Programmierkonzepte in der Physikalischen Chemie

5. Robuster und schneller Code

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Dr. Till Biskup
Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2013/14

Übersicht

Robuster Code

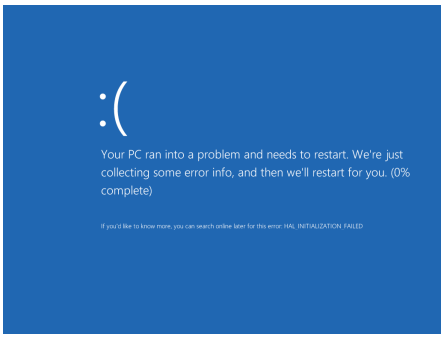
- Definierte Ausgangslage
- Definierter Rückgabestatus
- Fehler abfangen
- Programmierkonzepte

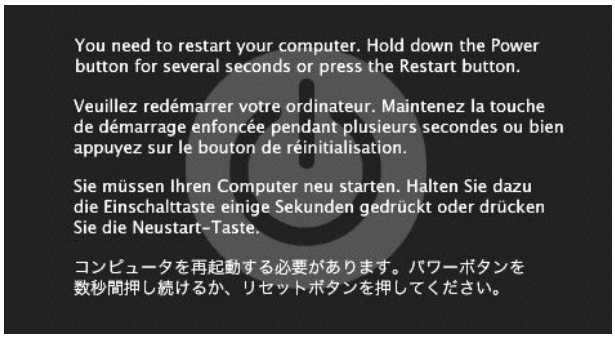
Schneller Code

- Optimierungsstrategien
- Funktionen zur Zeitmessung
- Besonderheiten von Matlab
- Wrapper in Matlab

Robuster Code

Notwendige Voraussetzung für nutzbare Programme





Listing 1: linux-2.0.38/arch/m68k/atari/atafb.c

```
1 /* Nobody will ever see this message :-) */
2 panic("Cannot initialize video hardware\n");
3
4 linux-2.0.38/arch/m68k/atari/atafb.c
```

Robuster Code (Anwendersicht)

- ist tolerant gegenüber (falschen) Nutzereingaben,
- sorgt immer für eindeutige Zustände,
- gibt klare und verständliche Fehlermeldungen zurück,
- sorgt im Fehlerfall für eine „weiche Landung“.

Robuster Code (Entwicklersicht)

- ist übersichtlich geschrieben,
- hat eine hohe Testabdeckung,
- lässt sich einfach weiterentwickeln,
- ist modular und gut weiterverwendbar.

Robuster Code (Anwendersicht)

```
Error using resfields (line 102)
D strain and g/A strain cannot be used at the same time.

Error in
/Users/till/Programme/Matlab/Toolboxen/easyspin-4.5.1/easyspin/pepper.p>pepper
(line 514)

Error in
/Users/till/Programme/Matlab/Toolboxen/easyspin-4.5.1/easyspin/pepper.p>pepper
(line 105)
```

- Nicht abgefangener Fehler
  - Bricht in Matlab die weitere Abarbeitung sofort ab.
  - Kein definierter Zustand (kann zu Instabilitäten führen)

☛ Strategien zum Umgang mit Fehlern

Robuster Code

Was heißt das für den Entwickler?

- Definierte Ausgangslage
  - Variablen am Anfang definieren/initialisieren
  - Nutzereingaben überprüfen
- Definierter Rückgabestatus
  - „graceful exit“
  - klare Fehlermeldungen
  - einfach abfragbarer Status
- Fehler abfangen
  - Nutzereingaben überprüfen (inputParser)
  - Fehlerbehandlung mit `try-catch`
- Programmierkonzepte
  - Unit Tests
  - Refactoring
  - Test Driven Development

Kontext der Ausführung beachten

- Skript
  - Alle im „Workspace“ definierten Variablen verfügbar.
  - Kann zu unberechenbaren Abhängigkeiten führen.
- Funktion
  - „Tabula rasa“
  - Nur übergebene oder intern definierte Variablen verfügbar.

Grundregeln

- Variablen vor der Benutzung initialisieren.
  - Möglichst an einer zentralen Stelle früh im Quellcode.
- Nutzereingaben (streng) überprüfen.

Grundregel

Traue niemals den Eingaben eines Nutzers.

- Jeden Eingabeparameter auf Existenz und Typ überprüfen, bevor auf ihn zugegriffen wird.

Listing 2: Eingabe initialisieren

```
1  function inputCheck(string)
2  % INPUTCHECK Demonstrate how to check input variables.
3
4  % Initialise input
5  if ~nargin || isempty(string) || ~ischar(string)
6    string = 'Hello world';
7  end
8
9  % And here comes your code
10 disp(string);
```

Robuster Code

Definierte Ausgangslage: Variablen initialisieren

UNI FREIBURG

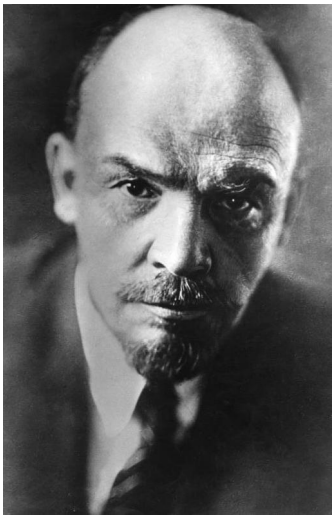
Listing 3: Rückgabe initialisieren

```
1  function [out] = outputCheck(in)
2  % OUTPUTCHECK Demonstrate how to initialise return variables.
3
4  % Initialise output
5  out = [];
6
7  % And here comes your code
8  if ~nargin
9    return;
10 end
```

- Ruft man eine Funktion ohne Rückgabeparameter auf, gibt es keinen Fehler, wenn sie nicht initialisiert wurden.
- Rückgabeparameter sollten im Code initialisiert werden, bevor ein (beliebiger) Fehler auftreten kann.
  - *vor* dem Parsen der Eingabeparameter

Vertrauen ist gut, Kontrolle ist besser!

Lenin (zugeschrieben)



Was sollte ggf. überprüft werden?

- Existenz
- Typ
- Größe
- Gültigkeit des Wertebereiches

Zwei Möglichkeiten (in Matlab)

1. manuelle Überprüfung
2. inputParser (kommt später)

Existenz

- Der Zugriff auf eine nicht deklarierte Variable bewirkt (u.a. in Matlab) einen Fehler während der Ausführung.

  Undefined function or variable '<name>'.

Überprüfung (in Matlab)

exist allgemeine Überprüfung der Existenz von Variablen (und Dateien, Funktionen, Klassen).
- Liefert den Typ (als Skalar) zurück
- Optional: Überprüfung auf nur einen Typ

isfield Existenz eines Feldes in einem `struct`-Array
- Nur für die aktuelle Ebene des Arrays

Typ

- Ist eine Variable nicht vom erwarteten Typ (string, integer, float, cell, struct, ...), kann es ernsthafte Probleme geben.
- Aber: Matlab ist eine schwach typisierte Sprache.
  $\Rightarrow$ Viele Typen werden implizit ineinander umgewandelt.

Überprüfung (in Matlab)

isa Überprüfung des Typs einer Variable
- Erlaubt die Überprüfung einer Objektklasse (OOP)

is* Serie von Funktionen zur Typ-Überprüfung
- auch „Metabefehle“: `isnumeric`, `isreal`, ...
- Vorsicht bei `isscalar`, `isvector`, `ismatrix`

Größe (Dimension)

- Die Größe (Dimension) ist oft wesentlich für die weitere Verarbeitung (Indizierungsfehler, ...).

```
Error using <operator>
Matrix dimensions must agree.
```

Überprüfung (in Matlab)

length Länge der größten Dimension
- Antwort immer ein Skalar
- Nie bei mehrdimensionalen Objekten verwenden!

size Dimensionen eines Arrays
- Liefert Größe jeder Dimension

Wertebereich

- Oft ist der (sinnvolle) Wertebereich einer (numerischen) Variablen beschränkt.
- Lässt sich nur im gegebenen Kontext überprüfen.

Überprüfung (in Matlab)

- Allgemein nur über explizite Abfragen (`if...else`) möglich
- Beispiel (numerisch):
```
if ~( (x>=0) && (x<=1) )
```
- Beispiel (Strings):
```
if ~strcmpi(needle,haystack)
```

Robuster Code

Definierter Rückgabestatus: „graceful exit“

UNI FREIBURG

- Eine Funktion sollte wann immer möglich beim Beenden einen definierten Zustand hinterlassen.
- Es gibt Situationen, in denen die weitere Abarbeitung einer Funktion nicht sinnvoll ist.

Abarbeitung beenden (Matlab)

error Zeigt eine Fehlermeldung und beendet sofort
- Besser als keine vernünftige Fehlerbehandlung.
- Kehrt auch aus Unterfunktionen sofort auf die Matlab-Kommandozeile zurück ($\Rightarrow$ unschön).

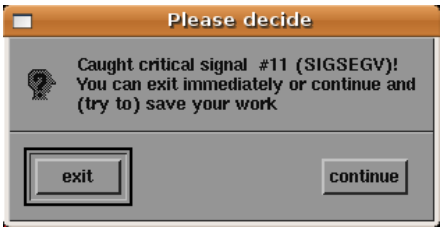
return Gibt Kontrolle an aufrufende Instanz zurück
- Fehlermeldung ausgeben (via `fprintf`, `disp`)
- Fehlerbehandlung in aufrufender Instanz

Aussagekräftige Fehlermeldungen

„Error -2051 is a bit of a 'catch-all' error that is returned if nothing more descriptive is available.“

Hilfreiche Fehlermeldungen

Wenn die Messung seit über 24 Stunden lief und noch nicht gespeichert wurde...



Arten von Fehlern

- Warnungen
- Fehler, die die normale Abarbeitung verhindern
- Fehler, die eine Weiterarbeit unmöglich machen

Kernaspekte einer Fehlermeldung

- Funktionsname der Funktion, in der der Fehler auftritt
- Kurze Beschreibung des Problems
- Schwere des Problems (Warnung, Fehler)

error

Listing 4: error-Funktion in Matlab

```
1 error('msgString')
```

- Zeigt `<msgString>` an und beendet die Funktion.
- Kann einen Identifizierungsstring mitbekommen.
- Sammelt Informationen in einem `MException`-Objekt.
- Oft nur sinnvoll mit `try-catch` verwendbar.
  - Wird später behandelt.

warning

Listing 5: warning-Funktion in Matlab

```
1 warning('msgString')
```

- Zeigt `<msgString>` an und fährt fort.
- Kann einen Identifizierungsstring mitbekommen.
  - Kann verwendet werden, um gezielt Warnungen ein-/auszuschalten

Alternative: Manuelle Nachrichten

- Größere Flexibilität
- Mehr Eigenverantwortung

Beispiel: `if-else`-Statement zum Abfangen von Fehlern

Listing 6: if-else zum Abfangen von Fehlern

```
1 if <condition>
2   % This is how it should be.
3 else
4   disp('Some error occurred...');
5 end
```

Tipp: Häufig reicht eine `if`-Abfrage mit invertierter Logik:

Listing 7: if mit invertierter Bedingung zum Abfangen von Fehlern

```
1 if ~<condition>
2   disp('Some error occurred...');
3   return;
4 end
5 % This is how it should be.
```

Beispiel: `switch`-Statement mit `otherwise`-Zweig

Listing 8: otherwise-Zweig zum Abfangen von Fehlern

```
1 switch <expression>
2   case '<something>'
3     % This is taken care of
4   case '<something else>'
5     % This is taken care of as well
6   otherwise
7     fprintf('Unknown expression "%s" in %s...',<expression>,mfilename);
8 end
```

- Möglichst `otherwise`-Zweig in `switch`-Statements
- Ausdruck sinnvollerweise in Fehlermeldung ausgeben
- Name der Funktion über `mfilename`
  - Alternative: `dbstack`
  - Funktioniert jeweils nur innerhalb von Funktionen.

Konvention in vielen Betriebssystemen

0 alles in Ordnung

$\neq 0$ Fehler

Situation in Matlab

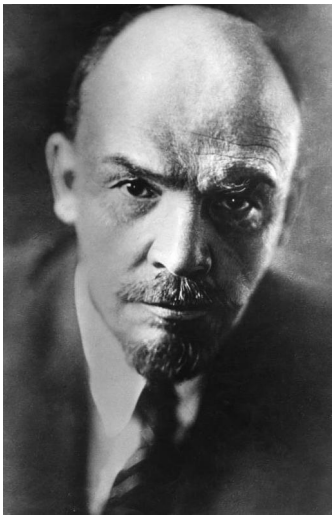
- Kein Status des letzten aufgerufenen Befehls verfügbar.
- Programmierer ist selbst dafür verantwortlich.

Möglichkeiten

- Status (Skalar) als Rückgabeparameter von Funktionen
- Leerer Rückgabeparameter bei Problemen

Vertrauen ist gut, Kontrolle ist besser!

Lenin (zugeschrieben)



Was sollte ggf. überprüft werden?

- Existenz
- Typ
- Größe
- Gültigkeit des Wertebereiches

Zwei Möglichkeiten (in Matlab)

1. manuelle Überprüfung
2. inputParser

UNI FREIBURG

inputParser

Matlab-Klasse, die den Typ und den Wert jedes Eingabeparameters einer Funktion überprüft.

- Konzepte der objektorientierten Programmierung (OOP)
  1. Objekt erzeugen (instanziieren)
  2. Eigenschaften des Objektes setzen
  3. Methoden des Objektes zum Parsen aufrufen
  4. Ergebnisse auswerten
- Beherrscht drei Arten von Parametern
  1. obligatorische Parameter
  2. optionale Parameter
  3. (optionale) Schlüssel-Wert-Paare

Listing 9: inputParser – ein reales Beispiel

```
1  function varargout = trEPRload(filename, varargin)
2  % TREPRLOAD Load files or scan whole directories for readable files
3  % ...
4  p = inputParser;              % Create an instance of the inputParser class.
5  p.FunctionName  = mfilename; % Include function name in error messages.
6  p.KeepUnmatched = true;       % Enable errors on unmatched arguments.
7  p.StructExpand  = true;       % Enable passing arguments in a structure.
8
9  p.addRequired('filename', @(x)ischar(x) || iscell(x) || isstruct(x));
10 p.addParamValue('combine',logical(false),@islogical);
11 p.addParamValue('loadInfoFile',logical(false),@islogical);
12 p.parse(filename,varargin{:});
13
14 % ...
15 if p.Results.combine
```

- Validierung hier u.a. über „anonyme Funktionen“
- Fehler beim Parsen führen zur sofortigen Beendigung.
  - Lösung: `try-catch`

Tipps zur Arbeit mit inputParser

- Rückgabeparameter zuerst definieren
  - Saubere Terminierung, wenn beim Parsen Fehler auftreten
- Eigentliche Überprüfung in ein `try-catch` verpacken
  - Ein Fehler im Parser führt zu einer Ausnahme (*Exception*), die sonst unmittelbar zum Abbruch führt.

Grenzen von inputParser

- Keine beliebige Reihenfolge optionaler Parameter
  - Optionale Parameter müssen beim Funktionsaufruf vor den Schlüssel-Wert-Paaren eingegeben werden.
  - Mehrere optionale Parametern werden in der eingegebenen Reihenfolge geparst.

Hilfe ausgeben bei fehlenden Eingabeparametern

Listing 10: Hilfe ausgeben bei fehlenden Eingabeparamtern

```
1 if ~nargin
2   help <functionname>
3   return;
4 end
```

- Kann bei konsequentem Einsatz nutzerfreundlich sein
- Nicht in jedem Fall umsetzbar
  - Es gibt Funktionen ohne Eingabeparameter.
- Initialisierung eventueller Rückgabeparameter essentiell
  - Führt ansonsten zu Fehlern

Listing 11: try-catch

```
1 try
2   ...
3 catch exception
4   ...
5 end
```

- Sollte jeden *kritischen* Funktionsaufruf einklammern.
- Erlaubt das Abfangen von Fehlern.
  - Verhindert das „Platzen“ von Programmen wegen Fehlern.
  - Erlaubt das definierte Beenden („graceful exit“).
- `exception` ist ein Objekt der Klasse `MException`
  - Enthält alle verfügbaren Informationen zum Fehler.
  - Kompletter „call stack“ (Hierarchie der Funktionsaufrufe)

Robuster Code *vs.* Geschwindigkeit

- Abfragen kosten (Rechen-)Zeit
  - In zeitkritischen Systemen nicht immer möglich
- Interne Funktionen kommen oft ohne Überprüfung aus.
  - Funktionen, die sich *nicht* vom Nutzer aufrufen lassen
  - Kein Grund, Funktionen unnnötig zu kapseln (DRY!)
  - Interne Funktionen in `private`-Verzeichnis auslagern

☛ Letztlich von Fall zu Fall entscheiden.

☛ Sauberes Schnittstellenkonzept mit guter Dokumentation und einer möglichst vollständigen Testabdeckung.

Warum Programmierkonzepte für robusten Code?

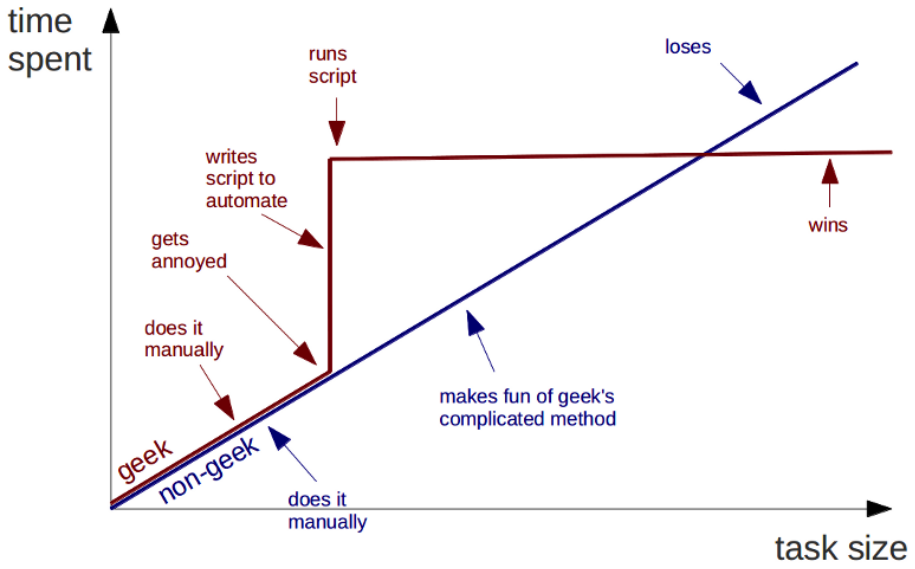
- Robuster Code ist wichtig für Entwickler und Anwender.
- Code wird robuster, wenn man gewissen Prinzipien folgt.
- Robuster Code erfordert Disziplin der Programmierer.

Programmierkonzepte – drei Beispiele

- Unit Tests
- Refactoring
- Test Driven Development

☛ Es gibt noch viel mehr Konzepte.

Geeks and repetitive tasks



http://imgur.com/Q8kV8

Unit Tests (Modultests)

Überprüfung der funktionalen Einzelteile (Module, Funktionen) eines Programms auf korrekte Funktionalität.

- Test des Verhaltens von Funktionen „von außen“.
- Vergleich der Rückgabe oder des Verhaltens einer Funktion mit dem Erwartungswert.
- Normalerweise durch Test-Frameworks realisiert (und automatisiert).
- Voraussetzung für andere Programmierkonzepte (Refactoring, Test Driven Development)

Vorteile

- Testdurchführung ist ein automatisiertes Verfahren.
- Hohe Testabdeckung ermöglicht sicheres Refactoring.
- Regelmäßige Tests erhöhen die Zuverlässigkeit.
- Tests können Aufgaben einer Funktion klarer machen.

Nachteile

- Am Anfang steile Lernkurve
- Tests müssen formuliert und geschrieben werden.
- 100% Testabdeckung nie erreichbar
- Komplexe Nutzerschnittstellen (CLI, GUI) schwer testbar

Unit Tests und Matlab

- Ab Matlab 2013a
  - Unit-Test-Framework integriert
  - Folgt im Wesentlichen den Konzepten von xUnit
- Ab Matlab 2008a
  - MATLAB xUnit Test Framework (File Exchange)
  - Wird nicht mehr weiterentwickelt (s.o.)
- Vor Matlab 2008a
  - keine oder rudimentäre OOP-Unterstützung
  - Test-Frameworks quasi unmöglich implementierbar

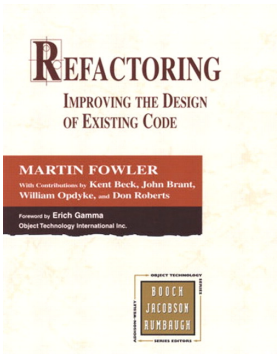
☛ xUnit kommt aus der Welt der Objektorientierung (OOP).

UNI FREIBURG

Refactoring (Refaktorierung, Restrukturierung)

Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens

- Grundprinzip
  - Immer nur kleine Änderungen
  - Funktionalität bleibt erhalten
  - Keine neuen Eigenschaften
- Voraussetzungen
  - Tests mit ausreichender Abdeckung
  - Versionsverwaltung



Improving the design [of code] after it has been written.

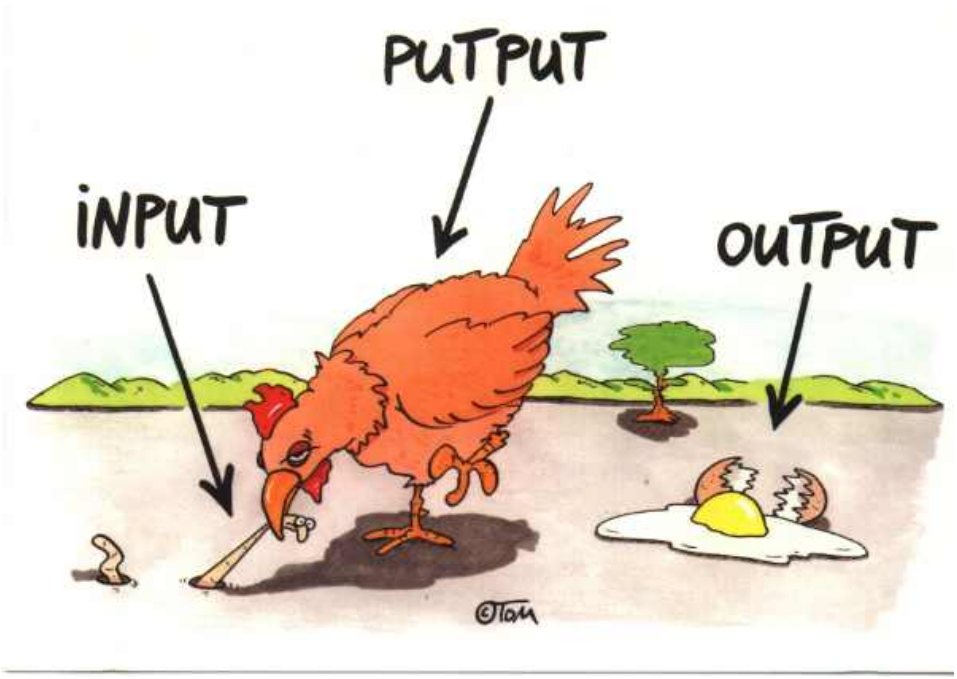
Ziel: Vereinfachung von Code

- besser verständlich
- leichter (und billiger) zu modifizieren und zu erweitern

Grundsatz: Keine Änderung am Programmverhalten

- Keine neuen Eigenschaften implementieren.
  - Auch nicht, wenn im bestehenden Design Fehler sind!
- Keine anderen Refactorings nebenher durchführen.
- Nach jedem Schritt Tests laufen lassen.

M. Fowler: Refactoring, Addison-Wesley 1999



Thomas Körner, alias ©TOM

Wann sollte man Code restrukturieren?

- Wenn neue Eigenschaften implementiert werden
  - Vor der Implementierung der neuen Eigenschaften
  - Zur Verbesserung der Lesbarkeit/Verständlichkeit
- Wenn Fehler behoben werden müssen
  - Zur Verbesserung der Lesbarkeit/Verständlichkeit
  - Führt oft zum schnelleren Auffinden des Fehlers.
- Wenn der Quellcode überprüft wird (Code Review)
  - Szenario: fremden Code übernehmen
  - Szenario: Code einer anderen Person erklären

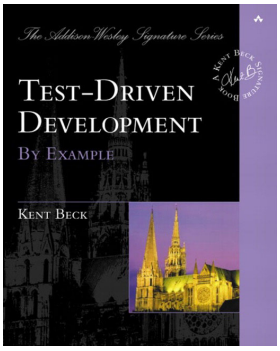
☛ Zeit für das Restrukturieren (Refactoring) fest einplanen

UNI FREIBURG

Testgetriebene Entwicklung

engl. *test-driven development*, konsequente Entwicklung von Tests *vor* den zu testenden Komponenten

- Zwei Prinzipien:
  1. Never write a single line of code unless you have a failing automated test.
  2. Eliminate duplication.
- Voraussetzungen
  - Tests mit ausreichender Abdeckung
  - Versionsverwaltung



Das TDD-Mantra: Drei Phasen der Entwicklung

1. rot
   - Schreibe einen kleinen Test, der nicht funktioniert, vielleicht noch nicht einmal kompiliert.
2. grün
   - Sorge dafür, dass der Test schnell durchläuft, egal welche Programmierersünden dafür notwendig sind.
3. Restrukturierung (Refactoring)
   - Eliminiere jede Form von Duplizierung, die eingeführt wurde, um den Test zu bestehen.

☛ Kurze Zyklen, am besten weniger als eine Stunde

Warum sollte man so etwas tun?

- Die meisten Programmieraufgaben sind komplex.
- Oft ist nicht klar, wie man ans Ziel gelangt.
- Kleine Schritte sorgen für die nötige (Selbst-)Sicherheit.

Deshalb

- *Kleine* Schritte – viel kleiner, als man sich vorstellt
- Automatisierte Tests
- Jeweils *nur eine* Designentscheidung hinzufügen

☛ Alles Weitere aufschreiben und später abarbeiten

Robuster Code

Programmierkonzepte: Vor- und Nachteile

Vorteile

- Robuster Code, leicht verständlich, einfach zu erweitern
- Verhindert Degenerierung von Code über die Zeit
- Zahlt sich auf Dauer aus (Zeit- und Geldersparnis)

Nachteile

- Zeitaufwendig bei Anwendung auf bestehende Projekte
- Erfordert Umdenken und Disziplin vom Programmierer

- ☛ Für große Projekte führt (fast) kein Weg daran vorbei.
- ☛ Konzepte werden oft unbewusst/unbenannt eingesetzt.

Robuster Code

Am Ende hilft nur noch: Testen, testen, testen

- Manchmal hilft nur noch das Testen in realer Umgebung (z.B. CLI, GUI).
- Hier hilft eine gute Infrastruktur (u.a. Versions- & Bugverwaltung).

Bananensoftware

Software, die vor der Weitergabe an den Nutzer nicht ausreichend getestet wurde und beim Nutzer „nachreift“.

- Gilt im strengen Sinne nur für kommerzielle Software, da freien Programmierern oft die Kapazität fehlt.

Schneller Code

Notwendige Voraussetzung für aufwendige Rechnungen

Warum ist schneller Code wichtig?

- Simulationen
  - Viele kleine Schritte, die sehr oft wiederholt werden
  - Beispiel: Pulvermittelung eines Spektrums
- Verarbeitung großer Datenmengen
  - Mehrdimensionale Datensätze
  - Viele Datensätze parallel
- (graphische) Nutzerschnittstellen
  - Nutzer erwarten eine zügige Reaktion des Programms.
  - Matlab-GUIs sind *per se* nicht die schnellsten...

Voraussetzungen für die Beschleunigung von Code

- Kenntnis der jeweiligen Programmiersprache
  - Jede Sprache hat ihre Eigenheiten.
  - Optimierungsstrategien sind nicht immer offensichtlich.
- Werkzeuge zur Zeitmessung
  - Einzelne Codeblöcke messen
  - Langsame Codeblöcke identifizieren
  - Potential für die größten Einsparungen lokalisieren
- Code, der ausreichend oft wiederverwendet wird
  - Optimierung ist nur sinnvoll, wenn sie sich auszahlt.

Optimierungsstrategien

1. Häufig aufgerufene Codeteile optimieren
   - Liefert meist die größte Zeitersparnis
   - Beispiele
     - Immer gleiche Berechnungen aus Schleifen herausziehen
     - Variablengrößen nicht in Schleifen modifizieren
2. Langsame Teile optimieren
3. Auslagerung zeitkritischer Funktionen
   - Maschinennahe Programmiersprachen (C, C++, Fortran)
   - Wrapper (kommt später)

☛ Voraussetzung: Funktionen zur Zeitmessung

Matlab: Eingebaute „Stoppuhr“

Listing 12: Matlab-Funktionen zur Zeitmessung

```
1 tic;
2 % Some code
3 toc
```

- Zeitangabe auf eine Mikrosekunde genau

Listing 13: Verschachtelte Zeitmessung

```
1 timer1 = tic;
2 % Some code
3 timer2 = tic;
4 % Some code
5 elapsedTime2 = toc(timer2);
6 % Some code
7 elapsedTime1 = toc(timer1);
```

Wichtige Details von Matlab

- Java-basiert und interpretiert
  - \+ (Halbwegs) plattformunabhängig
  - \+ Hervorragend geeignet für „Prototyping“
  - \+ Sehr flexibel und (verhältnismäßig) anwenderfreundlich
  - – Langsam
  - – Programmierung mitunter kompliziert
- Optimiert für Matrix-Operationen, nicht für Schleifen
- Ausgabe (Kommandozeile/Plotten) langsam

☛ Strategien zur Code-Beschleunigung

Speicher für Variablen vorreservieren („preallocate“)

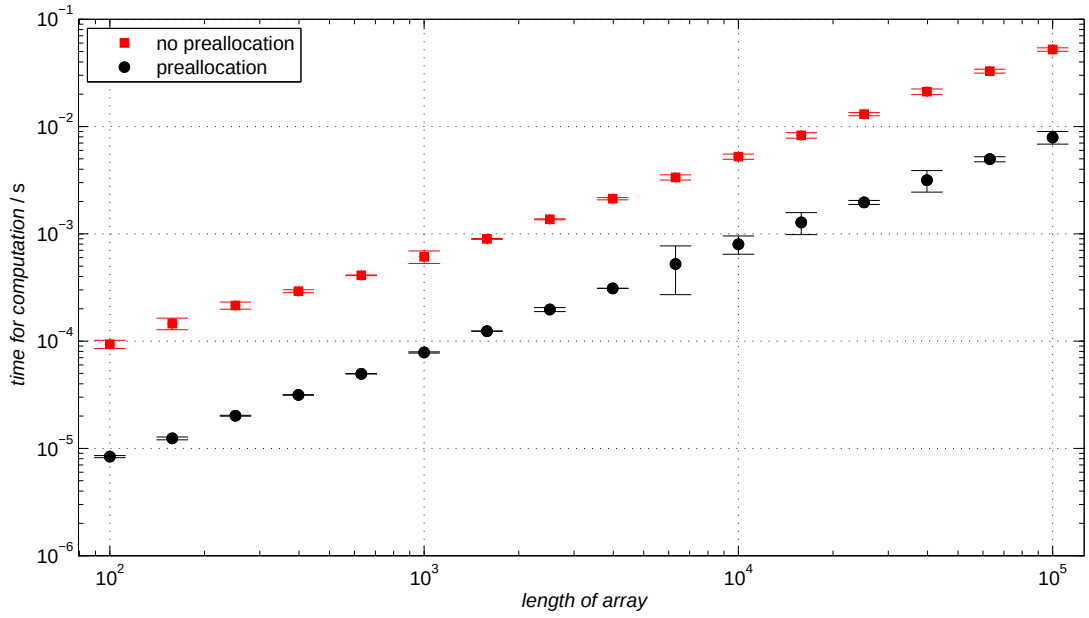
Listing 14: Langsamer Code

```
1 x = 1:0.1:4*pi;
2 for k=1:length(x)
3   % y changes size in every loop iteration
4   y(k) = sin(x(k));
5 end
```

Listing 15: Schneller Code

```
1 x = 1:0.1:4*pi;
2 y = zeros(length(x),1);
3 for k=1:length(x)
4   % y preallocated, doesn't change size
5   y(k) = sin(x(k));
6 end
```

Speicher für Variablen vorreservieren („preallocate“)



Leere Elemente löschen

- Speicher vorreservieren und leere Elemente löschen
- Vektoren

Listing 16: Leere Elemente aus Vektoren löschen

```
1 V = V(V>0);
```

- Cell Arrays

Listing 17: Leere Elemente aus Cell Arrays löschen

```
1 C = C(~cellfun('isempty',C));
```

Array-Funktionen statt Schleifen

Listing 18: Langsamer Code mit Schleife

```
1 for idx = length(cellArray):-1:1
2     if isempty(cellArray{idx})
3         cellArray(idx) = [];
4     end
5 end
```

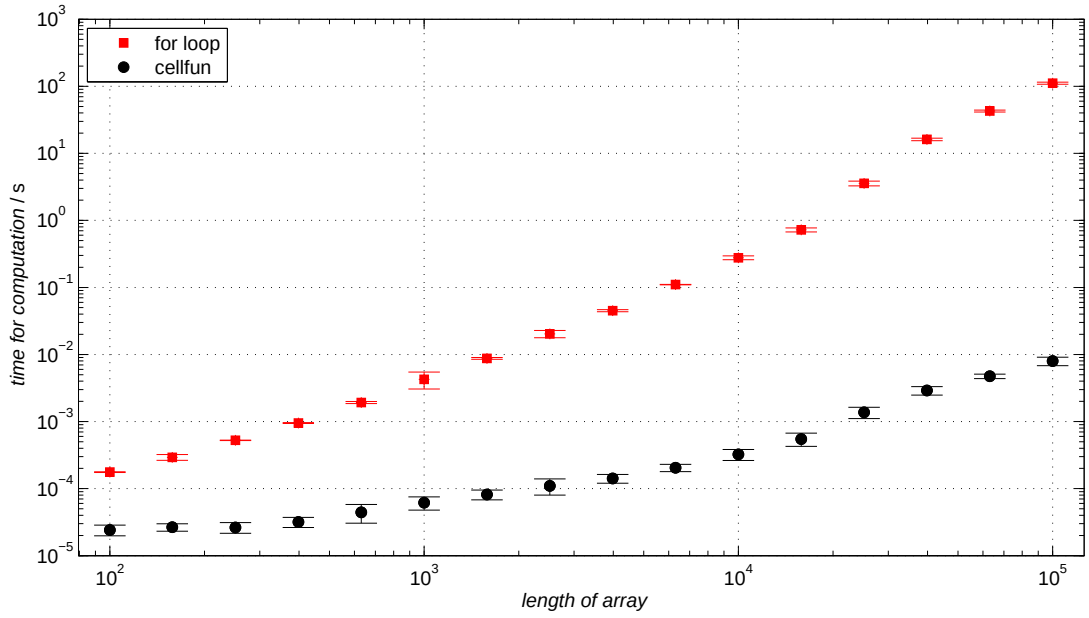
- Leere Elemente aus dem `cell array` entfernen

Listing 19: Schneller Code mit cellfun

```
1 cellArray = cellArray(~cellfun('isempty',cellArray));
```

- Übersichtlicherer und *deutlich* schnellerer Code

Array-Funktionen statt Schleifen



Matrix-Operationen statt Schleifen (Vektorisierung)

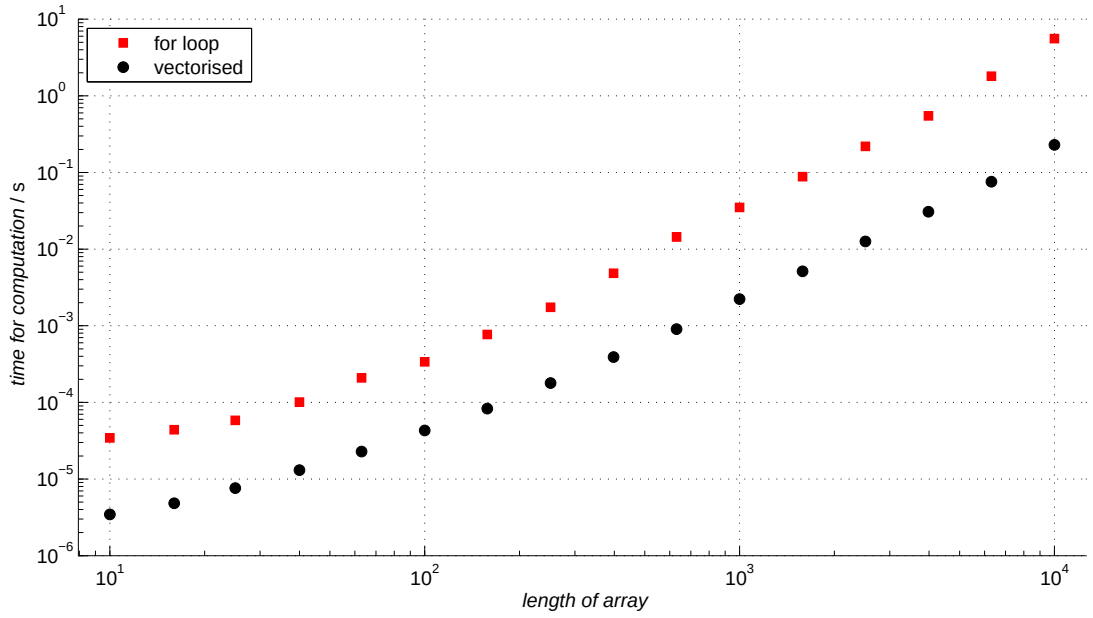
Listing 20: Langsamer Code mit Schleifen

```
1 x = -2:0.1:2;
2 y = -1.5:0.1:1.5;
3 F = zeros(length(x),length(y));
4 for xi = 1:length(x)
5     for yi = 1:length(y)
6         F(xi,yi) = x(xi)*exp(-x(xi)^2-y(yi)^2);
7     end
8 end
```

Listing 21: Schneller Code mit meshgrid und Matrix-Operationen

```
1 x = -2:0.1:2;
2 y = -1.5:0.1:1.5;
3 [X,Y] = meshgrid(x,y);
4 F = X.*exp(-X.^2-Y.^2);
```

Matrix-Operationen statt Schleifen (Vektorisierung)



Weitere Strategien (MATLAB)

- Immer gleiche Berechnungen aus Schleifen auslagern
  - Beispiel: $\sin(\theta)$ in ϕ-Schleife
- Typen existierender Variablen nicht ändern
  - Neue Variable anlegen
- Logische Operatoren
  - „&&“ und „||“ statt „&“ und „|“
  - „&&“ und „||“ brechen bei erster nicht erfüllter Bedingung ab
- Funktionen statt Skripte
  - Funktionen generell schneller
  - Weitere Vorteile: Modularität, Übersichtlichkeit

UNI FREIBURG

Wrapper (Adapter)

Software, die ein anderes Stück Software umgibt

- Schnittstelle zu (bestehenden) Funktionen
- Zusätzliche Abstraktionsschicht

Vorteile

- Verringerung von Querabhängigkeiten
- Einfacherer, lesbarerer, leichter zu pflegender Code

Beispiel

- Zugriff auf Funktionen in einer anderen Sprache

Wrapper in Matlab

- Matlab greift intern auf BLAS und LAPACK zurück.

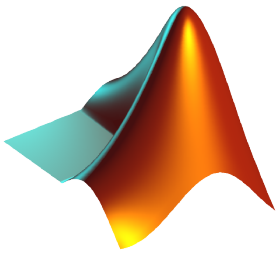
  BLAS Basic Linear Algebra Subprograms
  LAPACK Linear Algebra PACKage

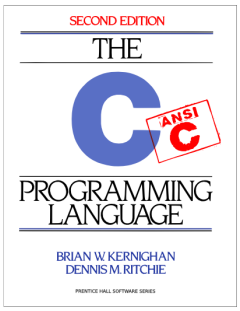
  - Hocheffiziente Bibliotheken für lineare Algebra
  - Geschrieben in Fortran 90
  - Nutzen moderne Prozessorarchitekturen aus
  - Frei verfügbar, für eigene Programme verwendbar

- Viele Matlab-Befehle sind strenggenommen „Wrapper“.
- Numerisch ist Matlab nicht besser als BLAS und LAPACK.

Warum Wrapper selber schreiben?

- Zugriff auf (existierende) Fortran/C/C++-Routinen
  - Beispiel: Simulationsroutinen für Spektren





- Fortran/C/C++ für numerische Probleme sehr schnell
  - Beispiel: EasySpin – viele numerische Routinen in C

Wrapper in der Praxis

- Funktion in Fortran/C/C++
- Schnittstelle zwischen Matlab und dem Programm
  - Konvertiert Datentypen und Ein-/Ausgabe
  - Lädt Bibliothek mit Hilfsfunktionen zum Konvertieren (Matlab-Bibliothek)

Mögliche Stolpersteine

- Systemarchitektur (32 bit vs. 64 bit)
- Speicherhandhabung

☛ Wrapper trennen von den eigentlichen Programmen (maximale Unabhängigkeit von Matlab)

Optimierungsstrategien – Kosten und Nutzen abwägen

- Einfach und immer möglich
  - Variablen vorher in korrekter/maximaler Größe definieren
  - Lange Skripte in einzelne Funktionen aufteilen
- Möglich mit detaillierten Kenntnissen von Matlab
  - Vektorisierung von Schleifen
  - Ausnutzung der Array-Manipulationen von Matlab
- Möglich mit Kenntnissen in Fortran/C/C++
  - Rechenintensive Funktionen maschinennah programmieren
  - Aus Matlab via Wrapper aufrufen

☛ Hängt u.a. von den eigenen Fähigkeiten
(und den verfügbaren Resourcen) ab

Ausblick

Das war's. Wie geht's weiter?

So long, and thanks for all the fish.

Vorschau: Datenverarbeitung und Nutzerschnittstellen

- Datenverarbeitung
- Nutzerschnittstellen

Douglas Adams