Durchführung von Softwareprojekten

Überblick:

- Typische Probleme
- einige Qualitätsfaktoren
- Entwicklungsprozeß
- Modularität
- Design und Spezifikation
- Programmierkonventionen
- Qualitätssicherung und Testkonzepte
- Dokumentation

Typische Probleme bei der Entwicklung von Software

- ungenaue Zielsetzungen
- fehlendes Konzept oder Spezifikation
- Zeitmanagement
- Überwachung der Durchführung
- ungenügende Absprachen im Projektteam
- keine Qualitätssicherung
- mangelnde Dokumentation

einige Qualitätsfaktoren

- **Korrektheit**
  die Fähigkeit, die Aufgaben exakt zu erfüllen, wie sie durch Anforderungen und Spezifikation definiert sind

- **Robustheit**
  die Fähigkeit, auch unter außergewöhnlichen Bedingungen zu funktionieren

- **Erweiterbarkeit**
  die Leichtigkeit, Anpassungen an Spezifikationsänderungen durchzuführen

- **Wiederverwendbarkeit**
  die Eigenschaft, ganz oder teilweise für neue Anwendungen wiederverwendet werden zu können

Der Entwicklungsprozeß

- eingehende **Problemanalyse**
  Eingrenzung und Definition der Aufgabe
- **Design und Spezifikation**
- **Implementation**
- **Testen**

Modularität

- **Zerlegbarkeit**
  Reduktion der Komplexität durch Aufteilung in Teilprobleme

- **Kombinierbarkeit**
  Softwareentwurf als Baukastentätigkeit

- **Verständlichkeit**
  Modul für sich verständlich

- **Beständigkeit**
  kleine Spezifikationsänderung bewirkt nur kleine Änderungen in einzelnen Moduln (Bsp.: symbolische Konstanten; einheitlicher Zugriff)

- **Geschütztheit**
  Ausnahmesituation auf Modul beschränkt

Modularität - abgeleitete Prinzipien

- **Sprachliche Moduleinheiten**
  Moduln müssen zu syntaktischen Einheiten der benutzten Sprache passen.
- **Wenige Schnittstellen**
  Modul sollte mit möglichst wenig anderen kommunizieren.
- **Schmale Schnittstellen**
  Wenn zwei Moduln kommunizieren, sollten sie so wenig Information wie möglich austauschen.
- **Explizite Schnittstellen**
  Wenn zwei Moduln kommunizieren, muß das deutlich sichtbar geschehen.
- **Geheimnisprinzip**
  Jede Information über ein Modul sollte modulintern sein, wenn sie nicht ausdrücklich als öffentlich erklärt wird.

Spezifikation und Design

- Design Patterns
- logisches vs. physikalisches Design
  Klassen vs. Komponenten
- Designcheckliste
- Durchsicht einer Schnittstelle auf
  Konsistenz, Klarheit und Nutzbarkeit

physikalisches Design I

- Problematik:
  Kompilier - und Linkzeitabhängigkeiten
- Komponenten als kleinste, unteilbare physikalische Einheiten
- ein Header- und ein Implementationsfile pro Komponente
- Definition und Deklaration einer logischen Einheit in derselben Komponente
- Vermeide Definitionen mit externer Bindung im .cc-File einer Komponente, die nicht explizit im entsprechenden .h-File deklariert worden sind.
- Vermeide Zugriff auf eine Definition mit externer Bindung in einer anderen Komponente über eine lokale Deklaration

physikalisches Design II

- Vermeide “friendship” über verschiedene Komponenten
- Vermeide Kreise physikalischer Abhängigkeiten
- Verbesserungsstrategien: Escalation, Demotion, Opaque Pointers, Dumb Data, Redundancy, Callbacks, Manager Classes, Factoring, Escalating Encapsulation
- Kapselung vs. Isolierung
- Vermeidung von Präprozessormakros
  Ausnahmen: “include guards”; bedingte Compilierung für Debugging oder Portabilität

Verwendung von Checklisten am Beispiel der Schnittstellenspezifikation einer Funktion in C++

1. Operator or non-operator function?
2. Free or member operator?
3. Virtual or non-virtual function?
4. Pure or non-pure virtual member function?
5. Static or non-static member function?
6. const or non-const member function?
7. Public, protected or private member function?

Checkliste - 2. Teil

8. Return by value, reference, pointer, or argument?

9. Return const or non-const?

10. Argument is optional or required?

11. Pass argument by value, reference, or pointer?

12. Pass argument as const or non-const?

13. Friend or non-friend function?

14. Inline or non-inline function?

Programmierkonventionen

- Namenskonventionen
- sprechende Namen
- Einrückungen
- einheitliches Klassenlayout
- Hauptsache konsistent!

Programmierstil

- so klar und einfach wie möglich
- keine Tricks

Qualitätssicherung

- Benutzung von Zusicherungen
  - als Unterstützung beim Schreiben korrekter Software
  - als Dokumentationshilfe
  - als Debuggingwerkzeug
  - zur Unterstützung von Fehlertoleranz (exception handling)
- Invariantenprüfung ein- und ausschaltbar
- jeder Algorithmus sollte unabhängigen Checker enthalten
- Testkonzepte
  - extra Testprogramme schreiben
  - Testdatensätze entwickeln
  - zunächst auf Klassenebene
  - Überdeckung des codes durch Test
- Warnungen beim Compilieren einschalten (Option -Wall); Purify benutzen

Dokumentation

- zwei Zielgruppen: für Entwickler — für Benutzer
- Dokumentation und code im selben File
- sofort dokumentieren !!
- in englischer Sprache
- Tools: Perceps, javadoc, DOC++
- Beispiel: Polymake
  http://www.math.tu-berlin.de/diskregeom/polymake/doc/
- Verwendung von Fehlerlisten

Literaturhinweise

- B. Mayer: Objektorientierte Softwareentwicklung, Hanser, 1990; englisches Original: Object-oriented software construction, Prentice Hall, 1988.
- J. Lakos: Large-Scale C++ Software Design, Addison-Wesley, 1996.
- E. Gamma et al.: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- K. Weihe: A Software Engineering Perspective on Algorithmics, Konstanzer Schriften in Mathematik und Informatik Nr. 50, 1998, http://www.informatik.uni-konstanz.de/Schriften