technische universität
dortmund

Software- Engineering für langlebige Systeme

Thomas Ruhroth | Dortmund
SoSe 2015

Softwareerosion

VL2

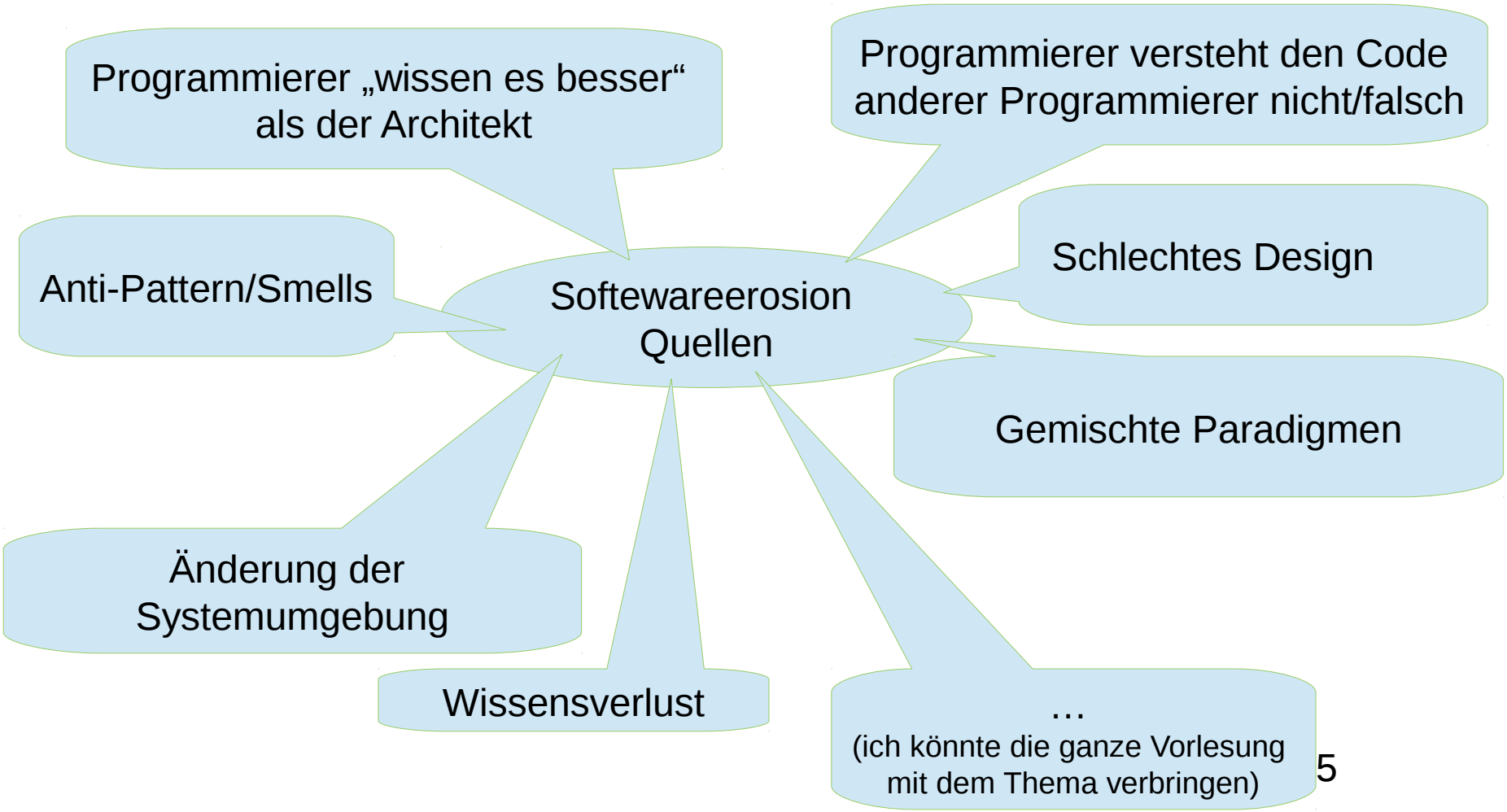
- Softwareerosion
  - Quellen
  - Erkennen und Messen

- Ziele:
  - Die wichtigsten Quellen von Softwareerosion kennen lernen.
  - Wissen, wie sich Softwareerosion bemerkbar macht.

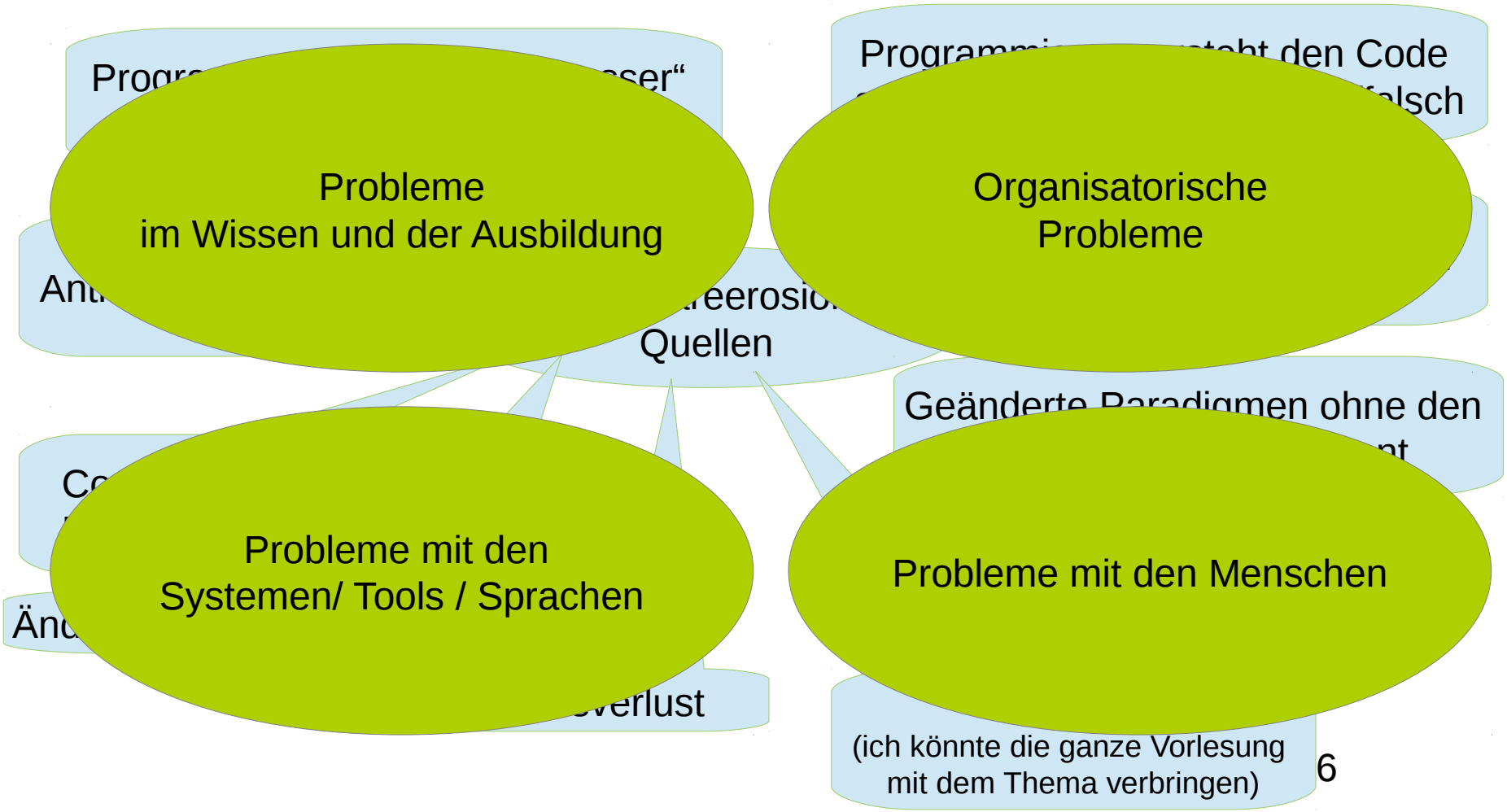
Softwareerosion

Quellen ...

Beispiele von Quellen



Gründe für Quellen



Beispiele:

B1: Programmierer versteht den Code anderer Programmierer nicht/falsch

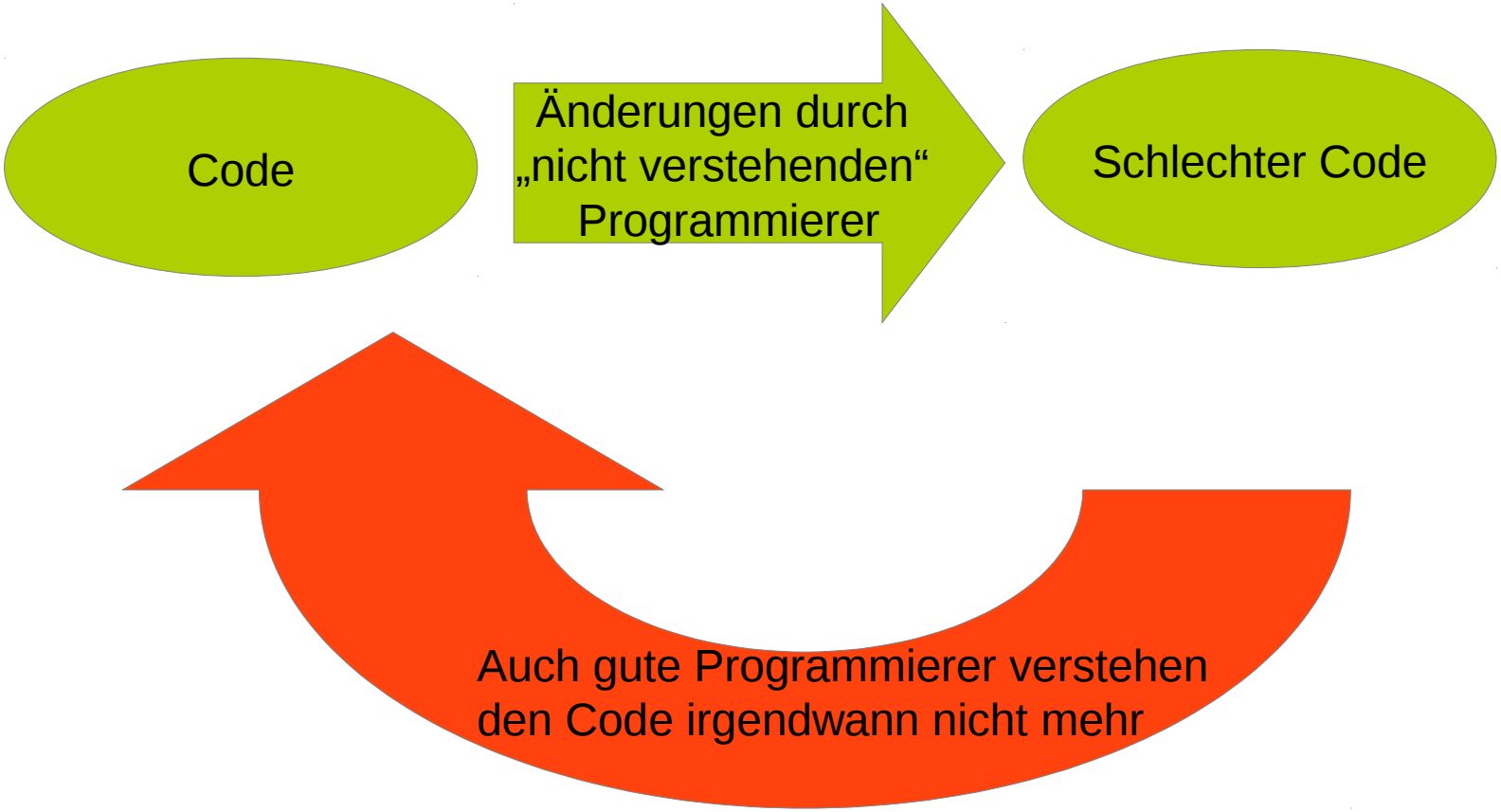
B2: Smells/Anti-Pattern

B3: Verlust von Wissen

Programmierer versteht den Code anderer Programmierer nicht/falsch

```
void get_crc( unsigned char d )
{
 int i;  c ^= d;
              for( i = 8; i; i-- ){
  c = (c >> 1) ^ ((c & 1) ? 0xA001 : 0 );
 }
}
```

Programmierer versteht den Code anderer Programmierer nicht/falsch



Programmierer versteht den Code anderer Programmierer nicht/falsch

Unterschiede in der Programmiererfahrung

- Code Anderer wird fehlinterpretiert
- „Verschlimmbessern“
- Unangemessene Verallgemeinerung/Vereinfachung

Schlechter Code

- Code nicht verständlich
- Wenn eine Stelle geändert wird, müssen viele (weit entfernte) Code-Stellen mit angepasst werden.
- Wenn der Code geändert wird, ergeben sich viele Folgefehler

Achtung:

- Schlechter Code ist Auswirkung und Quelle zugleich!
- Hier beginnt ein Teufelskreis

Änderung der Systemumgebung

- Prozessoränderungen
  - Umstellung 8 → 16 → 32 → 64 Bit-Systeme (s. Übung)
  - Prozessorgeschwindigkeit (TurboPascal- Zählerüberlauf)
- Änderungen im Betriebssystem
  - Geänderte Zugriffsrechtesysteme
  - Funktionen haben einen geänderten Effekt
- Neue/Geänderte Schnittstellen

Anti-Pattern

- Vorgehensweisen die schlecht sind, aber dennoch häufig vorgefunden werden

- Projektmanagement-Anti-Pattern
  - Smoke and mirrors, Feature creep, Scope creep, Brooks'sches Gesetz
- Architektur- bzw. Entwurfs-Anti-Pattern
  - Big Ball of Mud, God object
- Programmierungs-Anti-Pattern
  - Onion, Lavafluss, Magic Values
- Organisations-, Management- bzw. Prozess-Anti-Pattern
  - Reinventing the square wheel, Body ballooning, Warm body, Net Negative Producing Programme

Anti-Pattern: Copy-Und-Pase-Programmierung

- Programmierung mittels Kopieren und Einfügen
  - Code nicht neu entwickelt
  - existenter Quelltexte wird hrauskopiert und angepasst .

- Fehler werden mitkopiert
- Für den neuen Bereich nicht optimal einsatzbereit
- Der Entwickler reflektiert weniger über sein Programm
- Fehleranfälliges Vorgehen
  - Entwickler nicht weiß, was er eigentlich macht.
- Wartbarkeit des Codes ist reduziert
  - (fast) gleicher Programmcode an vielen Stellen

Verlust von Wissen

- Designentscheidungen können nicht mehr nachvollzogen werden
  - Wichtige Gründe werden bei Umbauten nicht bedacht
- Bedeutung von Magic Numbers geht verloren
- Funktionen werden außerhalb der vorgesehenen und geprüften Bereiche genutzt

Design-Probleme

- Das Design ist zu komplex/zu einfach
- Es existieren viele (unübersichtliche) Abhängigkeiten (s. Findbugs-Bsp)
- Entscheidungen sind nicht dokumentiert
- Verschiedene widersprüchliche Spezifikationen
- Design für Zielsprache nicht angemessen/passend

Achtung:

- Schlechtes Design ist Auswirkung und Quelle zugleich!
- Hier beginnt ein Teufelskreis

Design-Probleme: Inkonsistenzen zwischen Artefakten

- Code
- Datenbank
- Design/Modell
- Dokumentation
- Tests

Achtung:

- Inkonsistenzen zwischen Artefakten sind Auswirkung und Quelle zugleich!
- Hier beginnt ein Teufelskreis

Softwareerosion

Feststellen

Erkennen von Softwareerosion

- Fühlen
  - Wartungsaufwand steigt
  - Fehlersuche ständig kompliziert
- Auftreten von Anti-Patterns und Smells
  - Regelmäßig zu beobachtende Mängel
  - Nicht quantifiziert
- Metriken
  - Zahlenmäßiges erfassen der Qualität
  - Jede Metrik stellen nur einen Messpunkt dar
  - Veränderung hat mehr Aussagekraft als der eigentliche Wert

Smells/Anti-Pattern

Smells

- Codestellen können „stinken“
  - Immer wenn man denkt: Was soll das?!?
- Dabei gibt es verschiedene Gerüche

- Smells werden auch in der Softwareentwicklung unter Code-Qualität betrachtet. Sie spielen aber auch als Anzeichen von Softwareerosion eine wichtige Rolle

- Anti-Pattern können Smells hervorufen!

  Copy- und Past-Programmierung (Anti-Pattern) → Duplizierter Code (Smell)

Duplizierter Code

- Code ist in gleicher (oder nur leicht ähnlicher) Form an mehreren Stellen vorhanden.
- Problem: Wenn der eine Code angepasst wird, muss häufig auch der andere Code angepasst werden
- Fehlerquelle für unvollständige Fixes

Lange Methoden

- Eine Methode ist so lang, dass man im Editor die Methode nicht mehr ganz überblicken kann
- Problem: Die Methode ist dann meist so komplex, dass man sie nicht auf Anhieb verstehen und ändern kann.

- Große Klassen analog, werden jedoch wegen anderen „Gegenmaßnahmen“ meist getrennt behandelt.

Unangebrachte Intimität

- Zwei Klassen haben zu enge Verflechtungen miteinander
- Problem: Die Aufteilung auf die Klassen ist nicht gut gewählt.

Schrotkugeln herausoperieren (engl. Shotgun Surgery)

- Für eine Änderung müssen viele kleine Änderungen an vielen Klassen gemacht werden

Case-Anweisungen in objektorientiertem Code

- Polymorphismus macht Switch-Case-Anweisungen weitgehend überflüssig und erledigt das damit zusammenhängende Problem des duplizierten Codes

Temporäre Felder

- Ein Objekt verwendet eine Variable nur unter bestimmten Umständen – der Code ist schwer zu verstehen und zu debuggen, weil das Feld scheinbar nicht verwendet wird

Toter Code

- Ein Stück Code, das überhaupt nicht (mehr) verwendet wird

Spekulative Allgemeinheit

- Es wurden alle möglichen Spezialfälle vorgesehen, die überhaupt nie benötigt werden; solcher allgemeiner Code braucht nur Aufwand in der Pflege, ohne dass er etwas nützt

Zyklische Benutzungsbeziehungen zwischen Paketen, Schichten und Subsystemen

- Artefakte werden zyklisch genutzt
- Änderungen sind nicht mehr gekapselt.

Parallele Vererbungshierarchien

- Zu jeder Unterklasse in der einen Hierarchie gibt es immer auch eine Unterklasse in einer anderen Hierarchie

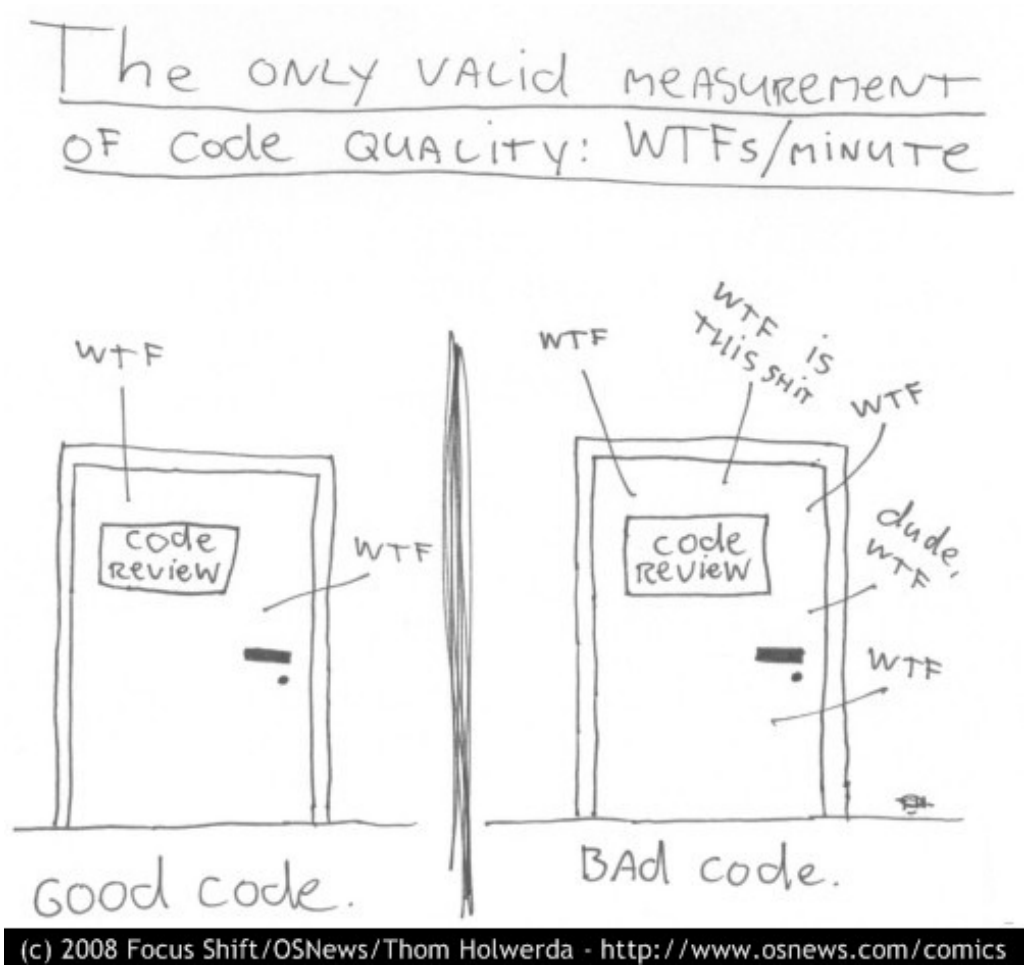
Metriken

Metriken

Nach Definition des IEEE ist:

> Eine Softwaremetrik [...] eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

WTF-Metrik



http://www.osnews.com/story/19266/wtfs_m

WTF-Metrik

http://www.osnews.com/story/19266/wtfs_m

Cyclomatic Complexity

- Die cyclomatic complexity v(G) wurde von Thomas McCabe in 1976 entwickelt
- Misst die Anzahl der linearly-independent paths durch eine Komponente/Modul (Control Flow).
- Die McCabe complexity ist relativ unabhängig von der verwendeten Programmiersprache

McCabe complexity - Berechnung

- v(G) die Anzahl von conditional branches.
- v(G) = 1 gilt, wenn es nur einen Ausführungspfad gibt
- Für eine einzelne Funktion ist
  v(G) = #conditional branching points - 1
- Je größer v(G) desto mehr Ausführungspfade
  - = schwerer verständlich

Berechnung

- Starte mit 1
- Erhöhe um 1 bei
  - if-statement (introduces a new branch to the program)
  - Iteration constructs such as for- and while-loops
  - Each case ...: part in the switch-statement
  - Each catch (...) part in a try-block
  - Construction exprl ? expr2 : expr3
  - Composit expressions (||, &&)

- Funktionsaufrufe werden ignoriert, obwohl sie auch Auswirkungen auf die Komplexität haben.

Beispiel (Wert 5)

```
// complexity == 1 (Grundwert)
public final void iterateComplex(final List<String> tokens) {
   if (tokens == null) {                    // complexity++
      return;
   }
   for (final String eachToken : tokens) {       // complexity++
      if (eachToken != null &&       // complexity++ complexity++
            ! "".equals(eachToken.trim())) {
         System.out.println(eachToken);
      }
   }
   return;
}
```

Halstead Metrics

- Entwickelt von Maurice Halstead (sen.)
- Aus dem Jahr 1977

- Sind mit die Ältesten und doch häufig genutzten Metriken
  - ➔ Starker Indikator für code complexity.
  - ➔ Häufige maintenance metric.
- Gruppe von (verwandten) Metriken

Halstead-Metriken

- Programmlänge (N)
- Vokabulargröße (n)
- Volumen des Programms (V)
- Schwierigkeitsgrad (D)
- Programmniveau/Program level (L)
- Implementieraufwand/Effort to implement (E)
- Implementierzeit/Time to implement (T)
- Anzahl der ausgelieferten Bugs (B)

Grundwerte

Die Halstead-Metriken betrachten den Quellcode als eine Aufeinanderfolge von Operatoren und Operanden.

Sie zählen

- die Anzahl der verschiedenen Operatoren (n1)
- die Anzahl der verschiedenen Operanden (n2)
- die Gesamtanzahl der Operatoren (N1)
- die Gesamtanzahl der Operanden (N2).

Alle anderen Halstead-Maße werden von diesen vier Werten abgeleitet.

Operators (1/2) – nach VersiSoft für C++

SCSPEC (storage class specifiers)

- Reserved words that specify storage class:
  - auto, extern, inlin, registerstatic, typedef, virtual, mtuable.

TYPE_QUAL (type qualifiers)

- Reserved words that qualify type:
  - const, friend volatile.

RESERVED

- Other reserved words of C++:
  - asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast,dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename.
-

Operators (2/2) – nach VersiSoft für C++

OPERATOR

- !, !=, %, %=, &, &&, ||,&=, ( ), *, *=, +, ++, +=, -, --, -=, →, ., ...,/,/=, :, ::, <, <<, <<=,<=, =, ==, >, >=, >>, >>=,?, [ ], ^, ^=, {, }, |, |=,~ts

Sonderfälle

Die folgenden Kontrollstrukturen case ...: for (...) if (...) seitch (...) while for (...) und catch (...) werden in einer besonderen Weise behandelt.

Der Doppelpunkt und die Klammern werden als Teil des Konstrukts betrachtet. Case und der Doppelpunkt oder for (...) if (...) switch (...) while for (...) und catch (...) sowie die Klammern werden zusammen als ein Operator gezählt.

Programmlänge (N)

Die Programmlänge (program length, N) ist die Summe der Gesamtzahl aller Operatoren und Operanden eines Programms:

$$N = N1 + N2$$

Vokabulargröße (n)

Die Vokabulargröße (vocabulary size, n) erhält man durch die Addition der Anzahl der verschiedenen Operatoren und Operanden:

$$n = n1 + n2$$

Volumen des Programms (V)

Das Volumen des Programm (program volume, V) gibt den Informationsgehalt der Software gemessen in mathematischen Bits an.

$$V = N * \log2(n)$$

Halsteads Volumen (V) beschreibt die Größe der Implementation. Die Berechnung erfolgt mit Hilfe der Anzahl der ausgeführten Operationen und der Bearbeiteten Operanden im Algorithmus. Der Wert V ist daher im Vergleich zu den Zeilenmetriken weniger vom Code-Layout abhängig.

V – gute Werte

Das Volumen einer Funktion sollte mindestens 20 und höchstens 1000 betragen. Eine parameterlose Funktion, die aus einer nicht leeren Zeile besteht beträgt etwa 20. Wenn das Volumen den Wert von 1000 übersteigt, macht die Funktion wahrscheinlich zu viele Dinge.

Das Volumen einer Datei sollte zwischen 100 und höchstens 8000 liegen.

Schwierigkeitsgrad (D)

Der Schwierigkeitsgrad (difficulty level, D) oder Fehlerneigung eines Programms ist proportional zur Anzahl der verschiedenen Operatoren in diesem Programm.

D ist ebenfalls proportional zum Verhältnis zwischen der Gesamtanzahl der Operatoren und der Anzahl der verschiedenen Operanden. Wird der gleiche Operand beispielsweise mehrmals im Programm benutzt, wird er dadurch fehleranfälliger.

$$D = (n1 / 2) * (N2 / n2)$$

Programmniveau/Program level (L)

Durch den Kehrwert des Schwierigkeitsgrades erhält man das Programmniveau.

Ein Programm mit einem niedrigen Niveau ist relativ anfällig für Fehler.

$$L = 1 / D$$

Implementieraufwand/Effort to implement (E)

Der Implementieraufwand (Effort to implement, E) ist proportional zum Volumen und zum Schwierigkeitsgrad des Programms.

$$E = V * D$$

Nächste Woche:

Psychologie
&
Softwarewartung auf Code-Ebene