Software-Entwicklung durch schrittweise Komplettierung

J. Ludewig, H. Färberböck, H. Lichter, H. Matheis, E. Wallmüller
Institut für Informatik, ETH-Zentrum, CH 8092 Zürich

Zusammenfassung

Die Arbeit beschreibt die Software-Entwicklung durch schrittweise Komplettierung. Darin sind die Prinzipien der traditionellen Programmentwicklung nach einem Phasenplan mit dem Prototyp-Ansatz verbunden. Auf diese Weise bleiben die Vorteile beider Ansätze erhalten. Das Vorgehen erfordert starke Unterstützung durch die Software-Entwicklungsdatenbank. Diese Datenbank wird gegenwärtig in unserer Gruppe realisiert.

Definitionen

Um möglichen Mißverständnissen vorzubeugen, geben wir zunächst unsere Definitionen der in diesem Kontext wichtigen Begriffe.

Zielsystem, Zielprogramm, Zielsprache

Das Zielsystem ist das System, das am Ende der Entwicklung an den Kunden ausgeliefert wird. Es enthält als maschinell übersetzbaren Teil das Zielprogramm. Es ist (bis auf Strukturangaben) in der Zielsprache codiert.

Dokumentation, Software, Programm

Dokumentation ist die Gesamtheit aller permanent gespeicherten, verfügbaren Information, die im Laufe einer Software-Entwicklung entsteht und für das Zielprogramm (Gebrauch und Wartung) relevant ist. Software wird als Synonym für Dokumentation betrachtet. Das Programm ist der maschinell übersetzbare Teil der Software.

Programm-Architektur, Software-Architektur

Die Programm-Architektur ist die Grobstruktur des Programms. Sie besteht aus

- der statischen Struktur des Systems, also Gliederung in Module und Angabe des wesentlichen Zwecks jedes Moduls und der Beziehungen zwischen den Modulen,
- der dynamischen Struktur des Programms, also den Datenfluß- und Ablauf-Beziehungen zwischen den Modulen.

Die Software-Architektur umfaßt außer der Programm-Architektur auch die Strukturen der übrigen Dokumente (beispielsweise Handbücher), also ihre Gliederung, ihren Zweck und ihre Beziehungen.

Modell eines Systems

Ein Modell ist ein zweites System, das mit dem Original in wichtigen Eigenschaften übereinstimmt und die Möglichkeit gibt, Aussagen über das Original zu treffen, die mit dem Original selbst nicht oder nur mit größerem Aufwand möglich sind (beispielsweise, weil das Original noch nicht existiert oder der Beobachtung nicht zugänglich ist).

Prototyp, Prototyp-Sprache

Ein Prototyp ist ein dynamisches (d.h. ausführbares) Modell des Zielsystems (System-Prototyp) oder eines Teils davon (Modul-Prototyp). Ein Modul-Prototyp ist in einer formalen Sprache, der Prototyp-Sprache, realisiert. Der System-Prototyp besteht aus Modulen (Modul-Prototypen und Module in Zielsprache) und einer Strukturbeschreibung (siehe Software-Architektur).

Prototyping

Prototyping ist die Anfertigung eines Prototyps.

Software-Architektur-Prototyping

Dies ist eine Form des Prototyping, bei der ein Modell der Software entsteht, das hinsichtlich der Architektur mit dem Zielsystem übereinstimmen soll. D.h. nach der Evolution des Prototyps definiert dieser die Architektur des Zielsystems.
Architektur-Prototyping unterscheidet sich damit nicht durch die Vorgehensweise, wohl aber durch die Zielsetzungen vom Benutzer-Schnittstellen-Prototyping.

Traditionelle Programm-Entwicklung, Prototyp-Ansatz und evolutionäre Programmentwicklung

In der Regel werden Programme heute nach einem *Phasenplan* entwickelt (Boehm, 1976; Boehm, 1983). Dieses Verfahren ist dadurch gekennzeichnet, daß das Produkt, also die Software, in einem einzigen Durchgang spezifiziert, entworfen, implementiert, integriert, getestet und korrigiert wird. In der Praxis werden oft nicht alle Phasen wirklich konsequent durchgezogen, doch das Prinzip ändert sich dadurch nicht.

Seit einigen Jahren ist alternativ zu diesem Vorgehen der *Prototyp-Ansatz (Rapid Prototyping)* propagiert worden (Boehm, Standish, 1983; Budde et al., 1984; Budde, 1986). Dabei wird zunächst ein System implementiert, das mit dem Zielsystem in wichtigen Punkten übereinstimmt, so daß es anstelle des Zielsystems wenigstens für Experimente gebraucht werden kann. Vor allem die Benutzerschnittstelle ist vielfach Gegenstand der Betrachtung. Die Ergebnisse der Versuche dienen dazu, die Anforderungen zu konkretisieren und zu korrigieren. Dies hat, wie Versuche gezeigt haben, deutliche Vorteile (Boehm, Gray, Seewald, 1984). Der Prototyp selbst wird nach dieser Phase weggeworfen. Damit bedeutet Prototyping, daß innerhalb der traditionellen Spezifikationsphase ein kompletter Entwicklungsgang abläuft.

Mit dem Prototyp-Ansatz verwandt ist die schrittweise oder evolutionäre

Entwicklung *(Iterative Enhancement)*. Dabei entsteht zunächst ein rudimentäres, aber bereits lauffähiges, eventuell bereits brauchbares Programm, das den Kern des Zielsystems bildet. Dieses wird dann schrittweise ergänzt, bis die volle Funktionalität bereitgestellt ist.

Beide neuen Ansätze haben den Vorteil, daß schneller als bei der traditionellen Entwicklung ein lauffähiges Programm entsteht. Dies erlaubt die frühe Überprüfung der Anforderungen ("Hat der Kunde wirklich gesagt, was er will, und steht das auch in den Anforderungen?"). Außerdem wirkt die Existenz eines laufenden Programms erfahrungsgemäß stimulierend auf die Entwickler: Der Prototyp-Ansatz und die schrittweise Entwicklung erhöhen die Motivation.

Beim Prototyping braucht keine effiziente, leicht wartbare Software zu entstehen, man arbeitet "quick and dirty", vorzugsweise mit Programmiersprachen, die diesen Stil unterstützen (etwa LISP). Dafür muß man das Ergebnis wegwerfen. Umgekehrt muß man bei der evolutionären Entwicklung von Beginn an qualitativ hochwertige Lösungen hervorbringen.
Dieser Beitrag beschreibt einen Ansatz, der darauf zielt, die Vorteile beider Verfahren zu verbinden.

Konzept für einen gleitenden Übergang vom Prototypen zum Produkt

Unser Ansatz ist dadurch gekennzeichnet, daß wir den Verlust der bei der Prototyp-Entwicklung zusammengetragenen Information vermeiden wollen, indem wir nicht den *vollständigen* Prototypen durch eine *vollständige* Implementierung in Zielsprache ersetzen, sondern den Übergang schrittweise vollziehen, jeweils für einzelne Komponenten. Dieser Ansatz hat folgende Vorteile:

- Wie beim Prototyp-Ansatz entsteht schnell ein lauffähiges Programm, das die Überprüfung der Spezifikation und des Lösungsansatzes erlaubt und die Motivation hebt.

- Einzelne Komponenten können in beliebiger Reihenfolge verfeinert werden. Damit ist es möglich, kritische Teile sehr früh zu implementieren (zum Nachweis der Realisierbarkeit unter den gegebenen Randbedingungen, beispielsweise Effizienzforderungen).

- Einmal in einer ersten Version erstellt, bleibt der Prototyp ständig ausführbar. Damit entfällt die Durststrecke zwischen der Fertigstellung des Prototyps und der des "harten" Programms (Zielprogramm).

- Die Struktur des Systems kann schnell und leicht verändert werden, solange es in einer permissiven Prototyp-Sprache beschrieben ist, die bezüglich Typbindung und Schnittstellen-Konsistenz weniger streng ist als die Zielsprache. Entschließt man sich, auf die Zielsprache überzugehen, so gehen die Strukturinformationen nicht verloren.

Die Abb. 1 zeigt den Verlauf einer schrittweisen Vervollständigung.

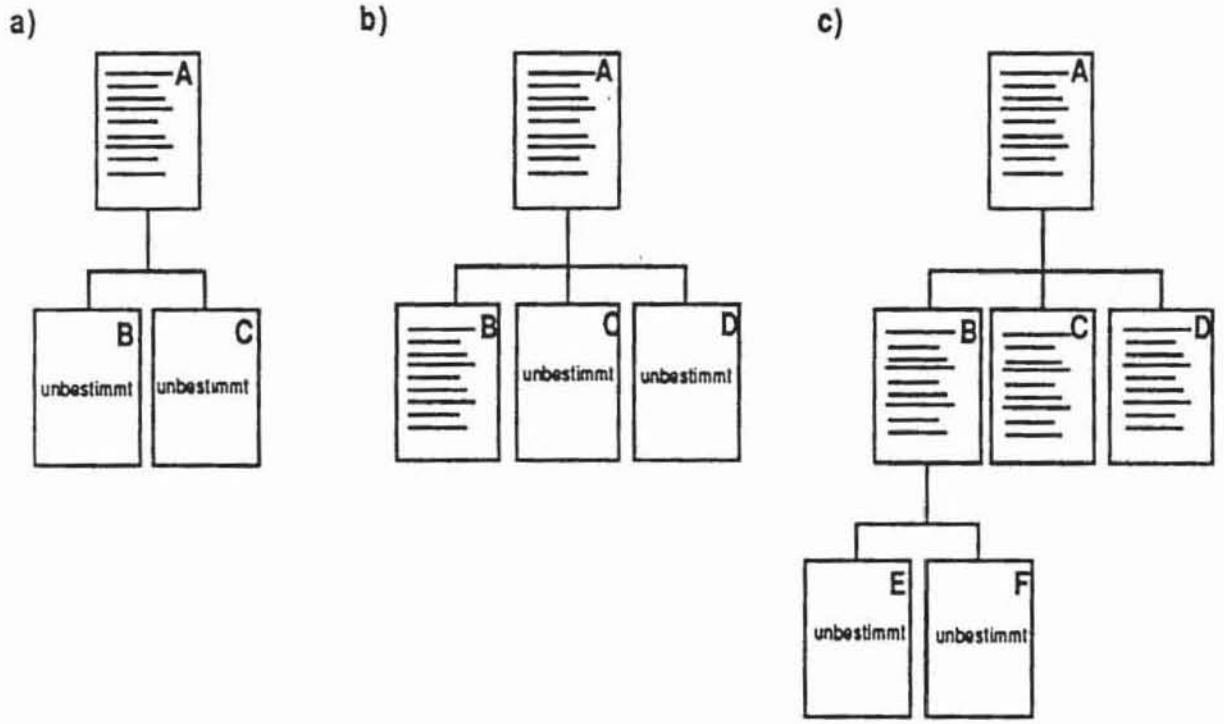

Abb. 1: Verlauf einer schrittweisen Vervollständigung beim Modulentwurf

In Abb. 1a ist der Modul A beschrieben. Die für die Leistungen von Modul A benötigten Module B und C sind als Rahmen dargestellt, jedoch noch nicht weiter beschrieben (unbestimmt). Die Leistung des Moduls A kann nur durch Verwendung eines zusätzlichen Moduls D vollständig erbracht werden. Dieser Modul wurde vergessen (fehlende Komponente).
In Abb. 1b wurde das Verhalten des Moduls B beschrieben, jedoch noch nicht weiter detailliert (fehlende Detaillierung) und der vergessene Modul D als Rahmen eingefügt.
In Abb. 1c schließlich wurden die Module C und D beschrieben und der Modul B durch die Modulrahmen E und F weiter detailliert.

Entsprechend der sukzessiven Vervollständigung, die vom vagen, unvollständigen Prototypen zum Zielprogramm führt, sprechen wir bei unserem Ansatz von **schrittweiser Komplettierung** (**P**rogram **D**evelopment by **S**tepwise **C**ompletion = **PDSC**). Er ist bestimmt durch die Vorstellung, daß eine scharfe Trennung zwischen der Spezifikations- und der Entwurfsphase unrealistisch und unzweckmäßig ist, weil sich die Anforderungen zu einem großen Teil erst durch Entwurfsentscheidungen ergeben (Swartout, Balzer, 1982; Ludewig, 1982).
Ähnliche Ideen wurden schon früher veröffentlicht, allerdings ohne konkreten Plan einer Realisierung (Beregi, 1984).

Sprachen für PDSC

Für PDSC benötigen wir neben einer natürlichen Sprache (für die informale Dokumentation) und einer Kommando-Sprache, auf die wir hier nicht weiter eingehen, folgende Sprachen (vgl. Ludewig, Glinz, Matheis, 1985):

1. eine Beschreibungssprache für die Software-Architektur, also für das "Programming in the Large" *(Architektur-Sprache)*

2. eine Sprache zur Realisierung der Modul-Prototypen *(Prototyp-Sprache)*

3. eine Sprache, in der das Zielprogramm schließlich codiert wird *(Ziel-Sprache)*

Die Architektur-Sprache muß folgenden Anforderungen genügen:

- Die Beschreibung aller Entitäten (Module, Dokumente) und ihrer Beziehungen muß leicht möglich sein.
- Es müssen unvollständige und vage Beschreibungen erlaubt sein.
- Die Informationen müssen ausreichen, um einen System-Prototypen zu konfigurieren.
- Die Sprache muß sich mit geringem Aufwand auf ein Schema der Software-Entwicklungsdatenbank abbilden lassen (aus der Architektur-Sprachen-Beschreibung muß sich das Schema der Datenbank generieren lassen).
- Die Sprache soll maschinell verarbeitbar, also formal sein.

Für die Prototyp-Sprache gilt:

- Sie soll übersetzbar oder (besser) interpretierbar sein.
- Sie soll die schnelle Realisierung der Module gestatten.
- Sie soll wie die Architektur-Sprache auch abstrakte, unvollständige und vage Formulierungen gestatten, die bei der Ausführung durch Standard-Behandlung oder interaktiv ergänzt werden.

Von der Zielsprache fordern wir:

- Die Sprache muß die Wartung des Programms erleichtern (durch eine gut lesbare Notation, Deklarationszwang, freie Syntax für Bezeichner usw.).
- Sie muß die Typ-Konsistenz sicherstellen (Strong Typing).
- Sie muß das Modul-Konzept anbieten und die separate Übersetzung gestatten.
- Sie muß in effizienten Code compilierbar sein.

Für alle Sprachen gilt:

- Sie sollen nicht auf spezielle Anwendungsbereiche zugeschnitten, sondern allgemein einsetzbar sein.
- Die Notation soll leicht lesbar sein.
- Die Konzepte der Sprachen müssen kompatibel sein, damit die Schnittstellen einfach und die Übergänge leicht möglich sind.

Vorgehen bei PDSC

Übersicht

Wie bisher steht am Anfang der Entwicklung eine Spezifikation, doch werden die Anforderungen an diese gegenüber dem traditionellen Modell stark reduziert. Das heißt nicht, daß in dieser Phase nachlässig gearbeitet werden soll. Stattdessen wird nicht krampfhaft versucht, eine Klarheit und Vollständigkeit zu erzwingen, für die zu diesem Zeitpunkt einfach nicht genügend Informationen vorliegen. Die Spezifikation wird also informal, unvollständig und vage sein.

Auf dieser Grundlage wird zunächst ein lauffähiges, i.a. sehr ineffizientes und unvollständiges Modell des Zielsystems geschaffen. Zweck dieses Systems ist die Überprüfung der Anforderungen und die Festlegung einer Lösungskonzeption. Das Modell wird ausgeführt, wobei Unvollständigkeiten interaktiv ausgeglichen werden.

Hat das Modell ein insgesamt befriedigendes Verhalten erreicht, so werden Komponenten darin (Module, Funktionen, Prozeduren) sukzessive ersetzt durch "gewöhnlichen" Code, also dargestellt mit den Mitteln einer traditionellen Programmiersprache (z.B. MODULA-2 oder Ada). Auf diese Weise findet eine Präzisierung und Fixierung der Programme statt, bis alle Elemente exakt dargestellt sind und das gleiche Ergebnis erreicht ist wie bei einer traditionellen Entwicklung. Die Ausführbarkeit bleibt dabei ständig erhalten.

Das Vorgehen läßt sich durch folgendes Tätigkeitsmodell beschreiben: Die einzelnen Aktivitäten beim PDSC sind:

A Aufnahme einer informalen Spezifikation

S Strukturierung der informalen Spezifikation

L Entwurf oder Modifikation einer Lösungsstruktur (d.h. Festlegung der Architektur)

P Implementierung von (rudimentären) Lösungen in der Prototyp-Sprache und Einbau in die Lösungsstruktur

E Erprobung und Analyse des System-Prototypen

Z Implementierung von Komponenten in der Zielsprache und Integration

Natürlich beginnt die Arbeit mit den Aktivitäten A, S, L, P und E. Danach ist die Reihenfolge weitgehend beliebig. Durch Prüfung der Zwischenergebnisse (insbesondere mit E) sollte erreicht werden, daß für jede einzelne Komponente die Tätigkeit Z nur einmal ausgeführt wird; dieses Ziel wird sich in kritischen Fällen nicht erreichen lassen.

Nachfolgend werden die Tätigkeiten einzeln diskutiert. Dabei sind jeweils Rollenbezeichnungen angegeben für die Personen, die die betreffende Tätigkeit ausführen. Eine bestimmte Person kann auch mehrere dieser Rollen innehaben (siehe beispielsweise Schritt E).

Natürlich kann sich während der Entwicklung eines Prototyps jederzeit herausstellen, daß das geplante Projekt unter den gegebenen Bedingungen nicht realisierbar ist. Dann müssen entweder die Randbedingungen geändert oder die Arbeiten eingestellt werden. Nach Fertigstellung des Prototyps sollte die Realisierbarkeit aber gesichert sein.

A Aufnahme einer informalen Spezifikation

Am Anfang der Entwicklung steht naturgemäß die Analyse *(Analytiker)*. Die Unterstützung durch PDSC ist gering. Da die Information zunächst kaum Struktur aufweist, ist ein Texteditor das geeignete Werkzeug.

S Strukturierung der informalen Spezifikation

Ziel dieser Aktivität ist die Gliederung des Problems in überschaubare Teilprobleme *(Spezifizierer)*, damit auch die Vorbereitung des Entwurfs (L). Wir kommen damit in den Einsatzbereich der Architektur-Sprache und der für diese vorgesehenen Werkzeuge. Diese müssen die Darstellung der Strukturen (Hierarchie, Querbezüge) gestatten.

L Entwurf und Modifikation einer Lösungsstruktur

Auf der Basis der Ergebnisse von S wird die Software-Architektur entworfen, eine Struktur, die die Bezeichner, den Zweck und die Beziehungen der Module (Programm-Architektur) und der übrigen Dokumentation festlegt *(Entwerfer)*.

Wir verwenden Objekte folgender Arten zur Beschreibung:

- Abstraktionsebenen (virtuelle Maschinen)
- Statische Einheiten: Dokumente, Module (incl. Daten-Kapseln und Abstrakte Datentypen)
- Dynamische Einheiten: Prozeduren
- Abstraktionen von Berechnungsvorschriften: Funktionen
- Es ist denkbar, auch endliche Automaten und Prozesse als spezielle Einheiten zu betrachten.

Die Beziehungen zwischen den Objekten lassen sich grob wie folgt einteilen:

- Statische Strukturen: Zuordnung (zu Schichten), Hierarchien
- Dynamische Strukturen: Aufrufbeziehungen (Benutzt-Relation)
- Andere logische Beziehungen, beispielsweise Querverweise
- Projektbezogene Beziehungen,
  z.B. "wird implementiert durch", "dokumentiert", "ersetzt".

P Implementierung der Module in der Prototyp-Sprache und Einbau in die Lösungsstruktur

In das Gerüst der Software-Architektur werden nun Prototypen der Module "eingehängt" *(Prototyp-Implementierer)*. Damit entsteht erstmals ein ablauffähiges System. Soweit Module bereits in Zielsprache vorliegen oder mit geringem Aufwand in Zielsprache codiert werden können, entfällt natürlich die Implementierung als Prototyp. Durch die Eigenschaften der Prototyp-Sprache ist gewährleistet, daß der Prototyp-Implementierer mit geringem Aufwand und ohne starke Einschränkung durch die "Sicherheitsgurte" einer Programmiersprache (z.B. Typüberwachung) zu Ergebnissen gelangt.

E Erprobung und Analyse des System-Prototypen

Sobald der Prototyp in einem minimalen Sinne vollständig ist (Ausführbarkeit), kann er erprobt und geprüft werden *(Prototyp-Prüfer)*. So prüfen

- Analytiker und Spezifizierer gegen die Anforderungen,
- der Kunde gegen seine (oft nicht formulierten) Erwartungen und
- Fachleute für spezielle Aspekte (z.B. Effizienz, Wartbarkeit) gegen die betreffenden (meist nicht-funktionalen) Anforderungen.

Die Erprobung findet jeweils auf der Grundlage der vollständigsten Konfiguration statt, die zu diesem Zeitpunkt möglich ist. So werden jüngere Versionen den älteren vorgezogen, solche in Zielsprache denen in Prototyp-Sprache.

Diese Tätigkeit hat primär zum Ziel, Mängel bezüglich der Anforderungsspezifikation und der Software-Architektur zu erkennen. Es wird dabei festgestellt, ob der Prototyp gewissen Anforderungen genügt, die vom Auftraggeber oder vom Hersteller an das Produkt gestellt werden. Die Beseitigung der Mängel erfordert in diesem Stadium wesentlich weniger Aufwand als nach der Codierung in Zielsprache.

Sekundäre Ziele sind:

- Unterstützung des Software-Engineering-Managements, insbesondere in Bezug auf die Planung, Kostenschätzung und Qualitätssicherung
- Validierung der Anforderungsspezifikation
- Steigerung der Motivation der Entwicklergruppen, da bereits sehr früh ein lauffähiges und damit auch demonstrierbares Modell des Produktes vorliegt
- Verbesserung der Kommunikation zwischen den an der Entwicklung beteiligten Personen und Gruppen, inbesondere Entwickler und Auftraggeber

Das Ergebnis dieser Erprobung ist eine Prototypbewertung. Die Erprobung endet, wenn das Ergebnis insgesamt akzeptabel ist. Andernfalls müssen die Tätigkeiten L, P und E erneut ausgeführt werden, im ungünstigsten Fall auch A und S. Abb. 2 zeigt das Tätigkeiten-Modell in graphischer Form. Der Prototyp-Zyklus (L, P, E) ist hervorgehoben.

Z Codierung von Komponenten in der Zielsprache und Integration

Ist eine Komponente des Prototyps mit ausreichender Sicherheit stabil, so kann sie in Zielsprache codiert werden. Die Implementierung wird schon früher stattfinden, wenn sie zur Klärung wichtiger Fragen notwendig ist, beispielsweise, wenn die Antwortzeiten eines Systems eine kritische Größe darstellen.

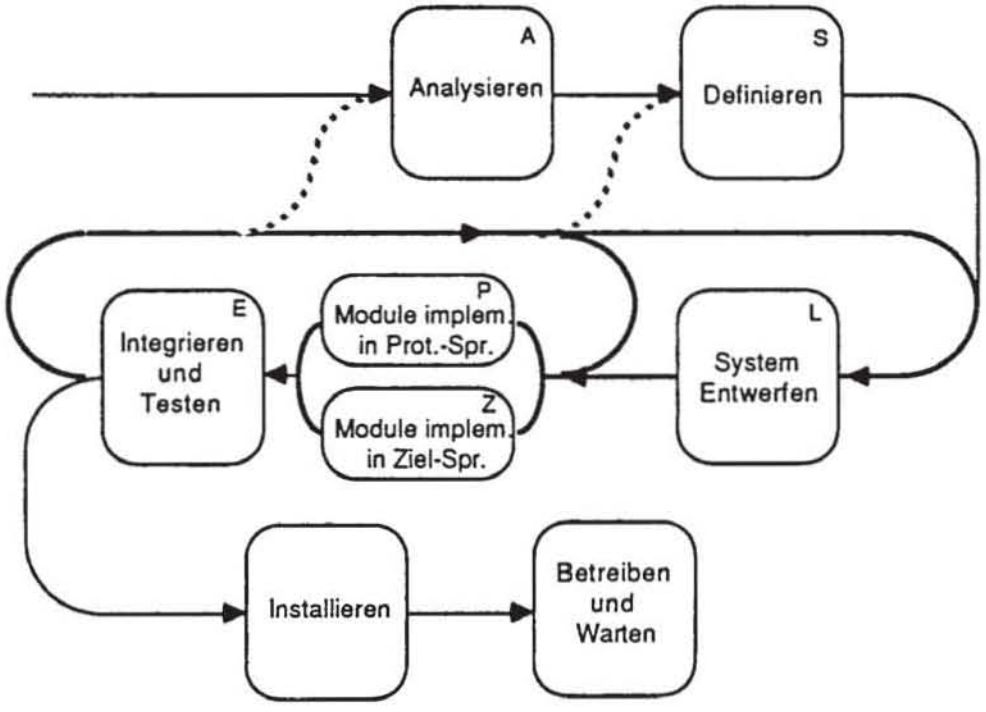

Abb. 2: Tätigkeiten-Modell für PDSC

Werkzeuge

Das beschriebene Konzept ist nur realisierbar mit einer Software-Entwicklungsumgebung (Software Engineering Environment), die folgende Komponenten enthält:

- Baustein zur syntaktischen Prüfung und Verarbeitung der Programmbeschreibungen (Architektur-, Prototyp- und Ziel-Sprache)

- Software-Entwicklungsdatenbank, die die Architektur-Beschreibung aufnimmt und damit alle anderen Objekte verknüpft.
  Aufgrund der Eigenschaften einer Architektur-Beschreibung ist vor allem eine Datenbank nach dem Entity-Relationship-Prinzip für diesen Zweck geeignet.

- System-Prototyp-Konfigurator
  Der Prototyp-Konfigurator wertet die Architektur-Beschreibung aus, wählt die präziseste verfügbare Beschreibung jeder Komponente aus und integriert das System. Auf diese Weise entsteht schließlich auch das Zielsystem.

- System-Prototyp-Simulator
  Solange das System noch nicht vollständig in Zielsprache implementiert ist, führt der Prototyp-Simulator den Prototypen (erzeugt vom Prototyp-Konfigurator) aus und kompensiert Unvollständigkeiten durch Interaktion oder durch Standard-Behandlung.

- Report-Generatoren
  Darunter fallen Werkzeuge, die den Inhalt der Datenbank nach gewissen Aspekten auswerten. Beispielsweise unterstützen Dokumenten-Generatoren bei der Erstellung von Handbüchern. Prüfungen auf Unvollständigkeiten oder auf Inkonsistenzen werden von Prüfwerkzeugen unterstützt.

- Compiler zur Übersetzung des Zielsystems

Die Abb. 3 zeigt die Struktur der Werkzeugumgebung.

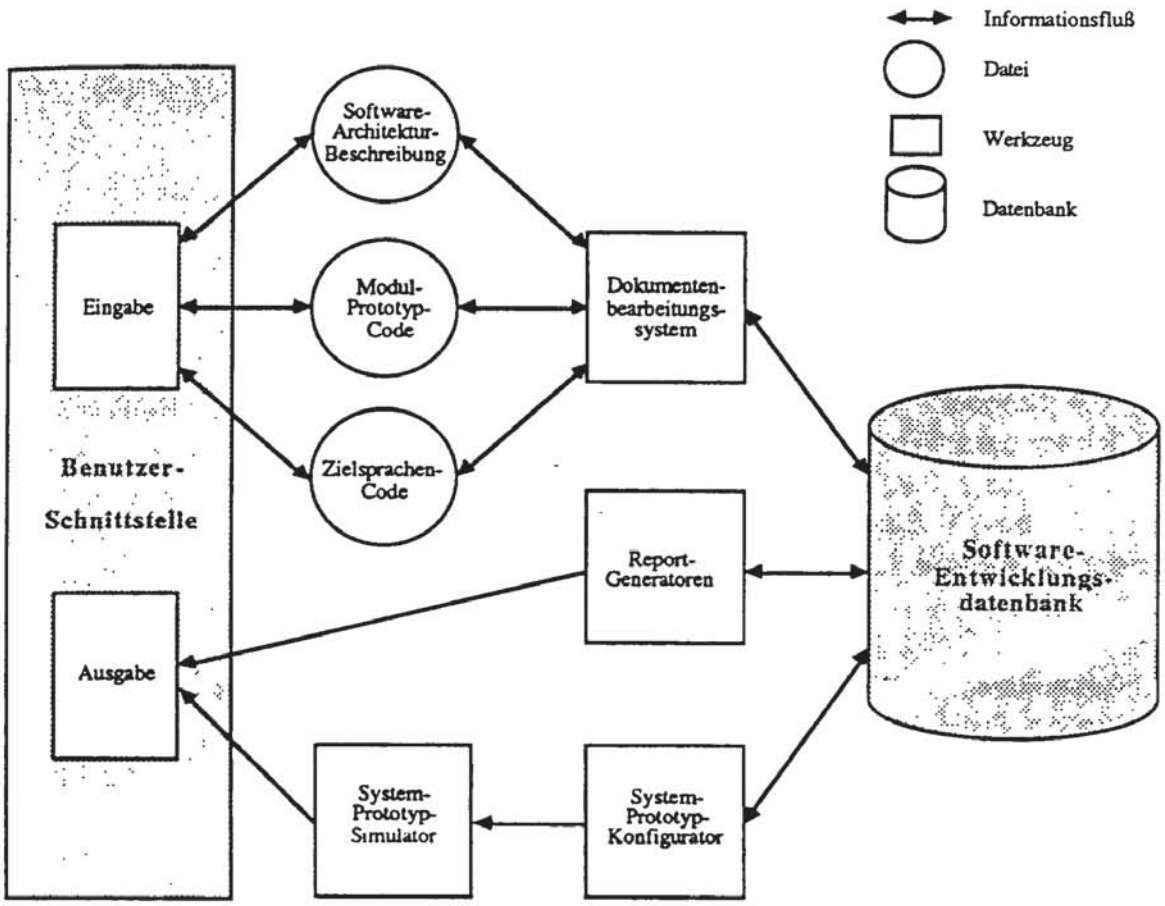

Abb. 3: Struktur der Werkzeugumgebung für PDSC

Stand der Arbeit, Planung (Februar 1987)

Die Arbeiten an einem System für "Program Development by stepwise completion" (PDSC) wurden im Herbst 1986 begonnen.

Die Architektur-Sprache wird gegenwärtig definiert; ihre Konzepte werden im wesentlichen von SPADES-L (Ludewig et al., 1985) übernommen.
Als Prototyp-Sprache wollen wir SMALLTALK-80 verwenden; das System ist auf unseren Workstations (SUN-3/50 mit UNIX) installiert.
Als Ziel-Sprache kommen nur MODULA-2 und Ada in Frage, die uns ebenfalls zur Verfügung stehen.

Für die Realisierung der Software-Entwicklungsdatenbank steht ein Basissystem (ERDAS, siehe Eigentumsvermerk unten) zur Verfügung (Glinz, Huser, Ludewig, 1985; Glinz, Ludewig, 1986; Huser, 1987), dessen Konzeption ausgezeichnet für die Ziele von PDSC geeignet ist. Dazu gehört auch ein Parser-Generator.
Im Verlauf des Jahres 1987 soll die Architektur-Sprache in ein Schema transformiert und damit einsetzbar werden. Im Sinne des Prototyping-Ansatzes versuchen wir selbst, möglichst rasch ein laufendes System zu erreichen, so daß wir - durch Bootstrapping - die PDSC-Werkzeuge mit diesen selbst bearbeiten können.

Literaturangaben

Beregi, W.E. (1984): Architecture prototyping in the software engineering environment. **IBM Systems Journal, 23,** 1, 4-18.

Boehm, B.W. (1976): Software Engineering. **IEEE Transactions on Computers, C-25**, 1226-1241.

Boehm, B.W. (1983): Seven basic principles of Software Engineering. **Journal of Systems and Software, 3,** 3-24.

Boehm, B.W., T.A. Standish (1983): Software technology in the 1990's: Using an evolutionary paradigm. **IEEE COMPUTER**, Nov. 1983, 30-35.

Boehm, B.W., T.E. Gray, T. Seewald (1984): Prototyping vs. specifying: a multiproject experiment. **IEEE Trans. on Software Engineering, SE-10**, 290-303.

Budde, R. K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (1984): **Approaches to Prototyping.** Springer-Verlag, Berlin usw., 1984.

Budde, R. (1986): **Very High Level Languages (VHLL) und Prototyping.** Manuskript, GMD F2G2, 20 S., unveröffentlicht.

Glinz, M., H.J. Huser, J. Ludewig (1985): SEED - A database system for software engineering environments. in Blaser, Pistor (Hrsg.): **Datenbanksysteme für Büro, Technik und Wissenschaft,** Informatik-FB 94, Springer, S.121-126.

Glinz, M., J. Ludewig (1986): SEED - a DBMS for Software Engineering applications based on the Entity Relationship approach. to appear in G. Wiederhold: **The Second Intern. Conf. on Data Engineering.** Los Angeles, CA, February 5-7, 1986.

Huser, H. (1987): Ein Datenbank-System für Software-Werkzeuge. **CRB-Bericht**, Brown Boveri Forschungszentrum, Baden/Schweiz, März 1987.

Ludewig, J. (1982): Computer aided specification of process control software. **IEEE COMPUTER, 15**, 5, 12-20.

Ludewig, J., M. Glinz, H. Matheis (1985): Software-Spezifikation durch halb-formale, anschauliche Modelle. in H.R. Hansen: **GI/OCG/ÖGI-Jahres-tagung 1985**, Informatik-FB 108, Springer, Berlin usw., S.193-204.

Ludewig, J., M. Glinz, H.J. Huser, G. Matheis, H. Matheis, M.F. Schmidt (1985): SPADES - A Specification and Design System and its Graphical Interface. **8th Intern. Conf. on Software Engineering**, IEEE CH2139-4/85/0000/0083, 83-89.

Swartout, W., R. Balzer (1982): On the inevitable intertwining of specification and implementation. **Commun. ACM, 25**, 7, 438-440.