Implementierung

Allmählich schließt sich der Kreis dieser Veranstaltung: wir kommen ans eigentliche Codieren, das Erstellen von Programmcode, der die abstrakte Maschine in der gewünschten Weise steuern soll. Damit kehren wir jetzt – nach dem Kennenlernen der vorhergehenden Phasen des Software-Lebenszyklus – zur elementaren Softwaretechnik zurück. Wir vertiefen bestimmte Aspekte in dieser Vorlesung.

Entwicklungsumgebung

In der arbeitsteiligen Entwicklung von Software spielt die Unterstützung durch die Entwicklungsumgebung eine wichtige Rolle.

In der Regel wird vor Beginn der Entwicklung entschieden, welche *Programmiersprache* eingesetzt wird, mit welchen *Klassenbibliotheken* und welchem *Framework* entwickelt wird. Ferner werden die *Werkzeuge* für die Entwicklung ausgewählt und in Betrieb genommen.

Gelegentlich gibt es „Glaubenskriege“ in Bezug auf Entwicklungsumgebungen, bei denen gerne die Wahl der Entwicklungsumgebung als wichtigster Erfolgsfaktor bei der Implementierung gehalten wird. Ähnlich wie bei den Vorgehensmodellen wird angenommen, die Entwicklungsumgebung sorge gewissermaßen automatisch für Qualität. Vorsicht: eine Entwicklungsumgebung *unterstützt* die Implementierung, die sorgfältige und bewusste Vorgehensweise des Entwicklers kann sie nicht erzeugen. Lassen wir Zuser zu Wort kommen:

> Die erfolgreiche Implementierung des Systems ist im Allgemeinen nicht ausschließlich von der Wahl der Programmiersprache, der Klassenbibliothek und der Werkzeuge abhängig. Die Implementierung wird auch stark beeinflusst von:
>
> - der Qualität des zugrunde liegenden Entwurfs
> - der zugrunde liegenden Infrastruktur
> - der konsequenten Einhaltung allgemeiner Prinzipien (z.B. objetorientierte Prinzipien wie Information Hiding, Vererbung, Polymorphismus usw.) und
> - dem generellen Arbeitsfeld und der individuellen Arbeitsweise der Programmierer (z.B. Code-Gestaltung, Kommentierung des Codes, systematisches inkrementelles Vorgehen).

> Schlechte Programmierer können auch mit einer noch so modernen Programmiersprache bzw. einer hoch automatisierten Entwicklunsumgebung kein qualitativ gutes Programm erzeugen.
>
> – Zuser et al S. 319

Programmiersprache

Oft hängt die Wahl der Programmiersprache für eine Entwicklung am jeweiligen Gebiet: im Bereich eingebetteter Software wird in der Regel C eingesetzt, im Bereich technischer Software vielleicht Fortran, für Geschäftsanwendungen Sprachen, die den Datenbankzugriff besonders integriert haben wie Delphi, PowerBuilder oder auch ABAP für SAP/R3, für webbasierte Anwendungen Java, C# oder auch Skriptsprachen wie PHP oder auch Perl.

C++, Java und C# kann man heute sicherlich als *die* „Allround"-Sprachen bezeichnen.

Einen Überblick mit Erläuterungen zu den wichtigsten Programmiersprachen finden Sie im Buch von Zuser et al. Abschnitt 11.2

Wenn man die Vielzahl der eingesetzten Programmiersprachen sieht, kann man Schlüsse ziehen:

- Ein Softwareentwickler muss in der Lage sein, verschiedene Sprachen einzusetzen, er muss auch schnell eine neue Sprache erlernen können. Die Tatsache, dass viele Prinzipien und Konstrukte in den Programmiersprachen gleich sind, unterstützt ihn darin. Es ist aber genauso eine Tatsache, dass es subtile Unterschiede zwischen Konzepten in den verschiedenen Programmiersprachen gibt, die der Entwickler genau beachten muss.
- Deshalb ist man gut beraten, die Programmiersprache möglichst geradlinig, klar und einfach zu verwenden (keine speziellen Tricks, die nur der Guru kennt!) – denn dadurch vereinfacht man nicht nur die erste Entwicklung, sondern die Wartung, Überarbeitung und Weiterentwicklung des Codes.

Deshalb sei noch ein Wort der Warnung gestattet: man beobachtet nicht selten Entwickler, die sich neue Programmiersprachen durch „Ausprobieren" aneignen. Nun gehört sicherlich zum Erlernen einer Sprache das Programmieren (von Beispielen) unbedingt dazu, aber ein Blick in ein Buch, das die Konzepte erklärt, erspart manches Probieren und hat

einen nachhaltigeren Effekt: Denn das *erkannte* Konzept kann man ohne Ausprobieren auch in analogen Situationen anwenden.

Frameworks & Klassenbibliotheken

Klassenbibliotheken, zusammengefasst in Frameworks, umfassen vorgefertigte Klassen und Komponenten für Aufgaben, die in verschiedenen Projekten immer wieder auftreten, etwa

- Klassen zur Gestaltung und Steuerung einer graphischen Benutzeroberfläche
- Klassen für den Datenbankzugriff
- Klassen für die Kommunikation zwischen Prozessen und Rechnern
- Klassen für gebräuchliche Datenformate wie XML

Für die Entwicklung von Anwendungen in Java oder C# stehen reichhaltige Frameworks zur Verfügung. In speziellen Bereichen, wie etwa der Entwicklung von eingebetteter Software gibt es Ähnliches, häufig jedoch von einem Hersteller selbst entwickelt und nur intern verwendet.

Die Entscheidung für ein solches Framework wird beeinflusst durch die Zielplattform auf der das zu entwickelnde System laufen soll. So wird etwa die .NET-Plattform vorwiegend in der „Windows-Welt“ eingesetzt.

Werkzeuge

Wichtige Werkzeuge für die Entwicklung sind

Codeditoren und Klassenbrowser, die bequemes Editieren der Programmquellen gestatten und Überblick über und leichtes Navigieren in beteiligten Klassen, Attribute und Methoden erlauben. Syntaxsensitive Editoren helfen dabei einfache Fehler zu vermeiden. (Aber: sie verleiten auch zum „Ausprobieren“ an Stelle von Wissen und Nachdenken mit möglicherweise unheilvollen Auswirkungen)

Make- und Build-Werkzeuge, die auf Basis einer Beschreibung der Abhängigkeiten der Quellen und (Teil-)produkte einer Software das Übersetzen, Linken, Konfigurieren, Deployen steuern. Vertreter sind make, ant, nant.

Testwerkzeuge, die etwa für den wiederholbaren, halbautomatischen Test von Programmteilen eingesetzt werden. Oder Testdatengeneratoren, die Daten für Belastungstests erzeugen; auch Ablaufrecorder, die einen Ablauf in einer interaktiven Benutzeroberfläche aufzeichnen und dann automatisch „nachspielen“ können. Besonders bekannt sind Werkzeuge für den Unit-Test wie JUnit.

Werkzeuge zur Codeanalyse, die Fehler oder problematische Stellen im Code finden können. Solche Werkzeuge werden auch eingesetzt, um die Einhaltung von Codierrichtlinien zu überwachen. Die „Mutter“ solcher Werkzeuge ist lint für die statische Analyse von C/C++-Code.

Debugger, mit deren Hilfe man ein Programm schrittweise durchlaufen kann und an jeder Stelle etwa den aktuellen Wert von Variablen inspizieren kann. Debugger können insbesondere von Nutzen eingesetzt werden, wenn zur Laufzeit Effekte auftreten, die man sich aus der Analyse des Codes nicht erklären kann. (Eine kritische Meinung zu Debuggern: „I've always believed that if you have to fire up a debugger to fix a problem, then you have a much more basic problem with your code that a debugger can't fix.“ (Karl Vogel DDJ 8/91) – will sagen: konzeptionelle Probleme im Code kann man mit einem Debugger nicht finden!)

Versionsverwaltung, mit deren Hilfe die Quellen versioniert werden und ihre Zugehörigkeit zu Projekten verwaltet werden kann. Beispiele sind MKS Source Integrity, Perforce, Microsoft Visual Source Safe, Rational ClearCase, im Open Source Bereich CVS und Subversion. Diskutiert am Anfang der Veranstaltung!

Änderungsverwaltung betrifft Werkzeuge, die für die Steuerung des Ablaufs bei Änderungen (Fehlerbehebung oder Weiterentwicklung) von Software eingesetzt werden. Sie unterstützen in der Regel einen Ablauf vom Anstoßen der Änderung und der Analyse des Vorgehens über die Entscheidung bis zu Durchführung und Test der Änderung. Da diese Fragen alle eng mit der Versionsverwaltung zusammenhängen, unterstützen die oben genannten Produkte in der Regel auch die Änderungsverwaltung (Change Request Management).

Dokumentationserstellung kann man weitgehend automatisieren, in dem die Spezifikation der Klassen und Methoden in der Quelle geschieht und dann durch Werkzeuge eine Systemdokumentation erstellt wird. Solche Werkzeuge sind Javadoc, Doxygen oder auch der C#-Compiler selbst. (Dies eignet sich natürlich nicht für die Benutzerdokumentation.)

Diese Werkzeuge sind heute in der Regel in einer *integrierten Entwicklungsumgebung* zusammengefasst, die durch weitere Werkzeuge ergänzt und erweitert werden kann (Plug-ins). Die wichtigsten dieser Entwicklungsumgebungen sind Visual Studio .NET für die Entwicklung auf der .NET-Plattform und Eclipse im Java-Umfeld.

Softwarehäuser verwenden oft speziell angepasste Varianten dieser oder anderer Entwicklungsumgebungen, die des jeweilige Vorgehensmodell unterstützen. Im Bereich eingebetteter Software werden häufig Entwicklungsumgebungen eingesetzt, die auch die Simulation und das Testen der entwickelten Systeme erlauben.

Aspekte der Implementierung

In diesem Abschnitt möchte ich auf einige besonders wichtige Aspekte hinweisen. Da wir in der Veranstaltung mit diesem Thema begonnen haben, genügen oft Hinweise auf Bekanntes.

Codierrichtlinie

Codierrichtlinien sorgen nicht nur für Lesbarkeit des Codes, sondern auch für die Entwicklung eines gemeinsamen Stils (man spricht auch von Idiomen) in einer Entwicklergruppe und helfen damit Fehler zu vermeiden, Test und Wartung zu erleichtern.

Codierrichtlinien sollten nicht als formales Gerüst gesehen werden, sondern als lebendiges Instrument zur Entwicklung. Deshalb sollten sie auch Fragen umfassen wie z.B. die einheitliche Fehlerbehandlung in einem Programm u.ä.

Was Codierrichtlinien alles beinhalten können, haben wir bereits diskutiert, siehe Elementare Softwaretechnik.

Versionsverwaltung

Auch hier genügt die Erinnerung an den Beginn der Veranstaltung. Ergänzend sei vermerkt, dass auch ein definierter Ablauf der Erstellung, des Testens, der Abnahme und Auslieferung und der Änderung bei der Wartung durch Versionsverwaltungssystem unterstützt werden. So kann durch eine geeignete Konfiguration des Versionsverwaltungssystems sichergestellt werden, dass eine Quelle nur dann in die neue Version des Produkts eingehen kann, wenn sie bestimmte Tests erfolgreich durchlaufen hat.

Solche Abläufe werden manchmal von Entwicklern als umständlich und „bürokratisch“ empfunden – was im Einzelfall sogar zutreffen mag. Aber andererseits sichert die Steuerung des Ablaufs, dass Werkzeuge auf definierte Weise eingesetzt werden. Dies hilft insbesondere dann, wenn neue Entwickler ins Team integriert werden müssen. Denn der Aufwand eine geeignete Entwicklungsumgebung aufzubauen (und zu verstehen) ist nicht unerheblich.

Integrationsstrategien

Bisher haben wir insbesondere betrachtet, wie man unterstützt durch Werkzeuge einzelne Klassen, Methoden oder Komponenten entwickelt. Als Ergebnis der arbeitsteiligen Entwicklung müssen diese Teile natürlich zu einem Gesamtsystem zusammengeführt werden – dies nennt man *Integration*.

Für die Integration gibt es folgende Strategien (siehe Zuser S. 328ff):

Big-Bang-Integration: Zuerst werden alle Teile entwickelt und dann „auf einen Schlag“ zusammengebaut. Es braucht wenig Phantasie, sich auszurechnen, dass dies in der Regel nicht klappen dürfte!

Top-down-Integration: Man baut zunächst einen Rahmen, in dem das komplette System laufen kann, wobei die systemnahen Bestandteile durch sogenannte „stubs“ simuliert werden. Der Vorteil besteht darin, dass schon sehr früh in der Integration die Anwenderschnittstelle gezeigt und überprüft werden kann. Nachteil ist, dass ein großer Aufwand für die Integration betrieben werden muss.

Bottom-up-Integration: Hierbei wird von den systemnahen Funktionen das Programm sukzessive bis hin zu der Benutzerschnittstelle aufgebaut. Bei dieser Strategie kann es sein, dass man erst sehr spät merkt, dass man „das Falsche“ (für die Anwender) gebaut hat.

Build-Integration: Bei diesem Vorgehen wird von den Anwendungsfällen ausgegangen und ein komplettes System für *einen* Anwendungsfall (mit systemnahen Komponenten und allen benötigten Teilen der Anwendungslogik und Benutzerschnittstelle) konstruiert. Das geht natürlich nur, wenn bereits alle die systemnahen Teile entwickelt wurden, damit man diesen einen Anwendungsfall durchführen kann – bei Entwicklungen, bei denen viel Infrastruktur entwickelt werden muss, kann es demzufolge lange dauern, bis der erste Anwendungsfall tatsächlich integriert werden kann.

Daily Build: Bei dieser Strategie wird täglich ein lauffähiges System (mittels der Build-Werkzeuge) erstellt und dann getestet. Dies ist natürlich nur möglich, wenn von Anfang an eine entsprechende Umgebung vorhanden ist, auf der das zuerst ja nur rudimentäre System ausgeführt und getestet werden kann. Der große Vorteil dieses Vorgehens besteht darin, dass Probleme sehr schnell entdeckt werden können und dadurch der Augenmerk im Projekt auf die wichtigsten Punkte gelenkt werden kann.

Überprüfen und Testen

Implementierung schließt das ständige Überprüfen der gerade entwickelten Quellen ein. Das macht jeder Entwickler gewissermaßen ohnehin, durch Techniken wie den Daily Build wird dieses Überprüfen auch zeitnah im laufenden System gemacht. Auch systematisches Testen der einzelnen Klassen und/oder des gesamten Systems gehören zu diesem Überprüfen (was wir in der nächsten Vorlesung noch genauer behandeln werden).

Es ist eine gute Idee, das Überprüfen und Verbessern von Code regelrecht in die normale Vorgehensweise einzubauen. Frank Buschmann hat in einem Vortrag berichtet, dass in Projekten an denen er beteiligt war, die Devise lautete: „3 1/2 Tage schnell vorwärts entwickeln, und dann 1 1/2 Tage überprüfen und verbessern“.

Dafür wird gerne die Technik des „Refactoring“ eingesetzt. Dabei handelt es sich um:

> Refactoring is the process of changing a software system in such a way that it does *not alter the external behavior* of the code yet *improves its internal structure*. It is a disciplined way to clean up code that minimizes the chance of introducting bugs. In essence when you refactor you are improving the design of the code after is has been written.
>
> – Martin Fowler

Refactoring wird durch kleine, elementare Transformationen, sogenannte Refactorings, schrittweise durchgeführt. Für jedes dieser Refactorings kann man leicht überprüfen, dass die Veränderung zwar die Struktur des Codes verbessert, nicht aber die Funktion verändert.

Freilich: oft stellt sich beim Testen und Überprüfen heraus, dass grundlegendere Probleme vorliegen. Es kann sein, dass die Anforderungen nicht wirklich verstanden wurden; es kommt vor, dass neue Ideen zur technische Durchführung aufkommen (auf Grund der Erfahrung in der

Entwicklung); man entdeckt Schwächen in der Architektur des Systems. Wie immer also: Refactoring ist ein sehr gutes Mittel zur Verbesserung des Codes „im Kleinen“, aber kein Allheilmittel gegen Schwächen in Analyse, Architektur und Entwurf.

Die Frage ist natürlich insbesondere, woran man erkennt, dass und wo im Code problematische Stellen sind. Damit befassen wir uns im folgenden Abschnitt.

Beurteilung von Code

Folgende Leitlinie für guten objektorientierte Code ist aus dem Buch von Zuser et al. Die Hinweise auf verdächtige Stellen im Code fassen Gedanken von Fowler zum Thema „übel riechenden Code“ zusammen.

Guter OO-Code

- Syntax und Struktur
  - Strukturieren Sie Code so, dass er immer leicht lesbar bleibt.
  - Verwenden Sie Einrückungen (Tabulatoren und Leerzeichen).
  - Beschränken Sie die maximale Zeilenlänge auf 80 Zeichen.
  - Geben Sie nur eine Anweisung pro Zeile.
  - Vermeiden Sie unverständliche Abkürzungen.
  - Legen Sie sich ein konsistentes Benennungssystem zurecht und halten Sie es ein. Vereinbaren Sie es im Team!
- Kommentare
  - Sparen Sie nicht mit Kommentaren, insbesondere bei nicht offensichtlichen Annahmen.
  - Beschreiben Sie vor jeder Methode ihre Funktionsweise. Verwenden Sie z.B. die Technik zum Spezifizieren aus dieser Veranstaltung!
  - Sparen Sie nicht an begleitender Dokumentation (z.B. kann eine Readme-Datei mit einer ausführlichen technischen Beschreibung sehr hilfreich sein).
- System
  - Vermeiden Sie den Gebrauch globaler Variablen. Eine Methode verwendet als Daten im Normalfall nur Parameter und lokale, initialisierte Variablen.

  - Implementieren Sie eine effektive Fehlerbehandlung (Exceptions sind in objektorientierten Sprachen ein hervorragendes Instrument für diesen Zweck).

- Klassenebene
  - Verwenden Sie Vererbung nur, wenn Klassen in inhaltlichem Zusammenhang stehen (setzen Sie keinesfalls Vererbung ein, um sich das Tippen einiger Variablen zu sparen). Denken Sie an das Liskovsche Substitutionsprinzip!
  - Fassen Sie in Subsystemen nur zusammen, was auch zusammengehört.
  - Vermeiden Sie zu allgemeine Klassen. Jede Klasse soll eine genau spezifizierte Funktionalität besitzen, für die sie Spezialist ist. Denken Sie an die Grasp-Entwurfsprinizipien!

- Methodenebene
  - Benutzen Sie aussagekräftige Variablen- und Methodenbezeichner. Verwenden Sie zur besseren Lesbarkeit Groß- und Kleinschreibung.
  - Halten Sie private und öffentliche Methoden strikt auseinander.
  - Jede Methode muss testbar sein, d.h. es soll zu entscheiden sein, ob sie korrekt implementiert wurde.
  - Jede Methode muss leicht verständlich sein. Halten Sie dazu Methoden so kurz wie möglich. Teilen Sie Funktionalität – wenn nötig – auf mehrere Methoden auf.
  - Informieren Sie sich in einschlägigen Büchern über geeignete Standardalgorithmen.

- Implementieren Sie Design by Contract[1]
  - Initialisieren Sie alle Variablen. Alle Annahmen über den Zustand von Variablen sollten durch Abfragen im Code überprüft werden.
  - Jede Methode überprüft Eingabeparameter auf Gültigkeit und Sonderfälle. Die Methode reagiert klar und sinnvoll auf Sonderfälle.
  - Überprüfen Sie immer die Rückgabewerte von Funktionen auf mögliche Fehler (z. B. kein Rückgabewert, Rückgabeweit liegt außerhalb des Wertebereichs usw.).

[1] So steht das im Buch von Zuser, ich glaube aber dass Bertrand Meyer nicht damit einverstanden wäre, das dann Folgende so zu bezeichnen. Die Bezeichnung „Defensives Programmieren“ trifft es besser. (Siehe auch Elementare Softwaretechnik)

Verdächtiges

Folgende Übersicht gibt Indizien für Fehler im Code – es sind auch genau die Punkte, die in einem Codereview besonders beachtet werden sollten.

In konkreten Umgebungen stellt man häufig fest, dass immer wieder ähnliche Fehler auftreten – diese kann man dann gezielt suchen und auch in die Liste der „verdächtigen“ Stellen aufnehmen.

- Duplizierter Code, duplizierte Logik
  - Gleiche Codestrukturen an mehreren Stellen
  - Identische Berechnungen in mehreren Methoden
- Lange Methoden
  - Methode passt nicht auf 1, 2 Bildschirmseiten
  - Kontrollfluss verwickelt
  - Hohe Schachtelungstiefe
- Große Klassen
  - Viele Variablen, Variablen, die als Flags verwendet werden
  - Viele Methoden (Kohäsion?)
- Lange Parameterlisten
  - Benutzergerecht entworfen?
  - Oft geändert? Drangestrickt?
- Unpassende, unverständliche Namen
  - Experimentell entstandener Code?
  - Kopierter Code?
  - Gut durchdacht?
- Kommentare
  - Können manchmal unverständlichen Code signalisieren
  - Mit Code übereinstimmend?
  - Code nicht selbsterklärend, warum?

Lektüre zum Thema Entwicklungsumgebungen: Einen grundsätzlicheren Blick auf das Thema wirft Udo Kelter. Seine Darstellung http://pi.informatik.uni-siegen.de/kelter/lehre/04w/lm/lm_seu_info.html geht über die hier gegebene weit hinaus.

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 2.1 – 7. März 2011