Prototyping in einer objektorientierten Entwicklungsumgebung*

Gerti Kappel
Oscar Nierstrasz

University of Geneva
Centre Universitaire d'Informatique
12 Rue du Lac, CH-1207 Geneva, Switzerland
E-mail: {kappel, oscar}@cui.unige.ch
(auch ...@cui.uucp, ...@cgeuge51.bitnet)

Kurzfassung

Prototyping von Software ist eine Entwurfstechnik, die durch einen zyklischen Entwurfsprozeß und durch die rasche Entwicklung von operationalen Systemen "bessere" Software, im Sinn von zuverlässiger und den Anforderungen entsprechend, erzeugen hilft. Objektorientierte Programmierung ist eine Programmiertechnik, die durch die Wiederverwendung bereits existierender Softwareobjekte ausgezeichnet ist. Die wichtigsten Mechanismen in objektorientierten Sprachen zur Wiederverwendung sind die (mehrfache) Vererbung und die Instantiierung von Objektklassen. Dabei zeigt sich, daß die objektorientierte Programmierung nicht nur verschiedene in der Literatur bekannte Prototypingansätze unterstützt, sondern auch daß Prototyping ein inhärentes Konzept im objektorientierten Software Lifecycle darstellt. Welche Werkzeuge und Entwicklungsumgebungen benötigt werden, um einen objektorientierten Prototypingansatz Realität werden zu lassen, wird diskutiert.

1 Erwartungen an einen objektorientierten Prototypingansatz

Bei der Entwicklung von Applikationssystemen konnten zwei grundlegende Erfahrungen gemacht werden. Erstens, es ist oft nicht möglich, die genauen Anforderungen an das Applikationssystem von Anfang an festzulegen. Zweitens, während des Einsatzes des Applikationssystems ändern sich sehr oft seine Umweltbedingungen, und damit die Anforderungen an das System. Mit dem traditionellen Software Lifecycle ("waterfall" Modell), der auf einer Sequenz von Entwicklungsphasen (Anforderungsanalyse, Systemdesign, Implementierung und Test, Wartung) aufbaut, kann diesen Erfahrungen nicht generell begegnet werden.

Ein Ziel von Prototyping ist es, besser und schneller die richtigen Anforderungen an ein Applikationssystem zu erkennen. Doch damit kann erst das erste Problem gelöst werden. Damit auch das

*Eine gekürzte Version dieses Artikels erschien in Handbuch der Modernen Datenverarbeitung, Nr.145, Forkel-Verlag, Januar 1989, S. 116-125.

zweite Problem zumindest teilweise in den Griff zu bringen ist, brauchen wir adaptierbare, d.h. anpaßbare Systeme. Mit anderen Worten, die Prototypingmöglichkeiten sollten nicht nur während der Erstentwicklung bestehen, sondern während des gesamten Lifecycles des Applikationssystems. Dieses sogenannte "evolutive Prototyping" wird von bestehenden Entwicklungsumgebungen wenig unterstützt.

Objektorientiertes Programmieren ist eine Programmiertechnik, die durch die Wiederverwendung bereits existierender Softwareobjekte ausgezeichnet ist. Dazu kommt, daß die atomaren Bausteine (Objektklassen) mit eindeutigen Schnittstellen definiert sind, sodaß ein Austausch von, beziehungsweise eine Erweiterung um Funktionalität erleichtert wird. Evolutives Prototyping kann als eine Charakteristik der objektorientierten Applikationsentwicklung angesehen werden, und objektorientiertes Prototyping heißt daher – für uns – objektorientierter Applikationsentwurf. Diese Behauptung zu erklären und zu begründen ist Ziel des Artikels.

Im folgenden geben wir einen Ueberblick über Prototyping im allgemeinen. Wir gehen auf die Zusammenhänge zwischen Prototyping und objektorientierter Applikationsentwicklung ein, und stellen einen objektorientierten Software Lifecycle vor, der die Entwicklung von adaptierbaren Systemen, und damit evolutives Prototyping unterstützt.

2 Prototyping

2.1 Arten von Prototyping

Eine Klassifikation von Prototypingansätzen wird in [9] vorgestellt. Wir verwenden diese Klassifikation und zeigen im nächsten Kapitel, wie objektorientiertes Prototyping hier einzuordnen ist.

Basierend auf Zielsetzungen, d.h. Erwartungen an den Prototyp, werden drei Ansätze unterschieden, nämlich das *explorative*, das *experimentelle*, und das *evolutive* Prototyping.

- *exploratives Prototyping:* hilft bei der Klärung von Benutzerwünschen in der Analyse- und Designphase. Dabei soll möglichst rasch eine lauffähige Version von Teilen des spezifizierten Systems dem Benutzer zur Verfügung gestellt werden. Der Benutzer soll in wiederholten "verwende und korrigiere" Phasen seine eigenen Anforderungen an das zukünftige System erkennen. Dadurch wird es auch dem Designer erleichtert, klare Vostellungen der Anforderungen an das Softwaresystem zu bekommen. Exploratives Prototyping bezieht explizit den Benutzer in den Entwicklungsprozeß mit ein, mit den möglichen Seiteneffekten, daß sowohl die Akzeptanz des Benutzers gegenüber der neuen Arbeitsumgebung, als auch seine Arbeitszufriedenheit gesteigert werden.
- *experimentelles Prototyping:* dient zur Evaluierung (Bewertung) verschiedener Lösungsansätze in der Designphase. Das Ziel dieses Prototypingansatzes ist die beste Lösung für die Implementierung des Softwaresystems zu finden.
- *evolutives Prototyping:* bei diesem Prototypingansatz wird der gesamte Softwareentwicklungsprozeß beeinflußt. Das Applikationssystem wird aus einzelnen – so weit als möglich unabhängigen

– Teilen aufgebaut werden. Jeder Teil wird in einem zyklischen Entwicklungsprozeß entworfen und verbessert. Dabei wird nicht mehr zwischen der Entwicklung des Prototyps und des Endproduktes unterschieden. Basierend auf der Erfahrung, daß die Anforderungen an ein mittelgroßes Softwaresystem schnelleren Aenderungen unterworfen ist, als eine geeignete Software (auch im Idealfall) hergestellt werden kann, ist die Idee des zyklischen Entwurfs mit Feedback-Möglichkeiten nicht nur erstrebenswert, sondern oft eine notwendige Voraussetzung für die Erstellung eines Systems. Jede "verbesserte" Version des Prototyps spiegelt die Weiterentwicklung des Softwaresystems wider.

Im Gegensatz zum letztgenannten Ansatz, wird im explorativen und im experimentellen Prototyping nicht auf die "physische" Wiederverwendung des Prototyps im Endprodukt Wert gelegt. Bei diesem "Wegwerf"-Prototyping gehen die Erfahrungen, die bei der Prototyperstellung gemacht wurden, in das Produktsystem ein; dieses wird aber in der Regel von Beginn an neu implementiert. Die drei Prototypingansätze stellen sich nicht ausschliessende Alternativen dar.

2.2 Anforderungen an Prototyping

Für jeden Prototypingansatz können spezielle Anforderungen an seine Methode und die benutzten Werkzeuge aufgestellt werden. Da das evolutive Prototyping die anderen Ansätze auch beinhalten kann, sind die Anforderungen sich nicht gegenseitig ausschliessend, sondern als sich ergänzend zu verstehen. Gemäß den drei Prototypingansätzen unterscheiden wir drei Gruppen von Anforderungen:

- *Schnelle Entwicklung:* Für einen explorativen Prototypingansatz ist der raschest mögliche Aufbau eines operationalen Systems die wichtigste Anforderung. Um dieses Ziel zu verwirklichen, werden Werkzeuge mit "vorgefertigter Funktionalität" benötigt. Die schnelle und billige Konstruktion eines Prototyps durch den Designer oder den Endbenutzer selbst, ermöglicht bereits praktische Erfahrung während die Applikation entworfen wird.
- *Evaluierung:* Die Evaluierung der verschiedenen Funktionen des Prototyps ist Voraussetzung für die nächste, verbesserte Version. Sowohl im explorativen als auch im experimentellen Prototyping brauchen wir Werkzeuge zur Evaluierung . Für eine erfolgreiche Evaluierung benötigen wir:
  - Demonstrierbarkeit und Verständnis: die verschiedenen Möglichkeiten des Prototyps sollen einfach für den Benutzer demonstrierbar sein. Dabei sollten Visualisierungswerkzeuge zum Verständnis der Struktur des Prototyps als auch seiner Funktionalität beitragen.
  - Flexibilität und Aenderungsmöglichkeit: der Prototyp soll einfach änderbar bzw. um weitere Funktionalität erweiterbar sein.
- *Inkrementeller Entwurf und Weiterentwicklung:* ist die wichtigste Anforderung an einen evolutiven Softwareentwicklungsprozeß. Dazu benötigen wir:
  - Wiederverwendungsmöglichkeit: der Prototyp sollte aus Teilen aufgebaut werden, die in verschiedenen Applikationen wiederverwendet werden können. Diese Teile werden

in einer allgemein zur Verfügung stehenden Bibliothek verwaltet. Wird ein Prototyp erzeugt, dessen Wiederverwendung wahrscheinlich ist, so soll dieser neue "Teil" in der vorhandenen Bibliothek abgespeichert werden.

– Erweiterungsmöglichkeit: sowohl der Prototyp als auch die Prototypentwicklungsumgebung, d.h. die verwendeten Werkzeuge, müssen neuen Anforderungen angepaßt und weiterentwickelt werden. Dies soll jedoch in einer integrierten Arbeitsumgebung erfolgen.

2.3 Werkzeuge zur Unterstützung des Prototyping

Abhängig von den Anforderungen an das Prototyping stehen unterschiedliche Werkzeuge zur Verfügung. Auch hier gilt, daß ein evolutives Prototyping eine Integration verschiedener Werkzeuge verlangen wird.

Für einen besseren Ueberblick behalten wir die Drei-Teilung auch bei den Werkzeugen bei. Wir unterscheiden "Toolkits" für das explorative Prototyping, "Very High Level Programming" (VHLP) Werkzeuge für das explorative und experimentelle Prototyping, und integrierte Programmier- und Entwicklungsumgebungen für das evolutive Prototyping.

2.3.1 Benutzerschnittstellen Toolkits

Mit hochauflösenden pixelorientierten Bildschirmen ist die Konstruktion von benutzerfreundlichen Schnittstellen nicht nur möglich geworden, sondern – quasi als Konsequenz – ein bedeutender Teil in einer Applikation. Parameterisierte Toolkits zur Erstellung von Benutzerschnittstellen sind daher ein wichtiges Werkzeug im explorativen Prototyping. Ein Vergleich verschiedener Benutzerschnittstellen Toolkits wird in [5] vorgestellt.

2.3.2 VHLP

Mit "Very High Level Programming" bezeichnen wir Sprachen und Werkzeuge, die eine anwendungsnahe Programmierumgebung zur Verfügung stellen. Darunter verstehen wir mächtige, vordefinierte Sprachkonstrukte zur Erstellung (vor allem) applikationsabhängiger Programme, graphische Schnittstellen zur Programmentwicklung, und Programmiermodelle, die eine realitätsnahe Spezifikation und Programmierung erlauben. Es können die folgenden Ansätze unterschieden werden:

- *Sprachen der 4.Generation (4GL)* werden hauptsächlich in (relationalen) Datenbankapplikationen verwendet [12, 19]. Die wichtigsten Merkmale von 4GL sind ihre Integration mit einem Datenbanksystem, vorgefertigte Funktionalität für datenorientierte Applikationen, und Schnittstellen, die auch einem Endbenutzer das Arbeiten mit 4GL ermöglichen sollen. Prototyping mit 4GL kann in den meisten Fällen als evolutives Prototyping angesehen werden, weil der erzeugte "Prototyp" auch als Produktionssystem verwendet wird. In [17] wird auf diesen Aspekt näher eingegangen. In [19] wird ein Ueberblick über existierende 4GL gegeben.

- *Visuelle Programmierung*, auch Graphische Programmierung genannt, basiert auf dem Entwicklungsparadigma, ein *Programm* nicht zu schreiben, sondern *graphisch zu entwerfen*. Dieses beinhaltet die Visualisierung von Datenstrukturen, Datenfluß, und Kontrollfluß [6]. Visuelle Programmierung unterstützt sowohl exploratives als auch evolutives Prototyping. Zwei verschiedene Sprachansätze sind zu nennen. Erstens, applikationsneutrale graphische Programmiersysteme, wo Programme durch Zusammenfügen von graphischen Repräsentationen (Ikonen) von Kontrollstrukturen und Datenbeschreibungen entwickelt werden. In diesem Fall spricht man auch von graphischen Editiersystemen, die in der Regel auf existierenden, textorientierten Sprachen aufbauen. Zweitens, visuelle Simulationssprachen, die für spezielle Applikationen entworfen werden.

- *Programming By Example (PBE)*, d.h. "Programmierung durch Spezifikation von Beispielen", wurde für Nichtprogrammierer entwickelt, um selbst (wenn auch) kleine Applikationen entwerfen zu können. Ein sehr guter Vergleich verschiedener Systeme wird in [24] durchgeführt.

  Obwohl PBE Systeme noch nicht in der Praxis als Prototypingwerkzeuge verwendet werden (die meisten Systeme befinden sich noch in der Entwicklung), ist es wert sie zu erwähnen. Erstens, weil sie sich für einen explorativen Prototypingansatz eignen würden, und weil sie vielversprechende Konzepte beinhalten, die in abgewandelter Form in visuellen Programmiersprachen anzutreffen sind. Der Benutzer macht seine ersten Erfahrungen mit dem System, indem er es selbst zumindest teilweise entwirft.

- *Deklarative Sprachen:* In deklarativen Sprachen wird das gewünschte Ziel und seine Nebenbedingungen beschrieben (das "Was"), jedoch nicht wie es gelöst werden soll (das "Wie"). Prolog, eine logische Programmiersprache, kann als Hauptvertreter dieser Richtung angesehen werden [25].

2.3.3 Programmier- und Entwicklungsumgebungen

Unabhängig von einem speziellen Prototypingansatz sind Sprachen und Werkzeuge notwendig, um den Programmier- und Entwicklungsprozeß zu vereinfachen, und ihn dadurch effizienter und produktiver zu machen. Eine offene Frage ist, wie können die Ziele eines evolutiven Prototypingansatzes mithilfe solcher Werkzeuge realisiert werden. Die erste Antwort ist, die Werkzeuge alleine werden nicht ausreichen. Wir brauchen eine Softwareentwicklungsmethode, die die Idee des Prototyping unterstützt. Die zweite Antwort ist, wir brauchen eine integrierte Sprach- und Werkzeugumgebung, auf der diese Softwareentwicklungsmethode aufbauen kann. Erste Ansätze dazu werden in [1, 23] vorgestellt.

2.4 Analyse

In Tabelle I fassen wir die besprochenen Ansätze, Anforderungen, und Werkzeuge zusammen.

Ansätze	**Anforderungen**	**Werkzeuge**
• explorativ • experimentell • evolutiv	• schnelle Entwicklung • Evaluierung • inkrementeller Entwurf und Weiterentwicklung	• UI Toolkits, • VHLP • Programmier- und Entwicklungsumgebungen

Tabelle I

Unter der Annahme, daß Applikationssysteme sich ändernden Umweltbedingungen angepaßt werden müssen, ist evolutives Prototyping ein möglicher Weg dazu. Dieses soll die Charakteristika des explorativen und experimentellen Prototypings (zum Vermeiden von Analyse- und Designfehler und zum Experimentieren mit verschiedenen Lösungen) ebenfalls beinhalten. Die Prototypversion i stellt dann die aktuelle Produktversion zum Zeitpunkt i dar.

Wir glauben, daß ein objektorientierter Entwicklungsansatz dazu beiträgt, diesem "Idealziel" näherzukommen. Der große Vorteil des objektorientierten Ansatzes ist die integrierte Entwicklungsumgebung, in welcher das Applikationssystem sowohl entworfen wird, als auch letztendlich eingebettet ist. Dies wird vor allem durch ein einheitliches Modell (das Objektmodell) für die Werkzeuge und die zu entwickelnde Applikation erreicht.

3 Objektorientiertes Prototyping

Objektorientierte Programmierung verspricht die rasche Entwicklung von Applikationen und deren Anpassung an geänderte Anforderungen durch die Konstruktion aus wiederverwendbaren Softwareobjekten. Die traditionelle Top-down Programmierung kann daher für die objektorientierte Programmierung nicht ausschließlich verwendet werden. Wir brauchen einen Software Lifecycle, der Bottom-up Entwicklung ebenso beinhaltet.

3.1 Der objektorientierte Software Lifecycle

Der historische Ursprung objektorientierter Programmierung ist die Simulationssprache SIMULA [3]. Mit der Entwicklung von Smalltalk [10] wurde klar, daß das objektorientierte Entwicklungsparadigma nicht nur zur Simulation von realen Objekten (z.B. Autos in einer Verkehrssimulation) verwendet werden kann, sondern neue Möglichkeiten im Entwurf von Softwaresystemen bietet. Diese Möglichkeiten sind eng mit den wichtigsten Charakteristika von objektorientierten Sprachen verbunden. Obwohl es verschiedene objektorientierte Sprachansätze gibt, wird derzeit als Minimalanforderung an eine objektorientierte Sprache akzeptiert, daß sie Objekte, Objektklassen, und Vererbung von Objektklassen [26] unterstützen soll. Diese Begriffe können folgendermaßen definiert werden:

1. Ein *Objekt* besteht aus einer *sichtbaren Schnittstelle* (eine Menge von Operationen, die auf dem Objekt ausgeführt werden können) und einer *unsichtbaren Repräsentation* (eine Menge von

Instanzvariablen und eine Menge von Methoden, die die Implementierung der Operationen darstellen). Dieses Konzept wird *Information Hiding* genannt. *Kapselung* bezeichnet die gemeinsame Darstellung von Instanzvariablen und Methoden auf diesen Variablen.

2. Eine *Objektklasse* spezifiziert das gemeinsame Verhalten (Instanzvariable und Methoden) einer Menge von Objekten. Es können beliebig viele Objekte einer Objektklasse kreiert werden.

3. *Vererbung von Objektklassen* wird benutzt, um neue Objektklassen, genannt Subklassen, zu spezifizieren. Diese Subklassen erben das Verhalten existierender Klassen, und können ihr Verhalten durch zusätzliche Definition von Instanzvariablen und Methoden erweitern.

Objektorientierte Sprachen variieren in der Funktionalität, die sie zusätzlich anbieten, z.B., strenge Typenbindung, mehrfache Vererbung, generische (parameterisierte) Klassendefinition, und persistente Objekte.

Ein wichtiger Aspekt, den objektorientierte Sprachen zur Verfügung stellen, ist die Art und Weise, wie mithilfe von Objekten programmiert werden kann. Dabei ist nicht nur von Bedeutung, daß ein objektorientierter Entwurf die Strukturierung der jeweiligen Applikation unterstützt, sondern auch, daß Objekte eine gute Basis für einen *synthetischen Programmieransatz* darstellen. Synthetisches Programmieren (Bottom-up Programmieren) baut aus vorhandenen Softwareteilen neue, komplexere Applikationen auf. Dieser Programmierstil, und im generellen dieser Entwicklungsansatz, ist eine wesentliche Voraussetzung, wenn ein evolutiver Softwareentwicklungsprozeß, und damit evolutives Prototyping realisiert werden soll.

Smalltalk unterstützt diesen Programmentwurfsstil, indem eine Bibliothek von einigen hundert Objektklassen zur Verfügung gestellt wird, aus denen der Programmierer seine jeweilge Applikation "zusammenbauen" kann. Ein "Browser" unterstützt das Finden von Objektklassen, indem er das Navigieren durch die Hierarchie der Objektklassen ermöglicht. Wie wichtig diese Idee der Wiederverwendung existierender Funktionalität, d.h. existierender Objektklassen, ist, wird am besten von Brad Cox ausgedrückt, wenn er schreibt:

> *"These reusable components are called Software-ICs to emphasize the parallel with the way hardware engineers build circuits from a stockroom of generic, reusable silicon chips."* [7], S. iv

Wiederverwendung als zentrales Thema in objektorientierten Sprachen wird auch durch die Tatsache bestärkt, daß vermehrte Anstrengungen in sämtlichen objektorientierten Sprachen unternommen werden, um erstens Bibliotheken mit wiederverwendbaren Objektklassen aufzubauen, und zweitens bessere Sprachkonstrukte zur Verfügung zu stellen, die das Wiederverwenden von Objektklassen und die inkrementelle Weiterentwicklung (Modifikation) von Objektklassen unterstützen. Dazu gehört auch das Prinzip der Kapselung, d.h. Unterscheidung von sichtbarer Schnittstelle und unsichtbarer Repräsentation. Dies ermöglicht eine hohe Modularität und eindeutig definierte Schnittstellen zwischen den Objektklassen, was eine Voraussetzung für die inkrementelle Konstruktion von Applikationen ist.

Das traditionelle Phasenmodell der Softwareentwicklung steht im Gegensatz zur Idee der Softwarewiederverwendung: Applikationen, die in einem Top-down Ansatz, beginnend mit den Anforderungen und Spezifikationen entworfen werden, lassen wenig Hoffnung übrig, daß vorhandene Software genau diese Spezifikationen erfüllen wird. Demgegenüber paßt die prinzipielle Idee des Protoyping besser zum objektorientierten Softwareentwurf, weil es ebenfalls ein Gleichgewicht zwischen einer Top-down Vorgangsweise (analytisches Programmieren) und dem Bottom-up Ansatz (synthetisches Programmieren) voraussetzt. "Analytisches Programmieren" heißt, daß mit der Definition eines Programmskeletts der zukünftigen Applikation begonnen wird, während "synthetisches Programmieren" den Schwerpunkt auf die Entwicklung eines evolutiven Prototyps, konstruiert aus vorhandener Software legt. Der Prototyp sollte – im Idealfall – sowohl als Spezifikation als auch als Zwischenprodukt im Entwicklungsprozeß dienen.

Wir vergleichen in Tabelle II die Anforderungen an Prototypingwerkzeuge mit den Eigenschaften, die objektorientierte Sprachen und Entwurfswerkzeuge haben, bzw. versprechen. Daraus kann abgeleitet werden, daß eine objektorientierte Entwurfsumgebung die grundlegenden Ideen des Prototyping enthält. Wir behaupten sogar, daß die Entwicklung eines Prototyps in einer objektorientierten Entwicklungsumgebung, und die Entwicklung eines lieferbaren Produktes in derselben Umgebung, die gleich Art von Tätigkeit darstellt. D.h., objektorientierte Softwareentwicklung kann auch als "Prototyping-in-the-large" angesehen werden.

Prototyping erfordert	**Objektorientierte Entwicklung ermöglicht**
• schnelle Entwicklung	• synthetisches Programmieren, basierend auf wiederverwendbaren Objektklassen
• Evaluierung	• Konfiguration und Rekonfiguration von Objektklassen • Objektmetapher, unterstützt sowohl die Beschreibung von Realweltproblemen, als auch die Visualisierung des Programmverhaltens
• Inkrementeller Entwurf und Weiterentwicklung	• Wiederverwendung durch Instantiierung, Vererbung und durch Klassenbibliotheken • inkrementelle Konstruktion

Tabelle II: Eigenschaften der objektorientierten Programmentwicklung

Was ist nun aber notwendig, um diese Behauptung zu begründen? Wir stellen im folgenden einen objektorientierten Software Lifecycle vor, der als Basis für die noch notwendigen Entwicklungen im Bereich objektorientierter Systeme angesehen werden kann.

Wie kann die Wiederverwendbarkeit von Software ermöglicht werden? Sicher nicht allein dadurch, daß wir nun Objektklassen anstatt Funktionen und Prozeduren verwenden. Die Behandlung dieser Fragestellung wird das *Objektdesign* Problem bezeichnet. Dabei gilt, eine Objektklasse so zu entwerfen, sodaß sie mit anderen Objektklassen kombiniert werden kann, um ein noch komplexeres Problem zu lösen, als es mit ihrer bisherigen Funktionalität möglich war. Dieses Problem hat sein Analogon im Hardwaredesign, wo ebenfalls mit klaren Schnittstellendefinitionen gearbeitet wird, um die einzelnen Teile besser verbinden zu können. Auch die Erfolge in der Softwarewiederverwendung, z.B. bei Lisp, Prolog, oder Unix Shell Scripts, beruhen auf der klaren Definition der verwendeten Datenstrukturen und Schnittstellen (z.B., Listen, Funktionen, und Datenströme).

Eine Objektklasse kann dann wiederverwendet werden, wenn ihre Funktionalität, d.h. ihr Verhalten klar definiert ist. Was bedeutet nun "klar definiert"? Eine Objektklasse soll genau das Verhalten aufweisen, das von ihr intuitiv, basierend auf dem Namen der Objektklasse, auf dem Namen der jeweiligen Operation, oder aufgrund einer graphischen Repräsentation erwartet werden kann. Ein Beispiel dazu: wer bereits mit graphischen Benutzerschnittstellen gearbeitet hat, kennt den Begriff des Fensters und er hat bereits gewisse Vorstellungen, welche Funktionalität er von einem Fenster erwarten kann. Dazu gehören Vergrössern, Verkleinern, Positionswechsel am Schirm, und einige andere. Nehmen wir an, es soll eine graphische Schnittstelle in einem objektorientierten Softwaresystem erstellt werden, und die Objektklasse "Fenster" befindet sich in einer Objektklassenbibliothek. Diese Objektklasse stellt unter anderem die Operation "Vergrössern" zur Verfügung. Wird diese Operation getestet, und das Ergebnis ist eine Vergrösserung des Fensters mit gleichzeitigem Positionswechsel am Schirm, so ist diese Operation nicht intuitiv definiert, und damit in unserem Sinn nicht "klar definiert". Es ist natürlich nicht immer so eindeutig feststellbar, was die "klare Definition" einer Operation ausmacht. In [14] werden Richtlinien zum Design wiederverwendbarer Klassen aufgestellt.

Aus dem oben gesagten wird ersichtlich, daß (1) Objektklassen nicht unabhängig voneinander entworfen werden können, und daß (2) das Design einer Menge von Objektklassen mit der Zeit und der gesammelten Erfahrung veränderbar sein muß.

Unser Modell eines objektorientierten Software Lifecycles muß daher die folgenden zwei Prinzipien beinhalten, nämlich (1) Software muß wiederverwendbar sein, und (2) Software muß weiterentwickelbar sein. Daraus lassen sich zwei miteinander verbundene Lebenszyklen ableiten, die in Abbildung 1 dargestellt sind. Applikationen, die selbst Objektklassen darstellen können, bestehen aus verschiedenen Konfigurationen von Objektklassen. Eine neue Applikation entsteht als Prototyp aus vorhandenen Objektklassen, und vorhandenen, ähnlichen Applikationen. Dieser Prototyp wird weiterentwickelt, indem neue Anforderungen dazukommen, aufgrund der Bewertung Fehler erkannt und ausgebessert werden, oder die Umweltbedingungen dieser Applikation geändert werden. Die Applikationen werden in einer Applikations_Software_Bibliothek verwaltet, und können für die Konfiguration komplexerer Applikationen weiterverwendet werden. Parallel mit der Entwicklung und Weiterentwicklung von Applikationen wird es auch notwendig werden neue Objektklassen zu entwickeln. Die Bibliothek der Objektklassen, die Objekt_Software_Bibliothek (OSB), wird durch neu entwickelte Objektklassen ergänzt. Dabei kann es auch zu einer Reorganisation der abgespeicherten Objektklassen kommen. Ein einfaches Beispiel dazu wird in Abbildung 2 vorgestellt.

In Abbildung 2(a) ist schematisch eine Objektklasse OK1 mit zwei Operationen op1 und op2 dargestellt. Eine zweite Objektklasse OK2 wird entworfen und ebenfalls in der OSB für eine spätere Wiederverwendung abgespeichert (Abbildung 2(b)). OK2 unterstützt die Operationen op2 und op3. Um ein klares Design zu gewährleisten ist es notwendig, die OSB, wie in Abbildung 2(c) dargestellt, umzuorganisieren. Die Operation op2 wird aus OK1 und OK2 ausgeklammert, und eine eigene Objektklasse OK3 mit der Operation op2 entworfen. Diese Klasse, d.h. ihre Funktionalität, kann nun von allen Klassen geerbt werden, so auch von OK1 und OK2.

Die Evolution (Weiterentwicklung) der OSB ist eine langfristige Tätigkeit. Dabei soll es zu einer Standardisierung von häufig verwendeten Objektklassen kommen. Sofern Applikationsentwicklung als Prototyping angesehen wird, ist die Wiederverwendung von Software das wichtigste Kriterium, nachdem eine OSB bewertet werden kann. Ein objektorientierter Software Lifecycle sollte die zwei

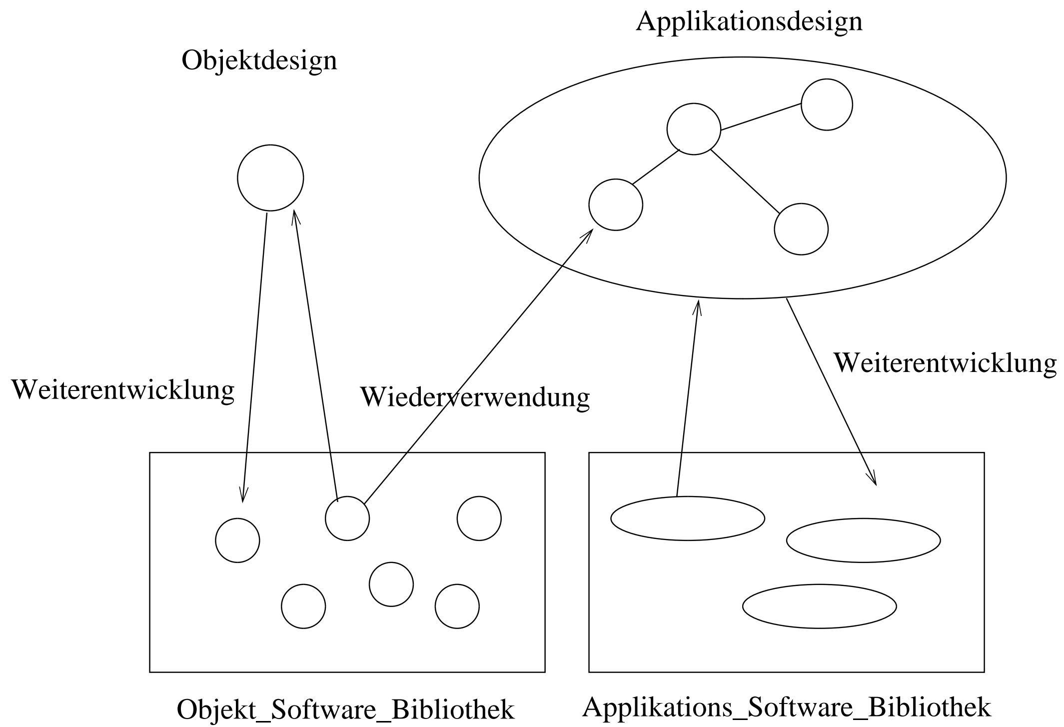

Figure 1: Abbildung 1: Objektorientierter Software Lifecycle

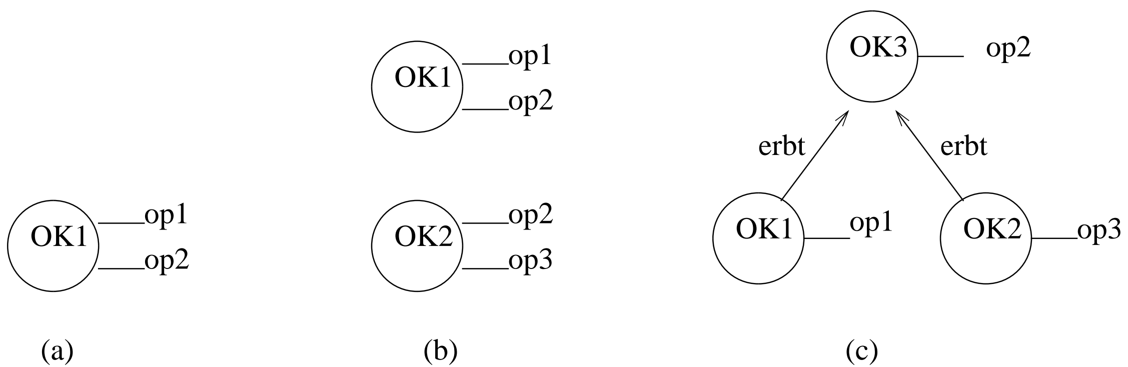

Figure 2: Abbildung 2: Reorganisation der Objekt_Software_Bibliothek

komplementären Aufgaben des Prototyping und des Objektdesigns unterstützen.

3.2 Werkzeuge zur Unterstützung des Prototyping in einem objektorientierten Software Lifecycle

Es existiert noch keine integrierte Entwicklungsumgebung, die den vorgestellten objektorientierten Software Lifecycle vollständig unterstützt. Es wurden aber bereits Werkzeuge, basierend auf objektorientierten Sprachen entwickelt, die spezifische Aufgaben erfüllen helfen. Für eine Einführung in diese Werkzeuge verwenden wir die bereits vorgestellte Klassifikation. Inwieweit diese Werkzeuge zum Design von Objektklassen und Applikationen beitragen können, wird untersucht.

3.2.1 Benutzerschnittstellen Toolkits

Die objektorientierte Beschreibung von Problemen, d.h. die Kapselung einer spezifischen Datenstruktur mit den erlaubten Operationen auf diesen Daten, eignet sich sehr gut für den Entwurf verschiedener graphischer Schnittstellen-Objektklassen, wie Fenster und Menüs. Neben der allgemeinen Bedeutung von Benutzerschnittstellen ist das sicher ein Grund für die Existenz von Bibliotheken für Schnittstellen-Objektklassen in verschiedenen objektorientierten Sprachen (z.B., für Objective C [7], Eiffel [20], und Smalltalk [10]). ET++ [27] stellt eine Bibliothek von wiederverwendbaren Objektklassen für Benutzerschnittstellen in C++ zur Verfügung.

Schnittstellen-Objektklassen helfen einheitliche, und damit leichter verständliche Benutzerschnittstellen für Objektklassen und Applikationen aufzubauen. Dies wird durch eine Trennung des Designs der Funktionalität vom Design der Repräsentation der Funktionaliät erreicht. Diese Idee ist speziell im *Model-View-Controller* Schnittstellenparadigma von Smalltalk-80 [11, 16] weiterentwickelt worden. Die *models* sind jene Teile des Applikationssytems, die die gewünschte Funktionalität zur Verfügung stellen. Eine *view* ist die Repräsentation eines Modells, und der *controller* ist für die Interaktion zwischen Ein- und Ausgabegeräten (z.B., Maus, Tastatur) und den Modellen und ihren zugeordneten Views verantwortlich.

3.2.2 VHLP

Objektorientierte Sprachen bauen auf Sprachkonzepten auf, die speziell den "rapid development" Aspekt des Prototyping unterstützen. Dazu gehören das Klassen- und Instantiierungskonzept, die Vererbung, das Overriding (Neuspezifikation von geerbten Instanzvariablen und Methoden), und die Kapselung. Mit diesen Eigenschaften ausgerüstet, können objektorientierte Sprachen selbst als "Very High Level Programming" Werkzeuge angesehen werden.

Doch objektorientierte Sprachen unterstützen nicht automatisch andere nützliche Programmierstile, wie das visuelle Programmieren, oder das Programmieren durch Spezifikation von Beispielen. Wir können aber feststellen, daß der objektorientierte Programmentwurf, d.h. die Objektklasse als kleinste Entwurfs- und Programmiereinheit und der Austausch von Nachrichten (Messages) als

einziger Kommunikationsmechanismus, sich sehr gut zur Visualisierung eignet, und damit den Aufbau graphischer Systeme unterstützt.

Nachdem Smalltalk die weitestverbreitete objektorientierte Sprache, vor allem im Forschungs- und Universitätsbereich darstellt, ist es nicht verwunderlich, daß die meisten Werkzeuge in und für Smalltalk entwickelt wurden. Dazu gehören die folgenden:

- *Visuelle Programmierung:* auch hier ist die Unterteilung in applikationsneutrale graphische Systeme und graphische Simulationssysteme gültig. Zum Beispiel, Fabrik [13] ist ein visuelles Progammiersystem, das aus einer Menge von Komponenten, dargestellt als Karteikarten, mit verschiedenen Funktionalitäten besteht. Die Komponenten haben "ports" für die Eingabe und Ausgabe von Daten. Komponenten können miteinander verbunden werden, wobei die Ausgabedaten einer Komponente, die Eingabedaten der nächsten Komponente darstellen. Dies entspricht dem klassischen Datenflußkonzept. Neue Komponenten können durch Kombination existierender Komponenten aufgebaut und in weiteren Applikationen wiederverwendet werden. Ein Beispiel für die zweite Kategorie von visuellen Sprachen ist ThingLab [4], eine graphische Simulationssprache. Beide Systeme sind in Smalltalk entwickelt.
- *PBE:* die Metapher des kommunizierenden Objektes hilft, einem Nicht-Computerspezialisten Datenstrukturen und Kontrollstrukturen in einer ihm verständlichen Form zu erklären. Zum Beispiel, es wurde ein Autorensystem für Lehrer entwickelt, die selbst die gewünschte Unterrichtssoftware damit programmieren können. Dieses System heißt "Programming By Rehearsal" [8], weil die Benutzerschnittstelle als Theaterbühne mit Schauspielern in verschiedenen Rollen entworfen ist.

Eine Schwierigkeit im Design von Objektklassen ist sicherlich die mangelnde Erfahrung, wie Objektklassen kompiniert werden können, um daraus neue Klassen aufzubauen. Visuelle Programmiersysteme bieten normalerweise genau eine Möglichkeit der Kombination. Zum Beispiel, Fabrik benutzt bidirektionalen Datenfluß, wobei die Objektklassen der jeweiligen Eingabe- und Ausgabeports, die verbunden werden, übereinstimmen müssen. Stehen mehrere Möglichkeiten der Verbindung von Objektklassen zur Verfügung, so wird auch die Wahrscheinlichkeit der Wiederverwendung dieser Klassen erhöht. Visueller Entwurf kann auch im Applikationsdesign helfen, indem von den Details innerhalb der Komponenten abstrahiert, und mit verschiedenen Konfigurationsarchitekturen experimentiert wird. Die Interdependenz zwischen den Komponenten sollte dadurch leichter verständlich werden.

3.2.3 Programmier- und Entwicklungsumgebungen

Um den objektorientierten Software Lifecycle zu realisieren, brauchen wir Sprachmechanismen, die speziell das Prinzip der Wiederverwendung unterstützen, und integrierte Werkzeuge, die uns im Objektdesign und im Applikationsdesign helfen. Neben den bereits besprochenen Benutzerschnittstellen Toolkits und den graphischen Programmiersystemen unterscheiden wir Werkzeuge zur Visualisierung von objektorientierten Applikationen und Werkzeuge zum Objektdesign.

Visualisierungswerkzeuge sollen ein besseres Verständnis der statischen und dynamischen Struktur einer objektorientierten Applikation ermöglichen. Unter statischer Struktur versteht man die Hierarchie der verwendeten Objektklassen. Die dynamische Struktur zeigt, welche Nachrichten zwischen den Objekten zur Laufzeit ausgetauscht wurden. Diese Werkzeuge können auch zum Debuggen des Softwaresystems verwendet werden. Visualisierungswerkzeuge werden manchmal auch als Werkzeuge zur Animation von Programmen bezeichnet. In [15, 18] werden Visualisierungswerkzeuge für Smalltalk vorgestellt.

Existierende Werkzeuge für Objektdesign beschäftigen sich in erster Linie mit der Verwaltung von wiederverwendbaren Objektklassen. Dabei ist unter Verwaltung das Speichern, Wiederfinden, und die Weiterentwicklung von Objektklassen zu verstehen. Wenn eine Objektklasse in einer OSB (Abbildung 1) abgespeichert wird, sollte möglichst viel Information über diese Objektklasse (z.B. ihre Funktionalität) und ihre Beziehungen zu anderen Objektklassen (z.B. Vererbungsbeziehungen) so aufbereitet werden, daß ein späteres Wiederfinden dieser Klasse ermöglicht wird. Ein Modell einer OSB wird in [2] vorgestellt.

Das nächste Problem ist das Finden von Objektklassen in einer OSB, die in der aktuellen Applikation verwendet werden können. Für diese Aufgabe werden "Browsing" Werkzeuge eingesetzt, die das Browsen (das Durchsuchen) erleichtern sollen. Smalltalk und Trellis [21] sind Beispiele von Programmierumgebungen, die Werkzeuge zum Browsen ihrer Bibliotheken zur Verfügung stellen. In [22] wird ein Browsingmechanismus vorgestellt, der auf verschiedenen und sich ändernden Beziehungen zwischen den Objektklassen aufbaut.

Die Weiterentwicklung von Objektklassen kann nur als "long-range" Aktivität verstanden werden. Indem Objektklassen immer wieder verwendet werden, wird ein Redesign sowohl von Objektklassen als auch von Beziehungen zwischen den Klassen notwendig werden. Objektorientierte Sprachen bieten dazu den Mechanismus der Kapselung an. Solange sich die sichtbare Schnittstelle einer Klasse nicht ändert, bleibt das richtige Funktionieren der Applikationen, die diese Klasse verwenden, gewährleistet.

4 Zusammenfassung und offene Probleme

Wir haben einen objektorientierten Software Lifecycle vorgestellt, der evolutives Prototyping, und damit die Anpassung einer Applikation an sich ändernde Anforderungen unterstützt. Dieser Software Lifecycle baut auf dem Prinzip der Wiederverwendung existierender Objektklassen und Applikationen auf. Obwohl objektorientierte Sprachen bereits Mechanismen für diese Aufgabe zur Verfügung stellen, und verschiedene objektorientierte Werkzeuge für den Entwicklungsprozeß existieren, gibt es noch einige ungelöste Probleme und Schwachstellen.

Es gibt noch keine anerkannte Methode für Objektdesign. Wie muß eine Objektklasse entworfen werden, damit sie wirklich wiederverwendbar ist? Wie wird die Reorganisation einer OSB vorgenommen, ohne die Konsistenz der darauf aufbauenden Applikationen zu gefährden? Wir glauben, daß diese Fragen aufgrund praktischer Erfahrungen gelöst werden können.

Objektorientierte Sprachen unterstützen nicht (ausreichend) Mechanismen, die sowohl in der "Prototyp"-

Entwicklung, als auch in der "Produkt"-Entwicklung gebraucht werden. Die bisherige Erfahrung war, daß Sprachen, die sich aufgrund ihrer Flexibilität und ihrem hohen Abstraktionsniveau zum Prototyping eignen , wegen mangelnder Effizienz nicht für den Produktentwurf verwendet werden, und umgekehrt. Um dieser Erfahrung gerecht zu werden, brauchen wir objektorientierte Sprachen mit folgenden Eigenschaften: Erstens, in der Entwicklung befindliche Objektklassen sollen interpretierbar sein, während sie mit Objektklassen kommunizieren, die bereits übersetzt wurden. Einige Smalltalk- und Lispdialekte bieten diese Möglichkeit. Zweitens, der Programmierer soll entscheiden können, ob bereits zur Uebersetzungszeit (StatischeBindung), festgelegt wird, welche Methode beim Aufruf einer Operation ausgeführt wird oder erst zur Laufzeit (Dynamische Bindung). C++ und Eiffel bieten Mechanismen in diese Richtung an. Drittens, der Programmierer soll zwischen einer Typüberprüfung zur Uebersetzungszeit und zur Laufzeit unterscheiden können.

Und schließlich brauchen wir Programmier- und Entwicklungsumgebungen, die die oben genannten Eigenschaften integrieren. Aufgrund der noch offenen Fragen ist ersichtlich, daß objektorientierte Entwicklungssysteme und objektorientiertes Prototyping sicher kein abgeschlossenes Forschungsgebiet darstellen. Dennoch, die bisherigen Erfahrungen mit objektorientierten Systemen und die vermehrten Forschungsaktivitäten auf diesem Gebiet lassen erwarten, daß ein objektorientierter Software Lifecycle wesentlich zum Entwurf anpassungsfähiger, und damit langfristig besserer Applikationen beitragen wird.

Literatur

[1] W.W. Agresti (ed.), *New Paradigms for Software Development*, IEEE Computer Society Press/North Holland, 1986.

[2] C. Arapis and G. Kappel, *Organizing Objects in an Object Software Base*, in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, S. 32-50, June 1988.

[3] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[4] A. Borning, *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, S. 353-387, Oct. 1981.

[5] R. Cazalens, A. Doucet, D. Plateau and B. Poyet, *Towards Benchmarking User Interface Toolkits*, Rapport Technique Altair 24-88, GIP ALTAIR, Oct. 1988.

[6] IEEE Computer - Special Issue on Visual Programming, vol. 18, no. 8, August, 1985.

[7] B.J. Cox, *Object Oriented Proggramming: An Evolutionary Approach*, Addison-Wesley, 1986.

[8] W. Finzer and L. Gould, *Programming by Rehearsal*, BYTE, vol. 9, no. 6, S. 187-210, June 1984 .

[9] C. Floyd, *A systematic look at prototyping*, in *Approaches to Prototyping*, ed. R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, S. 1-18, Springer, 1984.

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

[11] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, 1984.

[12] Handbuch der Modernen Datenverarbeitung - 4 Software-Generation, vol. 137, Sept 1987.

[13] D. Ingalls, *Fabrik: A Visual Programming Environment*, Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, S. 176-190, Nov. 1988.

[14] R.E. Johnson and B. Foote, *Designing reusable classes*, Journal of Object-Oriented Programming, vol. 1, no. 2, S. 22-35, 1988.

[15] M.F. Kleyn and P.C. Gingrich, *GraphTrace - Understanding object-oriented systems using concurrently animated views*, Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, S. 191-205, Nov. 1988.

[16] G.E. Krasner and S.T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, vol. 1, no. 3, S. 26-49, Aug. 1988.

[17] K.-D. Kreplin, *Prototyping mit Werkzeugen der vierten Generation*, Handbuch der Modernen Datenverarbeitung - 4 Software-Generation, vol. 137, S. 29-40, Sept 1987.

[18] R.L. London and R.A. Duisberg, *Animating programs using Smalltalk*, IEEE Computer - Special Issue on Visual Programming, vol. 18, no. 8, S. 61-71, Aug. 1985.

[19] J. Martin, *Fourth Generation Languages, Vol. I: Principles*, Prentice Hall, 1985.

[20] B. Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[21] P.D. O'Brien, D.C. Halbert and M.F. Kilian, *The Trellis Programming Environment*, Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 22, no. 12, S. 91-102, Dec 1987.

[22] X. Pintado and D. Tsichritzis, *An Affinity Browser*, in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, S. 51-60, June 1988.

[23] G. Pomberger et al, *Prototypingorientierte Softwareentwicklung*, Teil I. Tech.Report Nr. 8705, Institut für Informatik, Universität Zürich, 1987.

[24] R.V. Rubin, *Language Constructs for Pogramming by Example*, 3rd ACM-SIGOIS Conference on Office Information Systems, also SIGOIS Bulletin, vol. 7, no. 2-3, S. 92-103, 1986.

[25] R. Venken and M. Bruynooghe, *Prolog as a language of prototyping information systems*, in *Approaches to Prototyping*, ed. R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, S. 447-458, Springer, 1984.

[26] P. Wegner, *Dimensions of Object-Based Language Design*, Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 22, no. 12, S. 168-182, Dez. 1987.

[27] A. Weinand and E. Gamma, *ET++ - An object-oriented application framework in C++*, Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, S. 46-57, Nov. 1988.