Dirk Riehle und Heinz Züllighoven

Späte Erzeugung

Zusammenfassung

Ein zentrales Konzept des objektorientierten Entwurfs ist die Bildung abstrakter Oberklassen, deren Zusammenspiel die Funktionalität einer Anwendung realisiert. Konkretisiert wird dies durch Unterklassen, die offengelassene Aspekte der Oberklassen ausfüllen und so die Erzeugung von Objekten ermöglichen. Um diese Objekte zu erzeugen, müssen die konkreten Klassen benannt werden. Mit dem hier vorgestellten Konzept der s*päten Erzeugung* lassen sich Objekte konkreter Unterklassen nur mit Kenntnis ihrer abstrakten Oberklassen erzeugen, was die Flexibilität der Softwarearchitektur erhöht und ihre Komplexität reduziert.

1 Einleitung

Ein wesentliches Qualitätsmerkmal des objektorientierten Entwurfs ist die Wiederverwendbarkeit von Problemlösungen. Standen zunächst Bausteinsammlungen einzelner effektiver Klassen, von denen unmittelbar Exemplare erzeugt werden konnten, im Vordergrund, so wurde rasch deutlich, daß die eigentliche Stärke der Objektorientierung in der Wiederverwendung von Konzepten liegt. Folglich wurden allgemeine Problemlösungen in abstrakten Klassen beschrieben, die dann für die einzelne Anwendung durch entsprechende Unterklassen konkretisiert wurden. In fachlichen Rahmenwerken werden komplexe Problemlösungen oder Dienstleistungspakete mittlerweile durch das Zusammenspiel mehrerer Klassen und die Verteilung von Verantwortung unter ihnen beschrieben. Die sich dabei ergebenden Klassenstrukturen werden Designmuster genannt [Coa92, GHJ+93]. Die meisten der an einem Designmuster beteiligten Klassen sind abstrakt, d.h. sie spezifizieren zwar die vollständige Schnittstelle, ermöglichen aber nicht das Erzeugen konkreter Objekte. Die offenen Stellen der Implementation werden von Unterklassen ausgefüllt, die so das Designmuster für eine bestimmte Anwendung maßschneidern und erzeugbar machen.

Objekte dieser Unterklassen unterscheiden sich nur in ihrer Implementation von den Vorgaben ihrer abstrakten Oberklassen, so daß die Information, zu welcher konkreten Unterklasse ein Objekt gehört, nur beim Erzeugen gebraucht wird. Anschließend werden diese Objekte polymorph über Bezeichner der Oberklassen angesprochen.

Durch Bekanntgabe des Namens einer konkreten Unterklasse wird nicht nur diese Klasse, sondern ihr Klassenbaum bis zur abstrakten Oberklasse öffentlich und damit festgeschrieben.

Dies kann durch s*päte Erzeugung* vermieden werden. Späte Erzeugung ermöglicht, Objekte ohne Wissen ihrer exakten Klasse zu erzeugen. Das Konzept, die notwendige Konstruktion sowie die Einsatzmöglichkeiten später Erzeugung schildert dieses Papier.

2 Späte Erzeugung

Späte Erzeugung ermöglicht das Erzeugen eines Objekts unter ausschließlicher Kenntnis seiner abstrakten Oberklasse. Der Klassenname des zu erzeugenden Objekts braucht nicht einmal *außerhalb seiner eigenen Klasse* im Quelltext benannt zu werden. Ein Beispiel:

Mit einem grafischen Editor sollen verschiedene geometrische Objekte bearbeitet werden. Im Rahmen der Konstruktion nach Werkzeugen und Materialien [BCS92, KGZ93] ist der grafische Editor mit seinen Materialien über die Aspektklasse `GrafischEditierbar` verbunden

In: 39. Internationales Wissenschaftliches Kolloquium Ilmenau.
Thüringen, Technische Universität Ilmenau, 1994. Seiten 3-9.

(Abb. 1). Alle grafisch editierbaren Objekte können vom Benutzer erzeugt werden. Dazu steht eine Knopfleiste zur Verfügung (Abb. 2).

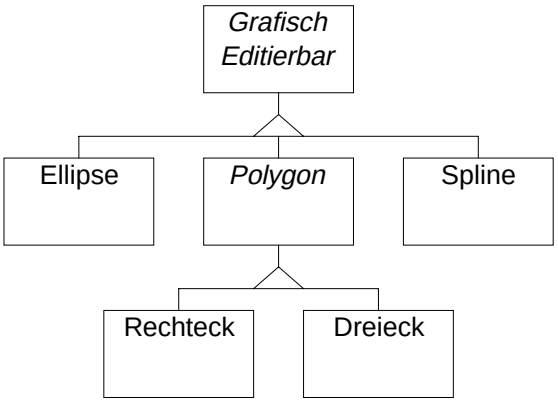

Abb. 1: Der Klassenbaum für geometrische Objekte mit der abstrakten Oberklasse `GrafischEditierbar` und konkreten Unterklassen.

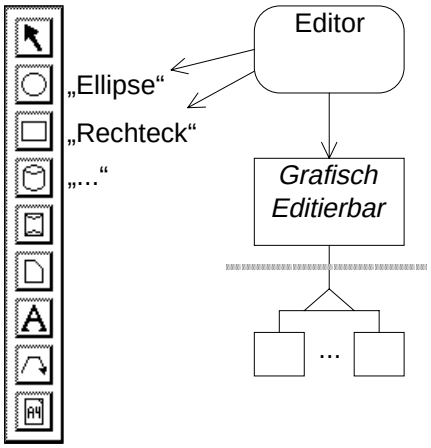

Abb. 2: Zur Auswahl eines geometrischen Objekts klickt der Benutzer auf einen Knopf, was vom System in ein konkretes grafisches Objekt umgesetzt wird.

Der Editor muß jedem Knopf der Leiste eine Klasse im Klassenbaum der geometrischen Objekt zuordnen, um auf Anforderung ein Exemplar dieser Klasse erzeugen zu können. Nach der Erzeugung wird üblicherweise das neue Objekt einem Bezeichner der Oberklasse `GrafischEditierbar` zugewiesen, weil deren Schnittstelle zur Manipulation der grafischen Objekte ausreicht. Der Name der konkreten Klasse wird bis auf das Erzeugen nicht benötigt. Doch für die erzeugende Operation muß der ganze Klassenbaum und seine Klassennamen offengelegt werden.

Nach dem Geheimnisprinzip sollte der Editor lediglich die Oberklasse aller grafischen Objekte kennen. Somit wäre der Klassenbaum bis auf die abstrakte Aspektklasse `GrafischEditierbar` verborgen (Abb. 2). Das Editorobjekt erhält vom Knopf eine Zeichenkette, die das editierbare Objekt spezifiziert. Die Umsetzung der Zeichenkette in ein grafisches Objekt soll ebenfalls vor ihm verborgen bleiben.

Die folgende Konstruktion löst das Problem. Jeder Klasse im Klassenbaum `GrafischEditierbar` wird eine eigene Operation `Create` hinzugefügt, deren Parameter Anforderungen an das zu erzeugende Objekt widerspiegeln. Die Create-Operation gibt ein neues Exemplar der Klasse zurück, sofern Objekte prinzipiell erzeugt werden können und die genannten Anforderungen für diese Klasse zutreffen. Andernfalls wird kein Objekt zurückgegeben.

Anforderungen an ein neues Objekt heißen im folgenden Klauseln, welche eine Spezifikation des zu erzeugenden Objekts darstellen. Diese Klauseln können aus mehreren Parametern oder, wie in unserem Beispiel, aus einer einzelnen Zeichenkette bestehen.

Die Create-Operationen der Klassen müssen so miteinander verbunden werden, daß alle Create-Operationen eines Klassenbaums nacheinander aufgerufen werden; so wird für jede Klasse des Baums, von ihr selbst geprüft, ob ein zur Klausel passendes Objekt geliefert werden kann.

Die Konstruktionsidee ist, zu jeder Klasse zur Laufzeit einen Prototyp (als Objekt) zu erzeugen, und sie der Klassenbaumstruktur nach untereinander zu verbinden. Die Benutzungsbeziehungen der Prototypen spiegeln die Vererbungsbeziehungen des Klassenbaums wieder (Abb. 3).

Anders als die nur von unten nach oben verlaufende Verknüpfungen der Vererbungsbeziehung besitzt jeder Prototyp eine Liste seiner Unterklassen-Prototypen. Wird nun die Create-Operation eines Prototypen aufgerufen, so prüft er anhand der übergebenen Klausel zuerst, ob ein Objekt der eigenen Klasse erzeugt werden soll. Wenn nicht, delegiert er die Anforderung sukzessive an die Create-Operationen seiner Unterklassen-Prototypen.

Der Editor wendet sich nur an die Oberklasse `GrafischEditierbar`, deren Create-Operation er mit einer den vom Knopf erhaltenen String versehenen Klausel als Parameter aufruft. Existiert irgendwo im Prototypenbaum eine zur Klausel passende Klasse, so erhält der Editor ein entsprechendes Objekt zurück. Für ihn ist bei der Erzeugung neuer Objekte nur die Oberklasse `GrafischEditierbar` sichtbar; der darunterliegende Klassenbaum und der Prototypenmechanismus bleiben verborgen. Die tatsächliche Erzeugung des angeforderten Objekts findet nur innerhalb der Create-Operation seiner Klasse statt.

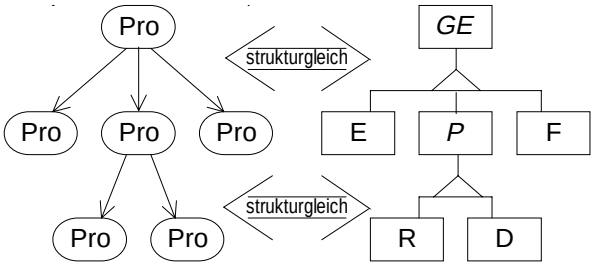

Abb. 3: Prototypen (Objekte) und die ihnen entsprechenden Klassen (*GE* für `GrafischEditierbar` usw.)

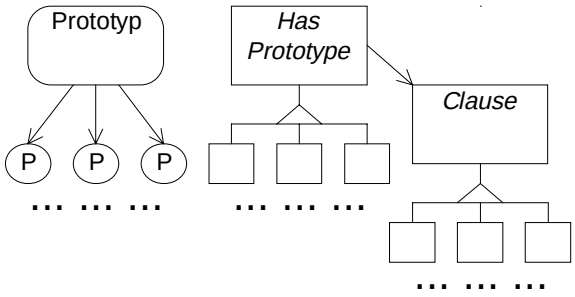

Abb. 4: Die beiden Oberklassen HasPrototype und Clause sowie der Prototypenbaum.

3 Späte Erzeugung im Allgemeinen

Bei *später Erzeugung* wird, ähnlich wie beim s*päten Binden* von Operationen, die Entscheidung, von welcher Klasse ein zu erzeugendes Objekt ist, erst zur Laufzeit gefällt. Im folgenden beschreiben wir die Strukturen und Beziehungen später Erzeugung allgemein.

3.1 Statische Struktur

Zu jeder Klasse, von der Objekte spät erzeugt werden sollen, wird zur Laufzeit des Systems ein Prototyp erzeugt. Er ist ein Repräsentant seiner Klasse. Wesentlich ist, daß die Prototypen zur Laufzeit in einer Benutzungshierarchie miteinander verbunden werden, welche dem Vererbungsbaum der Klassen entspricht, von denen Objekte spät erzeugt werden sollen. Dazu besitzt jeder Prototyp eine Liste von Unterklassen-Prototypen, in die sich Prototypen eintragen lassen können. Die Anmeldung geschieht von unten nach oben, d.h. Unterklassen-Prototypen melden sich bei ihren Oberklassen-Prototypen an. Das Prototyp-Prinzip wird bereits bei sog. Metaobjekt-Protokollen angewendet.

Wie können diese Prototypen dynamisch erzeugt werden können, ohne das ihre Klasse statisch in der Anwendung bekannt gemacht werden muß? Denn ohne die initiale Erzeugung und Verkettung der Prototypen können die zugehörigen neuen Klassen nicht in das System eingebunden und damit auch keine Objekte spät erzeugt werden.

Die notwendige Funktionalität wird durch die Klasse `HasPrototype` definiert. Klassen, deren Objekte spät erzeugt werden soll, müssen von ihr erben. Sie bietet folgende Operationen:

`isValidClause` überprüft, ob die übergebene Klausel für den *Klassenbaum* überhaupt Gültigkeit besitzt. Wenn nicht, kann das Traversieren des Prototypen-Baum bereits an dieser Stelle abgebrochen werden.

`Create` überprüft die übergebene Klausel auf Gültigkeit für die *Klasse*. Ist dies der Fall, so erzeugt der Prototyp ein Objekt seiner eigenen Klasse und gibt es zurück. Andernfalls wird null zurückgegeben.

`Delegate` wird aufgerufen, wenn die Klausel zwar Gültigkeit für den Klassenbaum (`isValidClause`), nicht aber für die Klasse (`Create`) besitzt. Der Prototyp reicht die Anfrage durch Iteration über die Unterklassenliste an deren Prototypen weiter.

Die Operationen von `HasPrototype` sind mit Klauseln der Klasse `Clause` parametrisiert. Unterschiedliche Klauseln werden durch Unterklassen von `Clause` ausgedrückt. So können bei Bedarf neue Klauselarten eingeführt werden (Abb. 4).

3.2 Dynamischer Ablauf

Soll ein Objekt einer Klasse spät erzeugt werden, so wendet sich der Kunde an eine Klasse und erhält ein Objekt zurück, das zwei Zusicherungen macht: Das Objekt bietet mindestens die Funktionalität der Klassenschnittstelle, bei der die späte Erzeugung begann, und es genügt den durch die Klausel verlangten zusätzlichen Anforderungen.

Bei später Erzeugung werden die drei Operationen `isValidClause`, `Create` und `Delegate` verwendet. Die Hilfsoperation `CreateLate` jeder Klasse prüft mittels `isValidClause`, ob die übergebene Klausel für den Klassenbaum Gültigkeit besitzt. Falls nicht, bricht die Operation ab. Falls ja, versucht sie mittels `Create` ein Objekt der aktuellen Klasse zu erzeugen. Gelingt dies, so wird das Objekt zurückgegeben und die Suche beendet. Wurde kein Objekt erzeugt, so wird mittels `Delegate` die Aufgabe an die Unterklassen-Prototypen weitergeleitet.

Auf diese Weise findet eine Tiefensuche über den gesamten Klassenbaum statt. Dabei wird jede Klasse befragt, ob sie ein Objekt erzeugen kann, das den Anforderungen genügend. Andere Strategien zur Traversierung des Klassenbaums lassen sich leicht realisieren.

3.3 Technische Diskussion

Die für späte Erzeugung notwendigen technischen Strukturen lassen sich gut in ein allgemeines Metaobjekt-Protokoll einbetten, welches für getypte Sprachen ohnehin benötigt wird. Bei der Implementation ergeben sich im wesentlichen drei Gesichtspunkte:

Klauseln sollten Doppeldeutigkeiten vermeiden, d.h. sie sollten sich wechselseitig ausschließen. Es muß sichergestellt werden, daß ein Prototyp als Stellvertreter für jede Klasse zur späten Erzeugung instantiiert wird, sobald die Klasse ins System eingebunden wird. In einer Sprache wie Smalltalk, in denen Klassen selbst Objekte sind, ist dies kein Problem. Das Klassenobjekt kann die Aufgaben des Prototyps übernehmen. In einer Sprachen wie C++ muß zu Hilfskonstruktionen gegriffen werden: Mit Hilfe klassenlokaler, sog. statischer Variablen, können bei Einbindung einer neuen Klasse vom Laufzeitsystem die Prototypen automatisch erzeugt werden.

Der dritte Gesichtspunkt betrifft das Zusammenbinden. Bei der späten Erzeugung gibt es keine explizite Referenz auf die einzubindenden Klassen. Heutige Binder fügen aber nur Objektcode in das ausführbare Programm ein, der referenziert ist. Deswegen muß dem Binder explizit mitgeteilt werden, welche Klassen eingebunden werden sollen. Dies muß immer auf einer Metaebene geschehen, also im Makefile oder einem eigenen Konfigurationswerkzeug.

4 Architekturmodellierung

Späte Erzeugung läßt sich in der Modellierung von Architekturen nicht nur auf Ebene einzelner Klassen anwenden. Vielmehr wird seine Stärke erst auf der Ebene größerer Softwarekomponenten deutlich.

Es liegt nahe, späte Erzeugung zu verwenden, um einen Typ mit mehreren Implementationen zu realisieren. Der Typ wird durch die Wurzel des Klassenbaums ausgedrückt, die verschiedenen Implementationen durch Unterklassen. Übliche Klauseln sind Auswahl- oder Performanzklauseln.

Subsysteme sind Einheiten fachlich definierter und abgegrenzter Funktionalität im Sinne von Dienstleistungen, welche durch das Zusammenspiel mehrerer Objekte realisiert werden und welche sich nach außen über eine einzelne Schnittstelle präsentieren [WWW90, Nag90]. Wird diese Schnittstelle durch einen Klassenbaum realisiert, deren Objekte als Aufhänger für unterschiedliche Subsysteme dienen, so kann über dieses Wurzel-/Schnittstellenobjekt das gesamte Subsystem spät erzeugt werden. Dies läßt sich entsprechend rekursiv erweitern.

Bis jetzt wurden die verschiedenen Klassenbäume über einzelne spät zu erzeugende Objekt in das System eingebunden. Wir haben aber bereits einleitend auf die wachsende Bedeutung von Designmustern hingewiesen, bei der der Zusammenhang mehrerer Objekte auf der Ebene abstrakter Oberklassen formuliert wird [Coa92, GHJ+93].

Läßt sich das Designmuster über ein einzelnes Objekt spät erzeugen, so braucht nichts verändert zu werden. Um aber ein komplexes Designmuster spät zu erzeugen, muß die Kapsel um den einzelnen Klassenbaum in eine Kapsel um die am Designmuster beteiligten Klassenbäume erweitert werden. Nur so können die am Designmuster beteiligten Unterklassen aufeinander Bezug nehmen und eine Spezialisierung ermöglichen, ohne daß die erweiterte Funktionalität von Unterklassen verlorengeht.

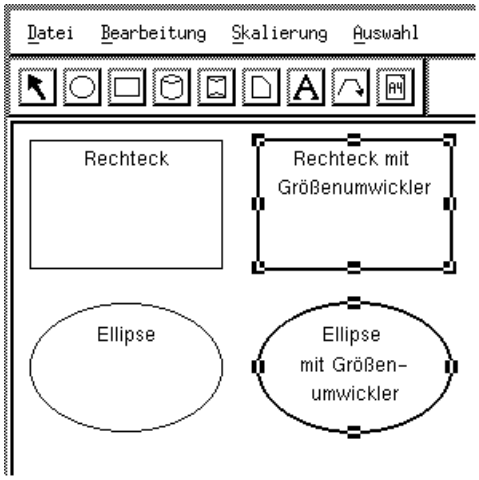

Abb. 5: Grafische Objekte in Größenumwicklern.

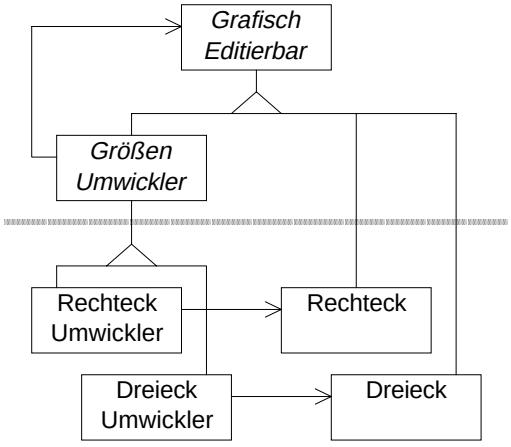

Abb. 6: Das Designmuster des (Größen-) Umwicklers (Wrapper).

Ein Beispiel (Abb. 5): In Grafikeditoren kann jedes dargestellte Objekt in seiner Größe verändern werden. Dazu wird z.B. ein das Objekt umfassendes Rechteck mit neun Anfaßpunkten dargestellt. Für andere Objekte (Dreieck, Ellipse) ist eine anderes umfassendes Objekt sinnvoll. Technisch wird dies durch einen Größenumwickler realisiert. Ein Größenumwickler legt einen Rahmen um ein grafisches Objekt und realisiert die Größenveränderung dieses Objekts. Für jede unterschiedliche Darstellung ist ein spezialisierter Größenumwickler notwendig (Abb. 5, 6). Die Erzeugung des jeweiligen Umwicklers hängt also vom grafischen Objekt ab. Diese Information wird über eine Abhängigkeitsklausel in die Create-Operation der Umwicklerklassen hereingereicht. Technisch steht in der Klausel für den Umwickler eine Referenz auf das zugehörige grafische Objekt. Die Create-Operation fragt die Klasse des grafischen Objektes ab.

Ein neu zu erzeugendes Objekt kann nicht nur von einem, sondern von beliebig vielen anderen Objekten abhängen. Ein komplexes Designmuster kann also auf zwei Arten erzeugt werden. Zum einem kann ein „Wurzel“-Objekt spät erzeugt werden, welches alle weiteren Objekte erzeugt. Oder es wird ein Objekt spät erzeugt, auf das sich die anderen zu erzeugenden Objekt beziehen können. Im ersten Fall wird das Designmuster versteckt, im zweiten Fall sind die Beziehungen zwischen den beteiligten Klassen sichtbar.

5 Anwendung der späten Erzeugung

Wir beschreiben nun ein Beispiel aus unserer Praxis, um die Mächtigkeit der späten Erzeugung zur Strukturierung im Großen zu verdeutlichen. In einem Anwendungsrahmen für die bereits erwähnte Konstruktion mit Werkzeugen und Materialien [Rie93] bestehen Softwarewerkzeuge aus einem interaktiven und einem funktionalen Teil. Die interaktive Komponente besteht im einfachsten Fall aus dem Objekt einer von `IkBasis` abgeleiteten Klasse (Ik = Interaktionskomponente), die Funktionskomponente aus einem Objekt einer von `FkBasis` abgeleiteten Klasse.

Werkzeuge und Materialien sind in eine Umgebung eingebettet, welche u.a.. die Koordinierung und Integration der Werkzeuge vornimmt. Die Umgebung ist ihrerseits ein Objekt, welches Werkzeuge erzeugt und verwaltet. Abb. 7 zeigt die genannten Klassenbeziehungen sowie zwei Werkzeuge (Editor und Browser).

Die Umgebung kennt nur die Klassen `FkBasis` und `IkBasis`. Beide können spät erzeugt werden. Die Funktionskomponente wird über den Werkzeugnamen, der üblicherweise vom Benutzer aus einem Menü ausgewählt wurde, spät erzeugt. Die dazu passende Interaktionskomponente wird mit einer Abhängigkeitsklausel, die sich auf die Funktionskomponente bezieht, spät erzeugt.

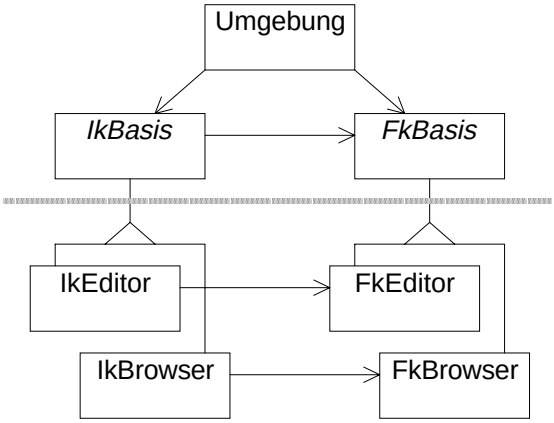

Abb. 7: Klassenbaum von Werkzeugen in einer Umgebung.

Somit ist die Kapselung der konkreten Werkzeuge im Klassenbaum unterhalb von `IkBasis` und `FkBasis` möglich. Dies ist vor allem dann nützlich, wenn Umgebungen dynamisch um neue Werkzeuge erweitert werden sollen.

6 Verwandte Arbeiten

Die Autoren der ET++ Klassenbibliothek verwenden zur Lösung der beschriebenen Probleme ein verwandtes Konzeptes. Zu jeder normalen Klasse gibt es ein Objekt der Klasse `Class`, welches als Klassendeskriptor bezeichnet wird [WGM89, Gam92] und klassenbezogene Information liefern kann. Der Klassendeskriptor kennt den Klassendeskriptor der Oberklasse sowie eine prototypische Instanz der normalen Klasse. Über den Klassendeskriptor können Objekte der normalen Klasse erzeugt werden, indem Kopien der prototypischen Instanz angelegt werden. Die möglichen Spezifikationen sind durch die Schnittstelle der Klasse `Class` festgelegt und somit für alle Klassen gleich. In der ET++ werden neben dem üblichen Metaobjekt-Protokoll die Klassendeskriptoren hauptsächlich für Objekt-I/O und zum anonymen Kopieren (Klonen) von Objekten verwendet. Alle Klassendeskriptoren werden vom `ClassManager` gesammelt und stehen an zentraler Stelle für die Aufgaben bereit.

Der Unterschied zu der von uns vorgeschlagenen Konstruktion besteht darin, daß die Klassendeskriptoren keinen Klauselbegriff und somit keinen Spezifikationsbegriff kennen. Zudem ermöglichen die Klassendeskriptoren nur eine Suche von den Blättern zur Klassenbaumwurzel und nicht umgekehrt. Letzteres aber ist für späte Erzeugung im hier vorgestellten Sinne unverzichtbar.

Anders geht Coplien an die Sache heran [Cop92]. Das als *autonomous generic exemplar idiom* bezeichnete Konzept schränkt den von Gamma/Weinand vorgesehenen `ClassManager` auf eine einzelne Klasse ein, welche in einer Liste die *Exemplar* genannten prototypischen Instanzen ausgewählter Klassen sammelt. Diese Exemplare werden über einen speziellen Konstruktor erzeugt, womit es also keine Metaklassen gibt. Dieser Vorschlag ist dem Konzept der Prototypen

vergleichbar. Über die Liste der Exemplare wird iteriert und sie werden befragt, ob die übergebene Spezifikation für sie gültig ist.

Copliens Ansatz ist dem hier vorgestellten in weiteren Aspekten ähnlich: Es gibt klassenbaumabhängige Spezifikationen und neue Klassen sind unproblematisch einfügbar. Aber es gibt Unterschiede: Da es nur eine Liste gibt, ist die Hierarchie der Exemplare einstufig. Die Exemplare werden nicht nach entsprechend der Klassenhierarchie verknüpft. Dadurch wird die Laufzeit linear statt logarithmisch. Der Klassenbaum wird als abgeschlossen betrachtet, was die fortschreitende Spezialisierung der Spezifikation und seine inkrementelle Weiterentwicklung verhindert. Zudem werden die Spezifikationsmöglichkeiten nicht in einer Klausel verpackt, sondern hart in der Klassenschnittstelle durch entsprechende Operationen kodiert.

7 Zusammenfassung

Späte Erzeugung konzentriert den Blick auf die im Entwurf jeweils zu verwendenden Konzepte und Abstraktionen. Unser Ansatz ermöglicht, die Erzeugung konkreter Objekte einer Abstraktion polymorph zu machen und überwindet damit eine wesentliche Hürde der universellen Verwendbarkeit von Polymorphie unter Beibehaltung des Typkonzepts.

Konsequenz unseres Ansatzes ist die softwaretechnische Entkopplung der Konkretisierung verschiedener Abstraktionen voneinander. Die Konkretisierung einer Abstraktion muß keine Information von den Konkretisierungen der verwendeten Abstraktionen besitzen.

Damit ist der Schritt zu einer weitgehenden Austauschbarkeit von Realisierungen und Implementationen von Abstraktionen getan, ohne daß dies im restlichen System Änderungen nach sich ziehen muß.

Literatur

BCS92 Reinhard Budde, Marie-Luise Christ-Neumann and Karl-Heinz Sylla. “Tools And Materials: An Analysis and Design Metaphor”. Tools-7, *Technology of Object-Oriented Languages and Systems, Europe ‘92*. Ed. G. Heeg, B. Magnusson and B. Meyer. Prentice-Hall, 1992. 135-146.

Coa92 Peter Coad. “Object-Oriented Patterns”. *Communications of the ACM* 35, 9 (September 1992): 152-159.

Cop92 James O. Coplien. *Advanced C++: Programming Styles and Idioms.* Reading, Massachusetts: Addison-Wesley, 1992.

Gam92 Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++*. Berlin, Heidelberg: Springer-Verlag, 1992.

GHJ+93 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. “Design Patterns: Abstraction and Reuse of Object-Oriented Design”. ECOOP-93, *LNCS- 707*, 1993. 406-431.

KGZ93 Klaus Kilberth, Guido Gryczan und Heinz Züllighoven. *Objektorientierte Anwendungsentwicklung – Konzepte, Strategien, Erfahrungen*. Braunschweig, Wiesbaden: Vieweg-Verlag, 1993.

Nag90 Manfred Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Berlin, Heidelberg: Springer -Verlag, 1990.

Rie93 Dirk Riehle. *Dokumentation zur FIAK-Bibliothek.* Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993.

WGM89 André Weinand, Erich Gamma and Rudolf Marty. “Design and Implementation of ET++, a Seamless Object-Oriented Application Framework”. *Structured Programming* 10, 2 (Juni 1989): 63-87.

WWW90 Rebecca Wirfs-Brock, Brian Wilkerson und Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

Autoren:
Dirk Riehle
Heinz Züllighoven
{riehle, zuelligh}@informatik.uni-hamburg.de
Universität Hamburg, FB Informatik, AB Softwaretechnik
Vogt-Kölln-Str. 30
D-22527 Hamburg