13 Programming by Contract

Literatur: Bertrand Meyer, OO Software Construction, Kapitel 7., 11.1.

Motivation

- Integriere formale Konzepte in die Praxis
- Zusicherungen als Spezifikation und Dokumentation
- **Aufdecken** und **Eingrenzen** von Fehlern
- Diszipliniere den Entwurf

13.1 Zusicherungen allgemein

```
assert(Bedingung); // Fehlermeldung falls !Bedir
```

- Kennen wir schon.
- Jetzt mehrere Spezialfälle
- In einem korrekten Programm wird NIE eine assertion ausgelöst ($\neq$ Ausnahmefallbehandlung)

Was ausliefern?

Zunächst: Möglichst Assertions in abfangbare Fehler umwandeln.

Debuggingversion: Mit allen Assertions. Immer im Beta-Test.

Produktionsversion: NUR zeikritische Assertions abgeschaltet

Schlanke Version: Keine assertions. Nur bei sehr platzkritischen stabilen Codes.

13.2 Vor- und Nachbedingungen

In Eiffel:

```
methodenname(parameterliste) is
require
  Vorbedingung
do
  eigentlicher Code
ensure
  Nachbedingung
end;
```

Vor- und Nachbedingungen definieren einen **Kontrakt** zwischen dem Programmierer der den Aufruf schreibt und dem Programmierer der Funktion

13.2.1 Vorteile vollständiger Vor- und Nachbedingungen

- Doku: Nur Vor- und Nachbedingung lesen (bzw. deren **Dokumentation**)
- Vorbedingung falsch $\Rightarrow$ Fehler vorher suchen. Nicht an Symptomen herumflicken!
- Nur Nachbedingung falsch $\Rightarrow$ Routine oder Spezifikation unvollständig
- Keine Fehlermeldung: Code richtig oder Spezifikation unvollständing (Code muß zunächst nicht betrachtet werden)

13.3 Alte Werte

```
pop is // loesche oberstes Stackelement
require
  top >= 0;
do
  top--;
ensure
  top == old top - 1;
end;
```


13.4 Nochange

Beispiel: pop, das bei leerem Stapel nichts tut

```
pop is // loesche oberstes Stackelement
do
  top--;
ensure
  top >= 0 and top == old top - 1
  or top < 0 and Nochange;
end;
```

Nochange (fast) implizit bei Funktionen!

13.5 Schleifeninvarianten und Varianten

Eiffel kennt nur eine Schleife

```
ggtF(a,b:INTEGER):INTEGER is
require a>0; b>0;
local x,y:INTEGER;
do
  from
    x := a; y := b;
  invariant
    x > 0;  y > 0;
    ggt(x,y) = ggt(a,b) -- eigentlich Kommentar
  variant
    max(x,y);
  until
    x = y;
  loop
  if x > y then x := x - y;
  else          y := y - x;
end;
```


13.6 Klasseninvarianten

Aussagen über Eigenschaften der members.

Beispiel Stapel: `-1 <= top < maxSize`

- Implizite Nachbedingung von Konstruktoren
- Implizite Vor- und Nachbedingung aller member functions
- Dito friend-functions
- Dito direkte Zugriffe auf public members

Spart Schreibarbeit!

13.7 Sonstige Assertions

- Einmalig verschärfte Vorbedingung bei Funktionsaufruf. (Dokumentation, man sucht den Fehler znuächst nicht in der Spezifikation.)
- Replikation der Vorbedingung wenn nicht offensichtlich ist das die hier gilt
- Dito Nachbedingung. (Z.B. weil `new` keine ausreichnede Nachbedingung hat. (→ lieber Wrapper?)
- Vor- oder Nachbedingung für Statements. Elegante Form der Dokumentation sonst schwer verständlichen Codes.
- So plazieren, daß Fehler möglichst frühzeitig erkannt werden. (Ergänzung zu *Scaffolding*.)

13.8 Zusicherungen und Vererbung

Klasseninvarianten: Logisches **und** mit Invarianten der Basisklassen. Beispiel: `SortedFixedSizeArray : FixedSizeArray`

Vorbedingungen können **abgeschwächt** werden. (Vererbung nur bei Gleichheit sinnvoll.)

Nachbedingungen können **verstärkt** werden. (könnten also auch **und** mit Basisklassen bilden.)


```
class A { ...
void f(...){
  require(pre )...ensure(post ) }}

class B : A {
void f(...){
  require(pre')...ensure(post') }}
```

pre ⇒ pre' und post' ⇒ post

Subcontracting ?

Wenn das nicht klappt: Designfehler?

13.9 Probleme mit Assertions

- Quantoren? Es gibt Präprozessoren (endliche Grundmengen!)
- Rekursion (ggt, ggtF) ⇒ Assertions innerhalb von Assertions abschalten.
- Vollständige Spezifikation kann viel zusätzlichen Code erfordern. Trotzdem ...
- Postconditions manchmal schwierig zu prüfen. Z.B. wie Beweisen, daß eine Frequenzzuordnung optimal ist?

14 Einige Common Sense Regeln

Quelle: Writing Solid Code von McGuire.

14.1 Bessere Ausnutzung von Compilerwarnungen

- Höchste Warnstufe einstellen
- Optimierung einschalten! Warum?
- **ALLE** Warnungen beseitigen. Wie wenn wirklich kein Fehler?

  Unused Variable `v`**:** `(void)v;` Teil der Doku!

  Unsaubere implizite Typwandlung: Sauber casten.

  Unsauberer Code von anderen: Separat übersetzen (geht nicht bei Templates)

- `if () {} else {}`

14.2 Neue Fehler bei Änderungen/Fixes Vermeiden

- Never change a running system
- Keine Umformatierungen von Fremdcode
- Testsuite vor Freigabe jeder noch so kleinen Änderung laufen lassen

14.3 Undefiniertes Verhalten vermeiden

- `memcpy` vs. `memmove`
- Komplizierte Ausdrücke mit `++` etc. vermeiden
- Fremden (infizierten) Code mit Vor- und Nachbedingungen verpacken („Quarantäne“).

14.4 Defensives Programmieren

Code der nicht bei unerwarteten Eingaben nicht abstürzt

+ Manchmal klappts
– Fehler werden verschleppt
– Verschlechtert Lesbarkeit

- Besser Assertions plus Error-handling
- Radikale Testversion und defensive Produktversion
- Spezialfall Länge Null? Umstritten.

14.5 Portatibiltät

- `sizeof` statt Annahmen über Datentypgrößen
- Eigene Datentypen mit spezifizierter Anzahl Bits. `Int8, Int16, Int32, (Int64) UInt8,...` Immer bei 64 Bit Typen!
- Standardisierte Grundfunktionalität von Unix-Bibliotheken etc. bevorzugen
- Java o.ä. benutzen

14.6 Module „befestigen“

- Kleines Interface (Statdtore)
- Vor- und Nachbedingungen für Interface (Türme im Stadttor)
- Was werden Benutzer falsch machen? (sich in den Angreifer versetzen)
- Welche Fehler können besonders lange unbemerkt bleiben? (Spione) wie das verhindern (Farbklecks?)
- Wrapper um existierende Bibliothek (neue Statdmauer ohne Häuser abzureißen)
- Vorhersagbares Verhalten (...)

14.6.1 Beispiel Speicherverwaltung

- Ungerade Adressen
- In Debugversion freigegenen Speicher mit speziellem Code (nicht 0) überschreiben
  - Deckt späteren unerlaubten Zugriff auf
  - Soll Exception bei Dereferenzierung auslösen
  - Soll illegaler Befehl bei Ausführung sein

  Allgemein: Wenn mehrere im korrekten Fall gleichwertige Optionen verfügbar, dann denke nach was besser (z.B. sicherer) ist.
- Bei resize immer neu allozieren (z.B. Crash bei falschen Zeigern).
- In Debugversion Tabelle, die sagt, welche Zeiger (Blöcke) gültig sind. Allgemein: Zusatzliche Testdatenstrukturen. Bei wichtigen, fehlerträchtigen Subsystemen kann sich auch eine mehrfach komplexere Debugimplementierung bezahlt machen.

14.7 Unintuitive Interfaces

- Funktionen ohne außen sichbaren Seiteneffekte! (Meyer) (z.B. `T top(); void pop()`) enweder „was tun“ oder „was sagen“. Anderes Extrem: Design Antipattern „Blob“
- Boolsche Paremeter sind oft unintuitiv. Beispiel: `unsignedToStr(unsigned int, bool base16)`. Besser: `unsignedToStr(unsigned int, int base)` sogar mehr Funktionalität. (Vermutlich kostenlos)
- Fehlercodes vermeiden. (d.h. unnötig machen)

14.8 Risiken vermeiden

Beispiel: Wie Steilwand herabkommen. Seil oder Drachenflieger? Programmiererantwort ? ...

- Vorsicht mit `char`-Ausdrücken
- Ausnahmefallbehandlungen durch orthogonalen Code vermeiden. Beispiel: Wurzelfenster nicht speziell behandeln. Jedes Statement weniger ist eine Fehlermöglichkeit weniger (keine Lizenz zum hacken.) Ifs meiden wie die Pest.
- Ist tunen wirklich nötig?

14.9 Typische „Sünden“

- Andererleut Daten zugreifen
- Gerade freigegebenen Speicher zugreifen (hatten wir schon?)
- Schreiben wenn man nur lesen sollte?
- Zeiger auf lokale Variablen zurückgeben
- Ausgabespeicher als Zwischenspeicher verwenden (vor allem wenn zu viel davon.)
- Ausgabe von Zeigern auf `static`-Speicher
- Ausnutzung nichtzugesicherter Eigenschaften aufgerufener Funktionen
- Mit dem Ergebnis von Vergleichen rumrechnen. (Geht schief wenn andere boolsche Werte auftauchen)
- Verwechslung von Quell- und Objektcodelänge. (`?:`-Operator) (APL-Syndrom) (C-Syndrom?)
- Tricks aus reiner Eitelkeit anwenden

14.10 Manches ist Einstellungssache

- Wenn ein Fehler einfach verschwindet. Nicht nachlassen. Nicht an Symptomen herumdoktorn.
- Nie auf Testphase verlassen
- Blind featurism, second system effect, gold plating, cool stuff
- There are no free lunches (Doku ...)
- Flexibilität brütet Fehler aus. (Bei Sonderfallbehandlungen zahlt man die Zeche)
- Portierter Code ist neuer Code
- Nicht „probieren“, lesen
- Testzeiten nicht dem Zeitplan opfern