Seminar „Ursachen und Vermeidung von Fehlern in der Softwaretechnik“

Programmiertips

Barbara Haupt
Institut für Informatik
FU Berlin
WS 05/06

Überblick

1. Programmier – PRINZIPIEN
2. Programmier - TECHNIKEN
   - Vermeide Wiederholungen!
   - Orthogonalität
   - Nichts ist endgültig!
   - Designe für Nebenläufigkeit
   - Design by Contract!
   - Designe für Tests!
3. Programmier – PSYCHOLOGIE
   - Keine eingeschlagenen Fenster!
   - Think!

Freie Universität Berlin

Vermeide Wiederholungen!

- Änderungen müssen an mehr als nur einer Stelle durchgeführt werden

  Problem: Irgendwann habe ich vergessen, wo sich die anderen Stellen befinden

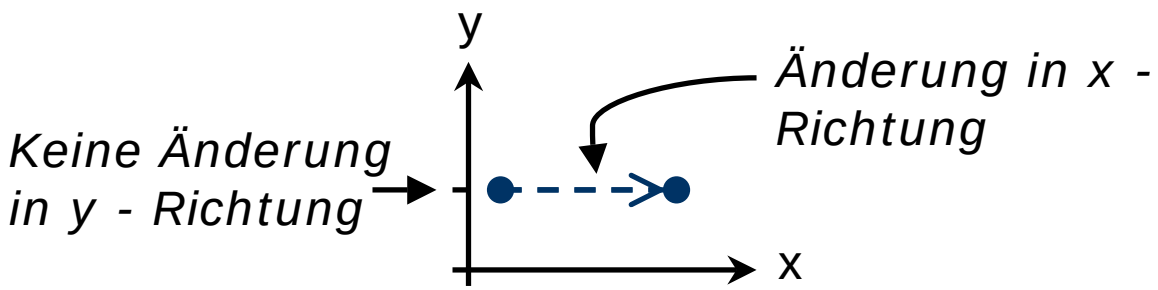
- Wiedersprüchliche Daten sind möglich!

```
class Line{
 public:
  Point start;
  Point end;
  double length;
};
```

vs.

```
class Line{
 public:
  Point start;
  Point end;
  double length(){
   return end.getDistanceTo(start);
  };
};
```

Orthogonalität



- Nicht-Orthogonale Systeme:
  - **komplex** (Änderungen in einem Teil sind nicht möglich, ohne dass ein anderer Teil davon betroffen ist)
  - **fehleranfällig** (Studien: [Car86] und [Sel91])
  - **weniger "mächtig"** als orthogonale Systeme (Analogon: Dimension bei Vektorräumen)

  Beispiel: einen Hubschrauber fliegen

Nichts ist endgültig!

- Szenario 1:

  Beginn des Projekts: Verwenden von Datenbank A

  Mitte des Projekts: Umstieg auf Datenbank B

  Ein Problem?

- Szenario 2:

  Beginn des Projekts: Client-/Server Model

  Mitte des Projekts: Stand-alone Produkt

  Und im umgkehrten Fall?

Alles neu schreiben? → Fehler bei der Planung!

Design für Nebenläufigkeit!

Schlechtes Beispiel:

Die C Library Routine `strtok`

1. Aufruf mit dem zu parsenden Argument
2. Alle nachfolgenden Aufrufe mit NULL

→ Es ist nicht möglich zwei Strings gleichzeitig zu parsen!

Wann darf der Status eines Objekts abgefragt werden?

Objekt im ungültigen Zustand: Kann ich wirklich sicher sein, dass niemand je zu diesem Zeitpunkt den Zustand abfragen wird?

Design by Contract!

- Verträge legen Rechte und Pflichten fest
- Verträge sind überprüfbar

Falls ich denke: “Das kann nie passieren!” sollte ich mich absichern, dass es nicht passiert

- Count kann ***nie*** negativ werden
- Dieser Code wird ***niemals*** mehr als 30 Jahre verwendet werden, also sind 2 Datumsstellen genug
- Diese Anwendung wird ***niemals*** woanders benutzt werden ...

- Korrekte Software tut nicht mehr und nicht weniger als das, was sie tun soll (Paradigma: Faule Software)

Designe für Tests!

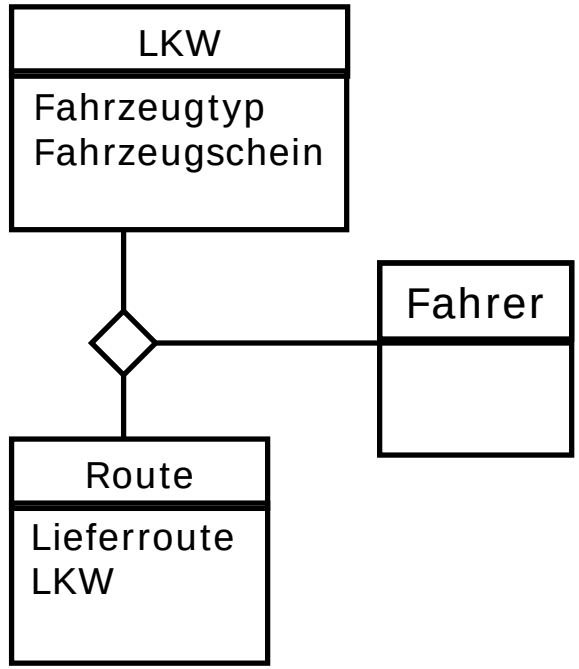
- Den Test schreiben, noch bevor die Routine implementiert ist!

Vorteile:

- Man kann das Interface ausprobieren, bevor man es verwendet.
- Code, den man auf's Testen hin entwirft, ist einfach zu testen!

Vermeide Wiederholungen!

- Unbeabsichtigte Wiederholung
  - Beispiel: class Line
  - Tip: Vermeide unnormalisierte Daten
- Ungeduld als Ursache von Wiederholung
  - “short cuts make for long delays”
- Wiederholung zwischen Entwicklern
  - Kommuniziere!
- Von außen aufgezwungene Wiederholung



2. Programmier - TECHNIKEN

Freie Universität Berlin

Vermeide Wiederholungen!

- **Problematisch**: Von außen aufgezwungene Wiederholung
  1. **Client-/ Server Architektur, wobei Client und Server in verschiedenen Sprachen geschrieben sind, aber Inhalte gemeinsam haben.**
  2. **Eine Klasse, deren Attribute ein Datenbankschema widerspiegeln**
  3. **C/ C++ Header Dateien**
  4. **Kommentare und Quellcode /Spezifikation und Quellcode**
- **Mögliche Lösungen:**
  1. u. 2. **Code automatisch generieren / Code filtern**
  3. **Verschiedene Ebenen:**
     - **High-Level Wissen in Kommentaren, Low-Level Wissen im Code**
     - **die Schnittstelle betreffendes in die *.h Datei, Implementierungsdetails in die *.cpp Datei**
  4. **Tests aus der Spezifikation generieren …**

Orthogonalität

- Eliminiere Auswirkungen zwischen unzusammenhängenden Dingen!
- Vermeide globale Variablen
- Vermeide ähnliche Funktionen
- Paradigma: schüchterner Code
- Kopplung (Coupling) minimieren, Bindung (Cohesion) maximieren
- “Law of Demeter”

Figure 5.1. Law of Demeter for functions

```
class Demeter {
private:
   A *a;
   int func();
public:
   //...
   void example(B& b);
}

void Demeter::example(B& b) {
   C  c;
   int f = func();          <-- itself

   b.invert();              <-- any parameters that were passed in to the method

   a = new A();
   a->setActive();          <-- any objects it created

   c.print();               <-- any directly held component objects
}
```

The Law of Demeter for functions states that any method of an object should call only methods belonging to:

Vorteile (LoD):

Weniger Fehler im Code (Studien hierzu: [BBM96])

Nachteile (LoD):

Evtl. mehrere Wrapperklassen notwendig, Performance-nachteile.

```
public void getPayment(Customer cust){
  Wallet walli = cust.getWallet();
  if (walli.getMoney(0.95)){
    myMoney+=0.95;
    return true;
  }else return false;
}

public boolean getPayment(Customer cust){
  if (cust.getMoney(0.95)){
    myMoney+=0.95;
    return true;
  }else return false;
}
```

Idee: Wenn ich einem Zeitungsverkäufer Geld für die Zeitung gebe, dann gebe ich ihm ja nicht mein Portemonnaie!

Nichts ist endgültig!

- Abstraktionen leben länger als Details!
  - Abstraktionen in den Quellcode
  - Details besser in **Metadata**

  (zwingt zu gutem Design)
- Text lebt länger als Binärdateien!

Design für Nebenläufigkeit!

- besseres Design!
- selbst Entscheidungen von Stand-alone auf n-tier Architektur umzusteigen keine Katastrophe

Gutes Design:
Der Java String Tokenizer

```
StringTokenizer st1 = new StringTokenizer(„Nä, wat wor dat früher“);
StringTokenizer st2 = new StringTokenizer(„en superjeile Zick“);

while(st1.hasMoreTokens() && st2.hasMoreTokens()){
        System.out.println(st1.nextToken());
        System.out.println(st2.nextToken());
}
```

Design by Contract! (Mach den Code schußsicher)

Bertrand Meyer: Programmiersprache Eiffel

- *Vorbedingung* (Precondition): Die Vorbedingungmuss erfüllt sein beim Aufruf der Routine.

- *Nachbedingung* (Postcondition): Der Status, nachdem die Routine durchlaufen wurde.

- *Invarianten*: Bedingungen, die während des Programms aus Sicht des Aufrufers stets erfüllt sind.

Wer sollte überprüfen, ob die Precondition erfüllt ist: **Aufrufer** oder **aufgerufene Methode**?

Hinweis zur Benutzung von Assertions/Exceptions

Designe für Tests!

- Tests automatisieren
- Einen Bug einmal finden und nie wieder
- Den Test testen...

1. Programmier – „PSYCHOLOGIE“

Keine eingeschlagenen Fenster!

- Das erste eingeschlagene Fenster: Ursache für den Verfall eines gesamten Hauses und das Entstehen von Ghettos! (Broken Window Theory [...70])
- Ein Fehler/ unsauberes Stück Code zieht weitere Fehler nach sich:
  - Weil sich eine “Egal-Einstellung” breit macht
  - Weil einer allein nicht alles reparieren kann
  - Weil eine kaputte Türe schwieriger auszutauschen ist, wenn auch der Türrahmen beschädigt ist.

3. Programmier – „PSYCHOLOGIE“

Think!

- Kritisch sein!
- Hat die Katze den Quellcode gegessen?
- Schuld suchen oder Probleme beheben / faule Ausreden oder Möglichkeiten suchen
- Ist “SELECT” kaputt?
- Das eigene Werk mit Stolz signieren!
- Sich nicht kochen lassen (sei kein Frosch)!
- Defensiv fahren.
- Schalte nie auf Autopilot. Think!

Vielen Dank!