

Software- Qualitätsmanagement

Kernfach Angewandte Informatik

Sommersemester 2005

Prof. Dr. Hans-Gert Gräbe

Bindung und Kopplung

Die Bindung innerhalb einer Systemkomponente und die Kopplung der Systemkomponenten untereinander bestimmen die Struktur eines Software-Systems.

Bindung (*cohesion*) ist ein qualitatives Maß für die Kompaktheit einer Systemkomponente. Es werden dazu die Beziehungen zwischen den Elementen einer Systemkomponente betrachtet.

Kopplung (*coupling*) ist ein qualitatives Maß für die Schnittstellen zwischen den Systemkomponenten. Es werden der Kopplungsmechanismus, die Schnittstellenbreite und die Art der Kommunikation betrachtet.

7. Analysierende Verfahren

2. Bindung und Kopplung

Bindung von Prozeduren und Funktionen

- Gute Bindung liegt vor, wenn nur solche Elemente zu einer Einheit zusammengefasst werden, die auch zusammen gehören.
- Bindung von Funktionen wird nur qualitativ erfasst.

[Stevens 81] unterscheidet folgende Bindungsarten :

1. zufällige Bindung
2. logische Bindung
3. zeitliche Bindung
4. prozedurale Bindung
5. kommunikative Bindung
6. sequentielle Bindung
7. funktionale Bindung

schwache Bindung



starke Bindung

7. Analysierende Verfahren

2. Bindung und Kopplung

Ziel: Erreichen einer funktionalen Bindung.

- Alle Elemente sind an der Verwirklichung einer einzigen, abgeschlossenen Funktion beteiligt.
- Komplexe Funktionen werden realisiert, indem importierte Funktionen verwendet werden, die selbst funktional gebunden sind.
- **Kennzeichen** einer funktionalen Bindung:
 - Alle Elemente tragen dazu bei, ein einzelnes spezifisches Ziel zu erreichen.
 - Es gibt keine überflüssigen Elemente.
 - Die Aufgabe kann mit genau einem Verb und genau einem Objekt beschrieben werden.
 - Austausch gegen anderes Element, welches denselben Zweck erfüllt, leicht möglich.
 - Hohe Kontextunabhängigkeit, d.h. einfache Beziehungen zur Umwelt.

7. Analysierende Verfahren

2. Bindung und Kopplung

Vorteile einer funktionalen Bindung:

- Hohe Kontextunabhängigkeit (die Bindungen befinden sich innerhalb der Prozedur, nicht zwischen Prozeduren).
 - Geringe Fehleranfälligkeit bei Änderungen,
 - Hoher Grad der Wiederverwendbarkeit,
 - Leichte Erweiterbarkeit und Wartbarkeit, da sich Änderungen auf isolierte, kleine Teile beschränken.
-
- Konzept der Bindung verallgemeinert die (konzeptuellen) Regeln für „guten Code“ zu Regeln für „guten Software-Entwurf“.
 - Die Bindungsart einer Prozedur lässt sich nicht automatisch ermitteln, sondern nur durch manuelle Prüfmethoden.
 - Diese Untersuchungen sind noch im experimentellen Stadium und haben heute weitgehend informellen (damit aber nicht weniger bindenden) Charakter.

7. Analysierende Verfahren

2. Bindung und Kopplung

Bindung von Datenabstraktionen/Klassen

Beschreibt das Zusammenwirken verschiedener Funktionen, welche derselben Datenabstraktion oder Klasse zuzuordnen sind.

- Voraussetzung: Alle Methoden sind funktional gebunden

Gute Bindung einer Klasse (*model cohesion*) liegt vor, wenn

- sie ein einzelnes semantisch bedeutungsvolles Konzept repräsentiert,
- die Klasse keine verborgenen Klassen enthält und
- keine Operationen enthält, die an andere Klassen delegiert werden können.

Wird in der Literatur auch als Kohärenz bezeichnet.

Für Klassen ist weiter die Bindung innerhalb von Vererbungsstrukturen wesentlich.

Informale Bindung

[Myers 78] fordert für abstrakte Datenobjekte informale Bindung.

- liegt vor, wenn mehrere, in sich abgeschlossene, funktional gebundene Zugriffsoperatoren, die zu einer Datenabstraktion gehören, auf einer einzigen Datenstruktur operieren.
- **Idee:** hinter der gemeinsamen Funktionalität liegt auch ein gemeinsames Datenmodell

Merkmale:

- Unterstützt das Geheimnisprinzip, d.h. die Datenstruktur gehört nur zu einer Datenabstraktion,
- Änderungen der Datenstruktur tangieren nur eine Datenabstraktion,
- Problem der Vermischung von Zugriffsoperationen, da alle auf derselben Datenstruktur operieren.

Bindung in Vererbungsstrukturen

- Die ganze Vererbungshierarchie muss untersucht werden.
- **Starke Vererbungsbindung** liegt vor, wenn die Hierarchie eine Generalisierungs-/Spezialisierungshierarchie im Sinne der konzeptuellen Modellierung ist.
- **Schwache Vererbungsbindung** liegt vor, wenn die Hierarchie nur zum "*code sharing*" verwendet wird.
- Das **Ziel** jeder neu definierten Unterklasse muss sein, ein einzelnes semantisches Konzept auszudrücken.

Einführung

Quantitative Aussagen über die Produktqualität einer Systemkomponente können mit Hilfe von **Metriken** ermittelt werden.

- Mit solchen Metriken sind heute nur einfache Aussagen über Eigenschaften einer Komponente möglich.
- Eine Metrik bewertet ein Software-System immer nur unter einem sehr speziellen Blickwinkel.
- Aussagekräftiger Gesamteindruck von einer Systemkomponente nur durch Auswertung einer Gruppe von Metriken, oft auch nur im Vergleich zu Parametern anderer, bereits im Einsatz befindlicher Komponenten.
- Metriken können nicht nur für bereits implementierte Komponenten, sondern auch schon entwicklungsbegleitend eingesetzt werden.

Metriken zum Erfassen der prozeduralen Komplexität

- **Umfangsmetriken**
 - sind die ältesten Metriken
 - stellen ab auf die textuelle Komplexität
 - verwenden einfach verfügbare Informationen (Anzahl an Programmzeilen, Dateigröße, Zahl der Funktionen, ...)
 - Vertreter: Halstead-Metrik, Function Points (zur Erfassung des Umfangs verbaler Anforderungen)
- **Logische Strukturmetriken** (Kontrollfluss-Metriken)
 - Analyse des Kontrollfluss-Graphen
 - Wird samt seiner Begleitobjekte (Symboltabelle) sowieso vom Compiler ausgewertet
 - Vertreter: McCabe-Metrik

- **Datenstrukturmetriken**
 - messen die Anzahl an Variablen, deren Gültigkeit und Lebensdauer sowie die Referenzierung der Variablen
- **Stilmetriken**
 - messen ob die Programme richtig eingerückt wurden und ob die Namenskonventionen eingehalten wurden
- **Interne Bindungsmetriken**
 - messen die syntaktische Bindung durch Prüfen des Codes jeder Komponente

7. Analysierende Verfahren

3.1. Die Halstead-Metrik

Umfangsmetriken – Die Halstead-Metrik

- Misst die textuelle Komplexität eines Programms, indem die Zahl der verwendeten Funktionen und der verwendeten Variablen ins Verhältnis gesetzt werden.
 - Es wird jeweils die Gesamtzahl (Programmtext) und die Zahl verschiedener Objekte (Symboltabelle) bestimmt.
 - η_1, N_1 = Zahl der (verschiedenen) Funktionen, Operatoren, Symbole oder Schlüsselwörter (z. B.: +, -, *, /, **while**, **if**, ...)
 - η_2, N_2 = Zahl der (verschiedenen) Variablen, Operanden ...
- **Interpretation:**
 - $\eta = \eta_1 + \eta_2$: Größe des Vokabulars
 - $N = N_1 + N_2$: Länge der Implementierung

7. Analysierende Verfahren

3.1. Die Halstead-Metrik

- **Vorteile:**
 - einfach zu ermitteln,
 - bei jeder Programmiersprache verwendbar und
 - gute Eignung der Metriken für die zu messenden Größen
- **Nachteile:**
 - nur der Implementierungsaspekt betrachtet und
 - Mehrdeutigkeiten im Messansatz, z. B. bei den Klassifikationsregeln für Operatoren und Operanden
- **abgeleitete Größen:**
 - $D = \eta_1/2 \cdot N_2/\eta_2$, Parameter für die Schwierigkeit, den Code zu verstehen
 - Interpretation: $N_2/\eta_2 =$ durchschnittliches Vorkommen jeder Variablen, $\eta_1 =$ Anzahl der verwendeten Funktionen

7. Analysierende Verfahren

3.1. Die Halstead-Metrik

```

int ZaehleVokale(String s) {
    int VokalAnzahl; char Zchn; int i;
    for(i=0; i < s.length(); i++) {
        Zchn=s[i];
        if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||
            (Zchn == 'O') || (Zchn == 'U')) VokalAnzahl++;
    }
    return VokalAnzahl;
}

```

ZaehleVokale	1	int	3	()	9	++	2
VokalAnzahl	3	String	1	{ }	2	[]	1
Zchn	7	char	1	;	8	==	5
s	3	for	1	=	2		4
i	5	if	1	<	1	.	1
Konstanten (6)	6	return	1	length	1		

$$\eta_1=17, N_1=44, \eta_2=11, N_2=25, D=19.32$$

7. Analysierende Verfahren

3.2 McCabe-Metrik

Kontrollfluss-Metriken – Die McCabe-Metrik

- Misst die strukturelle Komplexität eines Programms, indem eine Grapheninvariante, die **zyklomatische Zahl** $V(G)$ des Kontrollflussgraphen, bestimmt wird.
- $V(G) = e - n + 2p$ mit
 - e = Anzahl der Kanten des Graphen
 - n = Anzahl der Knoten
 - p = Anzahl der Zusammenhangskomponenten
- Kontrollflussgraph wird für jede Prozedur aufgestellt ($p=1$).
- Für solchen Graphen gilt

$$V(G) = b + 1$$

mit b = Anzahl der Bedingungen.

- Zyklomatische Zahl ist additiv auf Komponenten.
- Lineare Teilstücke können zusammengezogen werden.

7. Analysierende Verfahren

3.2 McCabe-Metrik

- **Vorteile:**
 - einfach zu berechnen,
 - grobes Maß für die Kontrollflusskomplexität: je größer, desto weiter weicht der Kontrollfluss vom linearen ($V(G)=1$) ab.
- **Nachteile:**
 - unterschiedliche Programmmerkmale werden stark vereinfacht
 - Quellprogramm als zentrales Messobjekt überbetont
 - Es wird nur das Programmgerüst, nicht aber die Komplexität einzelner und verschachtelter Anweisungen berücksichtigt

7. Analysierende Verfahren

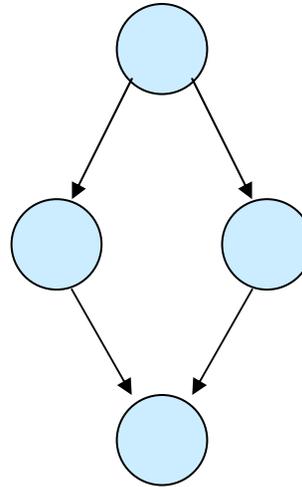
3.2 McCabe-Metrik - Beispiel

Sequenz



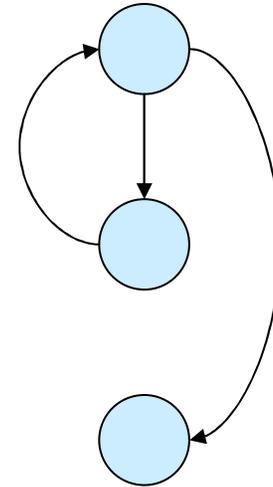
$$V(G) = 1 - 2 + 2 = 1$$

Auswahl



$$V(G) = 4 - 4 + 2 = 2$$

Abweisende Schleife



$$V(G) = 3 - 3 + 2 = 2$$

- das Programm ZaehleVokale hat die zyklomatische Zahl

$$V(G) = 7 - 6 + 2 = 3$$

ZaehleVokale enthält zwei Bedingungen → $V(G) = 3$

Metriken für objektorientierte Entwicklung

- Metriken der klassischen Software-Entwicklung sind in unveränderter Form für OO-Projekte nur bedingt aussagefähig.
 - Umfangsmetriken: Wie mit geerbtem Code umgehen?
 - Durch Vererbung und Polymorphismus sinkt die Zeilenzahl signifikant
 - McCabe-Metrik: Kontrollflusskomplexität bei OO meist sehr gering
- Frage nach Maßen für die OO-spezifischen Effekte
 - zusätzliche Maße waren erforderlich
 - Breite und Höhe der Vererbungshierarchie
 - Anzahl der Klassen, die eine spezielle Operation erben
 - Anteil wieder verwendeter Komponenten
 - Anzahl der Objekt- und Klassenattribute
 - Anzahl der Objekt- und Klassenoperationen

Typische Metriken (Beispiele)

- Objekt- und Klassenattribute einer Klasse:
 - Anzahl: $|OV|$, $|CV|$
 - gewichtete Anzahl: $\sum T(v)$
 - $T(v)$ = Gewicht des Typs des Attributs v
 - gewichtet nach Zahl der Vorkommen, nach Zahl der Vorkommen in verschiedenen Methoden der Klasse etc.
- Objekt- und Klassenmethoden einer Klasse:
 - $|OM|$, $|CM|$, evtl. wieder gewichtet nach Komplexität
 - Parameterkomplexität einer Methode
 - Zahl und Gewicht der Typen der Aufrufparameter
 - McCabe-Metrik
- durchschnittliche Komplexität von Klassen in einem Paket
 - Durchschnittswerte der einzelnen Klassenparameter
 - Kreuzreferenzparameter (Bindungsanalyse im Paket)

Folgende Metriken wurden als signifikant befunden
[Basili, Briand, Melo 1996], [Chidamber, Kemerer 1994]

- DIT (*Depth of Inheritance Tree*):
 - Tiefe des Vererbungsbaumes (Zahl der Vorfahren einer Klasse)
 - je höher der Wert von DIT, desto höher die Fehlerwahrscheinlichkeit (Hoher Nachnutzungsgrad, Nichtbeachtung verdeckter Annahmen)
- NOC (*Number of Children of a Class*):
 - Anzahl der direkten Nachfolger einer Klasse
 - je höher der Wert von NOC, desto geringer die Fehlerwahrscheinlichkeit (sehr präzise Abstraktion, Nichtvorhandensein verdeckter Annahmen)
- RFC (*Response For a Class*):
 - Anzahl der Funktionen, die direkt durch die Methoden einer Klasse aufgerufen werden.
 - je höher der Wert von RFC, desto größer die Fehlerwahrscheinlichkeit (Hoher Delegationsgrad, Nichtbeachtung verdeckter Annahmen)

- WMC (*Weighted Methods per Class*):
 - Anzahl aller neu definierten oder überschriebenen Methoden, die in jeder Klasse definiert sind.
 - Geerbte Methoden werden nicht gezählt.
 - je größer der Wert von WMC, desto höher die Fehlerwahrscheinlichkeit (??)
- CBO (*Coupling Between Object Classes*):
 - eine Klasse ist mit einer anderen Klasse gekoppelt, wenn sie deren Methoden und/oder Attribute benutzt.
 - CBO ist die Anzahl der Klassen, mit der eine Klasse gekoppelt ist.
 - je größer der Wert von CBO, desto größer die Fehlerwahrscheinlichkeit (höherer Verschränkungsgrad)

Vorteile:

- erste Ansätze zur Verbesserung objektorientierter Komponenten
- erste empirische Untersuchungen zeigen die Eignung einiger Metriken als Qualitätsindikatoren

Nachteile:

- die Ziele der Metriken sind nur implizit erkennbar
 - Zielrelevanz der Metriken noch ungenügend untersucht
- keine Metriken für dynamische Aspekte
 - z. B.: Zustandsautomaten
- keine semantische Unterscheidung der Methoden
 - Standardoperationen (lesen, schreiben, erzeugen,...) sind weniger fehleranfällig als auszuprogrammierende "fachkonzeptspezifische" Methoden
- keine Berücksichtigung der Oberklassenqualität
 - ob eigene, geerbte oder fremde Methoden
 - ob Vererbung von Methoden aus Standardklassen oder eigenen
- keine Metriken, die eine "gute" Vererbungsstruktur prüfen

Eine **Anomalie** ist jede Abweichung bestimmter Eigenschaften eines Programms von der korrekten Ausprägung dieser Eigenschaften.

- Konstruktive Sprachkonzepte erlauben das Aufdecken von Anomalien durch statische Quelltextanalyse
 - Beispiel: Typprüfung durch den Compiler
 - Oft ist keine unmittelbare Fehlererkennung, jedoch eine Identifikation von Fehlerort und -symptomen möglich.

Datenflussanomalien-Analyse

- **Ziel:** Aufdecken von Datenflussanomalien durch Analyse von Programmpfaden auf sinnvolle Datenfluss-Sequenzen

7. Analysierende Verfahren

3.4 Anomalienanalyse

Datenfluss-Eigenschaften von Variablen

- Auf eine Variable x kann entlang eines Programmpfades wie folgt zugegriffen werden:
 - x wird definiert (d),
 - x wird referenziert (r),
 - x wird undefiniert (u) (z.B. beim Verlassen einer Methode)
 - x wird „geleert“ (e), d.h. der Wert an einen anderen Ort übertragen
- Enthält die Sequenz Teile, die keinen Sinn ergeben, so liegt eine Datenfluss-Anomalie vor.
 - Beispiele:
 - **rdru** → die Sequenz beginnt mit einer Referenz, vor der Definition, diese Anomalie ist vom Typ **ur**
 - **ddrdu** → diese Sequenz beginnt mit einer doppelten Definition (Anomalientyp **dd**) und endet mit **du**

7. Analysierende Verfahren

3.4 Anomalienanalyse

Beispiel

```
/* swap (int a, int b) */
  int hilf; a=hilf; a=b; hilf=b;
```

Analyse:

a d : d d : r
b d : r r : r
hilf u : r d : e

Anomalietyp:

mehrfach nacheinander überschrieben
nie verändert
neu definiert vor e

```
/* swap (int a, int b) korrigierte Version */
  int hilf; hilf=b; b=a; a=hilf;
```

a d : r d : r
b d : r d : r
hilf u : d r : e

Leistung:

entdeckt Wertzuweisungen an falsche Variablen, Anweisungen an unkorrekter Stelle und fehlende Anweisungen

• **Vorteile:**

- sichere Entdeckung bestimmter Fehlertypen,
- geringer Aufwand im Vergleich zu dynamischen Verfahren,
- direkte Fehlerlokalisierung und
- gute Ergänzung zu anderen Testverfahren

• **Nachteile:**

- Leistungsfähigkeit auf schmalen Fehlerbereich begrenzt