Grundprinzipien

des Software-Entwurfs

Andreas Zeller + Gregor Snelting

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Qualitätskriterien

Was erhoffen wir von Software?

Wir betrachten *Qualitätskriterien* wie

- Korrektheit
- Zuverlässigkeit
- Robustheit
- Effizienz
- Benutzerfreundlichkeit
- Wartbarkeit
- Wiederverwendbarkeit
- Vertrauenswürdigkeit







Qualitätskriterien (2)

Korrekte Software entspricht der Spezifikation

Ist die Spezifikation unpräzise oder unvollständig, gibt es keine Korrektheit – nur Überraschungen :-)

Sichere Software ist korrekte Software!

Mittel: *Verifikation*, *Programmbeweise*, *Ableitung aus korrektem Modell*

Zuverlässige Software erfüllt ihre Aufgabe meistens

Mittel: *Validierung*, *Tests*

Robuste Software funktioniert unter unvorhergesehenen Umständen

Beispiele: Benutzereingaben, Netzausfall, Stromausfall

Mittel: *Abdeckung* dieser Umstände in Pflichtenheft, *Fail-Safe-Technik*

Qualitätskriterien (3)

Effiziente Software nutzt Hardware-Ressourcen ökonomisch

Korrektheit geht vor Effizienz! (Muss Software nicht korrekt sein, darf sie beliebig effizient sein :-)

Mittel: *Effiziente Algorithmen*, *Messen*, *Verbessern*

Benutzerfreundliche Software kann von Menschen leicht erlernt und benutzt werden

(Siehe auch → Benutzerschnittstellen)

Wartbare Software kann leicht an neue Anforderungen angepasst werden

Wartungskosten: 60% der gesamten Softwarekosten!

Mittel: gute Architektur, Dokumentation, Prozess







Qualitätskriterien (4)

Wiederverwendbare Software kann leicht für neue Anwendungen verwendet werden

Ziel: „Zusammenstecken“ von Software aus vorgefertigten *Komponenten,* die man aus Katalogen zusammensucht

Variante: *Portierbarkeit*, *Kompatibilität*

Mittel: Abstraktion, Abstraktion, Abstraktion!

Vertrauenswürdige Software hat auch im Fehlerfall keine katastrophalen Auswirkungen

Ein Schachprogramm ist stets vertrauenswürdig.
Ein Flugzeug-Navigationsprogramm sollte besser vertrauenswürdig sein!

Mittel: Korrektheit, Zuverlässigkeit, Robustheit

Qualitätskriterien (5)

Kostengünstige Software hat niedrige Kosten

„Quick-and-dirty“-Programmierung führt zu niedrigen Kosten – zunächst.

Qualitativ hochwertige (= korrekte, wartbare, benutzerfreundliche. . .) Software senkt die Kosten für Fehlerkorrekturen, Anpassungen, Versicherungen. . .

. . . benötigt aber zunächst höhere Investitionen!

Nutzen und Risiken solcher Investitionen müssen genau abgewogen werden!







Grundprinzipien

Wie erreicht man diese Software-Qualität?

Wir betrachten vier *Grundprinzipien:*

1. Strenge und Formalität
2. Separation der Interessen ⇒ Modularität
3. Abstraktion ⇒ Allgemeinheit
4. Evolutionsfähigkeit ⇒ Inkrementalität







Prinzip 1: Strenge und Formalität

Software-Entwicklung ist eine kreative Tätigkeit

Viele Software-Entwickler, insbesondere Studenten und Wissenschaftler, lassen ihrer Kreativität bei der Software-Entwicklung freien Lauf

⇒ geniale Software schlechter Qualität bei unkalkulierbaren Terminen und Kosten

Dies mag im Wissenschaftsbereich ok sein, für eine am Markt orientierte Organisation aber nicht

⇒ Strenge und formale Verfahren sind ein notwendiges Komplement

Strenge und Formalität (2)

Strenge: Vorgehen nach strengen Regeln – etwa

1. Pflichtenheft erstellen
2. Software gemäss Pflichtenheft entwerfen
3. Aus Entwurf Feinspezifikation ableiten
4. Feinspezifikation validieren. . .







Strenge und Formalität (3)

Formalität ist mehr als Strenge:
Bei formalem Vorgehen muss in jedem Schritt ein mathematisches Gesetz angewendet werden.

Beispiel: Lehrbücher der Mathematik sind streng, aber nicht formal!

Formale Verfahren in der Softwareentwicklung:

- Algebraische Spezifikation
- Programmverifikation

Formalität und Strenge engen nicht die Kreativität ein, sondern erhöhen Korrektheit und Zuverlässigkeit







Prinzip 2: Separation der Interessen

Wer alles auf einmal machen will, erreicht nichts

In der Software-Entwicklung werden verschiedene Aspekte wie

- Funktionalität,
- Qualitätsmerkmale,
- Hard- und Softwareumgebung,
- Teamorganisation,
- Kostenkontrolle . . .

soweit möglich *isoliert betrachtet* („Teile und Herrsche“).

Dies ist die einzige Möglichkeit, die Komplexität der Entwicklung in den Griff zu bekommen!

Separation der Interessen (2)

- Trennung verschiedener *Entwicklungsphasen*
  - Erst wird entworfen
  - Dann implementiert
  - Dann getestet
- Trennung verschiedener *Qualitätsaspekte*
  - Korrektheit
  - Effizienz
  - Benutzerschnittstelle







Separation der Interessen (3)

- Trennung verschiedener *Sichten*
  - Statischer Aufbau (Objekt-Modell in UML)
  - Dynamisches Verhalten (Sequenzdiagramme in UML)
- Trennung verschiedener *Teile*
  - Zerlegung eines Systems in Elemente
  - Festlegung der Schnittstelle zwischen Elementen

„Trennung“ bedeutet hier oft auch „personelle Trennung“:
Jede/r sollte das machen, was er/sie am besten kann

In der Software-Entwicklung ist es selten,
daß eine ganzheitliche Sicht der Dinge mehr bringt







Spezialfall: Modularität

Modul = Software-Komponente mit wohldefinierter Schnittstelle

Schnittstelle: Dienste, die der Anwender nutzen kann

Im objektorientierten Sinne: Modul = Klasse

Interne Realisierungsdetails (lokale Datenstrukturen und Algorithmen) sind nach aussen unbekannt!
Beispiel: Der Benutzer einer Warteschlangen-Klasse braucht nicht zu wissen, ob die Warteschlange als Feld oder verkettete Liste implementiert ist; er muss aber wissen, wie die Methoden zum Anfügen / Entfernen benutzt werden.

Ziel: Änderungen in einem Modul bedingen aussen keine weiteren Änderungen







Grundprinzipien der Modularisierung

Hohe Kohäsion: Module sollten logisch zusammengehörende Funktionen enthalten

Wenige Schnittstellen: Module sollten mit möglichst wenig anderen Modulen kommunizieren

Schwache Kopplung: Module sollten so wenig Information wie möglich austauschen

Geheimnisprinzip: *Niemand darf modulinterne Daten abfragen oder manipulieren. Auf die Dienste eines Moduls darf nur durch Aufruf von Schnittstellenfunktionen zugegriffen werden.*

Prinzip 3: Abstraktion

Abstraktion: Ignorieren von irrelevanten Details;
Herausarbeiten des (für einen Zweck) Wesentlichen

Oft hat man Abstraktionshierarchien, wobei jede Schicht die adäquate Beschreibung für einen bestimmten Zweck ist:

- Elektronen / Quantenmechanik
- Transistoren / Festkörperphysik
- Gatter / Boolesche Algebra
- Maschinensprache / Rechnerorganisation
- höhere Sprache / Compilertheorie
- Module und Komponenten / Software Engineering

Jede höhere Schicht ignoriert Details der darunterliegenden







Abstraktion (2)

Klassischer Abstraktionsmechanismus in der Informatik: *höhere Programmiersprachen.* Diese verstecken die Details der Hardware.

Auch die Konstruktion eines Algorithmus ist eine Abstraktion: man abstrahiert von konkreten Eingabedaten und sucht ein allgemeines Verfahren.

Software-Entwurf: Definition geeigneter Funktionsbausteine
Von der konkreten Realisierung der Bausteine wird abstrahiert







Abstraktion (3)

Abstraktion ist der *Königsweg zur Wiederverwendbarkeit*:

- Ersetze anwendungsspezifische Teile durch formale *Parameter*
- *Instantiiere* diese Parameter je nach Bedarf der Anwendung

Das Finden geeigneter Abstraktionen ist der Kern der Informatik

Spezialfall: Allgemeinheit

„Every time you are asked to solve a problem, try to discover a more general problem that may be hidden behind the problem at hand.

It may happen that the generalized problem is not more complex—it may even be simpler—than the original problem. Being more general, it is likely that the solution to the more general problem has more potential for being reused.

It may even happen that the solution is provided by some off-the-shelf package. Also, it may happen that by generalizing a problem you end up designing a module that is invoked at more than one point of the application, rather than having several specialized solutions.“ (Ghezzi)

Spezialfall: Allgemeinheit (2)

Nimm...	**statt...**
Allgemeines Sortierprogramm	Quicksort für Arrays fester Größe
Tabellenkalkulation	Selbstgebastelter Taschenrechner
Datenbanksystem	Handimplementierte Rot/Schwarz-Bäume
Grafik-Bibliothek	Direktes Ansprechen der Grafikkarte
Textprozessoren (Perl)	Selbstrealisierter Knuth-Morris-Pratt-Algorithmus zur Textsuche

Allgemein verwendbare Software verkauft sich besser!







Prinzip 4: Evolutionsfähigkeit

In Software ist nichts beständiger als der Wandel

Bereits bei der Erstellung muß der zukünftigen *Software-Evolution* Rechnung getragen werden

Eigenschaften, die sich absehbar ändern, sollten von vornherein in spezifischen Komponenten isoliert werden („Antizipation des Wandels“)

Dadurch ist nur ein kleiner Teil der Software von der Änderung betroffen.

Beispiele für Evolutionsfähigkeit

Feldgrößen

Schlechte Lösung: Überall im Programm steht
`if (i < 100) { ...a[i]... }...`
Gute Lösung: Verwende symbolische Konstante

Datumsfunktionen

Schlechte Lösung: im Programm stehen Umrechungen wie
```
if (year > 30 && year < 99) { year += 1900 }
...
```
Gute Lösung: Verwende explizite Datumsfunktionen

Fehlermeldungen

Schlechte Lösung: Englische Texte sind im Code verstreut
Gute Lösung: Meldungen werden aus Datei eingelesen

⇒ *Wiederverwendbarkeit* wird deutlich verbessert!







Spezialfall: Inkrementalität

Inkrementalität = Vorgehen in kleinen Schritten

Inkrementalität im Software Engineering: ein Produkt wird als Sequenz von Approximationen realisiert („evolutionäres Vorgehen“)

Jede Approximation entsteht durch kleine Änderungen/Erweiterungen aus der vorangegangenen Version

Vorteile:

- Teilsysteme mit eingeschränkter Funktionalität können frühzeitig ausgeliefert werden
- Frühzeitiges Feedback durch Benutzer
- Vollständige Spezifikation ist am Anfang nicht nötig







Spezialfall: Rapid Prototyping

Das System wird zunächst in einer ausführbaren Spezifikationssprache (VDM, ASF, RAP) realisiert
Später kann eine effiziente Version mit endgültiger Benutzerschnittstelle implementiert werden

Variante: Realisiere zunächst eine hohle Benutzerschnittstelle und benutze diese zu Präsentationen / Diskussionen mit dem Kunden. Später wird dann die echte Funktionalität ergänzt

Hohle Schnittstellen zu bauen ist gängige Praxis!







Grundprinzipien und Qualitätsmerkmale

++, + (sehr) positive Auswirkung des Grundprinzips auf Qualitätsmerkmal
– negative Auswirkung des Grundprinzips auf Qualitätsmerkmal

	Strenge, Formalität	Separation der Interessen	Modularität	Abstraktion	Allgemeinheit	Evolutionsfähigkeit	Inkrementalität
Korrektheit	++	+	+	+		+	
Zuverlässigkeit	++	+	+	+	+	+	
Robustheit	++	++	++	+			
Effizienz			–	–	–		
Benutzerfreundlichkeit		+	+	+	+		++
Wartbarkeit	+	++	++	+		++	+
Wiederverwendbarkeit		++	++	++	++	++	+
Portierbarkeit	–	++	++	+			
Kompatibilität		+	+	+	+		
Vertrauenswürdigkeit	++		++	++	–		–
Niedrige Kosten	–	+	+	++	++	++	++