Über die Konstruktion robuster objektorientierter Klassenbibliotheken

Arne Frick
Universität Karlsruhe
Institut für Programmstrukturen
und Datenorganisation

Walter Zimmer
Forschungszentrum Informatik
Gruppe Programmstrukturen

Wolf Zimmermann
Universität Karlsruhe
Institut für Programmstrukturen
und Datenorganisation

Zusammenfassung

Software-Bibliotheken sind ein wichtiges Hilfsmittel, um sowohl Wiederverwendung von Entwürfen als auch Wiederverwendung von Programmen zu erreichen. Um diese Wiederverwendung zu ermöglichen, muß eine Bibliothek flexibel sein: eine Komponente soll in möglichst vielen Kontexten einsetzbar sein. Andererseits darf Wiederverwendung nicht zur Entstehung von Fehlern führen. Die Komponenten einer Programmbibliothek müssen auch bei Benutzung in unbekannten Umgebungen funktionieren (Robustheit). Flexibilität und Robustheit stehen oft im Widerspruch zueinander, was in objektorientierten Klassenbibliotheken zu einer einseitigen Betonung der Flexibilität führt.

In diesem Beitrag zeigen wir, daß flexible Wiederverwendung nicht notwendig dazu führt, daß Bibliotheken ihre Robustheitseigenschaften verlieren. Ferner diskutieren wir den Einsatz von Entwurfsmustern zur Strukturierung einer großen objektorientierten Klassenbibliothek.

Die in diesem Beitrag vorgestellten Konzepte wurden in der Algorithmen– und Datenstrukturbibliothek KARLA realisiert. Zur Zeit enthält KARLA mehr als 200 Klassen und wird im Rahmen von Praktika, Studien– und Diplomarbeiten stetig weiterentwickelt.

Stichworte: objektorientierte Klassenbibliotheken, Robustheit, Flexibilität, Effizienz, Entwurfsmuster

1 Einleitung

Zur Einsparung von Entwicklungs– und Wartungskosten sollten Softwarekomponenten wiederverwendbar sein. Daher stehen solche Komponenten häufig in Form von *Programmbibliotheken* zur Verfügung. In diesem Beitrag konzentrieren wir uns auf *objektorientierte Klassenbibliotheken* [10] als diejenige Form von Bibliothek, die das größte Wiederverwendungs-Potential besitzt.

Beim Entwurf von Bibliotheken sollen neben der funktionalen Korrektheit oft auch nicht-funktionale Eigenschaften wie *Robustheit*, *Flexibilität* und *Effizienz* berücksichtigt werden. Wir werden diese Begriffe in Abschnitt 3 formalisieren und erläutern, warum es sich um wünschenswerte Eigenschaften handelt. In diesem Papier zeigen wir Methoden und Techniken zur Konstruktion robuster wiederverwendbarer Klassenbibliotheken auf, wobei innerhalb dieses Rahmens maximale Flexibilität bzw. Effizienz erreicht werden soll.

Der Rest dieses Beitrages ist wie folgt aufgebaut. In Abschnitt 4 werden verschiedene Möglichkeiten der Vererbung von Klassen auf Basis der Spezifikation von Klassen definiert und unter dem Gesichtspunkt der Robustheit untersucht. Damit werden bisherige Ansätze verallgemeinert. In Abschnitt 5 betrachten wir die Instantiierung generischer Klassen und geben Einschränkungen an, die ein Bibliothekskonstrukteurs beachten muß, damit eine generische Klasse nur so instantiiert werden kann, daß dadurch keine neuen Fehler entstehen. In Abschnitt 6 stoßen wir aus der Motivation heraus, die Austauschbarkeit von Klassen- und Methoden zur Laufzeit robust zu gestalten, auf *Entwurfsmuster*. Diese werden sowohl aus Bibliotheks– als auch aus Anwendersicht betrachtet. Dabei erweitern wir die Definition und Anwendbarkeit einiger Entwurfsmuster. Ferner geben wir Beispiele zur konkreten Umsetzung an, die auf Polymorphie und Generizität basieren. In Abschnitt 7 bewerten wir bisherige Ansätze zur Konstruktion objektorientierter Klassenbibliotheken unter den dieser Arbeit zugrundegelegten Anforderungen. Im letzten Abschnitt berichten wir über die Erfahrungen, die wir mit den hier vorgestellten Ideen im Rahmen der Algorithmen– und Datenstrukturbibliothek KARLA gemacht haben. Der Anhang enthält ein durchgängig im Text verwendetes Beispiel.

2 Notation

Als *abstrakte Klassen* bezeichnen wir Klassen, von denen es keine Objekte als Instanzen geben kann. Dies ist der Fall, wenn eine oder mehrere Methoden-Implementierungen fehlen, sondern nur als Schnittstelle vorhanden sind. Alle Klassen, die nicht abstrakt sind, heißen *konkret*.

Eine Klasse B *erbt* von einer Klasse A, wenn die Definitionen von A ganz oder teilweise in B sichtbar sind. Man

sagt auch, daß B *Unterklasse* von A ist.

Unter *Polymorphie* wird verstanden, daß anstelle jeder Klasse auch eine beliebige ihrer Unterklassen eingesetzt werden kann.

Wir folgen der Semantik von SATHER-K und setzen Typen und Klassen gleich: Jede Klasse definiert einen gleichnamigen Typ.

Die Beispiele beziehen sich grundsätzlich auf die im Anhang erläuterte Notation der Sprache SATHER-K.

3 Robustheit, Flexibilität und Effizienz

In diesem Abschnitt definieren wir die Begriffe Robustheit, Korrektheit und Flexibilität. Dazu nehmen wir an, daß Klassen vollständig durch Invarianten, Vor– und Nachbedingungen spezifiziert sind. Eine Klasse A heißt *lokal korrekt* genau dann, wenn

(i) nach der Erzeugung eines Objekts der Klasse A die Invariante Inv_A erfüllt ist und

(ii) nach Aufruf eines Merkmals m eines Objekts der Klasse A das Objekt die Invariante Inv_A und die Nachbedingung $Post_{m,A}$ erfüllt, sofern bei Aufruf Inv_A und die Vorbedingung $Pre_{m,A}$ erfüllt war.

Insbesondere erfüllen die Objekte der Klasse A zu jedem Zeitpunkt die Klasseninvariante Inv_A. Die lokale Korrektheit ist der einzige Korrektheitsbegriff, den der Bibliotheksentwerfer sicherstellen kann. Die Einbeziehung von Vererbung und Generizität erfordert die Definition des Zusammenspiels von Klassen.

Eine Klassenbibliothek heißt *korrekt*, wenn jede Klasse lokal korrekt ist – auch wenn sie geerbt hat oder als Argumente generische Parameter verwendet. Dazu geben wir in Abschnitt 4 zwei Möglichkeiten zur Definition von Vererbung an, mit denen von der lokalen Korrektheit jeder Klasse in einer Bibliothek auf die Korrektheit der Bibliothek als Ganzes geschlossen werden kann. Ferner geben wir in Abschnitt 5 Voraussetzungen an generische Parameter an, um von lokaler auf globale Korrektheit schließen zu können.

Natürlich kann der Programmierer immer noch eine Klassenbibliothek falsch verwenden. So kann es beispielsweise passieren, daß die Vorbedingung einer Methode bei deren Aufruf nicht erfüllt ist oder die in den Abschnitten 4 und 5 diskutierten Eigenschaften nicht eingehalten werden. Schlimmstenfalls tritt damit der Fall ein, daß irgendwo tief in der Aufrufhierarchie einer Bibliothek Fehlersymptome auftreten oder gar der Programmlauf ohne Fehlermeldung abstürzt. Wir fordern an dieser Stelle, daß in solchen Fällen der Programmierer möglichst präzise auf seinen Fehler hingewiesen werden muß. Dazu sollte ihm mitgeteilt werden, welche Vorbedingung beim Aufruf einer Methode verletzt wurde. Eine Klassenbibliothek, die diese Eigenschaften erfüllt, nennen wir *robust*.

Eine Bibliothek, die sowohl korrekt als auch robust ist, nennen wir *zuverlässig*. Zuverlässigkeit ist ist die wichtigste Eigenschaft einer Bibliothek, denn eine unzuverlässige Bibliothek ist nicht verwendbar, geschweige denn wiederverwendbar. Der Erfolg der FORTRAN–Bibliotheken wie etwa *LAPACK* oder *BLAS* ist unter anderem auf ihre hohe Zuverlässigkeit zurückzuführen.

Der Programmierer soll aus der Bibliothek verwendete Klassen austauschen können, wenn sich seine Anforderungen (insbesondere nicht-funktionale) ändern, ohne daß ein kompletter Neuentwurf notwendig ist. Wir werden zeigen, unter welchen Voraussetzungen diese Flexibilität gewährleistet werden kann, ohne daß Korrektheits– und Robustheitseigenschaften einer Bibliothek verloren gehen. Eine Klassenbibliothek, die einen solchen Austausch von Klassen erlaubt, heißt *flexibel*. Die oben erwähnten FORTRAN–Bibliotheken sind unflexibel, da eine Bibliotheksroutine nur für die ihr zugedachte Aufgabe eingesetzt werden kann.

Implementierungen von Klassen sollen möglichst *effizient* hinsichtlich Zeit- und/oder Speicherbedarf sein. Wir werden sehen, daß bei vorausgesetzter Robustheit und geforderter maximaler Flexibilität der Fall eintreten kann, daß nur noch ineffiziente Implementierungen einer Klasse oder einer Methode möglich sind[1]. Die Bibliotheksstruktur muß also so angelegt sein, daß sie neben der Flexibilität auch effiziente Implementierungen von Klassen erlaubt.

4 Vererbungsbeziehungen

In diesem Abschnitt betrachten wir vier mögliche Vererbungsbeziehungen auf der Basis einer formalen Spezifikation der beteiligten Klassen. Wir untersuchen jeweils, wie sich Polymorphie auf das Entwurfsziel Robustheit auswirkt. Die einzigen uns bekannten Arbeiten zu diesem Thema, sind [14, 8], die allerdings nur eine der vier Beziehungen untersuchen, nämlich die *Konformität*[2].

Auf Basis von Klassen-Spezifikationen sind insgesamt vier verschiedene Möglichkeiten zur Definition einer Untertyp-Beziehung mit anschließender polymorpher Verwendung denkbar[3], die in Abbildung 1 dargestellt sind. Diese vier Möglichkeiten werden unter den Gesichtspunkten Robustheit, Kapselung, Flexibilität und praktischer Notwendigkeit diskutiert. Formal erhält man folgende Definitionen:

Seien A und B zwei Klassen. Klasse B heißt *konform* zu Klasse A genau dann, wenn B jede Methode von A enthält und folgende Implikationen gelten (vgl. Abbildung 1a):

(i) $Inv_B \Rightarrow Inv_A$

(ii) Für jede Methode m von A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ und $Post_{m,B} \Rightarrow Post_{m,A}$.

Klasse B heißt *spezieller* als Klasse A genau dann, wenn B jede Methode von A enthält und folgende Implikationen

[1] Das geht unter Umständen soweit, daß nur noch Implementierungen möglich sind, die exponentiellen Zeitaufwand erfordern.

[2] Im Papier von [8] wird zwar eine "Spezialisierung" betrachtet, die sich aber als Spezialfall der Konformität erweist. Ferner wird dort zum Nachweis der Korrektheit kompliziert über Ausführungsgeschichten von Objekten argumentiert. Diese Voraussetzung entfällt im vorliegenden Papier.

[3] Zwar existieren durch Umkehrung der Implikationen bei den Invarianten weitere vier, die aber keine neuen Möglichkeiten erschließen.

gelten (vgl. Abbildung 1b):

(i) $Inv_B \Rightarrow Inv_A$

(ii) Für jede Methode m von A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ und $Post_{m,A} \Rightarrow Post_{m,B}$.

Klasse B heißt *kovariant* zu Klasse A genau dann, wenn B jede Methode von A enthält und folgende Implikationen gelten (vgl. Abbildung 1c):

(i) $Inv_B \Rightarrow Inv_A$

(ii) Für jede Methode m von A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ und $Post_{m,B} \Rightarrow Post_{m,A}$.

Bemerkung: Die Spezialisierungsbeziehung von [8] erfordert die Äquivalenz der Vorbedingungen, was in unserem Szenario ein Spezialfall sowohl der Konformität als auch der Kovarianz ist.

Klasse B heißt *kontravariant* zu Klasse A genau dann, wenn B jede Methode von A enthält und folgende Implikationen gelten (vgl. Abbildung 1d):

(i) $Inv_B \Rightarrow Inv_A$

(ii) Für jede Methode m von A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ und $Post_{m,A} \Rightarrow Post_{m,B}$.

Durch die polymorphe Verwendung von Typen ist es möglich, daß der konkrete Typ eines Bezeichners erst zur Laufzeit feststeht. Fordert man diese durch das Prinzip der Polymorphie gegebene Flexibilität, dann kann im Sinne der Robustheit nur eine aufgeführten Vererbungsbeziehungen eingesetzt werden, nämlich die Konformität. Wie man leicht sieht, ist die Korrektheit eines Programms gewährleistet, wenn man einer Variable vom polymorphen Typ A zur Laufzeit ein Objekt einer zu A konformen Unterklasse B zuweist. Die übrigen Vererbungsbeziehungen sind nicht robust gegenüber Polymorphie, sondern man muß zu jedem Zeitpunkt genau den Typ von Objekten kennen, um Aussagen über die Korrektheit von Programmen treffen zu können. Wir halten fest, daß das Entwurfsziel der Konstruktion robuster Bibliotheken nur konforme Vererbung unterstützt wird.

Sind nun die übrigen Vererbungsbeziehungen irrelevant? Dies ist im Sinne der Wiederverwendung sicherlich nicht der Fall. Zumindest die Spezialisierung stellt sich als ebenso wichtig heraus wie die Konformität.

Beispiel 1: Wir betrachten das im Anhang angeführte Beispiel. Bei der robusten Modellierung von Graphen stellt man fest, daß die Klasse der gerichteten azyklischen Graphen (*dags*) spezieller als die Klasse der gerichteten Graphen sein muß. Man sieht nämlich leicht, daß in einen *dag* nicht mehr beliebige Kanten einfügt werden können, da unter Umständen die Invariante, Azyklizität, verletzt wird. Ein weiteres Beispiel sind größenbeschränkte Datenstrukturen (wie etwa größenbeschränkte Schlangen, Keller, Listen oder Mengen). Solche Strukturen sind spezieller als ihre unbeschränkten Versionen, denn im Gegensatz zu diesen ist nun ein Überlauf möglich. Ein drittes Beispiel findet man bei relationalen Datenbanken, wo Integritätsbedingungen und Normalformen von Relationen zu Spezialisierungen führen.

Wie man an diesen Beispielen sieht, ist die Spezialisierungsbeziehung durchaus natürlich und ebenso wichtig wie die Konformität. Daraus folgt, daß eine Klassenbibliothek unter Robustheitsgesichtspunkten *mindestens zwei Arten von Vererbung* zur Verfügung stellen muß.

Da nur konforme Vererbung zu einer robusten Polymorphie führt, könnte man sich aus der Sicht der Bibliothek heraus könnte man nicht-konforme Vererbungen rein unter dem Wiederverwendungsaspekt betrachten. Andererseits kann die Information, welche Beziehung zwischen Klassen besteht, dem Bibliotheksanwender weiterhelfen. Wir diskutieren abschließend die drei anderen Beziehungen unter diesem Aspekt. Dazu nehmen wir an, daß der Programmierer ein Objekt x der Klasse A deklariert hat und beim Entwurf feststellt, daß dort eigentlich ein Objekt x der Klasse B gefordert ist. Wir beantworten die Frage, ob Wiederverwendung stattfinden kann, oder ob ein kompletter Neuentwurf erforderlich ist.

1. A ist spezieller als B. Dann muß nur geprüft werden, ob der Korrektheitsbeweis auch mit der schwächeren Invariante Inv_A geführt werden kann.

2. B ist spezieller als A. Dann muß nur geprüft werden, ob der Korrektheitsbeweis auch mit den stärkeren Vorbedingungen $Pre_{m,B}$ und den schwächeren Nachbedingungen $Post_{m,B}$ geführt werden kann.

3. B ist kovariant zu A. Dann muß nur geprüft werden, ob der Korrektheitsbeweis auch mit den stärkeren Vorbedingungen $Pre_{m,B}$ geführt werden kann.

4. B ist kontravariant zu A. Dann muß nur geprüft werden, ob der Korrektheitsbeweis auch mit den schwächeren Nachbedingungen $Post_{m,B}$ geführt werden kann.

Falls in diesen Fällen die erforderlichen Prüfungen ergeben, daß der Korrektheitsbeweis so nicht geführt werden kann, dann muß zu diesem Zweck zusätzlicher Code an denjenigen Stellen eingefügt werden, dessen Stellen sich aus den Punkten ergibt, an denen der Korrektheitsbeweis mit den neuen Zusicherungen nicht geführt werden kann.

Es verbleiben zwei Fälle: wenn A kovariant oder kontravariant zu B ist, dann müssen an jeder Stelle, an der eine Methode von x verwendet wird, die Invarianten und die Vor– und die Nachbedingungen geprüft werden, was einem vollständigen Neuentwurf gleichkommt.

Zusammenfassend kommt man also zu folgenden Aussagen. Die einzige Vererbung, die unter Polymorphie robust ist, ist die konforme Vererbung. Alle anderen Vererbungsbeziehungen sind nicht robust unter Polymorphie. Jedoch erlauben alle Arten der Vererbung Code–Wiederverwendung und unterstützen die Wiederverwendung von Entwürfen. Eine Bibliothek sollte daher alle vier Vererbungsbeziehungen unterstützen.

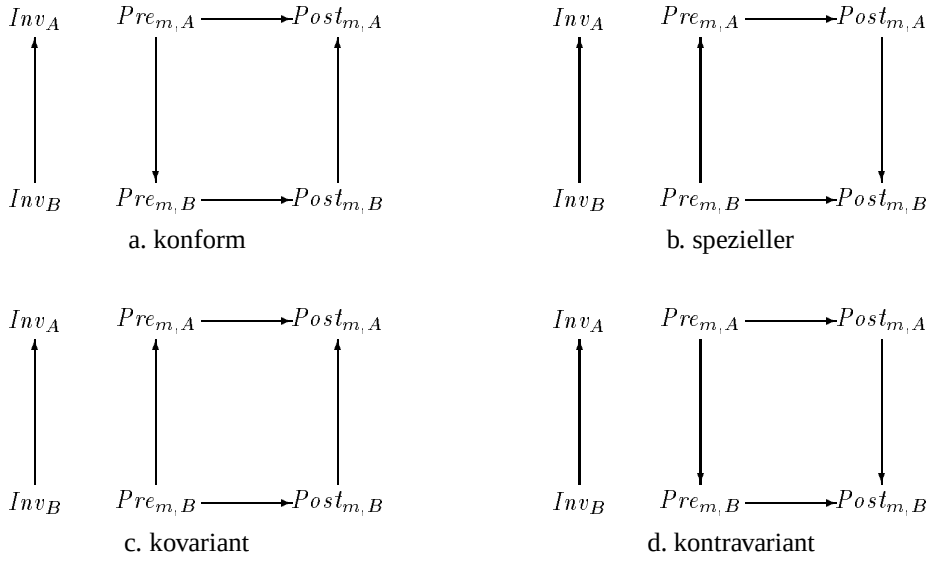

Abbildung 1: Verschiedene Vererbungsbeziehungen

5 Generische Klassen

Die Hauptanwendung der Generizität besteht in der Konstruktion von *Containerklassen* $C(T)$, für die bei der Instantiierung anstelle von T beliebige Typen eingesetzt werden dürfen[4]. Die Definition von Container-Klassen erfolgt unabhängig vom *Typparameter* T, denn aus Gründen der Zuverlässigkeit dürfen weder in den Implementierungen noch in den Methoden einer Containerklasse $C(T)$ irgendwelche Annahmen über die Struktur von T getroffen werden. Andernfalls wäre die Zuverlässigkeit der Bibliothek verletzt: der Programmierer könnte eine Containerklasse mit einem Typ instantiieren, der diese Annahmen nicht erfüllt, was dann zwangsläufig zu Übersetzungs– oder Laufzeitfehlern führt.

Wir nennen diese Generizität, bei der keine Annahmen über den Typparameter getroffen werden, *unbeschränkte Generizität*. Die unbeschränkte Generizität schließt viele Möglichkeiten der Wiederverwendung aus. Beispielsweise kann eine Klasse *Liste* keine Operation enthalten, die die Elemente der Liste sortiert, denn sonst wird implizit bereits die Existenz einer Relation $<$ angenommen, mittels derer der Typ T total geordnet ist. In einer zuverlässigen Bibliothek sollten also auch eingeschränkte Instantiierungen möglich sein (*beschränkte Generizität*).

Beispiel 2: Im Graphen-Beispiel ist die Klasse *EDGETYPE* eine Typschranke für die in *D_GRAPH* erlaubten Instanzen.

Wenn man für eine Containerklasse an einen generischen Parameter mittels sogenannter *Typschranken* Mindestanforderungen definiert hat, dann dürfen Anforderungen in der Spezifikation, der Implementierung und beim Nachweis der Korrektheit verwendet werden, denn es dürfen zur Laufzeit nur solche Typen T eingesetzt werden, die den Mindestanforderungen genügen. Aus der Diskussion des vorigen Abschnitts ergibt sich eine elegante Möglichkeit zur Definition der Mindestanforderungen. Sei $C(T)$ eine generische Klasse. Mindestanforderungen umfassen einerseits Eigenschaften der Objekte der Klasse T, andererseits werden bestimmte Methoden (wie etwa die Ordnungsrelation) samt deren Vor– und Nachbedingungen gefordert. Es liegt also nahe, solche Anforderungen selbst durch Klassen A zu definieren. In KARLA wurde dies in Form von *eigenschaftsdefinierenden Klassen* realisiert. Die Existenz einer Typschranke A an einen generischen Parameter T wird durch $C(T < A)$ notiert.

Welche Klassen dürfen nun anstelle von T eingesetzt werden? Dazu beobachten wir, daß der Beweis der lokalen Korrektheit von $C(T < A)$ mittels Inv_A, $Pre_{m,A}$ und $Post_{m,A}$ geführt werden darf. Damit die Korrektheit für jede Instanz erhalten bleibt, dürfen also nur noch solche Klassen für T eingesetzt werden, die konform zu A sind.

6 Entwurfsmuster

Ein objektorientiertes Entwurfsmuster besteht aus mehreren miteinander kollaborierenden Klassen, die als Gesamtheit ein wiederverwendbares, bewährtes Lösungsschema zu einem typischen Entwurfsproblem darstellen. Die erste bedeutende Sammlung von objektorientierten Entwurfsmustern wurde kürzlich von GAMMA ET.AL. [5] veröffentlicht.

In diesem Abschnitt untersuchen wir einige der in [5] aufgeführten Entwurfsmuster auf die Frage hin, wie sie im Rahmen des Bibliotheksentwurfs zur Unterstützung der Entwurfsziele Zuverlässigkeit, Flexibilität und Effizienz

[4] Beispiele für solche Container-Klassen sind sind abstrakte Datentypen wie *Mengen*, *Listen*, *Keller* und *Schlangen*, die als Elemente Objekte beliebiger Typen enthalten können

eingesetzt werden können. Die Beschreibung der betrachteten Muster wird um eine Diskussion ihrer Trade-Offs zwischen Zuverlässigkeit vs. Flexibilität vs. Effizienz erweitert. Insbesondere wird das Entwurfsmuster *Brücke* signifikant erweitert.

6.1 Strategie

Das Entwurfsmuster Strategie (*Strategy* [5]) "definiert eine Familie von Algorithmen, kapselt jeden einzelnen, und macht sie austauschbar" [5]. Kapselung heißt, daß man eigene Klassen für die verschiedenen Algorithmen definiert, so daß "Algorithmenobjekte" ausgetauscht werden können. Die Verwendung solcher *Algorithmenklassen* als generische Parameter einer Datenstrukturklasse ergibt eine statische Variante von Strategie, wobei der üblicherweise benötigte polymorphe Aufruf entfällt.

6.2 Brücken

Das Entwurfsmuster *Brücke* (*Bridge* [5]) trennt die Schnittstelle eine Klasse von ihrer Implementierung. Eine Brücke für eine (abstrakte) Klasse erlaubt den dynamischen Austausch von Implementierungen einer Klasse.

Beispiel 3: Wir betrachten das Beispiel Graphen-Modellierung im Anhang. Ziel ist es, zur Laufzeit die vorgegebene Mengen-Implementierung eines Objekts $x : \$D\_GRAPH(VERTEX, EDGE)$ durch eine Implementierung mittels Adjazenzlisten bzw. Adjazenzmatrizen auszutauschen.

Eine Brücke für eine Klasse A kann wie folgt durch eine Klasse A_b realisiert werden, die konforme Unterklasse von $\$A$ ist und ein Attribut $repr : \$A$ enthält, das auf die aktuell gültige Repräsentation verweist. Aufrufe von Methoden m in der Klasse A_b werden $repr$ delegiert[5]. Ein Austausch der Repräsentation durch eine neue Implementierung A_i erfolgt einfach durch Zuweisung eines Objektes der Klasse A_i an $repr$. Um den abstrakten Zustand des Objekts zu erhalten, führt man eine Methode $change\_repr$ ein, die den Repräsentationswechsel konsistent und für Benutzer von A_b transparent durchführt.

Man beachte, daß dies nur den Austausch einer Implementierung *samt aller* Methoden erlaubt. Für den Anwender einer Bibliothek genügt dies jedoch nicht. Ideal dafür ist eine Erweiterung der *Brücke* um Strategien. Das ermöglicht dem Programmierer beliebigen Austausch von Repräsentationen einer Klasse A und Algorithmen, die Methoden von A implementieren. Eine solche Brücke (die wir nun *dynamische Brücke* nennen) hat neben dem Attribut *repr* auch noch für jede Methode m, bei der der Algorithmus ausgetauscht werden soll ein Algorithmenobjekt $m_a : \$M_a$, an das der Aufruf der Methode m delegiert wird. Die Klasse M_a sorgt für irgendeinen Algorithmus der m implementiert, z.B. kann dort an die Methode m des Objekts $repr$ delegiert werden.

Dieser Ansatz ist sehr allgemein. Aus Robustheitsgründen ist er sogar zu allgemein, wie das folgende Beispiel zeigt.

Beispiel 4: In unserem Beispiel *Graphen* lassen sich für einige Methoden wie etwa $are\_connected$ und $is\_acyclic$ bereits in der (abstrakten) Klasse $D\_GRAPH(VERTEX, EDGE)$ Implementierungen angeben. Methoden, für die dies möglich ist, nennen wir *abgeleitete Methoden*. Nach Definition von abstrakten Klassen (siehe Abschnitt 2 ist dies nicht für alle Methoden möglich, sonst wäre die Klasse nicht mehr abstrakt. Beispiele dafür sind die Methoden $init, V, E, addvertex, addedge, delvertex$ und $deledge$, die eng mit der konkreten Repräsentation zusammenhängen und daher nicht bereits in der abstrakten Klasse selbst implementiert werden können. Man spricht in diesem Fall von *Kernmethoden*.

Weil abstrakte Implementierungen nicht auf die konkrete Repräsentation zugreifen können, sind sie im allgemeinen ziemlich ineffizient. Daher ist ein Austausch der Implementierung zur Laufzeit schon unter Effizienzgesichtspunkten sinnvoll. Genau dies wird durch die Verwendung von dynamischen Brücke erlaubt.

Abbildung 2 zeigt die Realisierung einer solchen Brücke für eine abstrakte Klasse A, wobei beispielhaft die Behandlung einer Kernmethode $k : SAME$ und eine abgeleitete Methode $m : T$ gezeigt ist. Daneben wird noch die *allgemeinste* Strategie M_a für m gezeigt. Angenommen, man will einen neuen Algorithmus für m zur Laufzeit ersetzen. Dieser Algorithmus muß in einer Unterklasse N von M_a definiert werden. Der Austausch erfolgt dann einfach durch die Zuweisung $m_a := \#N$, die ein neu zu erzeugendes Objekt der Klasse N an m_a zuweist. Man beachte, daß *init* irgendeine Implementierung und den dort definierten Algorithmus für m auswählt. Hat m Parameter, dann werden diese bei der Definition von m in M_a entsprechend hinzugefügt. Hat die Klasse A generische Parameter, so hat die Klasse A_b ebenso wie die Klasse M_a dieselben generischen Parameter mit denselben Schranken.

Aus Gründen der Robustheit können bei einer dynamischen Brücke nicht alle möglichen Algorithmen für eine Methode m eingesetzt werden.

Beispiel 5: Beispielsweise beruht ein effizienter Algorithmus zur Berechnung aller kürzesten Pfade in einem Graphen auf einer Repräsentation mittels Adjazenzmatrizen [4, Kapitel 26.1]. Dieser Algorithmus kann und darf nicht in einer Graphen-Implementierung mittels Mengen benutzt werden, da er auf einer konkreten Repräsentation beruht.

Wir unterscheiden daher *repräsentationsabhängige* von *repräsentationsunabhängigen* Algorithmen. Generell muß vermieden werden, daß repräsentationsabhängige Algorithmen bei falscher Repräsentation eingesetzt werden. Da bei dynamischen Brücken die Repräsentation einer Klasse nicht vorhersehbar ist (man kann ja jederzeit die Repräsentation wechseln) erlauben wir dort nur repräsentationsunabhängige Algorithmen. Dies kann beispielsweise durch das Schema in Abbildung 2 erreicht werden, das nur repräsentationsunabhängige Algorithmen als Untertyp von M_a zuläßt.

Leider sind meist gerade die repräsentationsabhängigen Algorithmen die effizientesten. Um auch solche Algorithmen benutzen zu können, wird die passende Repräsentation

[5] Diese SATHER-K-spezifische Notation besagt, daß *repr* zur Laufzeit einen beliebigen Untertyp von A annehmen kann (siehe Anhang).

```
(1)  class A_b is
(2)  subtype of A
(3)  private repr : $A
(4)  m_a : $M_a;
(5)  ...
(6)  init : SAME is
(7)      repr := #A_i.init;
(8)      m_a := #M_a;
(9)      ...
(10)     res := self;
(11) end;
(12) ...
(13) k : SAME is
(14)     repr := repr.k;
(15) end;
(16) ...
(17) m : T is
(18)     res := m_a.m(repr);
(19) end;
(20) ...
(21) end;
(22)
(23) class M_a is
(24)     m(repr : $A) : T is
(25)         res := repr.m;
(26)     end;
(27) end;
```

Abbildung 2: Realisierung einer dynamischen Brücke der Klasse A

benötigt. Dazu definieren wir eine *statische Brücke*, etwa nach folgendem Muster. Im Gegensatz zur dynamischen Brücke wird die Repräsentation ein generischer Parameter. Ebenso wird dieser generische Parameter ein generischer Parameter der Klasse M_a. In Abbildung 2 werden dazu folgende Ersetzungen durchgeführt:

```
(1)  class A_b(REPR < A) is
(2)  subtype of A
(3)  private repr : REPR
(4)  m_a : $M_a(REPR;
(5)  ...
(6)  init : SAME is
(7)      repr := #REPR.init;
(8)      m_a := #M_a(REPR);
(9)      ...
(10)     res := self;
(11) end;
```

Zusammenfassend halten wir fest, daß sich statische und dynamische Brücken aus Sicht der Programmierung ausschließlich im Klassenkopf, der Definition der brückenspezifischen Attribute sowie der Initialisierungsroutine unterscheiden.

Basiert ein Algorithmus für m auf einer speziellen Implementierung A_i von A, dann ist die entsprechende Algorithmenklasse einfach Untertyp von $M_a(A_i)$. Wir verdeutlichen das Konzept am Beispiel der Berechnung der kürzesten Pfade im Graphen aus Beispiel 5.

Beispiel 6: Eine statische Brücke der Graphen-Klasse definiere ein Attribut

$$s\_path : \$S\_PATH(VERTEX, EDGE, REPR),$$

so daß der Aufruf von *s_path* zur Berechnung der kürzesten Pfade an eine von der konkreten Repräsentation *repr* abhängige Version des Algorithmus (hier: Implementierung mittels Adjazenzmatrizen) delegiert wird. Diese repräsentationsabhängige Version des Algorithmus wird in einer Unterklasse von

S_PATH(VERTEX, EDGE,
REPR<D_GRAPH_AM(VERTEX,EDGE))

definiert, wobei $D\_GRAPH\_AM(VERTEX, EDGE)$ für die in Beispiel 5 eingeführte Implementierung der Datenstruktur *Graph* mittels Adjazenzmatrizen stehen soll.

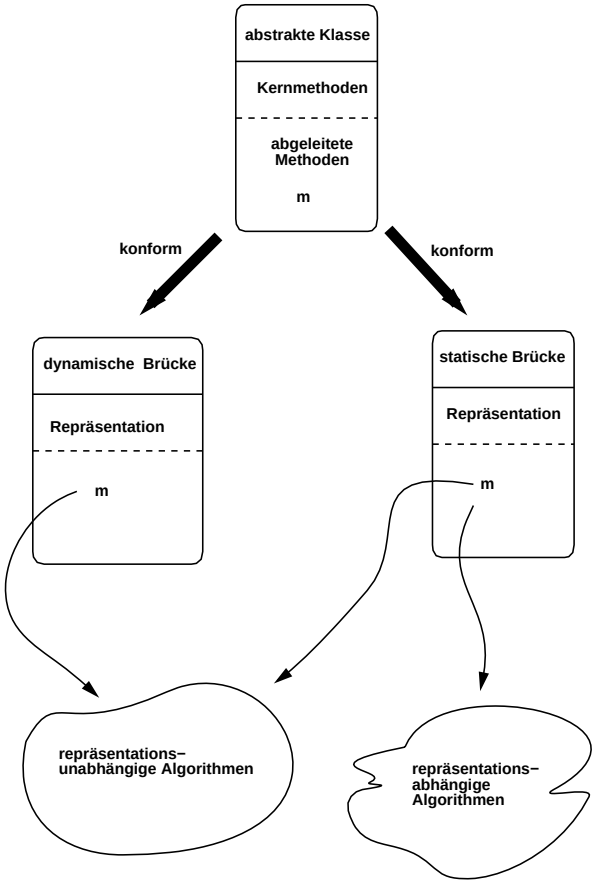

Abbildung 3: Dynamische und statische Brücken

Abbildung 3 stellt den Zusammenhang dar, den dynamische und statische Brücken zwischen Schnittstelle und Implementierung bilden. Dynamische Brücken für eine Klasse A erlauben sowohl den Austausch der Implementierung von A als auch der einzelner Methoden, wobei allerdings nur repräsentationsunabhängige Algorithmen verwendet werden dürfen. Umgekehrt erfordern statische Brücken für A die Festlegung der Implementierung bei der Objektdeklaration, erlauben dafür aber den beliebigen Austausch von Methoden-Implementierungen, auch durch solche, die von der Repräsentation abhängig sind.

6.3 Dekorateur

Das Entwurfsmuster Dekorateur (*Decorator* [5]) dient dazu, die Funktionalität oder die Eigenschaften eines Objekts dynamisch zu erweitern.

Technisch werden diese Ziele erreicht durch *Dekorateur*-Objekte (Klasse D), die ein Objekt (Klasse O) einschließen, um seine Eigenschaften zu erweitern, wobei das Dekorateur-Objekt Anfragen zu dem Originalobjekt delegiert. Die Dekorateur-Klasse D und die Klasse O sind dabei Unterklassen von einer gemeinsamen Abstraktion (Klasse A). GAMMA ET.AL. bemerken, daß die Schnittstellen von D und O konform zu denen von A sein müssen. Es stellt sich aber in der Praxis heraus, daß daneben weitere Aspekte zu beachten sind. Konformität muß, wie wir in refsec:inherit gezeigt habe, aus Robustheitsgründen nicht nur auf der Ebene der Schnittstelle, sondern auch auf der semantischen Ebene (Verhaltenskonformität wie beschrieben in Abschnitt 4) erfüllt sein, beschreibt also des Verhältnis zwischen den Objekten zur Laufzeit. Dies drückt sich in folgendem Beispiel aus.

Beispiel 7:

Sei *BOUNDED* eine Unterklasse von *D_GRAPH*, mit Hilfe derer man Objekte konkreter Unterklassen von *D_GRAPH* wie z.B. *D_GRAPH_SET* dekorieren kann, so daß diese Objekte nur eine begrenzte Anzahl von Knoten verwalten können.

Falls man Objekte vom Typ *D_GRAPH_SET* mit Hilfe der Klasse *BOUNDED* dekoriert, dann ändert sich die Schnittstelle nicht, und nach GAMMA ET.AL. sollte dies daher eine gültige Dekoration sein. Leider wird jedoch die Konformitätsbeziehung zwischen beiden Objekten verletzt, da sich die Vorbedingung der Methode *addVertex(v:VERTEX)* verschärft hat, so daß nur eine begrenzte Anzahl von Elementen aufgenommen wird. Man kann also das dekorierte Objekt nicht ohne die Gefahr von Laufzeitfehlern an Stelle eines normalen Objektes einsetzen.

Des weiteren erlauben verschachtelte Dekorateure zwar die flexible Kombination verschiedener Eigenschaften, aber Typ und (an der Schnittstelle) sichtbare Eigenschaften des dekorierten Objekts sind vom äußersten Dekorateurs abhängig. Daher kann man Eigenschaften von inneren Dekorateuren nicht immer ausnutzen für Effizienzsteigerungen bzw. in dynamisch getypten Sprachen beinhaltet dies die Gefahr von Laufzeitfehlern. Daher werden Dekorateure üblicherweise nur durch die Erweiterung von existierenden Methoden ohne Änderung der Schnittstelle realisiert.

Durch (Mehrfach-) Vererbung kann man eine statische Variante des Dekorateur Entwurfsmusters erzielen, allerdings mit Nachteilen wie Klassenexplosion und geringerer Flexibilität, aber mit einer erhöhten Zuverlässigkeit. Zur Zeit untersuchen wir an Hand von KARLA, inwieweit Generatortechnologie eingesetzt werden kann, um eine statische, sichere Variante von Dekorateur bereitzustellen, mit der ein Teil der Flexibilität erhalten werden kann. Ein geeignetes Konfigurationsystem soll dabei die durch die Mehrfachvererbung verbundenen Nachteile wie erhöhte Komplexität für den Benutzer weitgehend beseitigen.

6.4 Adapter

Das Entwurfsmuster *Adapter* stellt eine Lösung bereit, um eine gegebene Schnittstelle (*I1*) an eine andere Schnittstelle (*I2*) anzupassen. Im Bereich von Datenstrukturbibliotheken erlaubt Adapter dadurch die Anpassung der Schnittstellen von Komponenten anderer Bibliotheken an die Schnittstellen der vorhandenen abstrakten Klassen. Somit können viele komplexe Algorithmen existierender Bibliotheken wiederverwendet werden.

Weiterhin kann Adapter verwendet werden bei der Realisierung von Brücken. Eine spezielle Implementierung kann man in unterschiedlichen Brücken unverändert benutzen, indem man die Schnittstelle der Implementierungsvariante an die jeweilige abstrakte Klasse der Brücke anpaßt. So kann zum Beispiel eine Listenimplementierung verwendet werden als Implementierungsvariante für abstrakte Datentypen wie *List*, *Mapping*, *Set*, *Bag*, usw. Falls die Konformitätsbeziehung zwischen einer abstrakten Klasse und ihren Implementierungsvarianten erfüllt ist, dann ist dies eine zuverlässige Kombination der Entwurfsmuster Adapter und Brücke.

6.5 Zusammenfassung und Bewertung

Eine flexible Bibliothek sollte Mechanismen zur Kombination ihrer Klassen bereitstellen. Eine wichtige Voraussetzung ist hierbei, für die abstrakten Klassen klare Verantwortlichkeiten festzulegen und diese über wohldefinierte Schnittstellen zur Verfügung zu stellen. Die Kombinationsmöglichkeiten der abstrakten Klassen übertragen sich auf ihre konkreten Unterklassen, sofern hier die Konformitätsbeziehung erfüllt ist.

Dieses Prinzip wird in vielen Entwurfsmustern angewendet. Durch Anwendung von Entwurfsmustern wie Dekorateur, Brücke oder Strategie werden abstrakte Klassen (gemeinsame Schnittstelle) zur Verfügung gestellt mit Unterklassen wie z.B.: Klassen, die dekoriert werden; Klassen, die verschiedene Implementierungsvarianten enthalten; Klassen, die verschiedene Strategien implementieren.

Das Finden solcher Standardschnittstellen für die Abstraktionen ist oft schwierig, wie im folgenden erläutert wird. Manchmal unterstützen konkrete Klassen die Schnittstelle der abstrakten Oberklasse, aber sie können nicht die Funktionalität hinter jeder Methodenschnittstelle realisieren, oder das Verhalten ist implementiert, jedoch ist es nicht konform zu der Vor/Nachbedingung der Oberklasse. Dies bedeutet aber, daß die Austauschbarkeit und damit die Voraussetzung für die Kombination dieser konkreten Klassen nicht gegeben ist, beziehungsweise, daß die Kombination der Komponenten Fehler hervorrufen kann. Wenn ein solcher Fall also vorliegt, dann wurde der Standardisierungsprozeß zu weit vollzogen, und man hat Zuverlässigkeit zugunsten von Flexibilität aufgegeben.

Dieser Trade-Off zwischen Flexibilität und wenigen standardisierten Schnittstellen auf der einen Seite und Zuverlässigkeit auf der anderen Seite sollte beim Bibliotheksentwurf immer beachtet werden. Die Priorität (Flexibilität oder Zuverlässigkeit) hängt ab vom Typ der Applikation, vom Grad der geplanten Wiederverwendung, wieviel Flexibilität oder Zuverlässigkeit benötigt wird, usw.

Die oben eingeführten statischen Varianten von Entwurfsmustern erlauben durch die Ausnutzung von repräsentationsabhängigen Merkmalen die Erhöhung der Effizienz bei gleichzeitigem Erhalt der Zuverlässigkeit, wobei

ein Teil der Flexibilität (u.a. Kombination von Klassen zur Laufzeit) verloren geht[6]. Die Entwurfsmuster aus [5] legen viel Wert auf Flexibilität, wohingegen beim Entwurf von KARLA die Anforderungen bzgl. der Zuverlässigkeit klar die höchste Priorität hatten. Für KARLA gilt ferner, daß die angedeuteten Implementierungen für die Brücken automatisch aus der abstrakten Klasse erzeugt werden können.

7 Untersuchung existierender Klassenbibliotheken

In diesem Abschnitt werden existierende objektorientierte Bibliotheken im Hinblick auf die folgenden Fragen untersucht und verglichen:

- Inwieweit wurde beim Entwurf der Bibliotheken auf die oft gegensätzlichen Anforderungen — Zuverlässigkeit vs. Flexibilität vs. Effizienz — eingegangen? Welche Konzepte und Techniken werden genutzt, um die verschiedenen Aspekte zu berücksichtigen?
- Welche Rolle spielt Vererbung für die Struktur der Bibliothek? Für welche Zwecke wird Vererbung eingesetzt und was sind die Folgen dieser Verwendung?

Eiffel

Die Sprache Eiffel [11, 12] unterstützt die Anforderung Zuverlässigkeit durch die Möglichkeit, Vor- und Nachbedingungen für Methoden und Klasseninvarianten für Klassen anzugeben.

Wie CASAIS [1] heraushebt, wird innerhalb der gesamten Eiffel Bibliothek (Mehrfach-) Vererbung für verschiedenste Zwecke eingesetzt: für Gleichheit und Erweiterung (beide ähnlich zu Konformität), für Einschränkung und Code Wiederverwendung (ähnlich zu Spezialisierung), und für Kombinationen dieser Vererbungsbeziehungen.

Aus diesem Grund gibt es meist keine klare Trennung zwischen Schnittstelle und Implementierung, und der Austausch der Implementierung einer Datenstruktur oder eines Algorithmus' ist nicht einfach. Die sich dadurch ergebende Bibliotheksstruktur ist recht komplex, so daß ein großer Teil der Vorteile, die sich durch die Standardisierung der Eiffel Schnittstellen ergeben hatte, wieder verloren scheint.

Ziel der Restrukturierungs-Operationen (Dekomposition und Faktorisierung), die CASAIS durchgeführt hat [1], war es, Vererbung ausschließlich für Spezialisierung und Verfeinerung (Konformität) einzusetzen. Wie in Abschnitt 4 werden Spezialisierung und Konformität als die beiden Hauptanwendungsfälle für Vererbung angesehen (in Eiffel können diese allerdings nicht explizit unterschieden werden).

LEDA

LEDA [7, 9] steht für *Library of Efficient Data Structures and Algorithms*. Die Bibliothek ist implementiert in C++ und legt den Schwerpunkt auf Datenstrukturen (Basisdatentypen, Wörterbücher, Warteschlangen, Graphen, algorithmische Geometrie) und effiziente Algorithmen.

Um der Anforderung Effizienz gerecht zu werden, wird für alle Datentypen die asymptotisch effizienteste Implementierung, die bekannt ist, zur Verfügung gestellt. Die Parametrisierung eines Datentyps (Schnittstelle) mit einer Implementierungsvariante ist möglich, allerdings nur zur Übersetzungszeit, da die Parametrisierung durch Mehrfachvererbung erzielt wird, einer statischen Beziehung (statische Brücke). LEDA unterstützt das sogenannte *Item*-Konzept, d.h. LEDA stellt dem Benutzer eine sichere Abstraktion einer Position in einer Datenstruktur zur Verfügung, um dadurch einen effizienten Zugriff auf eine solche Position zu erhalten.

LEDA's Vererbungstruktur ist im wesentlichen flach, da Vererbung nur zur internen Strukturierung der Bibliothek eingesetzt wird. Daher wird keine Wiederverwendung durch Verwendung abgeleiteter Methoden erzielt. Bei der Parametrisierung von Datentypen wie Wörterbüchern mit Schlüssel- und Informations-Typen gibt es keine Einschränkungen auf diesen Typen (in KARLA wird dies zum Beispiel mit eingeschränkter Generizität erreicht), was das Robustheitskriterium verletzt. Daher müssen bei der Parametrisierung zusätzlich Funktionen angegeben werden, die eine Ordnung auf den Schlüsseln definieren bzw. es erlauben, Informationselemente auf Gleichheit zu überprüfen.

Die Dokumentation weist zwar Vor- und Nachbedingungen für einige Methoden auf, aber diese werden von der Implementierung zur Laufzeit nicht überprüft, eine weitere Verletzung des Robustheitskriteriums.

Die neueste Version von LEDA stellt für einige ihrer Datentypen mehrere Implementierungsvarianten zur Verfügung. Allerdings kann man die benötigte Implementierungsvariante nur zur Übersetzungszeit austauschen (statische Brücke), nicht zur Laufzeit.

So stellt LEDA zwar Implementierungen der aktuell effizientesten Datentypen zur Verfügung, kann aber Anforderungen nach Zuverlässigkeit und Flexibilität nicht genügen.

Smalltalk

Smalltalk [6] als dynamisch getypte Sprache unterstützt natürlich die Flexibilität, aber damit ist auch ein Verlust in Effizienz und speziell in Zuverlässigkeit verbunden.

In [3] wird von COOK die Smalltalk-80 Datenstruktur Bibliothek untersucht, und aus der Vererbungshierarchie wird eine *saubere* Hierarchie von Schnittstellen hergeleitet. Diese Entwicklung und der Vergleich der beiden Hierarchien zeigte einige der im folgenden beschriebenen Schwächen der Bibliothek auf.

Vererbung wird genutzt für die Wiederverwendung von Schnittstellen, von Code, oder einem Gemisch aus beidem. Methoden werden oft von Unterklassen gelöscht oder Methoden von Unterklassen verletzen die durch die Oberklassen gegebenen Vor- oder Nachbedingungen. Der Nutzen von Spezifikationen und klaren Schnittstellen beim Ent-

[6] Man beachte jedoch, daß durch geschickte Kombination statischer und dynamischer Brücken diese Flexibilität fast wiedergewonnen werden kann

wurf von Bibliotheken wird im Papier deutlich demonstriert. Weiterhin schlägt COOK vor, Vererbung nur für Konformität einzusetzen und andere Methoden (z.B. Mehrfachvererbung) bereitzustellen, um Code wiederzuverwenden.

Die Flexibilität von Smalltalk führt also oft zu einer undisziplinierten Benutzung von Mechanismen wie Vererbung, was die Zuverlässigkeit von Anwendungen beeinträchtigen kann. Effizienz stellt für zeitkritische Anwendungen immer noch ein Problem dar.

Coherent Design

RÜPING, WEBER UND ZIMMER[13] diskutieren, wie die verschiedenen Anforderungen Erweiterbarkeit, Verständlichkeit und Effizienz in Einklang gebracht werden können. Das Papier zeigt an Hand einer Datenstrukturbibliothek, wie die Entwurfsprinzipien *Trennung von Schnittstelle und Implementierung*, *Imitations*-Beziehungen zwischen Klassen (ähnlich zu Konformität) und *konzeptionelle Abstraktion* eingesetzt werden können, um diese Anforderungen zu unterstützen.

Dabei führen die klaren Entwurfsprinzipien, insbesondere der disziplinierte Einsatz von Vererbung als Imitation und Techniken wie abgeleitete Methoden, Items (siehe Diskussion über LEDA), Vor- und Nachbedingungen und Klasseninvarianten, zu einem hohen Grad der internen und externen Wiederverwendung und zu einer guten Verständlichkeit.

Das Konzept der *Imitation*, das die Autoren anwenden, ist eine etwas schwächere Form der Kompatibilität als Konformität. Beim Einsatz von Imitation kann man innerhalb des Rumpfes einer Klasse A alle Vorkommen des Namens einer Klasse B sicher ersetzen durch den Namen einer Klasse C, die B imitiert. Es ist jedoch nicht erlaubt, nur ein einzelnes Vorkommen von B zu ersetzen.

Falls der Entwickler sich an diese Regel nicht hält, können Typfehler auftreten, d.h. die Anforderung an die Zuverlässigkeit ist hier verletzt. Der massive Einsatz der Mehrfachvererbung in der oben angesprochenen Bibliothek schränkt die Flexibilität der Bibliothek ein (keine flexiblen Brücken) und wird von vielen als kritisch beurteilt.

OBST

Das objektorientierte Datenbanksystem OBST [2] stellt eine kleine, aber nützliche Bibliothek von Datenstrukturen wie Mengen, Multimengen, Wörterbücher usw. zur Verfügung. Die klare Trennung von Schnittstelle und Implementierung erlaubt den Austausch von Implementierungsvarianten.

Durch die Anwendung des Entwurfsmusters *Brücke* wird dem Datentyp *Mapping* ermöglicht, während der Laufzeit zwischen seinen Implementierungsvarianten *List* und *Hashtable* hin- und herzuschalten. Dieser Austausch kann automatisch oder benutzergesteuert erfolgen. Diese Flexibilität, die jeweils angemessene Implementierungsvariante für verschiedene Applikationen wählen zu können, resultierte in Effizienzgewinnen bzgl. Laufzeit und Speicherplatz. Natürlich verliert man durch die zusätzliche Indirektion zwischen Datentyp (Abstraktion) und Implementierung wieder etwas an Effizienz.

KARLA

Beim Entwurf von KARLA, der *Karl*sruher Bibliothek von *A*lgorithmen und Datenstrukturen, die in SATHER-K geschrieben ist, liegt der Hauptaugenmerk auf dem Aspekt der Zuverlässigkeit. Dazu werden, wie in Abschnitt 4 und 5 diskutiert, sowohl die Konformitäts- als auch die Spezialisierungsbeziehung unterstützt.

Durch Spezifikation von Klasseninvarianten und Vor- und Nachbedingungen für Methoden, welche von einem entsprechenden Konfigurations- und Informationssystem (*skit*) ausgewertet werden, wird die Zuverlässigkeit weiter erhöht. Weiterhin erlaubt *skit* die automatische Generierung von statischen und flexiblen Brücken aus einer gegebenen abstrakten Klasse. Eigenschaftsdefinierende Klassen werden als Typschranken für die Elemente der Container-Klassen verwendet, so daß die Existenz benötigter Methoden statisch vom Übersetzer überprüft werden kann.

Zusammenfassung

Jeder der oben diskutierten Bibliotheken weist verschiedene Charakteristika auf: LEDA stellt sehr effiziente Einzelkomponenten bereit; Smalltalk bietet eine große Flexibilität an; die Datenstrukturbibliothek von OBST ist klein, aber flexibel; die Eiffel Bibliothek stellt eine große Funktionalität zur Verfügung; die Datenstrukturbibliothek, die gemäß den Prinzipien von *Coherent Design* entworfen wurde, unterstützt sehr gut Wiederverwendung und Erweiterbarkeit; in KARLA wird großer Wert auf die Zuverlässigkeit der enthaltenen Komponenten gelegt.

Leider vernachlässigen die meisten Bibliotheken die Anforderungen an die Zuverlässigkeit in weiten Teilen. Mit der wachsenden Anzahl von sicherheitskritischer Software wird die Bedeutung dieses Aspekts sicherlich zunehmen, und man wird diesem Punkt, der in KARLA besondere Unterstützung findet, mehr Aufmerksamkeit widmen.

8 Zusammenfassung und Ausblick

In diesem Papier haben wir mehrere Techniken zur Konstruktion zuverlässiger Bibliotheken vorgestellt. Dabei haben wir zwischen abstrakten und konkreten Klassen unterschieden, was zur Trennung von Schnittstelle und Implementierung führt. Wir haben gezeigt, daß zusätzlich zur *Benutzt*-Beziehung zwischen Klassen mindestens zwei weitere Beziehungen existieren, um sowohl Flexibilität als auch Effizienz zu erlauben. Diese Beziehungen — Konformität und Spezialisierung — wurden formal mittels Vor- und Nachbedingungen sowie Invarianten definiert. Dabei haben wir gesehen, daß Konformität zu maximaler Flexibilität bei der Benutzung von Klassen führt, da jede Klasse, die zu einer gegebenen Klasse K konform ist, an deren Stelle verwendet werden kann. Spezialisierung hingegen ermöglicht die Wiederverwendung von Code sowie eine

Effizienzsteigerung durch Benutzung von repräsentationsabhängigen Algorithmen.

Die Trennung zwischen Konformität und Spezialisierung wurde in objektorientierten Bibliotheken bisher so nicht getroffen. Dort existiert entweder kein im Sinne der Robustheit "sauberes" Vererbungskonzept, oder es wird nur eine der Beziehungen unterstützt. Das vorliegende Papier weist nach, daß solche Bibliotheken entweder inhärent unzuverlässig sind oder ihren Schwerpunkt einseitig entweder auf Flexibilität oder Effizienz legen (meistens ersteres).

Der Spielraum zwischen Flexibilität und Effizienz wurde untersucht mit dem Ziel, Anforderungen für die zuverlässige Benutzung von Implementierungen abstrakter Klassen anzugeben. Um maximale Flexibilität zu erreichen, sollten sowohl die Repräsentation einer Schnittstelle (also die Implementierung der Kernmethoden insgesamt) als auch einzelne abgeleitete Methoden zur Laufzeit ausgewechselt werden können. Zu diesem Zweck haben wir das Entwurfsmuster der flexiblen Brücke eingeführt. Wir haben gezeigt, daß in diesem Szenario nur repräsentationsunabhängige Algorithmen ausgetauscht werden dürfen, da ansonsten die Robustheit verletzt ist. Auf der anderen Seite erfordern effiziente Algorithmen häufig die Kenntnis der vorliegenden Repräsentation. Aus diesem Grund haben wir ferner statische Brücken definiert, die es zur Laufzeit erlauben, sowohl repräsentationsabhängige als auch repräsentationsunabhängige Algorithmen auszutauschen. Es steht daher dem Anwender frei, selbst durch die Wahl des Brückentyps festzulegen, ob ihm Flexibilität oder Effizienz wichtiger ist.

Dieses Papier entstand anhand der Erfahrungen, die wir mit KARLA gemacht haben, der *Karl*sruher Bibliothek von *A*lgorithmen und Datenstrukturen, die in SATHER-K geschrieben ist. Weil die erste Version von KARLA nur Konformität als Vererbungsbeziehung kannte, benutzten Anwender Konformität, um Wiederverwendung von Code zu betreiben. Der Versuch, Spezialisierungsbeziehungen durch Konformität auszudrücken, führte zu einer unnötig komplexen und dadurch unverständlichen Struktur der Bibliothek. Nachdem dieser Zustand erkannt war, wurden große Teile der Bibliothek neu entworfen und Spezialisierungsbeziehungen unterstützt. Seither sind keine größeren Änderungen der Struktur mehr erforderlich gewesen. Zur Zeit enthält KARLA ca. 200 Klassen, von denen etwa 80 abstrakte Klassen als Schnittstellen dienen, die auch untereinander durch Konformitäts- und Spezialisierungsbeziehungen Code-Wiederverwendung erlauben.

Die Experimente mit KARLA begannen vor 15 Monaten, indem wir Studenten im Rahmen eines Algorithmentechnik-Praktikums die Bibliothek zur Verfügung gestellt haben mit dem Ziel, daß diese die Bibliothek zunächst benutzen und später erweitern sollten. Unsere Beobachtung ist, daß Studenten schnell in der Lage sind, im konkreten Fall zu unterscheiden, ob ihre Klassen in einer Konformitäts- oder Spezialisierungsbeziehung zu Klassen aus der Bibliothek stehen. Ferner benutzen sie diese Konzepte sowie die Entwurfsmuster (Brücken, Adapter, Dekorateure, Strategien), um aus den Bausteinen, die die Bibliothek zur Verfügung stellt, eine Anwendung mit geforderten Charakteristika zusammenzustellen. Dies zeigt, daß es sich um natürliche und benutzbare Konzepte handelt. Wie bereits ausgeführt wurde, ist die automatische Generierung von Brücken möglich. Der Nachweis ist für statische Brücken in KARLA bereits geführt und wird derzeit auf dynamische Brücken erweitert.

Die bisherige Erfahrung hat die Nützlichkeit der in diesem Papier vorgestellten Entwurfskonzepte demonstriert. Wir sind sicher, daß die Fortsetzung unserer Experimente weitere Probleme aufgeworfen wird, deren Lösung uns neue Einsichten in Konzepte und Muster zum Entwurf zuverlässiger Bibliotheken erlauben wird. Insbesondere erhebt sich die Frage, ob die rein kovarianten (kontravarianten) Beziehungen zwischen Klassen (siehe Abschnitt 4) in der Praxis auftreten.

Literatur

[1] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 114–132, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[2] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer. Obst - an overview. Technical report, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, June 1992. FZI.039.1.

[3] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 1–15, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley, 1994.

[6] A. Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison-Wesley, May 1983.

[7] S. Näher K. Mehlhorn. *Algorithms, Software, Architectures, Information Processing 92*, volume 1, chapter Algorithm Design and Software Libraries: Recent Developments in the LEDA Project. Elsevier Science Publishers B.V., 1992.

[8] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on*

Programming Languages and Systems, 16(6):1811–1841, 1994.

[9] Kurt Mehlhorn and Stefan Näher. Leda — a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[10] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[11] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[12] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[13] Andreas Rüping, Franz Weber, and Walter Zimmer. Demonstrating coherent design: A data structure catalogue. In Raimund Ege, Madhu Singh, and Bertrand Meyer, editors, *Proceedings 11th TOOLS Conference*, pages 363–377. Prentice Hall, 1993.

[14] Heinz W. Schmidt and Richard Walker. Tof: An efficient type system for objects and functions. Technical Report TR-CS-92-17, Australian National University, November 1992.

A Beispiel: Graphen

Das untenstehende Beispiel wird im Papier durchgängig verwendet. Es illustriert die in diesem Papier behandelten Ideen und Konzepte. Das Beispiel ist in der imperativen objektorientierten Programmiersprache SATHER-K definiert. Es genügt weitestgehend, sich darunter eine Sprache wie Eiffel vorstellen. Der Hauptunterschied zu Eiffel besteht in der Trennung von polymorphen und monomorphen Typen. Polymorphe Typen werden durch ein vorangestelltes $-Zeichen gekennzeichnet. Die Klausel **subtype of** bezeichnet konforme Vererbung und nur diese Unterklassen werden bei der Polymorphie berücksichtigt. Die Klausel **specialization of** bezeichnet dann die Spezialisierung[7]. Typschranken werden entsprechend Abschnitt 5 verwendet. Die mit dem Schlüsselwort **abstract** eingeleiteten Klassendefinitionen sind im Sinne von Abschnitt 2 abstrakt und dienen ausschließlich als Schnittstellenbeschreibungen ihrer Implementierungen. Um die Klasseninvariante initial herzustellen, fordern wir, daß nach jedem Erzeugen eines Objekts eine Methode *init* aufgerufen wird.

Die Klasse $SET(T)$ wird als gegeben angenommen. Sie soll die üblichen Methoden zum Erzeugen, Einfügen, Löschen und Enthaltensein ($\in$ zur Verfügung stellen. Außerdem machen wir noch von Schnitt ($\cap$), Vereinigung ($\cup$) und Mengendifferenz ($\setminus$) Gebrauch. Ein Zustand nach der Ausführung einer Methode wird mit $'$ gekennzeichnet. Kanten von Graphen sollen der Anforderung genügen, daß zwei Attribute $from$ und to zur Charakterisierung ihrer zugehörigen Ecken existieren und dementsprechend die Gleichheit zweier Kanten so definiert ist, daß diese Attribute übereinstimmen. Wir erhalten also eine Klasse

```
(1) class EDGETYPE(VERTEX) is
(2)     from, to : VERTEX;
(3) end
```

als Typschranke. Anstatt $e.from$ und $e.to$ schreiben wir der Einfachheit halber e_1 und e_2. Jeder in der folgenden Definition benutzte Kantentyp muß mindestens diese beiden Attribute enthalten.

```
(1)  abstract class D_GRAPH(
(2)      VERTEX,
(3)      EDGE < EDGETYPE(VERTEX)) is
(4)  -- definiert gerichtete Graphen;
(5)  -- Invariante: ∀e ∈ E : e1, e2 ∈ V;
(6)
(7)  V : $SET(VERTEX) is deferred;
(8)      -- liefert die Eckenmenge
(9)
(10) E : $SET(EDGE) is deferred;
(11)     -- liefert die Kantenmenge
(12)
(13) init : SAME is deferred
(14)     -- erzeugt einen leeren Graph
(15)     -- Vorbedingung: true
(16)     -- Nachbedingung: res.V.empty ∧ res.E.empty
(17)
(18) addvertex(v : VERTEX) : SAME is deferred
(19)     -- fügt eine Ecke ein
(20)     -- Vorbedingung: true
(21)     -- Nachbedingung: res.V = V.ins(v) ∧ res.E = E
(22)
(23) addedge(e : EDGE) : SAME is deferred
(24)     -- fügt eine Kante ein
(25)     -- Vorbedingung: e1 ∈ V ∧ e2 ∈ V
(26)     -- Nachbedingung: res.V = V ∧ res.E = E.ins(e)
(27)
(28) delvertex(v : VERTEX) : SAME is deferred
(29)     -- löscht eine Ecke samt den adjazenten Kanten
(30)     -- Vorbedingung: true
(31)     -- Nachbedingung: res.V = V.del(v)∧
(32)     --          res.E = E \ {e ∈ E : e1 = v ∨ e2 = v}
(33)
(34) deledge(e : EDGE) : SAME is deferred
(35)     -- löscht eine Kante
(36)     -- Vorbedingung: true
(37)     -- Nachbedingung: res.V = V ∧ res.E = E.del(e)
(38)
(39) are_connected(v, w : VERTEX) : BOOL is
(40)     -- ergibt true falls es einen Pfad von v nach w gibt.
(41)     -- Abkürzung: v →* w
(42)     -- Vorbedingung: v ∈ V ∧ w ∈ V
(43)     -- Nachbedingung: res = true ↔ w = v∨
(44)     --          ∀e ∈ E : e1 = v ⇒ e2 →* w
(45)     ...
(46) end;
(47)
(48) is_acyclic : BOOL is
(49)     -- ergibt true falls der Graph azyklisch ist
(50)     -- Vorbedingung: true
(51)     -- Nachbedingung: res = true ⇔
(52)     --          ∄v, w ∈ V : v ≠ w ∧ v →* w ∧ w →* v
(53)     ...
(54) end;
(55)
```

[7] In der gegenwärtigen Version von Sather-K gibt es diese Klausel noch nicht. Stattdessen gibt es eine **include**-Klausel, mit der ausschließlich Quelltext wiederverwendet werden kann.

(56) $\cdots$;
(57) **end**;

Zeile (56) kann steht für beliebig viele weitere Methoden wie etwa für das Finden kürzester Pfade, starke Zusammenhangskomponenten etc. Alle Methoden außer den mit **deferred** gekennzeichneten können (evt. mit Hilfe der **deferred**-Methoden) implementiert werden. Für eine Implementierung müssen zumindest die **deferred**-Methoden definiert werden. Nachfolgend geben wir eine einfache Implementierung an, die direkt aus der Mengendefinition des ADT Graph abgeleitet werden kann.

(1) **class** $D\_GRAPH\_SET($
(2) $\quad VERTEX,$
(3) $\quad EDGE < EDGETYPE(VERTEX))$ **is**
(4) $--$ implementiert gerichtete Graphen direkt durch Mengen
(5) $--$ Invariante: $\forall e \in EE : e_1, e_2 \in VV$;
(6)
(7) **subtype of** $D\_GRAPH(VERTEX, EDGE)$;
(8)
(9) $VV : \$SET(VERTEX)$;
(10) $EE : \$SET(EDGE)$;
(11)
(12) $V : \$SET(VERTEX)$ **is** ;
(13) $\quad --$ liefert die Eckenmenge
(14) $\quad res := VV$;
(15) **end**;
(16)
(17) $E : \$SET(EDGE)$ **is**;
(18) $\quad --$ liefert die Kantenmenge
(19) $\quad res := EE$;
(20) **end**;
(21)
(22) $init : SAME$ **is**
(23) $\quad --$ erzeugt einen leeren Graph
(24) $\quad --$ Vorbedingung: *true*
(25) $\quad --$ Nachbedingung: $res.VV.empty \wedge res.EE.empty$
(26) $\quad --$ $\quad \wedge self' = res$
(27) $\quad VV := \#SET\_LIST(VERTEX).init$;
(28) $\quad EE := \#SET\_LIST(EDGE).init$;
(29) $\quad res := self$;
(30) **end**;
(31)
(32) $addvertex(v : VERTEX) : SAME$ **is**
(33) $\quad --$ fügt eine Ecke ein
(34) $\quad --$ Vorbedingung: *true*
(35) $\quad --$ Nachbedingung: $res.VV = VV.ins(v) \wedge res.EE = E$
(36) $\quad --$ $\quad \wedge self' = res$
(37) $\quad \cdots$;
(38) **end**;
(39)
(40) $addedge(e : EDGE) : SAME$ **is**
(41) $\quad --$ fügt eine Kante ein
(42) $\quad --$ Vorbedingung: $e_1 \in V \wedge e_2 \in V$
(43) $\quad --$ Nachbedingung: $res.VV = VV \wedge res.EE = EE.ins(e)$
(44) $\quad --$ $\quad \wedge self' = res$
(45) $\quad \cdots$;
(46) **end**;
(47)
(48) $delvertex(v : VERTEX) : SAME$ **is**
(49) $\quad --$ löscht eine Ecke samt den adjazenten Kanten
(50) $\quad --$ Vorbedingung: *true*
(51) $\quad --$ Nachbedingung: $res.V = V.del(v) \wedge$
(52) $\quad --$ $\quad res.E = E \setminus \{e \in E : e_1 = v \vee e_2 = v\} \wedge self' = res$
(53) $\quad \cdots$;
(54) **end**;
(55)
(56) $deledge(e : EDGE) : SAME$ **is**
(57) $\quad --$ löscht eine Kante
(58) $\quad --$ Vorbedingung: true
(59) $\quad --$ Nachbedingung: $res.V = V \wedge res.E = E.del(e)$
(60) $\quad --$ $\quad \wedge self' = res$
(61) $\quad \cdots$;
(62) **end**;
(63)
(64) **end**;

Zum Schluß definieren wir noch gerichtete azyklische Graphen

(1) **abstract class** $DAG($
(2) $\quad VERTEX,$
(3) $\quad EDGE < EDGETYPE(VERTEX))$ **is**
(4) $--$ definiert azyklische gerichtete Graphen;
(5) $--$ Invariante: $\forall e \in E : e_1, e_2 \in V \wedge is\_acyclic$;
(6)
(7) **specialization of** $D\_GRAPH(VERTEX, EDGE)$;
(8)
(9) $addedge(e : EDGE) : SAME$ **is deferred**
(10) $\quad --$ fügt eine Kante ein
(11) $\quad --$ Vorbedingung: $e_1 \in V \wedge e_2 \in V \wedge \neg e_2 \xrightarrow{*} e_1$
(12) $\quad --$ Nachbedingung: $res.V = V \wedge res.E = E.ins(e)$
(13)
(14) $topsort : \$LIST(VERTEX$ **is**
(15) $\quad --$ sortiert den gerichteten azyklischen Graph topologisch
(16) $\quad \cdots$;
(17) **end**;
(18)
(19) **end**;

Man beachte, daß die Methode *topsort* nur bei gerichteten azyklischen Graphen Sinn macht.