8. Realisierung

Das Thema Realisierung wird in diesem Skript derzeit nur summarisch behandelt, da es nach dem bisherigen Curriculum an der Universität Zürich nicht in der Grundvorlesung, sondern in der Kernvorlesung Software Engineering behandelt wurde. Eine vollständige Überarbeitung des Kapitels ist für 2006 geplant. Inspektions- und Testverfahren werden im Kapitel 9 behandelt.

8.1 Möglichkeiten der Realisierung

Ein Lösungskonzept (d.h. eine Systemarchitektur) lässt sich auf verschiedene Arten realisieren:
- durch Entwerfen und Codieren
- durch Entwerfen und Generieren
- durch Konfigurieren vorhandener Komponenten.

In allen Fällen müssen die realisierten Einzelkomponenten zu einem System integriert werden, und müssen die Komponenten wie auch das System geprüft werden, um mögliche Fehler zu finden und sie anschließend zu beheben. Sowohl beim Entwerfen wie beim Codieren werden wo möglich vorhandene Komponenten und vorhandene Ideen (z. B. Entwurfsmuster) genutzt.

Im klassischen Detailentwurf werden die zu codierenden Algorithmen und Datenstrukturen festgelegt. Dort, wo das Lösungskonzept noch zuwenig detailliert ist, werden die nötigen Details erarbeitet. In der Codierung wird dieser Entwurf in einer Programmiersprache ausformuliert.

Dort, wo der Code generiert werden kann bzw. soll, entsteht ein formelles Entwurfsdokument, dessen Syntax den Vorschriften des Generators genügen muss. Typische Kandidaten für die Generierung sind Datenbanken und einfache Datenbankanwendungen einschließlich der zugehörigen Bildschirmmasken.

Aus hinreichend präzisen teilformalen Entwurfsmodellen können mit geeigneten Werkzeugen Code*rahmen* generiert werden, die dann von Hand auszuprogrammieren sind.

8.2 Systematisches Programmieren

Die Qualität der resultierenden Software wie auch der für die Prüfung erforderliche Aufwand lassen sich durch systematisches Programmieren erheblich beeinflussen. Besonders wichtige Punkte sind:
- Geeignete Wahl von Algorithmen und Datenstrukturen und deren adäquate Programmierung
- Verwendung geschlossener Ablaufkonstrukte (solche mit einem Eingang und einem Ausgang)
- Saubere Gliederung der Software in Prozeduren und Module
- Verwendung symbolischer Konstanten
- Verwendung selbstdokumentierender Namen für Prozeduren, Variablen, Konstanten und Typen
- Dokumentation aller Definitionen durch Kommentare
  - Bei Prozeduren: Voraussetzungen, die beim Aufruf erfüllt sein müssen, Ergebniszusicherungen (Aussagen, die nach der Ausführung der Prozedur gelten), Bedeutung und Kommunikationsrichtung der Parameter

  - Bei Variablen und Konstanten: Bedeutung der Variable bzw. Konstante im Kontext des Programms
- Defensives Programmieren
- Vermeiden von Trickprogrammierung und Programmierung mit Nebenwirkungen.

Häufig werden Vorschriften zur Programmiersystematik und zum Programmierstil in Entwicklungsrichtlinien festgehalten.

Beispiel 8.1 zeigt ein Beispiel für ein nach den obenstehenden Kriterien erstelltes Programm. Beispiel 8.2 demonstriert, wie man es nicht machen sollte. Das Java-Programm in Beispiel 1 wurde nach einer Entwicklungsrichtlinie für Java (Berner et al. 1996) dokumentiert.

Beispiel 8.1: Systematisches Programmieren (in Java)

Dargestellt ist ein Auszug aus einer Klasse zur Verwaltung von Messreihen. Da die Messreihen unterschiedlich lang sein können, sind sie intern als verkettete Listen von Datenobjekten realisiert. Nach außen ist jedoch für jede Messreihe nur ein Objekt sichtbar, das mit den von der Klasse bereitgestellten Operationen manipuliert wird.

```
/* ---------------------------------------------------------------------
AUTOR:        Martin Arnold
PROJEKT:      Beispiel 8.1, Vorlesung Software Engineering I
JAVA:         Symantec Café, Projectmanager 8.2b3
COPYRIGHT:    Forschungsgruppe Requirements Engineering,
              Institut für Informatik, Universität Zürich
KLASSE:       MeasurementList
VERSION:      1.01 von M. Glinz am 18.12.1996
---------------------------------------------------------------------------

ZWECK
Die Objekte der Klasse MeasurementList modellieren Messreihen.

VERANTWORTLICHKEITEN
Speicherung, Skalierung, Filterung und Abfrage von Messreihen.
*/
import java.awt.List;

public class MeasurementList extends List      //  MeasurementList wird Unterklasse von List
     {

/* --- KLASSEN-Variablen ---*/
     final private static double TOLERANZ = 0.30; // 30% zugelassene TOLERANZ nach unten

/* --- INSTANZ-Variablen ---*/
     double              wert;          // Listeneintrag: Messwert als reelle Zahl
     MeasurementList  naechstes;    // "Zeiger" auf nächstes Element der verketteten Liste

/*   Hinweis zur Dokumentation: Ein Objekt, auf das eine Methode (Operation) angewendet
     wird, wird aktuelles Objekt genannt.*/
/*   Hinweis zur Implementierung: Jede (auch die leere) Liste wird durch ein Objekt
     abgeschlossen, dessen Wert undefiniert ist und dessen Zeiger (naechstes) auf kein
     weiteres Objekt, sondern auf "null" zeigt.*/

void add(double element)
     /*   Erste Version von Martin Arnold, 06.12.1996
          Letzte Aenderung von Martin Glinz, 18.12.1996

          PRE       –
```

```
        POST    An die aktuelle Liste (d.h. diejenige, die mit dem aktuellen Objekt beginnt)
                wird ein Element mit der Zahl 'element' als Wert angehängt. Das aktuelle
                Objekt bleibt erstes Element der Liste.
    */
    {
    MeasurementList temp = this;    // Referenz auf aktuellen Listenbeginn (erstes Element)

    while ( temp.naechstes != null )  // Suche nach letztem Element
        { temp = temp.naechstes; }

    temp.naechstes = new MeasurementList();      // Zuweisung neuer Listenzeiger
    temp.wert = element;   // Zuweisung des einzufügenden Wertes
    }

/*
    Weitere Methoden; in diesem Beispiel weggelassen ...
*/

MeasurementList ausreisserEntfernen(double referenzwert)
    /*  Erste Version in Modula-2 von Martin Glinz, 1993
        Reimplementiert in JAVA von Martin Arnold, 6.12.1996
        Letzte Version von Martin Glinz, 18.12.1996

    PRE     Liste ist ordnungsgemäss verkettet. Beim letzten Element ist der Zeiger
            naechstes = null.

    POST    Die aktuelle Liste ist wie folgt modifiziert: Alle Listenelemente, deren Wert um
            mehr als TOLERANZ (in Prozent) kleiner als der Referenzwert ist, sind aus der
            Liste entfernt. Als Funktionswert wird das erste Listenelement (Listenanker) der
            bereinigten Liste zurückgegeben.
    */
    {
    MeasurementList temp = this; // Referenz auf aktuellen Listenbeginn (erstes Element)
    MeasurementList vorgaenger = new MeasurementList(); //  Initialisierung eines
                                                            Vorgaengerelements
    MeasurementList listenanker = temp;// Listenanker = erstes Listenelement

    /*  Die Liste wird hier vollständig durchlaufen und sämtliche
         Elemente unterhalb der TOLERANZgrenze werden entfernt.*/
    while ( temp.naechstes != null ) // Liste durchlaufen bis zum letzten Eintrag
        {
        if (temp.wert < (referenzwert * (1.0 - TOLERANZ)))
            { /*  Element unterhalb der TOLERANZgrenze */
            if (vorgaenger.naechstes != null)
                { /* aktuelles Element ist nicht erstes Listenelement (Listenanker) */
                vorgaenger.naechstes = temp.naechstes; // Element ausketten
                temp = vorgaenger.naechstes;
                }
            else
                { /* aktuelles Element ist erstes Listenelement (Listenanker) */
                listenanker = temp.naechstes; // Listenanker neu setzen
                temp = listenanker;
                }
            }
        else
            { /* Element im TOLERANZbereich */
            vorgaenger = temp;
            temp = temp.naechstes;
```

```
                }
            }
        return listenanker; // Rückgabe: Erstes Listenelement der bereinigten Liste
        }
}
```

Beispiel 8.2: Trick-Programmierung mit Nebenwirkungen (in C)

Die nachfolgende Prozedur hängt die Zeichenkette s an die Zeichenkette t an und speichert das Ergebnis in t. Man beachte, dass die Rümpfe der beiden verwendeten Schleifen leer sind, d.h. das Programm hat scheinbar keine Wirkungen. Alle Effekte sind Nebenwirkungen der Schleifensteuerungen.

```
void strcat (char *s, char *t)
{
WHILE (*t++); FOR (t--;*t++=*s++;); /* t <- t concat s */
}
```

8.3 Prüfung und Integration

Software wird typischerweise komponentenweise realisiert. Jede Komponente wird nach ihrer Fertigstellung zunächst für sich allein *geprüft*. Die wichtigsten Prüfverfahren sind:

- Prüfung auf syntaktische Richtigkeit durch einen Compiler
- Prüfung auf inhaltliche Richtigkeit durch Inspektion
  - Selbstinspektion durch die Autorin oder den Autor
  - Formale Inspektion (vgl. Kapitel über Qualitätsmanagement) durch Dritte
- Komponententest (vgl. Kapitel über Qualitätsmanagement).

Die fertiggestellten und geprüften Komponenten müssen anschließend zu einem System *integriert* werden. Die Integration erfolgt schrittweise. Jeder Integrationsschritt ist mit einem zugehörigen *Integrationstest* verbunden, der die Richtigkeit des Integrationsschritts sichert.

Nach Abschluss der Integration wird das fertiggestellte System einem gründlichen *Systemtest* unterworfen. Im Systemtest sollen möglichst viele derjenigen Fehler gefunden werden, die bei den Komponenten- und Integrationstests übersehen wurden, weil sie nur beim Zusammenspiel zwischen verschiedenen Komponenten auftreten.

Je nach Vertragsverhältnis zwischen Software-Ersteller und Auftraggeber erfolgt anschließend noch eine formale *Abnahme*. Dabei werden vorher vereinbarte Testfälle durchgespielt, mit denen dargelegt wird, dass die fertiggestellte Software die in der Anforderungsspezifikation gestellten Anforderungen erfüllt.

Zitierte und weiterführende Literatur

Berner, S., M. Arnold, S. Joos, M. Glinz (1996). *Java-Entwicklungsrichtlinien*. Institut für Informatik der Universität Zürich, Forschungsgruppe Requirements Engineering.

Berner, S., S. Joos, M. Glinz (1997). Entwicklungsrichtlinien für die Programmiersprache Java. *Informatik/Informatique* **4**, 3 (Jun 1997). 8-11.

Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Prüfung. Eine Fibel.* Zürich: vdf und Stuttgart: Teubner.

Wirth, N. (1993). *Systematisches Programmieren: eine Einführung*. 6. Auflage. Stuttgart: Teubner.