Methoden und Werkzeuge für das Prototyping und ihre Integration

G. Pomberger, W. Pree, A. Stritzinger
Institut für Wirtschaftsinformatik
J.K.-Universität Linz, A-4040 Linz,
e-mail: {pomberger, pree, stritzinger}@swe.uni-linz.ac.at

Zusammenfassung

Prototyping ist zu einem Schlagwort geworden, ähnlich dem Begriff *Strukturierte Programmierung* anfang der siebziger Jahre. Es wird vielfach angenommen, daß eine Prototyp-orientierte Software-Entwicklungsstrategie substantiell zur Lösung eines Teils der Probleme bei der Software-Entwicklung beitragen kann. Ausgehend von den Erfahrungen eines Forschungsprojektes, werden die Konzepte einer Prototyp-orientierten Software-Entwicklung, die dazu erforderlichen Werkzeuge und ihre Integration beschrieben. Ein Werkzeugkasten zur Prototyp-orientierten Software-Entwicklung wird detailliert vorgestellt.

Schlüsselwörter:
Prototyp, Prototyping, Software-Entwicklungsumgebung, Software-Lebenszyklus, Architektur-Prototyp, Benutzerschnittstellen-Prototyp, Explorative Software-Entwicklung

Abstract

Prototyping has become a key word during the last years, just like *structured programming* in the seventies. It is often assumed that a prototyping-oriented software development strategy may substantially contribute to the solution of problems associated with conventional software development. Based on experience from a research project, concepts of prototyping-oriented software development, necessary tools and their integration are described. A toolset for prototyping-oriented software development is presented in detail.

Keywords:
Prototype, Prototyping, Software Development Environment, Software Life Cycle, Architecture Prototyping, User Interface Prototyping, Explorative Software Development

1. Nachteile der klassischen Software-Entwicklungsmethode

Das Software Lebenszyklus-Modell basiert auf der Annahme, daß der Entwicklungsprozeß in der Regel linear verläuft und daß Wiederholungen einzelner Phasen selten notwendige Ausnahmen sind.

Eine dogmatische Anwendung dieser Entwicklungsmethode schreibt vor, daß eine Phase erst dann begonnen werden darf, wenn die vorhergehende vollständig abgeschlossen ist. In der Praxis sind vollständige Spezifikationen oder eine exakte Beschreibung der Systemarchitektur eine selten erreichte Ausnahme. Normalerweise haben die späteren Phasen einen starken Einfluß auf die vorangehenden. Die strikte Trennung der einzelnen Phasen ist daher eine unzulässige Idealisierung. Normalerweise gibt es Überlappungen zwischen den einzelnen Phasen, und die wechselseitigen Einflüsse sind viel komplexer als sie im sequentiellen Ein-Ausgangsmodell (siehe Abb. 1) dargestellt sind.

Die streng sequentielle Vorgehensweise führt dazu, daß experimentell überprüfbare Produkte oder Komponenten erst in sehr späten Projektphasen verfügbar sind. Erfahrungen haben gezeigt, daß keine zuverlässige Produktabnahme möglich ist, wenn Tests nicht unter realitätsnahen Bedingungen durchgeführt werden können. Einer der größten Nachteile ist, daß Änderungswünsche des Kunden erst sehr spät (nach Vorliegen einer ersten ausführbaren Systemversion) zum Vorschein kommen. Die Folge davon ist, daß notwendige Änderungen nur mit beträchtlichem Aufwand zu bewerkstelligen sind.

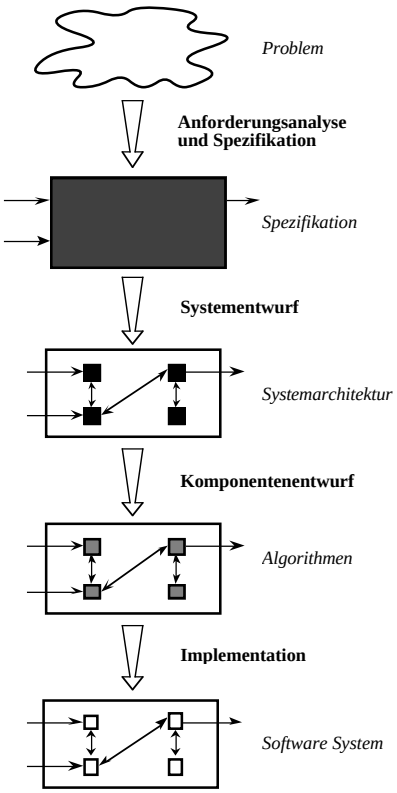

Abb. 1: Schrittweise Verfeinerung nach dem Software Lebenszyklus-Modell

Zusammenfassend kann man sagen, daß die Lebenszyklus-orientierte Software-Entwicklungsmethode zwar eine erprobte und weit verbreitete Vorgehensweise ist, jedoch beträchtliche Nachteile aufweist. Es wird häufig angenommen – und neuere Berichte aus Forschung und Praxis bestätigen diese Annahme, daß eine Prototyp-orientierte Entwicklungsstrategie einige der Schwächen der Lebenszyklus-orientierten Vorgehensweise beheben kann.

2. Prototyp-orientierte Software-Entwicklungsstrategie

Seit mehr als zehn Jahren gibt es Berichte über Software-Projekte, in denen Prototypen entwickelt wurden, um die Anforderungen des Benutzers umfaßend abzuklären. Wichtige Beiträge finden sich unter anderem in [4, 5, 6, 8].

Eine Analyse dieser Berichte machte deutlich, daß es keine einheitlichen Begriffsdefinitionen von *Prototyp* und *Prototyping* gibt. Konsens besteht darüber, daß ein Prototyp ausführbar sein muß. Ob "ausführbar" bedeutet, daß die Funktionen tatsächlich in irgend einer Weise realisiert sind, oder ob deren Simulation genügt, ist bereits kontrovers. Einigkeit herrscht in der Ansicht, daß Software-Prototypen – im Gegensatz zu anderen Prototypen – rasch und kostengünstig zu erstellen sein müssen. Einige Autoren empfehlen die Entwicklung von "echten" Prototypen, sodaß voll funktionsfähige Modelle mit allen relevanten Merkmalen der späteren Applikation verfügbar sind, die dann als Basis für den Software-Entwicklungsprozeß dienen. Andere Autoren meinen, daß Prototyping eine Bottom up-Vorgehensweise ist, d.h., daß einige wenige (einfache) Funktionen rasch implementiert und dann vom Benutzer getestet und verbessert werden. Danach erfolgt die Implementierung weiterer Funktionen. Dieser Prozeß läuft solange, bis schließlich das Produkt fertig ist.

Da es keine allgemein anerkannte Definition von Prototyp und Prototyping gibt, wollen wir zuerst in Anlehnung an [6] umschreiben, was wir unter diesen Begriffen verstehen.

Ein Software-Prototyp ist ein ausführbares Modell mit wesentlichen Eigenschaften des Zielsystems, das Grundlage für die Systemspezifikation ist und die Kommunikation zwischen Kunden und Entwickler unterstützt. Das Modell bildet wesentliche Eigenschaften des geplanten Systems in einer anschaulichen und leicht modifizierbaren Form nach. Ein Prototyp soll die Funktionalität wesentlich wirkungsvoller darstellen als textuelle Beschreibungen oder rein statische (= nicht ausführbare) Modelle.

Prototypen lassen sich wie folgt charakterisieren:

- Sie können schnell und billig entwickelt werden.
- Sie bieten dem Benutzer der geplanten Applikation ein funktionales und ausführbares Modell der wesentlichen Teile des Systems vor dessen Implementierung.
- Sie sind flexibel, d.h. sie lassen sich leicht modifizieren und erweitern.
- Sie modellieren nicht notwendigerweise ein vollständiges System.

Prototyping umfaßt alle Tätigkeiten, die erforderlich sind, um solche Prototypen zu erstellen.

Diese Umschreibung ist bewußt allgemein gehalten. Ihr Zweck ist es, klarzumachen, daß das Experimentieren auch bei der Entwicklung

großer Software-Systeme überaus nützlich und sinnvoll ist und daß die Vorgehensweise dabei ähnlich wie bei der Entwicklung anderer technischer Systeme ist.

Eine Prototyp-orientierte Software-Entwicklung ist nicht grundsätzlich verschieden von einer Lebenszyklus-orientierten Vorgehensweise. Vielmehr können wir beide Methoden als einander ergänzend, anstatt als konkurrierend ansehen, d.h. ein modifiziertes Software Lebenszyklus-Modell angeben (siehe dazu [19, 20]). Das modifizierte Software Lebenszyklus-Modell besagt, daß die Vorgehensweise nicht mehr streng sequentiell, sondern an bestimmten Stellen zyklisch ist. Die Wiederholung bestimmter (genau definierter) Aktivitäten ist nicht nur möglich, sondern notwendig. Dieser modifizierte Software Lebenszyklus ist in Abb. 2 dargestellt.

Die Prototyp-orientierte Entwicklungsmethode (wie wir sie sehen) unterscheidet sich von der konventionellen primär in den Tätigkeiten und Ergebnissen der einzelnen Projektphasen. Die Einteilung in Phasen wird zwar beibehalten, jedoch überlappen sich Anforderungsanalyse und Systemspezifikation zeitlich stark.

Der Spezifikationsprozeß wird durch Prototyping-Aktivitäten ergänzt, d.h. eine neue Tätigkeit (Benutzerschnittstellen-Prototyping) wird eingeführt und quasi ein Lebenszyklus im Lebenszyklus etabliert (siehe Abb. 2). Das Ergebnis dieses Entwicklungszyklus ist nicht mehr bloß ein Pflichtenheft, sondern ein ausführbares Modell zusammen mit einer textuellen Spezifikation all dessen was das Modell nicht ausdrückt. Der Textumfang des Pflichtenheftes wird geringer, da der Prototyp einen Teil des Pflichtenheftes in nicht-textueller Form darstellt.

Die Entwurfsphase wird ebenfalls ergänzt durch das sogenannte *Architekturprototyping*. Während des Systementwurfs wird die Architektur in Form eines Prototyps realisiert und anhand realer Anwendungsfälle überprüft. Dieser Prozeß stellt wiederum einen eigenen Lebenszyklus dar. Das Ergebnis der Designphase umfaßt neben der Architekturbeschreibung einen Prototypen der Architektur und gegebenenfalls Prototypen für einzelne (kritische) Systemkomponenten. Das Ziel ist, die Adäquatheit der Architektur und die Vollständigkeit der Komponentenschnittstellen und ihrer Wechselwirkungen nachzuweisen, bevor das System tatsächlich implementiert wird.

Die Phasen sind nun keine chronologischen Abschnitte eines kontinuierlichen Entwicklungsprozesses mehr. Daher sprechen wir besser von Aktivitäten anstelle von Phasen; die exakte Trennung der einzelnen Aktivitäten gemäß dem Software Lebenszyklus wird aufgeweicht.

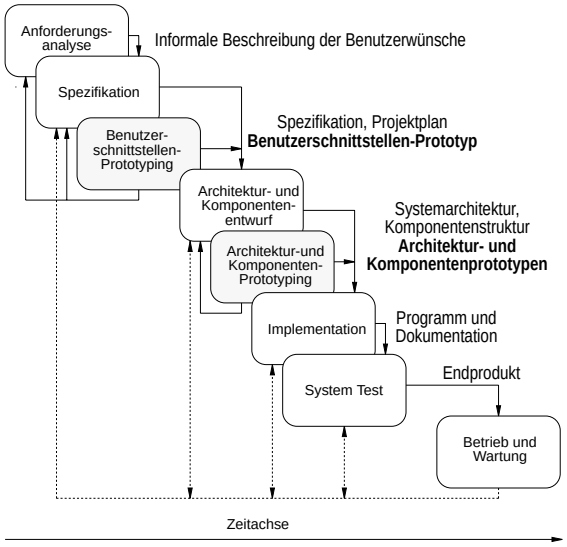

Abb. 2: Prototyp-orientierter Software Lebenszyklus

Die Erstellung eines Prototypen ist ein iterativer Prozeß (Implementierung - Evaluation - Adaptierung). Es wird jeweils soviel Funktionalität implementiert, wie notwendig ist, um das angestrebte Phasenergebnis (Spezifikation, Architekturbeschreibung) zu überprüfen.

Das führt zu einem weiteren wesentlichen Unterschied der beiden Vorgehensweisen. Bei der Lebenszyklus-orientierten Vorgehensweise wird die Implementierung so spät als möglich vorgenommen – erst nachdem das Zielsystem vollständig spezifiziert und entworfen ist. Bei der Prototyp-orientierten Methode andererseits wird so früh als möglich (ein Prototyp) implementiert.

Erfahrungen haben gezeigt, daß die Wahrscheinlichkeit, die gesteckten Ziele zu erreichen, drastisch erhöht wird, wenn die Anforderungsdefinition und die Systemarchitektur schrittweise anhand eines Modells, das auch die Dynamik des Systems zeigt, entwickelt wird.

Durch die Überlappung verschiedener Aktivitäten und der daraus resultierenden Art der Zwischenergebnisse – statt statischer Beschreibungen ausführbare Prototypen – wird das Risiko für falsche Entscheidungen signifikant reduziert. Der Verbesserungseffekt, der mittels Experimentieren erreicht wird, stellt eine neue Form der Qualitätssicherung dar. Darüber hinaus ist auch der Lerneffekt der Benutzer durch die Experimente entscheidend für die Akzeptanz des zu entwickelnden Systems.

Ob die einzelnen Prototypen (für Benutzerschnittstellen, Systemarchitektur und bestimmte Komponenten) weggeworfen oder Bestandteil des Systems werden, ist für die allgemeine Methode der Prototyp-orientierten Software-Entwicklung nicht relevant. Die Ziele sind Risikobegrenzung, Reduktion von Kommunikationsproblemen, Qualitätssicherung und die Nutzung der Lerneffekte durch Experimentieren unter realitätsnahen Bedingungen. In Bezug auf die Rationalisierung des aufwendigen Herstellungsprozesses ist natürlich die Integration der Prototypen in das Produkt wünschenswert, d.h. eine evolutionäre Entwicklungsstrategie wäre von Vorteil. Ob eine solche möglich ist, hängt stark davon ab, ob geeignete Werkzeuge verfügbar sind.

3. Prototyping-Werkzeuge

Die Prototyp-orientierte Vorgehensweise bei der Software-Entwicklung setzt voraus, daß Prototypen rasch und kostengünstig erstellt werden können. Der Grund für das relativ späte Auftauchen der Idee des Prototyping liegt unter anderem darin, daß zu wenig Werkzeuge mit hinreichender Qualität verfügbar waren. Die benutzten Werkzeuge sind in zweierlei Hinsicht von entscheidender Bedeutung: Sie haben einerseits großen Einfluß auf die Qualität der zu erstellenden Prototypen, andererseits auf den notwendigen Aufwand zur Modifikation und Erweiterung der Prototypen. Im Folgenden werden einige typische Klassen von Prototyping-Werkzeugen erörtert.

Generatoren

Generatorsysteme bestehen typischerweise aus zwei Komponenten: Einem Editor, mit dessen Hife Spezifikationen erstellt werden und einem Compiler/Interpreter/Laufzeitsystem zur Syntaxprüfung, zur Transformation der Spezifikation in ein ablauffähiges System und zu dessen Ausführung.

Die Idealvorstellung ist, daß der Prototyp-Entwickler lediglich spezifizieren muß, “was” zu geschehen hat; “wie” das ablauffähige System realisiert wird, darum braucht er sich nicht zu kümmern. Die Implementierung des Programms ist Aufgabe des Generators, der Systementwickler hat keinen Einfluß darauf.

Eine wichtige Eigenschaft eines Generators als Werkzeug zur Prototyperstellung ist das Abstraktionsniveau der Spezifikation. Beispiele für Spezifikationssprachen sind die Dialogbeschreibungssprache UISL [15], attributierte Grammatiken [16] und grafische Spezifikationssysteme [21].

Typische Vertreter von Generatoren, die den Prototyping-Prozeß unterstützen, sind: Benutzerschnittstellengeneratoren, Grammatikgesteuerte Generatoren und Generatoren für Informationssysteme.

Generatoren sind besonders nützliche Prototyping-Werkzeuge, weil sie die Spezifikation auf einem hohen Abstraktionsniveau ermöglichen, und weil sie eine rasche Implementierung sowie einfache Änder- und Erweiterbarkeit sicherstellen.

Der Nachteil der Generatorsysteme liegt häufig in der zu geringen Flexibiltät – Prototypen, die nicht in der vorgebenen Sprache spezifiziert werden können, sind nicht erzeugbar, auch wenn die Abweichungen gering sind. Ein weiterer Nachteil von Generatoren ist, daß sie häufig unzureichend effiziente Implementierungen erzeugen.

Application Frameworks

Konventionelle Modulbibliotheken sind nur in einem sehr beschränkten Ausmaß für Prototyping geeignet, da einerseits ihr Abstraktionsniveau zu niedrig ist, und da sie andererseits den Entwickler zwingen, große Teile der Applikation mittels einer konventionellen Programmiersprache manuell zu implementieren; dies ist für den Prototypingprozeß viel zu aufwendig.

Application Frameworks sind eine Sammlung von wiederverwendbaren, erweiterbaren und modifizierbaren Bausteinen zur Entwicklung von – vorzugsweise dialogorientierten – Applikationen. Sie geben einen Rahmen für die Architektur einer speziellen Applikation vor. Der Rahmen besteht aus einem zentralen Baustein, in dem Ereignisse (Benutzereingaben) entgegengenommen werden, und deren weitere Behandlung initiiert wird (*Main Event Loop*). Neben einem solchen zentralen Baustein bieten derartige Systeme noch eine Menge von Bausteinen zur Ereignisbehandlung (*Event Handler*) an. Diese Bausteine können an bestimmten vorgegebenen Stellen in den Rahmen eingesetzt werden. Ereignisbehandlungsobjekte sind häufig Komponenten der Benutzerschnittstelle (Fenster, Menüs, Rollbalken, Buttons, editierbare Textbereiche, etc.). Der zentrale Baustein und die verschiedenen Ereignisbehandlungsobjekte sind so aufeinander abgestimmt, daß sie in koordinierter Zusammenarbeit die Benutzereingaben verarbeiten können.

Application Frameworks werden (nahezu) ausschließlich in objektorientierter Form implementiert. Das heißt, alle vordefinierten Komponenten sind üblicherweise in Form von Klassen beschrieben. Diese Klassen bilden Schablonen, welche Struktur und Verhalten der daraus erzeugten Objekte festlegen. Die objektorientierte Programmierung erlaubt, daß Klassen mit meist sehr geringem Aufwand strukturell ergänzt und deren Verhalten modifiziert oder erweitert werden kann. Damit hat der Programmierer die Möglichkeit, applikationsspezifische Funktionen an die vorgegebenen Komponenten anzuhängen, oder das Standardverhalten der Bausteine zu modifizieren.

Application Frameworks sind in der Regel sehr umfangreiche und komplexe Baukästen. Die meisten Application Frameworks erfordern einen beträchtlichen Lernaufwand, um gewinnbringend eingesetzt werden zu können.

Aus der Sicht des Prototyping sind Application Frameworks eine wesentliche Verbesserung gegenüber Modulbibliotheken und konventioneller Programmierung. Durch das hohe Abstraktionsniveau können einfache Applikationen mit anspruchsvoller und funktionsfähiger Benutzerschnittstelle (=Benutzerschnittstellen-Prototypen) in einem deklarativen Programmierstil erstellt werden. Setzt man die Beherrschung eines geeigneten Application Frameworks voraus, ist die Prototypenerstellung eine einfach und effektiv ausführbare Tätigkeit. Diese Prototypen können in der Folge vom Systementwickler schrittweise mit anwendungsspezifischer Funktionalität erweitert werden.

Application Frameworks unterstützen daher die Methoden des explorativen und des evolutionären Prototyping in besonderer Weise. Die Nachteile der Application Frameworks liegen einerseits im hohen Einarbeitungsaufwand und andererseits im doch in der Regel geringeren Komfort im Vergleich mit Spezifikationssprachen oder interaktiven Dialog-Editoren. Insbesondere gestatten Application Frameworks dem späteren Systembenutzer nicht, Prototypen selbst zu gestalten; dazu sind fundierte Programmierkenntnisse erforderlich. Trotz dieser Nachteile gehören Application Frameworks wegen ihrer hohen Flexibilität zu den wichtigen Prototyping-Werkzeugen. Beispiele für Application Frameworks, die für Prototyping geeignet sind, finden sich in [25, 27, 28].

Architektursimulatoren

Das Ergebnis des Entwurfsprozesses ist eine Beschreibung der Systemarchitektur, d.h. eine Spezifikation aller notwendigen Komponenten. Architektursimulatoren sind Werkzeuge, die eine Überprüfung der wechselseitigen Einflüsse der Komponenten erlauben, bevor diese tatsächlich realisiert sind. Ein solches Werkzeug muß mindestens folgende Funktionalität zur Verfügung stellen:

- Simulation des Informationsflusses zwischen den Komponenten
- Verbindung der Systemarchitektur mit dem Benutzerschnittstellen-Prototyp
- Darstellung des aktuellen Systemzustandes und einer Auswahl der nächsten Zustände
- Automatische Protokollierung der Zustandsübergänge während einer Simulation
- Wiederholen von früheren Simulationen
- Unterstützung eines inkrementellen Implementierungsprozesses

Die Entwicklung derartiger Werkzeuge steht noch am Anfang. In Abschnitt 4 wird ein solches Werkzeug im Detail vorgestellt.

Datenbank- und 4.-Generationssysteme

Die Erstellung von Prototypen für Informationssysteme erfordert:
- Werkzeuge zur (interaktiven) Definition von Datenmodellen u. Datenbankschemata
- Werkzeuge zur interaktiven Gestaltung von Benutzerschnittstellen
- Werkzeuge zur Datenbankgenerierung
- Werkzeuge zur Simulation des Informationssystems

Moderne 4.-Generationssyteme, die für Prototyping geeignet sind, integrieren die oben angegebenen Werkzeugklassen und sollen darüber hinaus noch folgende Anforderungen erfüllen:
- Grafische Schemadarstellung, symbolorientierte, nicht prozedurale Programmiersprache
- Kurze Turn-Around-Zeiten
- Schnittstellen zu algorithmischen Sprachen und zu SQL.

Auch sogenannte Datendefinitionssprachen (*Data Definition Languages*, DDLs) sind als Prototyping-Werkzeuge geeignet. Allerdings ist der Komfort für den Entwickler geringer als bei grafischen Schema-Editoren. Die Ausdruckskraft von DDLs übertrifft dafür in manchen Fällen die Möglichkeiten grafischer Darstellungen. Zur Erstellung der Schema-Spezifikation wird häufig ein normaler Texteditor benutzt. Das hat den Nachteil, daß strukturelle Fehler erst bei der Datenbankgenerierung erkannt werden.

4. TOPOS – Eine Werkzeugsammlung für Prototyp-orientierte Software-Entwicklung

In den vorhergehenden Abschnitten wurde eine Prototyp-orientierte Entwicklungsmethode vorgestellt und die wichtigsten Werkzeugklassen erörtert. Ausgehend von einem Scenario für Prototyp-orientierte, inkrementelle Software-Entwicklung wird in diesem Abschnitt ein neues Konzept für diese Vorgehensweise diskutiert und die Integration verschiedener Werkzeuge zu einer Prototyp-orientierten Software-Entwicklungsumgebung vorgestellt. Das Ziel ist nicht, ein Produkt zu beschreiben, sondern vielmehr zu skizzieren, wie eine Prototyp-orientierte Entwicklungsumgebung aufgrund der Anforderungen aufgebaut sein kann, um einen Diskussionsbeitrag zur Gestaltung von Software-Entwicklungsumgebungen zu leisten. Der vorgestellte Werkzeugkasten ist jedoch nicht nur als Konzept vorhanden, sondern selbstverständlich auch implementiert. TOPOS steht für ***TO**olset for **P**rototyping-**O**riented **S**oftware Development*.

Die TOPOS-Struktur

Die Überlegungen, die zum Prototyp-orientierten Software Lebenszyklus (Abb. 2) geführt haben, machen deutlich, daß eine Reihe von Werkzeugen erforderlich ist, um den Entwicklungsprozeß zu unterstützen.

1) Werkzeuge zur Erstellung von Benutzerschnittstellen-Prototypen (Unterstützung der Anforderungsanalyse und Spezifikation)
2) Werkzeuge zur Architektur-Verifikation und Integration von Schnittstellen-Prototyp und anderen Systemkomponenten sowie Werkzeuge zur Unterstützung einer inkrementellen Implementierung
3) Werkzeuge, die die Verwaltung der im Software-Entwicklungsprozeß anfallenden Komponenten unterstützen.
4) Werkzeuge, welche die Dokumentation und Wartung unterstützen.

Darauf aufbauend wurde die Grobstruktur von TOPOS wie aus Abb. 3 ersichtlich festgelegt.

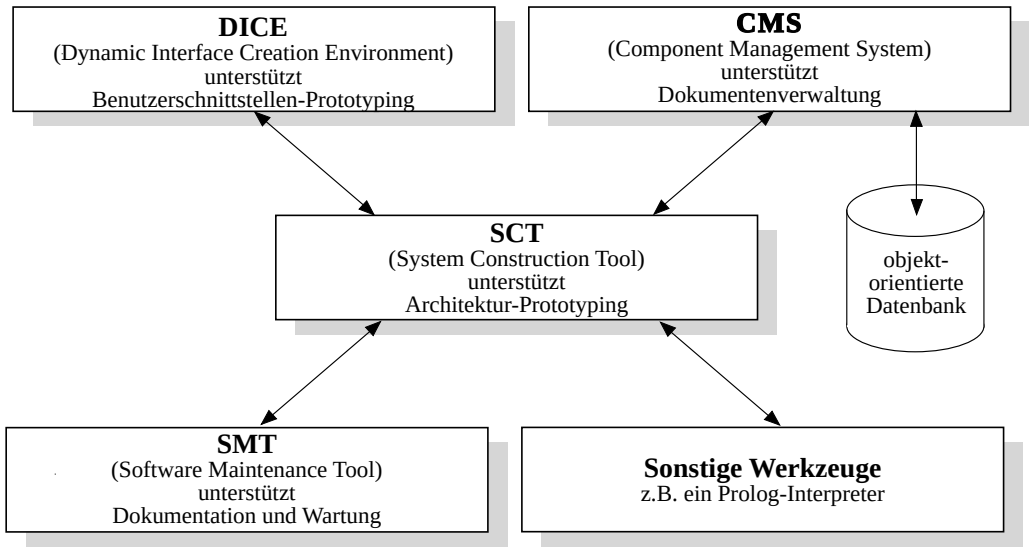

Abb. 3: TOPOS-Grobstruktur

In diesem Aufsatz beschreiben wir nur die Werkzeuge, die in direktem Zusammenhang mit *Prototyping* stehen. Dazu gehören ein Werkzeug zur Erzeugung und Modifikation eines Benutzerschnittstellen-Prototyps (DICE) und ein Werkzeug zur Unterstützung des Architektur-Prototypings (SCT). Eine Beschreibung des Komponentenverwaltungssystems findet sich in [24]. Das Software Maintenance Tool (SMT) ist in [23] beschrieben.

Das Benutzerschnittstellen-Prototyping-Werkzeug DICE

Wir zeigen anhand der Konzepte für das ***D**ynamic **I**nterface **C**reation **E**nvironment*, wie Application Frameworks ergänzt werden können, um diese als mächtige Prototyping-Werkzeuge zu nutzen. Dabei wird insbesondere auf die Spezifikation dynamischen Verhaltens eingegangen.

DICE ist mit der objektorientierten Klassenbibliothek ET++ implementiert. ET++ bietet neben den Klassen zur Programmierung hoch interaktiver grafischer Benutzeroberflächen auch Klassen zur Verwaltung verschiedener Datenstrukturen (dieser Teil entspricht den *Container Classes* in Smalltalk [10] und Applikationsbausteine auf hohem Abstraktionsniveau. ET++ ist in C++ unter UNIX implementiert und ist kompatibel mit den Fenstersystemen SunWindows, NeWS und X11. Eine genauere Beschreibung des Designs und der Implementierung von ET++ findet sich in [9, 27, 28].

Benutzerschnittstelle

Der wichtigste Baustein einer DICE-Applikation ist ein Fenster. Um ein leeres Fenster zu erhalten, wird der Button *Edit Window* im Bedienungspanel von DICE aktiviert (siehe Abb. 4). Um Benutzerschnittstellenelemente in ein Fenster hinzuzufügen, wird das gewünschte Element im Bedienungspanel ausgewählt und im korrespondierenden Fenster an der gewünschten Stelle eingefügt. Neben den üblichen Schnittstellenelementen grafischer Benutzeroberflächen (Buttons, Text- und Zahlenfeldern, Popup-Menü-Elementen, und Teilfenstern) gibt es zwei Gruppierungselemente – *Cluster* und *Expander*. Diese unterstützen die Spezifikation des Fenster-Layouts und des Verhaltens der Benutzerschnittstellenelemente, wenn die Größe des sie umgebenden Fensters verändert wird.

Abb. 4: DICE's Bedienungspanel

Innerhalb jedes Fensters stehen in DICE Editierfunktionen zur Verfügung, wie sie von gängigen WYSIWYG-Grafikeditoren bekannt sind: Die Benutzerschnittstellenelemente können (einzeln oder in Gruppen) verschoben oder in ihrer Größe verändert werden. Die *Cut/Copy/Paste*-Metapher wird innerhalb eines Fensters , zwischen den Fenstern eines Prototypen und zwischen verschiedenen Prototypen unterstützt. Die element-spezifischen Parameter werden über Dialoge angegeben. Abb. 5 zeigt beispielsweise die Attributspezifikation für einen Action Button.

Abb. 5: Beispiel für eine mit DICE spezifizierte Benutzerschnittstelle

Dynamische Aspekte

Ein Software-System ist insbesondere durch sein dynamisches Verhalten charakterisiert. Deshalb muß ein adäquates Benutzerschnittstellen-Prototyping-Werkzeug neben der Spezifikation des statischen Aussehens der Benutzerschnittstelle auch Möglichkeiten bieten, das dynamische Verhalten auf hoher Abstraktionsebene zu definieren und zu simulieren. Um den Implementierungsaufwand hochinteraktiver, grafischer Benutzeroberflächen stark reduzieren zu können, ist es wichtig, die evolutionäre Weiterentwicklung des Prototypen zur fertigen Applikation zu ermöglichen. Dazu bietet DICE verschiedene Möglichkeiten, die im folgenden näher erläutert werden:

- Es können Operationen spezifiziert werden, die den Zustand von Schnittstellenelementen verändern.
- Unterklassen des Application Frameworks ET++, die den spezifizierten Prototyp realisieren, können generiert werden.
- Im Sinne eines UIMS (User Interface Management System, siehe z.B. in [2, 12]) mit gemischter Kontrolle kann der Prototyp mit jedem beliebigen UNIX-Prozeß gekoppelt werden.

Den einzelnen Benutzerschnittstellenelementen sind bestimmte, vordefinierte Nachrichten zugeordnet (z.B. einem Fenster die Nachrichten *Open* und *Close*, einem Textfeld die Nachrichten *Enable*, *Disable* und *SetText(...)*, etc.). Betrachten wir als Beispiel die Benutzerschnittstelle eines einfachen Geldausgabeautomaten (*Cash Dispenser*) aus Abb. 5. Es soll z.B. das *Cash Dispenser* Fenster geschlossen werden, wenn der Button *Stop* aktiviert wird. Um dieses (dynamische) Verhalten zu spezifizieren, aktiviert man den Button *Link* im Parameterdialog des Buttons *Stop* (siehe Abb. 5). Mittels eines sogenannten “Message-Editors” (siehe Abb. 6) wird das erwünschte dynamische Verhalten spezifiziert, nämlich daß die Nachricht *Close* an das Fenster *Cash Dispenser* gesendet wird, wenn der Button *Stop* gedrückt wird.

Abb. 6: Nachrichteneditor

Für jedes aktivierbare Element (das sind die verschiedenen Arten von Buttons und Menüeinträge) kann eine beliebige Anzahl von Nachrichten definiert werden. Wenn der Prototyp simuliert wird (durch Aktivierung des Buttons *Test Prototype* im Bedienungspanel, siehe Abb. 4), werden bei Aktivierung eines Elements die dafür definierten Nachrichten an die Empfänger abgesendet, was die spezifizierte Änderung der Benutzerschnittstelle bewirkt. Auf diese Weise kann rudimentäres dynamisches Verhalten des Prototypen ohne eine einzige Zeile Programmcode realisiert werden.

DICE simuliert das statische und dynamische Verhalten einer Applikation, wenn der Prototyp getestet wird. Es ist somit kein Compile/Link/Go-Zyklus nötig. Um den Prototyp evolutionär weiterentwickeln zu können, bietet DICE die Möglichkeit, Unterklassen von ET++ Klassen zu generieren. Diese Klassen ergeben nach der Übersetzung eine Applikation, die mit dem in DICE spezifizierten Prototypen identisch sind. Der Quelltext der generierten Klassen muß (und soll) nicht geändert werden, wenn zusätzliche Funktionalität hinzugefügt wird. Applikationsspezifisches Verhalten wird in Unterklassen der generierten Klassen imp-

lementiert, indem entsprechende, dynamisch gebundene Methoden überschrieben werden.

Integration der Benutzerschnittstelle mit konventionellen oder objekt-orientierten Systemen

Es ist wünschenswert, daß ein Benutzerschnittstellen-*Prototyping*-Werkzeug die Kommunikation mit anderen (schon vorhandenen) Systemen erlaubt. Da ET++ auf UNIX-Rechnern implementiert ist, wird dazu der UNIX Interprozeß-Kommunikations-Mechanismus verwendet. Die mit DICE spezifizierte Benutzerschnittstelle und der damit kommunizierende Prozeß bilden ein User Interface Management System mit gemischter Kontrolle.

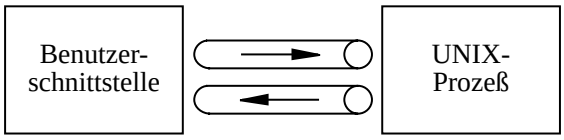

Abb. 8: Kommunikation zwischen Applikation und Benutzerschnittstelle

Die Kommunikation zwischen der eigentlichen Applikation und der Benutzerschnittstelle (siehe Abb. 8) basiert auf einem einfachen Protokoll: Wenn im simulierten Prototypen ein Schnittstellenelement aktiviert wird, sendet der Prototyp den Namen und den Wert des aktivierten Elements an den angeschlossenen Prozeß. Ein Radio Button sendet beispielsweise den Wert *true* oder *false*. Andererseits kann der angeschlossene Prozeß den Wert eines jeden Benutzerschnittstellenelements abfragen.

Auf diese Art kann eine mit DICE entwickelte Benutzerschnittstelle mit jedem anderen Software-System verbunden werden. Die Funktionalität des in Abb. 5 dargestellten Geldausgabeautomaten ist z.B. in C realisiert worden. Änderungen der Funktionalität können unmittelbar getestet werden, nachdem das mit der Benutzerschnittstelle verbundene Software-System neu compiliert und gestartet ist.

Die beschriebene Vorgehensweise ist insbesondere dann sinnvoll, wenn das mit der Benutzerschnittstelle zu verbindende Software-System mit anderen verfügbaren Methoden und Werkzeugen erstellt worden ist.

DICE ist also das Benutzerschnittstellen-Prototyping-Werkzeug der Software-Entwicklungsumgebung TOPOS. Erfahrungen im praktischen Einsatz haben gezeigt, daß Prototypen beträchtlicher Komplexität mit sehr geringem Aufwand (in der Größenordnung von Manntagen) realisiert werden können und die Benutzung von DICE ohne aufwendige Schulung möglich ist.

Das Architektur-Prototyping- und Implementierungswerkzeug SCT

Das SCT (*System Construction Tool*) wurde für die Unterstützung des Prototyp-orientierten, inkrementellen Software-Entwicklungsprozesses konzipiert. Dabei wurden folgende Ziele verfolgt:

(1) Die Entwicklung einer komfortablen Programmierumgebung für exploratives Programmieren in einer Sprache mit strenger Typenprüfung.
(2) Die Erweiterung des explorativen Programmierens hin zum explorativen Entwurf.
(3) Die Unterstützung verschiedener Programmierparadigmen (modulorientiertes, objektorientiertes, funktionales und logisches Programmieren).
(4) Die Unterstützung der Prototyp-orientierten, inkrementellen Software-Entwicklung, indem ein System zur Verfügung gestellt wird, das die Ausführung hybrider Software-Systeme zuläßt.

Unter einem hybriden System bzw. einer hybriden Ausführung verstehen wir die Ausführung eines Systems dessen Komponenten zum Teil direkt ausgeführt und zum Teil interpretativ ausgeführt werden.

Das erste Ziel wurde durch eine interpretative Programmierumgebung für Modula-2 erreicht, deren Funktionalität mit einer explorativen Programmierumgebung wie Smalltalk-80 vergleichbar ist. Um eine zufriedenstellende Ausführungsgeschwindigkeit während der Phase der explorativen Programmierung zu erreichen, wurde ein hybrider Ausführungsmechanismus implementiert, der sowohl die Interpretation der gerade in Entwicklung befindlichen Module als auch die direkte Ausführung der bereits implementierten und getesteten Module erlaubt.

Um das zweite Ziel zu realisieren, ist ein zusätzlicher Ausführungsmechanismus – der *Simulator* – zum hybriden Ausführungsmechanismus hinzugefügt worden. Mit Hilfe des Simulators kann die Architektur eines Software-Systems geprüft werden noch ehe die einzelnen Komponenten implementiert sind, oder wenn Änderungen der Architektur erwogen werden. Dazu werden die Kontroll- und Datenflüsse einer neu entworfenen Modulschnittstelle simuliert, während die anderen Teile der zu implementierenden Applikation interpretiert oder direkt ausgeführt werden.

Durch das Hinzufügen beliebiger anderer Ausführungsmechanismen zu SCT werden die Ziele drei und vier erreicht. Wird beispielsweise ein Prolog-Interpreter integriert, können hybride Software-Systeme ausgeführt werden, von denen Teile in Prolog und Teile in Modula-2 implementiert sind. Ein weiteres typisches Beispiel ist die Kopplung des hybriden Ausführungsmechanismus von SCT mit DICE. Wie im Abschnitt über die "Integration der Benutzerschnittstelle mit konventionellen oder objektorientierten Systemen" beschrieben, fungiert DICE bei der Simulation der Benutzerschnittstelle als Interpreter. Durch die Integration von DICE und SCT kann ein Software-System ausgeführt werden, das aus einer auf sehr abstraktem Niveau spezifizierten Benutzerschnittstelle einerseits und den in Modula-2 oder anderen Formalismen implementierten funktionalen Teilen andererseits besteht.

Die Möglichkeit, hybride Software-Systeme auszuführen, ist Voraussetzung für das Paradigma der Prototyp-orientierten, inkrementellen Software-Entwicklung. Dem Entwickler muß es möglich sein, komplexe hybride Software-Systeme auszuführen und weiterzuentwickeln, ohne ständig "manuell" die Integration von Komponenten, die in verschiedenen Formalismen beschrieben sind, vornehmen zu müssen.

Das Konzept der hybriden Ausführung hybrider Software-Systeme

Hybride Software-Systeme bestehen aus Komponenten, die in verschiedenen Formalismen (Sprachen) implementiert sind. Aus heutiger Sicht kann man Werkzeuge, die diese Art der System-Entwicklung unterstützen, in folgende drei Gruppen einteilen:

(1) Werkzeuge, die einen Code-Generator zur Verfügung stellen, der aus einer auf hohem Abstraktionsniveau beschriebenen Komponente Code in einer algorithmischen Sprache erzeugt. Dieser Code wird manuell mit den anderen Applikationsteilen integriert. Beispiele für solche Werkzeuge sind in [7, 13] beschrieben.
(2) Werkzeuge mit einer interpretativen Entwicklungsumgebung für einen Formalismus auf hoher Abstraktionsebene und einer Schnittstelle zu einer algorithmischen Sprache. Bei diesen Werkzeugen müssen die in der algorithmischen Sprache entwickelten Teile bei der Ausführung der gesamten Applikation mit den auf hoher Abstraktionsebene beschriebenen Teilen entsprechend verbunden werden. Beispiele für derartige Werkzeuge sind in [15, 26] beschrieben.
(3) Werkzeuge, die das Zusammenspiel verschiedener interpretativer Ausführungsmechanismen erlauben, wobei normalerweise einer davon ein Interpreter einer 3.-Generationssprache ist. [1, 11] beschreiben solche Werkzeuge.

Bei Prototyp-orientierter, inkrementeller Software-Entwicklung werden Teile auf niedrigem und hohem Abstraktionsniveau parallel entwickelt. Um einen derartigen Entwicklungsprozeß zu ermöglichen, ist es wichtig, daß das gesamte Software-System sofort wieder ausführbar ist, wenn an einer der Komponenten Änderungen durchgeführt wurden.
Diese Anforderung kann nur von Werkzeugen erfüllt werden, die zur dritten Gruppe gehören, da für Werkzeuge aus der ersten und zweiten Gruppe mehrere Generierungs-, Übersetzungs- und Integrationsschritte nötig sind, bevor das Gesamtsystem wieder ausführbar ist.

SCT ist in die dritte Gruppe einzuordnen. Es unterscheidet sich von anderen Werkzeugen dieser Art dadurch, daß das Zusammenspiel verschiedener interpretativer Ausführungsmechanismen nicht eingeschränkt ist – weder Art noch Anzahl der zu integrierenden Ausführungsmechanismen sind vorgegeben.

Der *Project Manager*

Der *Project Manager* (PM) erleichtert die Verwaltung komplexer Systeme, indem er alle Informationen verwaltet, die zur inkrementellen Entwicklung eines hybriden Software-Systems gehören. Die dazu benötigten Informationen können in folgende Gruppen eingeteilt werden:

- *Dokumente*, die Information über ein Software-System beinhalten (z.B. Spezifikationen, Benutzerschnittstellenprototypen, Modula-2 Quellcode, Diagramme, Schnittstellenbeschreibungen von integrierten Ausführungsmechanismen)
- Attribute, die diese Dokumente klassifizieren
- Pfade, die angeben, an welchen Stellen im UNIX-Verzeichnisbaum die Dokumente gespeichert sind
- Objektcode-Bibliotheken, die zur Ausführung des zu entwickelnden Software-Systems nötig sind
- Informationen über die integrierten Ausführungsmechanismen

Abb. 9: Die Benutzerschnittstelle des *Project Managers*

Die vom PM verwalteten Dokumente sind UNIX-Dateien. Dem Benutzer des PM bleiben alle unnötigen Details des Dateisystems verborgen. Die Dokumentidentifikatoren werden in Listenform präsentiert. Anstatt der Ordnung der Dokumente in einer Verzeichnishierarchie, gibt es Attribute, die den Dokumenten zugeordnet werden. Jedem Dokument können beliebig viele Attribute (z.B. Eigentümer, Zustand, Art des Dokuments, etc.) zugeordnet werden. Basierend auf diesen Attributen sind verschiedene Selektionskriterien anwendbar, um eine Untermenge der im PM verwalteten Dokumente zu erhalten.
Diese auf Attributen aufbauende Verwaltung von Dokumenten hat gegenüber einer Verzeichnishierarchie den Vorteil, daß Dokumente mehreren Kategorien zugeordnet werden können. So ist es beispielsweise möglich, Projekte in (sich zum Teil überlappende) Teilprojekte aufzuteilen. Der Software-Entwickler braucht sich also nicht um das darunterliegende Dateisystem zu kümmern.

UNIX-Applikationen benutzen häufig Funktionen aus Objektcode-Bibliotheken oder binden kompatible Objektcode-Dateien mit ein, die von Compilern unterschiedlicher Sprachen erzeugt wurden. Um solche Applikationen ausführen zu können, verwaltet der PM eine Liste von Objektcode-Bibliotheken und Dateien, die zum SCT wie der Objekt-Code von direkt ausführbaren Modulen dazugebunden werden. Andere Ausführungsmechanismen (z.B. DICE) werden durch eine Interprozeßkomunikationsschnittstelle mit SCT verbunden.

Beim Starten des SCT wird der Project Manager als erstes Werkzeug aktiviert (siehe Abb. 9). Die rechte Liste zeigt die Attribute an, die linke Liste die Dokumente, die diese

Attribute aufweisen. Die Buttons an der linken Seite gestatten es, Aktionen auszulösen, die unabhängig von gerade selektierten Dokumenten oder Attributen sind. Die Buttons unterhalb der Attribut- und Dokumentlisten beziehen sich auf die Selektion von Dokumenten bzw. Attributen. Bei Auswahl eines Dokuments in der Dokumentenliste wird ein sogenanntes *Implementor-Objekt* erzeugt. Ein Implementor gestattet nun die Bearbeitung eines Dokuments, d.h. er stellt die entsprechende Funktionalität dafür zur Verfügung (z.B. eine Programmierumgebung für die Bearbeitung eines Modula-2 Moduls).

Der *Implementor*

Mit einem Implementor kann z.B. ein Modul editiert werden und es kann dann ein Software-System ausgeführt werden, das diesen Modul als Wurzel enthält. Die Anzahl der aktiven Implementoren ist nicht beschränkt. Es kann jedoch nur von einem einzigen Implementor die Ausführung gestartet werden. Ein Implementor wird gestartet, indem im Project Manager der Button *Edit* bzw. *Execute* gedrückt wird. Abb. 10 zeigt die Benutzerschnittstelle eines *Implementors*. Diese besteht aus einer Reihe von Buttons im oberen Teil, die die Analyse, das Ausführen und Debugging steuern, sowie einem komfortablen Texteditor.

Moduln werden aus drei Gründen analysiert: um ihre syntaktische Korrektheit zu prüfen, um Informationen zu erhalten, die zum *Browsing* nützlich sind, und um sie für die Ausführung vorzubereiten.

Bevor ein Modul ausgeführt werden kann, müssen die Implementierungsteile aller interpretierbaren Module, von denen importiert wird, analysiert werden, da eine von dort importierte Funktion oder Prozedur aufgerufen werden könnte. Eine Analyse der direkt ausführbaren Moduln ist ebenfalls nötig, um den Browser-Werkzeugen die benötigten Informationen zur Verfügung zu stellen.

Die Ausführung der interpretierbaren und direkt ausführbaren Module wird durch Drücken des Buttons *Execute* gestartet. Die Ausführung läßt sich durch das Setzen von Haltepunkten und durch die schrittweise Abarbeitung von Anweisungen steuern.

Exploratives Programmieren wird einerseits durch Browser-Werkzeuge unterstützt, die vom *Implementor* aus gestartet werden. Damit erhält man verschiedene Informationen über das Software-System (z.B. die Abfolge von Prozeduraufrufen, komplexe Datenstrukturen, Import- und Export-Abhängigkeiten, usw.). Zur Laufzeit können Daten editiert werden.

Abb. 10: Die Benutzerschnittstelle eines *Implementors*

Andererseits muß bei explorativer Programmierung und für das Architektur-Prototyping die Möglichkeit bestehen, bei der Ausführung des Systems jene Komponenten, die zwar entworfen (d.h. für die eine Schnittstelle definiert ist) aber noch nicht implementiert sind, zu simulieren. Das Werkzeug dazu ist der Architektur-Prototyper (oder Simulator) in Verbindung mit dem hybriden Ausführungssystem.

Der *Simulator*

Der Simulator (Architektur-*Prototyper*) bietet mehrere Dienste im Rahmen der Entwicklung eines Software-Systems mit dem SCT: Zunächst wird er benutzt, um Teile des Software-Systems auszuführen, die zwar entworfen, aber nicht implementiert sind, d.h. für die es z.B. bei einer Modula-2 Implementierung nur die Definitionsmodule gibt. Wird bei der Systemausführung eine als "simulierbar" gekennzeichnete Komponente angetroffen, wird die Ausführung unterbrochen und automatisch der Simulator aufgestartet (Abb. 12 zeigt seine Benutzerschnittstelle). Der Simulator zeigt folgende Informationen an: Die Schnittstelle der zu simulierenden Komponente, die Namen aller bereits existierenden Systemkomponenten (sie können im Zuge der Simulation aufgerufen werden) und weitere Informationen wie z.B. die Aufrufreihenfolge der Komponenten und den Datenfluß im bisherigen Simulationsprozeß.

Der Simulator gestattet nun, die Ausführung der (noch nicht implementierten) Komponenten zu simulieren, d.h. die Input- und Output-Objekte einzustellen, und die Kontrolle an andere Komponenten weiterzureichen. Dieser Verwendungszweck unterstützt das Prototyp-orientierte Ausführen der Systemarchitektur, indem verschiedene Scenarios durchgespielt werden können. Diese Scenarios können protokolliert und wiederholt werden, um die Systemarchitektur nach Veränderungen wieder zu überprüfen. Die protokollierten Simulationsläufe können aber auch später zum Testen der Implementierung verwendet werden. Dabei wird automatisch geprüft, ob sich die Implementierung tatsächlich so verhält, wie in dem protokollierten Scenario aus dem Prototyping-Prozeß vorgesehen.

Der vom Simulator zur Verfügung gestellte Mechanismus zum Komponentenaufruf kann weiters zum Testen von Modulen und Applikationsteilen verschiedenen Fertigungsgrades verwendet werden. Um einen Modul zu testen, werden dessen Prozeduren mit verschiedenen Eingabeparametern aufgerufen und die jeweils erhaltenen Ergebnisparameter auf Korrektheit geprüft. Die Testfälle können

gespeichert und wieder durchgespielt werden, wenn die Implementierung des Moduls geändert wurde. Beim der Wiederholung des Tests vergleicht der *Simulator* die erwarteten Werte der Ergebnisparameter mit den tatsächlich Werten. Dadurch wird der Aufwand für die Modultests stark reduziert.

Die Benutzerschnittstelle des Simulators (Abb. 11) besteht aus drei Gruppen von Buttons, die im oberen Teil des Fensters angeordnet sind, und sechs Teilfenstern, die die verbleibende Fensterfläche ausfüllen.

In den drei Teilfenstern an der linken Seite werden Moduln, die darin definierten Prozeduren und deren Parameter inspiziert. Das Modul-Teilfenster enthält eine Liste aller gerade verfügbaren Moduln. Nach der Selektion eines Moduls wird im darunterliegenden Prozedur-Teilfenster die Liste aller in diesem Modul definierten Prozeduren angezeigt. Die Selektion einer Prozedur bewirkt schließlich die Anzeige deren Schnittstelle im Parameter-Teilfenster.

Die drei rechts angeordneten Teilfenster enthalten die für die Simulation relevanten Daten: Im Teilfenster *Local Data* werden die Parameter der gerade simulierten Komponente angezeigt. Das Teilfenster *Called Procedures* enthält eine Liste aller Prozeduren, die von der gerade simulierten Prozedur aus aufgerufen wurden. Im Teilfenster *Procedure Call Chain* ist eine Liste aller gerade aktivierten simulierten, interpretierten oder direkt ausführbaren Prozeduren dargestellt.

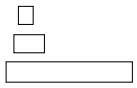

Abb. 11: Die Benutzerschnittstelle des *Simulators*

Zur Simulation eines Software-Systems müssen zwei Voraussetzungen erfüllt sein: Zum einen muß ein Komponentenaufruf simuliert werden können: Im Simulator erfolgt dies durch die Selektion der aufzurufenden Komponente im Prozedur-Teilfenster. Anschließend werden die lokalen Daten mit dem Befehl *Activate* angelegt und dann die Eingabeparameter editiert. Schließlich wird die Prozedur mit dem *Execute*-Befehl simuliert. Die zweite Voraussetzung zur Simulation ist die Möglichkeit der Rückkehr von einer simulierten Prozedur: Im Simulator wird dazu der Befehl *Return* im Teilfenster *Procedure Call Chain* abgesetzt.

Der *Workspace Manager*

Das SCT erlaubt die Ausführung von Teilen einer Applikation, für die Eingabeparameter zur Verfügung gestellt werden, und von denen Ausgabeparameter erwartet werden. Wenn dabei Datenstrukturen editiert werden müssen (z.B. bei Simulation einer noch nicht implementierten Systemkomponente), dann kann dies mit Hilfe der Runtime-Editoren bewerkstelligt werden. Mit Hilfe des Workspace Managers können komplexe Datenstrukturen gespeichert und für andere Komponenten zur Weiterverarbeitung verfügbar gemacht werden.

Bei den meisten interpretativen Programmierumgebungen gibt es nur einen globalen Workspace, in dem alle globalen Daten gespeichert sind. Das genügt auch für den Systementwickler, um die Ergebnisparameter von Applikationsteilen für spätere Verwendungszwecke zu speichern. Außer diesem einen, globalen Workspace stellt SCT für jeden einzelnen Systembaustein einen Workspace für Simulationszwecke zur Verfügung.

Hinzufügen anderer Ausführungsmechanismen

Zur Integration interpretativer Ausführungsmechanismen in eine hybride Umgebung müssen vier Bedingungen erfüllt sein:

(1) Das zu integrierende Werkzeug muß eine Schnittstelle bieten, um mit anderen Formalismen kooperieren zu können.
(2) Der Ausführungsmechanismus muß ein offenes System sein. D.h., eine Schnittstelle muß vorhanden sein, von der bestimmte Leistungen von anderen Werkzeugen aus verlangt werden können. Die meisten interpretativen Ausführungsmechanismen haben außerdem eine Schnittstelle, von der aus Leistungen anderer Werkzeuge in Anspruch genommen werden können.
(3) Die Schnittstellen der verschiedenen Werkzeuge müssen semantisch zueinander passen. Die Formalismen müssen zueinander insofern kompatibel sein, daß eine sinnvolle Synchronisation und ein Datenaustausch möglich sind.
(4) Es muß eine gemeinsame Plattform geben, in der die Schnittstellen der verschiedenen Werkzeuge integriert werden können.

SCT's hybrides Ausführungssystem für Modula-2 ist eine solche Plattform, in der interpretative Ausführungsmechanismen, die unter UNIX geboten werden, integrierbar sind.

Um eine solche Integration durchzuführen, muß die Schnittstelle des zu integrierenden Systems zur Applikation hinzugefügt werden. Die neue Schnittstelle ermöglicht den bereits integrierten Werkzeugen sowie anderen Teilen der Applikation, darauf zuzugreifen, indem wiederum der hybride Prozeduraufrufmechanismus benutzt wird. Ein dynamisches Binden eines Aufrufes ist mit SCT's *ProcCallDispatcher*-Schnittstelle möglich. Bisher wurden ein Prolog-Interpreter (für Expertensystemimplementierungen), eine relationale Datenbank (für Datenbank-orientierte Applikationsentwicklung) und das DICE in das SCT integriert.

Implementierung

TOPOS wurde auf SUN-Workstations unter dem Betriebssystem UNIX implementiert und Teile davon wurden auf Siemens PC 16-20 Rechner übertragen. Zur Implementierung sind die Sprachen Modula-2, C und C++ sowie die objektorientierte Klassenbibliothek ET++ verwendet worden. Eine Beschreibung der Implementierung ist in [3] enthalten.

Erfahrungen

Das *Dynamic Interface Creation Environment*, das *System Construction Tool* sowie das *Component Management System* werden seit Frühjahr 1989 angewendet, das *Software Maintenance Tool* ist seit Frühjahr 1990 in Verwendung.
Der praktische Einsatz dieser Werkzeuge hat gezeigt, daß sich die Art und Weise, wie wir Software entwickeln, drastisch geändert hat (sowohl die Vorgehensweise als auch die Architektur und die Aufwandsverteilung auf die verschiedenen Phasen), und daß beträchtliche Rationalisierungs- und Qualitätssteigerungseffekte erzielt werden konnten. Die Integration von Prototyping und explorativem Programmieren hat sich als äußerst nützlich erwiesen.

Literatur

1. Acius Inc., *4th Dimension*. User Documentation, 1987.

2. Betts, B. et al.: *Goals and Objectives for User Interface Software*. Computer Graphics, Vol. 21, No. 2, 1987.

3. Bischofberger W.R.: *Prototyping-Oriented Incremental Software Development: Paradigms, Methods, Tools and Implications*. Dissertation, Johannes Kepler Universität Linz, 1990.

4. Boehm, B.W.: *A Spiral Model of Software Development and Enhancement*. IEEE Computer, Vol. 21, No. 5, 1988.

5. Budde, R. et al.: *Approaches to Prototyping*. Proceedings of the Working Conference on Prototyping, Namur, 1983, Berlin, Heidelberg, New York: Springer 1984.

6. Connell J.L., Shafer L.: *Structured Rapid Prototyping*. Yourdon Press series, Englewood Cliffs: Prentice Hall 1989.

7. Cossey, G.R.: *Prototyper*. Users Guide, Smethers Barnes Inc., Portland, Or., 1987.

8. Floyd, Ch.: *A Systematic Look at Prototyping*. Approaches to Prototyping, Berlin, Heidelberg, New York: Springer 1984.

9. Gamma E., Weinand A., Marty R.: *Integration of a Programming Environment into ET++ A Case Study*. Proceedings of the 1989 ECOOP, July 1989.

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

11. Goodman, D.: *Danny Goodman's HyperCard Developer's Guide*. New York: Bantam Books 1988.

12. Hayes, P.J., Szekely, P.A., Lerner, R.A.: *Design Alternatives for User Interface Management Systems Based on Experience with COUSIN*. Human Factors in Computing Systems: CHI'85 Conference Proceedings, Boston, Mass., 1985.

13. Hulot, J-M.: *Exper Interface Builder*. Users Guide, 1987.

14. Kaufer, S., Lopez, R., Pratap, S.: *Saber-C, An Interpreter-Based Programming Environment for the C Language*. Proceedings of USENIX, San Francisco, 1988.

15. Keller, R.K.: *Prototyping-orientierte Systemspezifikation–Konzepte, Methoden, Werkzeuge und Konsequenzen*. Hamburg: Dr. Kovac, 1989.

16. Knuth, D.E.: *Semantics of Context-free Languages*. Mathematical Systems Theory 2, 1968.

17. LaLonde, W.R., Pugh, J.R.: *Inside Smalltalk*. Volume I, Englewood Cliffs: Prentice Hall 1990.

18. LPA Ltd., *MacPROLOG Reference Manual*. 1987.

19. Pomberger, G.: *Integration von Prototyping in Software-Entwicklungsumgebungen*. Oesterle, H. (Hrsg.): Anleitung zu einer praxisorientierten Softwareentwicklungsumgebung, München: Angewandte Informationstechnik Verlagsgesellschaft GmbH 1988.

20. Pomberger, G.: *Methodik der Software-Entwicklung*. Handbuch der Wirtschaftsinformatik, Stuttgart: Poeschl 1990.

21. Pree, W.: *DICE–An Object-Oriented Tool for Rapid Prototyping*. Proceedings of the Tools Pacific 1990 Conference, Sydney, Australia 1990.

22. Saber Inc.: *Saber-C Enhances Development for C Programmers*. The Sun Observer, Vol. 1, No. 4, 1988.

23. Sametinger, J.: *A Tool for the Maintenance of Object-Oiented Systems*. Proceedings of the Conference on Software Maintenance 1990, San Diego, 1990.

24. Schmidt, D.: *Persistente Objekte und objektorientierte Datenbanksysteme: Konzepte, Architektur, Implementierung und Anwendung*. Dissertation, Universität Zürich, 1990.

25. Stritzinger, A.: *Architecture and Dynamics of Smallkit: An Easy-To-Learn Application Framework*. Proceedings of the Tools Pacific 1990 Conference, Sydney, Australia, 1990.

26. SUN Microsystems Inc.: *SUN Unify*. User Documentation, 1986.

27. Weinand A., Gamma E., Marty R.: *ET++ - An Object-Oriented Application Framework in C++*. OOPSLA 88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.

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

29. Xerox Corporation, *Interlisp*. User Documentation, 1983.

Eingetragene Warenzeichen:

SunWindows und NeWS sind eingetragene Warenzeichen von Sun Microsystems.

UNIX ist ein eingetragenes Warenzeichen von AT&T.