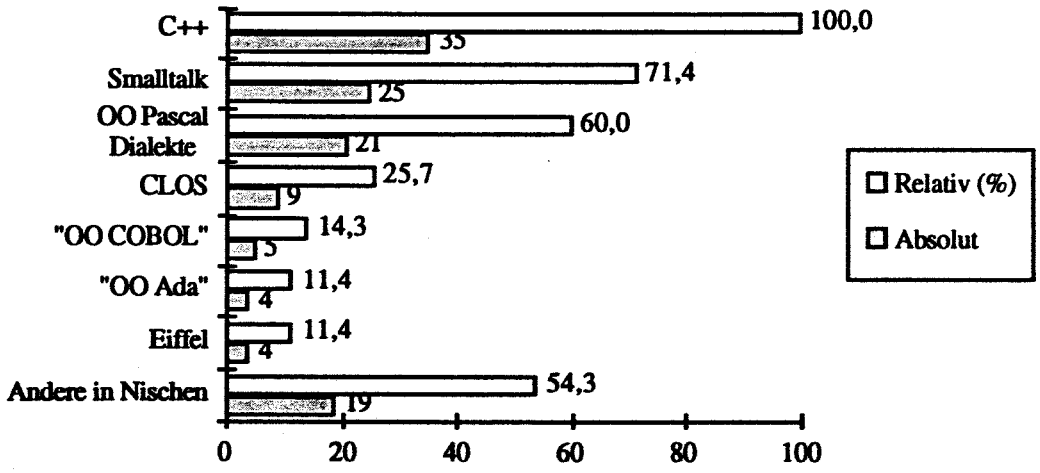

Abb. 8: "Welche OOPLs werden in der Zukunft von Bedeutung sein?"
(mehr als eine Nennung pro Person möglich)

Abbildung 8 zeigt das Ergebnis, wobei nur Sprachen aufgenommen wurden, *die von mindestens 10% der Befragten genannt wurden.* Der Eintrag "Andere in Nischen" stellt keine vom Autor eingeführte Nennung dar, die die unter 10% liegenden Sprachen erfaßt, sondern zeigt, wie oft tatsächlich eine

Nennung der Art *"Andere, von mir nicht genannte, aber heute bedeutende Sprachen werden Chancen in Nischenbereichen haben"* erfolgte.

Die die Nennungen begleitenden Kommentare werden zusammengefaßt für die drei Sprachen mit den meisten Nennungen wiedergegeben.

◊ *Begleitende Kommentare der Befragten für C++:*

- "Nur de-facto Standards von Großherstellern wie C++ werden für den großen industriellen Markt eine wirkliche Bedeutung erlangen."
- "Investitionsschutz wird höher bewertet als Innovation oder Qualität der Sprache selbst."
- "C++ baut auf einer weitverbreiteten Sprache auf und hat das Potential einer 'Sprache für alle'."

◊ *Begleitende Kommentare für Smalltalk:*

- "Smalltalk ist bei bewußter Auswahl schon aus historischen Gründen die erste Wahl."
- "Große Anwender werden es sich nicht leisten können, über kein Smalltalk Know-How zu verfügen."

◊ *Begleitender Kommentar für objektorientierte Pascal-Dialekte:*

- "Objektorientierte Pascal-Dialekte werden die Rolle von Pascal weiterführen (hauptsächlich im Heim- und Ausbildungsbereich), ohne jedoch an die industrielle Bedeutung von C++ heranzukommen."

4 Zusammenfassung

Die Wartung bzw. die Wartbarkeit wird besonders häufig als Vorteil der objektorientierten Software-Entwicklung gegenüber der herkömmlichen Software-Entwicklung genannt. Wenn man von verbesserter Wartbarkeit, einfacherer Wartung u.a.m. spricht, dann müßte streng genommen die Vergleichbarkeit sichergestellt sein, d.h. es müßte zumindest klar sein, wie Wartbarkeit, Wartungsaufwand usw. gemessen wird. Erst auf einer solchen Basis können unterschiedliche Softwaresysteme und Entwicklungsansätze systematisch verglichen werden. Der wissenschaftliche Fortschritt in bezug auf diese Fragen ist noch recht unbefriedigend. Im vorliegenden Artikel wurde daher versucht, in Weiterführung der erst spärlich vorhandenen Literatur, Aussagen zur Wartungssituation zu gewinnen. Zu diesem Zweck wurden die Ergebnisse einer Expertenbefragung unter Wartungsgesichtspunkten ausgewertet. Die wichtigsten Erkenntnisse werden nachfolgend nocheinmal zusammengefaßt.

- **Redesign**
  In der OOP tritt Redesign häufiger auf als in der konventionellen Programmierung. Das Redesign ist in der OOP allerdings einfacher durchzuführen als in der konventionellen Programmierung. Modifikationen können ebenfalls einfacher durchgeführt und, falls nötig, wieder entfernt werden.

- **Adaptierende Wartung und Weiterentwicklung**
  Die OOP bietet generell einen flexiblen Ansatz, indem der Entwickler viele Sachverhalte für eine spätere Modifikation offen lassen kann, ohne daß die Zielapplikation darunter leidet. Die Modifikationen können dann meist auch von einem anderen Autor ohne große Reibungsverluste vorgenommen werden. Wartungsaktivitäten im klassischen Verständnis (im Unterschied zum bereits erwähnten Redesign) sind einfacher durchzuführen, da alle auf einem AF basierenden Applikationen die gleiche Grundstruktur aufweisen.

- **Komplexität von objektorientierten Softwaresystemen**
  Bei der objektorientierten Softwareentwicklung entsteht der Programm-Code in einer sehr konzentrierten und kompakten Form. Damit steigt auch die Komplexität bei größeren Systemen rasch an. Die Wartbarkeit solcher Systeme hängt in der Folge stark von der Design-Qualität ab. Es gibt allerdings noch kaum Werkzeuge, welche bei der Bewältigung dieser Komplexität helfen.

- **Dokumentation**
  Die Systemdokumentation stellt ein wesentliches Hilfsmittel für die Wartungstätigkeiten dar. Das prototyping-orientierte Vorgehen bei der objektorientierten Softwareentwicklung

führt häufig dazu, daß die Dokumentation vernachlässigt wird. Objektorientierte Systeme sind im allgemeinen nicht selbstdokumentierend.

- **Archivierung**
  Ein wesentlicher Unterschied gegenüber der konventionellen Softwareentwicklung besteht in der Archivierung der entstehenden Dokumente, vor allem der Code-Dateien. Die Gesamtfunktionalität verteilt sich (meist) über zahlreiche Dateien, wodurch sich bei der OOP die Verwaltung und Zugriffskontrolle komplexer gestaltet.

- **Test und Debugging**
  Die heute verfügbaren Hilfsmittel lehnen sich konzeptionell meist noch stark an die herkömmliche prozedurale Softwareentwicklung an. Werkzeuge, die auf die Besonderheiten der OOP abgestimmt sind, bilden noch die Ausnahme.

- **Arbeitsteiligkeit bei der Softwareentwicklung**
  Es gibt noch wenig Erfahrung über die Entkopplung von Tätigkeiten, um die arbeitsteilige Softwareentwicklung aktiv zu unterstützen. Ziel ist immer die Konstruktion einer Klassenhierarchie, was in einem viel stärkeren Ausmaß Vereinbarungen und Informationsflüsse zwischen den Teams verlangt, als dies in der konventionellen Programmierung der Fall ist. Übereinkünfte über Schnittstellen sind schwieriger zu erreichen, da mit der Hierarchie eine zusätzliche Dimension berücksichtigt werden muß.

In verschiedenen Studien wurde u.a. festgestellt, daß sowohl die Änderungsrate als auch der Änderungsaufwand deutlich gesenkt werden konnten [vgl. z.B. HENR90, MANC90]. Die hier dargestellten Untersuchungsergebnisse lassen quantitative Schlußfolgerungen nicht zu. Festgestellt werden kann aufgrund der Expertenbefragung, daß die objektorientierte Softwareentwicklung manche Wartungsprobleme löst, oder sie zumindest reduzieren kann. Zwischen Wunsch und Wirklichkeit besteht allerdings noch eine große Diskrepanz. Dabei sollte bei der hier gewählten, primär software-technischen Betrachtungsweise nicht übersehen werden, daß ein wesentlicher Engpaß beim Personal besteht. Es ist außerordentlich schwierig, Programmierer zu finden, welche in der objektorientierten Softwareentwicklung Erfahrung und Kompetenz besitzen, und diese dann noch für Wartungsaufgaben zu motivieren.

Verwendete Abkürzungen:

AF	Application Framework
OO	objektorientiert(e)
OOP	objektorientierte Programmierung
OOPL	object-oriented programming language, objektorientierte Programmiersprache
KB	Klassenbibliothek
MI	Multiple Inheritance, Mehrfachvererbung
QA	Quality Assurance, Qualitätssicherung
SI	Single Inheritance, Einfachvererbung

Literatur

[BLAS91] Blaschek, G.: Type-Safe Object-Oriented Programming with Prototypes. The Concepts of Omega. Structured Programming, Vol. 12, No. 4, 1991

[BOOC91] Booch, G.: Object-Oriented Design with Applications. Redwood City, CA: The Benjamin/Cummings Publishing Company 1991

[BUTH91] Buth, A.: Softwaremetriken für objekt-orientierte Programmiersprachen. GMD Arbeitspapier Nr. 545, Bonn/st.Augustin Juni 1991

[COAD91] Coad, P., E. Yourdon: Object-Oriented Analysis. Prentice-Hall International Editions 1991 (2nd Ed.)

[COX86] Cox, B.J., A.J. Novobilski: Object-Oriented Programming. An Evolutionary Approach. Reading, MA: Addison-Wesley 1986

[DAHL66] Dahl, O.-J.: Simula, an Algol-Based Simulation Language. Communications of the ACM, Vol. 9, No. 9, 1966

[DAHL70] Dahl, O.-J., B. Myrhaug, K. Nygaard: Simula 67, Common Base Language; Publication S-22, Norsk Regnesentral, Oslo, October 1970

[DeMA78] DeMarco, T.: Structured Analysis and System Specification. Englewood Cliffs, NJ: Prentice-Hall 1978

[DUMK92] Dumke, R.: Softwareentwicklung nach Maß. Braunschweig/Wiesbaden 1992

[FANG93] Fang, Z., Lowther, B., Oman, P., Hagemeister, J.: Constructing and Testing Software Maintainability Assessment Models. In: Proceedings of the IEEE-CS International Software Metrics Symposium, May 21-22, Baltimore MD, USA, 1993 (in Druck)

[GAMM91] Gamma, E.: Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design. Dissertation. Universität Zürich, 1991

[GANE79] Gane, C., T. Sarson: Structured Systems Analysis: Tools and Techniques. Englewood Cliffs, NJ: Prentice-Hall 1979

[GOLD76] Goldberg, A., A. Kay (Eds.): Smalltalk-72 Instruction Manual. Technical Report SSL-76-6, Xerox Palo Alto Research Center, March 1976

[GOLD83] Goldberg, A., D. Robson: Smalltalk-80: The Language and its Implementation. Reading, MA: Addison-Wesley 1983

[GOLD85] Goldberg, A.: Smalltalk-80: The Interactive Programming Environment. Reading, MA: Addison-Wesley 1985

[HENR90] Henry, S. M., Humphrey, M.: A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software. In: Proceedings of the IEEE Conference on Software Maintenance, San Diego/Los Alamitos CA, 1990, 258-265

[LEHN91] Lehner, F.: Softwarewartung. München 1991

[LEHN92] Lehner, F., Hofmann, H. F., Setzer, R.: Wartung und Pflege von Wissensbanken. Forschungspapier der Wissenschaftlichen Hochschule für Unternehmensführung Koblenz, Nr. 15, Vallendar, November 1992

[LIHE93] Li, W., Henry, S.: Maintenance Metrics for the Object Oriented Paradigm. In: Proceedings of the IEEE-CS International Software Metrics Symposium, May 21-22, Baltimore MD, USA, 1993 (in Druck)

[MANC90] Mancl, D., Havanas, W.: A study of the impact of C++ on software maintenance. In: Proceedings of the IEEE Conference on Software Maintenance, Los Alamitos CA, 1990, 63-69

[MATT87] Matthews, M. H.: Maintenance & Language Choice. In: AI Expert 9/1987, 42-49

[McME84] McMenamin, S.M., J.F. Palmer: Essential Systems Analysis. New York: Yourdon Press 1984

[MEEG92] Van Meegen, M., Schless, P.: Programmierkonventionen für C++. In: Softwaretechnik-Trends. Fachgruppe für "Software Engineering" und "Requirements Engineering" der Gesellschaft für Informatik (Hrsg.), Band 12, Heft 3, 1992, 29-47

[MEYE88] Meyer, B.: Object-oriented Software Construction. Englewood Cliffs, NJ: Prentice-Hall 1988

[ODEL92] Odell, J.J., J. Martin: Object-Oriented Analysis and Design. Englewood Cliffs, NJ: Prentice-Hall 1992

[PAGE89] Page-Jones, M. et al.: Synthesis: An Object-Oriented Analysis and Design Method. In: American Programmer 2 (7-8), Summer 1989, 64-67

[PAGE90] Page-Jones, M. et al.: Modeling Object-Oriented Systems: The Uniform Object Notation. In: Computer Language, Oct. 1990, 69-86

[PARN72] Parnas, D. L.: On the Criteria to be used in Decomposing Systems into Modules. Communications of the ACM, Vol 15, No. 12, 1972, 1035-1058

[RUMB91] Rumbaugh, J. et al.: Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice-Hall 1991

[SAME90] Sametinger, J.: A Tool for the Maintenance of C++ Programs. In: Proceedings of the Conference on Software Maintenance, San Diego, CA, 1990, 54-59

[SAME91] Sametinger, J.: DOgMA: A Tool for the Documentation & Maintenance of Software Systems. Dissertation, Universität Linz, 1991

[SCHA92] Schaschinger, H.: Objektorientierte Analyse und Modellierung. Dissertation, Universität Linz: Institut für Informatik 1992

[SCHM86] Schmucker, K.: Object-Oriented Programming for the Macintosh. Hasbrouck Heights NJ, 1986

[SHLA88] Shlaer, S., S.J. Mellor: Object-Oriented System Analysis: Modeling the World in Data. Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall 1988

[SHLA89] Shlaer, S., S.J. Mellor: An Object-Oriented Approach to Domain Analysis. In: ACM Sigsoft Software Engineering Notes, Vol. 14, No. 5, July 1989, 66-77

[SHLA92] Shlaer, S., S.J. Mellor: Object Lifecycles: Modeling the World in States. Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall 1992

[SIKO92] Sikora, H.: Problembereiche und Trends der objektorientierten Systementwicklung: Eine empirische Untersuchung. Informatik-Berichte, ALLINF 4/92, Institut für Informatik, Abteilung Allgemeine Informatik, Universität Linz, September 1992

[STRO86] Stroustrup, B.: The C++ Programming Language. Reading, MA: Addison-Wesley 1986

[UNGA91] Ungar, D. et al.: Organizing Programs Without Classes. Journal of Lisp and Symbolic Computation, Vol. 1, No. 4, 1991

[WEIN88] Weinand, A., E. Gamma, R. Marty: ET++ — An Object-Oriented Application Framework in C++. In: Meyrowitz, N. (Ed.): OOPSLA '88 Conference Proceedings: Object-Oriented Programming: Systems, Languages and Applications. Special Issue of SIGPLAN Notices. Vol. 23, No. 11. New York: ACM Press 1988

[WEIN89] Weinand, A., E. Gamma, R. Marty: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. In: Structured Programming, Vol. 10, No. 2, Springer: New York 1989

[WEIN91] Weinand, A.: Objektorientierter Entwurf und Implementierung portabler Fensterumgebungen am Beispiel des Application Frameworks ET++. Dissertation. Universität Zürich, 1992

[WILD91] Wilde, N., Huitt, R.: Maintenance Support for Object-Oriented Programs. In: Proceedings of the IEEE Conference on Software Maintenance, Sorento, 1991, 162-170

[WIRF90] Wirfs-Brock, R. et al.: Designing Object-Oriented Software. Englewood Cliffs, NJ: Prentice-Hall 1990

[YOUR89] Yourdon, E.: Modern Structured Analysis. New York: Yourdon Press, Prentice-Hall 1989

[ZUSE91] Zuse, H.: Software Complexity. Measures and Methods. Berlin/New York 1991

Forschungspapiere der Wissenschaftlichen Hochschule für Unternehmensführung Koblenz

Lfd. Nr.	Autor	Titel
1	Weber, Jürgen	Theoretische Herleitung eines Controlling in Software-Unternehmen (Juni 1991)
2	Heinzl, Armin	Spinning off the Information Systems Support Function (Juni 1991)
3	Setzer, Ralf	Ergebnisse einer Befragung zur Beschaffungsplanung für zentrale Rechnersysteme (August 1991)
4.	Pfähler, Wilhelm Lambert, Peter	Die Messung von Progressionswirkungen (Oktober 1991)
5.	Pfähler, Wilhelm Lambert, Peter	Income Tax Progression and Redistributive Effect: The Influence of Changes in the Pre-Tax Income Distribution
6.	Pfähler, Wilhelm Leder, Thomas	Operative Synergie - von der Theorie zur Unternehmenspraxis (z.Zt. nicht erhältlich, wird überarbeitet)
7.	Wiese, Harald	Network Effects and Learning Effects in a Heterogeneous Dyopoly (Dezember 1991)
8.	Heinzl, Armin Stoffel, Karl	Formen, Motive und Risiken der Auslagerung der betrieblichen Datenverarbeitung (Januar 1992)
9.	Bungenstock, C. Holzwarth, J. Weber, Jürgen	Wegfallkosten als Informationsbasis strategischer Entscheidungen (Januar 1992)
10.	Lehner, Franz	Messung der Software-Dokumentationsqualität (August 1992)
11.	Heinzl, Armin Sinß, Michael	Zwischenbetriebliche Kooperationen zur kollektiven Entwicklung von Anwendungssystemen (August 1992)
12.	Heinzl, Armin	Die Ausgliederung der betrieblichen Datenverarbeitung

13.	Lehner, Franz	Expertensysteme zur Unterstützung der strukturorganisatorischen Gestaltung von Unternehmen
14.	Lehner, Franz	Brauchen wir eine Theorie der Wirtschaftsinformatik?
15.	Lehner,Franz, Setzer, Ralf Hofmann, Hubert	Wartung und Pflege von Wissensbanken
16.	Müller, Klein	Grundzüge einer verhaltensorientierten Preistheorie im Dienstleistungsmarketing
17.	Lehner,Franz	Considerations on Information System Strategies Based on an Empirical Study
18.	Lehner, Franz Hofmann, Hubert Setzer, Ralf	Maintance of Knowledge Based Systems
19.	Lehner, Franz Sikora, Hermann	Wartung objektorientierter Softwaresysteme

Rektorat / Juni 1993