Programmierkonzepte in der Physikalischen Chemie

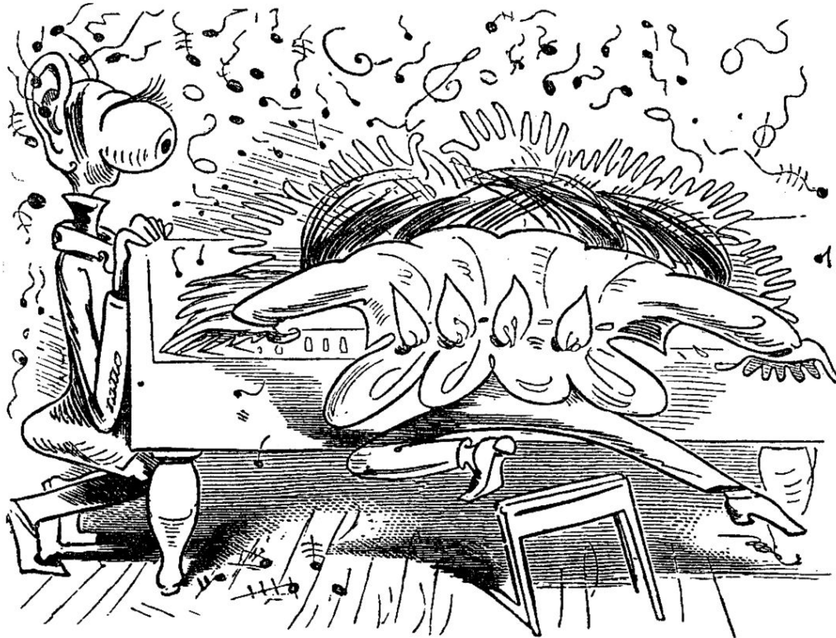
9. Programmentwicklung

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

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

UNI FREIBURG



Wilhelm Busch: Der Virtuos (1865)

Übersicht

Wohin soll die Reise gehen?

Infrastruktur

Konzepte

Datenverarbeitung

Nutzerschnittstellen

Projektplanung

Wohin soll die Reise gehen?

Wer hohe Türme bauen will...

UNI FREIBURG

Pieter Brueghel d.Ä. (1563)

Wohin soll die Reise gehen?

Klarheit über Ziele, Ressourcen, Anforderungen

UNI FREIBURG

Wer hohe Türme bauen will,
muss lange beim Fundament verweilen.

- Zielstellung
  - Was soll am Ende herauskommen?
- Anforderungen
  - Welche Aufgaben müssen bearbeitet werden?
  - Wer soll mit dem Programm arbeiten?
  - Einmaliger Vorgang oder (künftige) Routineaufgabe?
- Ressourcen
  - Vorwissen und Vorarbeiten
  - Verfügbarer Zeitrahmen
  - Hilfestellung durch das Umfeld

Anton Bruckner

UNI FREIBURG

Zielstellung

- Nicht immer von Anfang an klar
  - Wissenschaft ist in Teilen unvorhersehbar.
  - Umsichtig planen und auf Eventualitäten vorbereiten
  - Zielstellung mit der Erfahrung weiterentwickeln
- Oft von außen (weich) vorgegeben
  - Wir denken uns Themen selten selbst aus...
  - Fragestellungen erfordern entsprechende Auswertungen
- Zwei Grundsätze
  1. Es wird Daten geben.
  2. Es wird Code zur Datenauswertung geben.

☛ Je klarer das Ziel, desto besser lässt sich planen.

UNI FREIBURG

Anforderungen

- Welche Aufgaben müssen bearbeitet werden?
  - Möglichst modular und detailliert beschreiben
  - Unabhängig von der Implementierung/Umsetzung
  - Brainstorming: Alle Eventualitäten auflisten
- Wer soll mit dem Programm arbeiten?
  - Infrastruktur und konzeptionelle Dokumentation
  - Ausführlichkeit der Fehlerbehandlung
  - Kommandozeile oder (grafische) Schnittstelle?
- Einmaliger Vorgang oder (künftige) Routineaufgabe?
  - Kosten-Nutzen-Rechnung
  - Entscheidend für verwendete Konzepte/Infrastruktur
  - Kommandozeile oder (grafische) Schnittstelle?

Ressourcen

- Vorwissen und Vorarbeiten
  - Welche Vorkenntnisse bringt der Programmierer mit?
  - Welche Vorarbeiten in welcher Sprache sind vorhanden?
- Verfügbarer Zeitrahmen
  - Welcher Zeitrahmen steht global zur Verfügung? (BSc-, MSc-, Doktorarbeit)
  - Wie hoch ist der Programmieranteil am Gesamtprojekt?
- Hilfestellung durch das Umfeld
  - Welche Programme werden im Arbeitsumfeld verwendet?
  - Hilfestellungen? (Bücher, Internet, Betreuer, Kollegen)

UNI FREIBURG

Wer hohe Türme bauen will,
muss lange beim Fundament verweilen.

- Ziele, Anforderungen und Ressourcen durchdenken und schriftlich festhalten
  - Nicht statisch, entwickeln sich mit dem Projekt
- Programmiersprache festlegen
  - Eigenes Vorwissen und vorhandene Vorarbeiten
  - Know-How im und Hilfestellung aus dem Arbeitsumfeld
  - Generelle Eignung für das Projekt
- Programmierkonzepte berücksichtigen
  - Infrastruktur und „Best Practices“

Anton Bruckner

Infrastruktur

Definition des Begriffs

Infrastruktur (allg.)

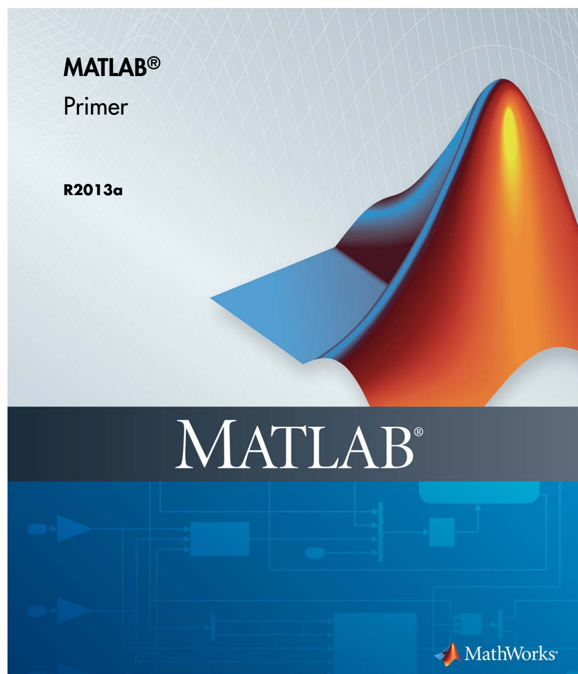
Personelle, sachliche und finanzielle Ausstattung,
um ein angestrebtes Ziel zu erreichen

- personell: Wir sind meist auf uns alleine gestellt.
- finanziell: Programmierung ist meist ein Hobby.
- sachlich: Es gibt Hilfsmittel, die das Leben erleichtern.
  - Vertrautheit mit der Programmiersprache
  - Integrierte Entwicklungsumgebungen (IDE)
  - Versionsverwaltung (VCS)
  - Bug-Verwaltung
  - Planung und konzeptionelle Dokumentation (Wiki)

Schubert, Klein: Das Politiklexikon. 5., aktual. Aufl. Bonn: Dietz 2011

Infrastruktur

„Kenne Deine Programmiersprache“



Inhalte

- Quick Start
- Language Fundamentals
- Mathematics
- Graphics
- Programming

Satz

Man muss nicht alles wissen, sollte aber wissen, wo es steht.

- Programmieren lernen ist wie eine Sprache lernen.
- Grundlegende Sprachkonzepte müssen bekannt sein.
- Details können in der Dokumentation nachgeschlagen werden.

☛ Kenntnis der vorhandenen Dokumentation und wie man sie nutzt.

Editor – /Users/till/Documents/Uni/FR/Lehre/Methodenkurse/Programmierkonzepte/aufgabenblaetter/02/sourceCode/badCode.m

```
x=[1:0.1:40*pi];
y=sin(5*x)*sin(0.1*x).*cos(2*x)
tic; figure(); maxvalue = 100;
set(gca,'XLim',[0 maxvalue]); set(gca,'YLim',[0 1]);
hold on; plot([0 maxvalue],[.5 .5])
for k=1:maxvalue value = rand;
  if value<0.5 disp('Lower half'); else disp('Upper half'); end
plot(k,value,'rx'); pause(0.02);
  end
hold off
toc
tic; t = 0:0.01:2*pi; x = 4 * cos(3*t); y = 4 * cos(4*t+pi/2);
figure; hold on; for k=1:length(x) plot(x(k),y(k),'r.'); pause(0.001); end
```

⚠ Terminate statement with semicolon to suppress output (within a script). Details Fix

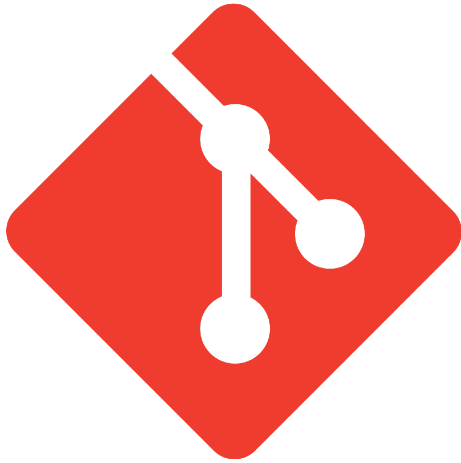
badCode.m | trEPRload.m | hyscorePlot.m | makePlots20131217.m | goodCode.m | badC

8 usages of "x" found Ln 1 Col 1

Was muss ein guter Editor können?

- automatische Codevervollständigung
- Codeüberprüfung während der Eingabe
- Hilfe aus dem Editor heraus erreichbar
- Syntaxhervorhebung („Syntax highlighting“)
- automatische Codeeinrückung
- Zusammenfalten von Codeteilen („Code folding“)
- Refaktorierung („Refactoring“)

☛ Der Matlab-Editor unterstützt die meisten der genannten Kriterien mittlerweile recht gut.


git

Versionsverwaltung

engl. *version control system* (VCS), System zur Erfassung von Änderungen an Dokumenten oder Dateien.

- Alle Versionen werden in einem Archiv gesichert.
- Jeweils mit Zeitstempel und Benutzerkennung
- Jede Version kann später wiederhergestellt werden.

☛ Typischer Einsatz: Softwareentwicklung

☛ VCS hat erst einmal nichts mit Versionsnummern zu tun.

UNI FREIBURG

Warum Versionsverwaltung?

- Auswirkungen auf die Art des Programmierens
  - Befreiend: Funktionierende Vorversion immer erreichbar
  - Strukturierend: Klarer Verweis auf eine Version möglich
- Unumgänglich für verteilte Programmierung
  - Zusammenführung unterschiedlicher Änderungen
  - Verantwortliche für Änderungen nachvollziehbar

Warum Versionsverwaltung als einzelner Nutzer?

- Historie einer Entwicklung verfügbar
- Auswirkungen auf die Art des Programmierens (s.o.)

☛ Voraussetzung für zentrale Programmierkonzepte

Infrastruktur

Bug-Verwaltung

U.S. Naval Historical Center Online Library Photograph NH 96566-KN

Warum eine Bug-Verwaltung?

- Um den Überblick zu behalten.
- Um einen Ort zu haben, wo alle Bugs auflaufen.
- Um dem jeweiligen Berichter
  ein Maximum an Transparenz zu gewährleisten.
- Um Bugs zu archivieren –
  damit lassen sich doppelte Berichte handhaben.

☛ Um den Programmierer zu entlasten.

☛ Lässt sich bis zu einem gewissen Grad
für „Feature Requests“ verwenden.

Infrastruktur

Konzeptionelle Dokumentation jenseits des Quellcodes

c't, Thomas Saur, Heise Zeitschriftenverlag

UNI FREIBURG

Konzeptionelle Dokumentation

- Eine Schnittstellen-Dokumentation reicht meist nicht aus.
- Grundlegende Konzepte und Ideen sollten zusammenhängend beschrieben werden.
- Ein statisches Dokument ist oft nicht flexibel genug.
- Ein Wiki ist eine mögliche Lösung.
  - Flexibel
  - Erlaubt einfache Aktualisierungen
  - Geeignet als primäre Informationsquelle für Anwender

☛ Hat sich in der Praxis bewährt

☛ Qualität ist (auch hier) eine Frage persönlicher Disziplin

UNI FREIBURG

Wichtige Aspekte in der Umsetzung

- Verwendung von Standard-Komponenten
  - keine Insel- oder Speziallösungen!
  - idealerweise freie Software (kostenneutral)
  - ausreichend getestet
  - einfach zu integrieren
  - hohe Chance auf Interoperabilität
- Einfache Bedienbarkeit
  - Nur was möglichst einfach und intuitiv bedienbar ist, wird in der Praxis auch eingesetzt werden.
- Hohe Verfügbarkeit
  - Programmierung oft auch nachts und am Wochenende
  - Erreichbarkeit von außerhalb der Arbeit sicherstellen

UNI FREIBURG

Infrastruktur (allg.)

personelle, sachliche und finanzielle Ausstattung,
um ein angestrebtes Ziel zu erreichen

Ziel: Das Leben erleichtern

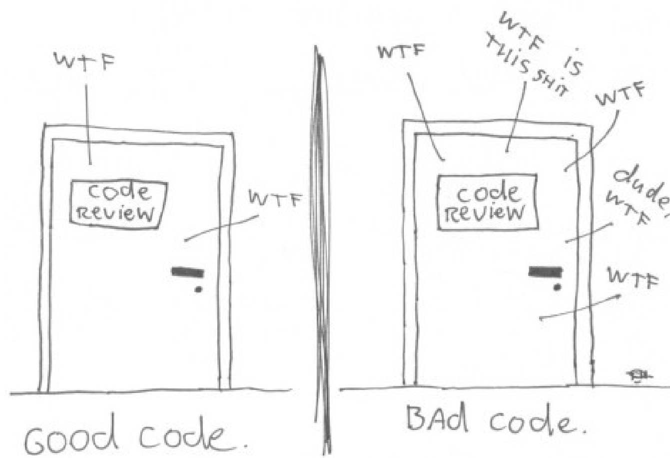
- Nutzen muss für die Anwender klar ersichtlich sein

☛ **Eigene Erfahrung:**
Infrastruktur wirkt befreiend und steigert die Produktivität

Schubert, Klein: Das Politiklexikon. 5., aktual. Aufl. Bonn: Dietz 2011

Konzepte

Warum Programmierkonzepte?



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Konzepte
Übersicht

Grundlegend

- Dokumentation
- Modularität
- Robustheit
- Schnelligkeit

Weiterführend

- Unit Tests
- Refactoring
- Test Driven Development

Voraussetzung

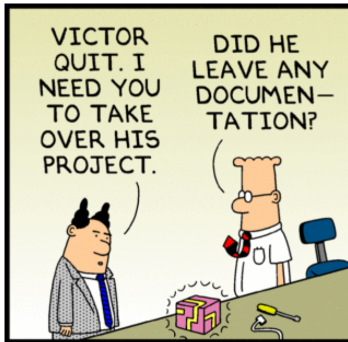
- Grundlegende Infrastruktur (VCS, IDE)

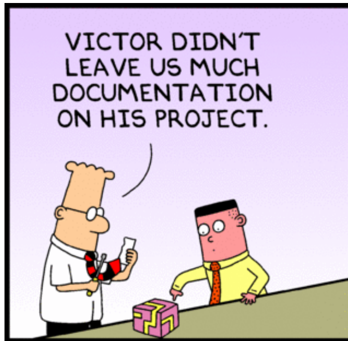
Ziel

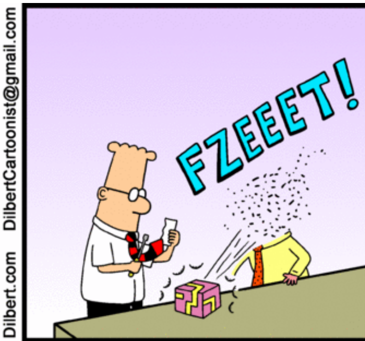
- Wiederverwertbarer, robuster, einfach zu wartender Code

Konzepte

Dokumentation







Real programmers don't comment their code.
If it was hard to write, it should be hard to read.

Warum dokumentieren?

- Weil andere das Programm nutzen/verstehen wollen.
- Weil man sich selbst nach zwei Monaten nicht mehr daran erinnern kann, was man da programmiert hat.
- Weil wir in aller Regel zu schlecht programmieren.
- Weil Dokumentation (gerade von Konzepten) für die weitere Entwicklung sehr hilfreich ist.

☛ Weil nur sauberer (dokumentierter) Code Zukunft hat.

Regel

Keine Datei (Routine, Skript) ohne Dokumentation am Anfang.

- Dokumentation ist eine Frage der Disziplin.
- Zum „Rapid Prototyping“ eignet sich die Kommandozeile.
  Skripte werden dokumentiert!
- Dokumentation muss zur Routine werden.
  Im Nachhinein zu dokumentieren ist keine Option,
  weil es dann nie gemacht wird.

Listing 1: Dokumentationsblock zu Beginn einer Routine

```
1  function spectrum = simCO2spectrum(x,fwhm,pos,height)
2  % SIMCO2SPECTRUM Function for simulating CO2 spectra using Lorentzians.
3  %
4  % Usage
5  %   spectrum = simCO2spectrum(x,fwhm,pos,height)
6  %
7  %   x        - vector
8  %              wavelength axis for spectrum
9  %
10 %   fwhm     - vector
11 %              spectral width of each Lorentzian
12 %
13 %   pos      - vector
14 %              position of each Lorentzian
15 %
16 %   height   - vector
17 %              height of each Lorentzian
18 %
19 %   spectrum - vector
20 %              calculated spectrum
21 %              same length as x
22 %
23 % The vectors fwhm, pos, and height have to be of the same size.
```

Real Programmers don’t need comments–
the code is obvious.

— Ed Post, 1983

Grenzen von Dokumentation

- Nicht das Offensichtliche dokumentieren
- Code lieber umschreiben (*Refactoring*)
  als schlechten Code ausführlich dokumentieren

Ed Post, „Real Programmers Don’t Use Pascal“, Datamation 29(7), 1983

Konzepte

Modularität

Alan Chi, Flickr

Das „Unix-Prinzip“

Eine Aufgabe, eine Routine

Don’t repeat yourself (DRY)

- Grundkonzept der Modularität
  - Eine Aufgabe wird von *genau einer* Routine erledigt.
  - Eine Routine erledigt *genau eine* Aufgabe.
- Vorteile
  - *Genau eine* Stelle im Code muss gepflegt werden.
  - Maximale Flexibilität

UNI FREIBURG

Objektorientierte Programmierung (OOP)

- Daten und verarbeitende Routinen in einem Objekt
- (Eigentlich) intuitiver als prozedurale Programmierung
- Erzwingt und erleichtert Modularisierung

Vererbung: Die abgeleitete Klasse „VW Käfer“

- Was *hat* ein Auto? (Attribute)
  - 4 Räder, 3 Türen
  - ...
- Was *kann* ein Auto? (Methoden)
  - Motor starten und stoppen
  - ...

Konzepte
Robustheit

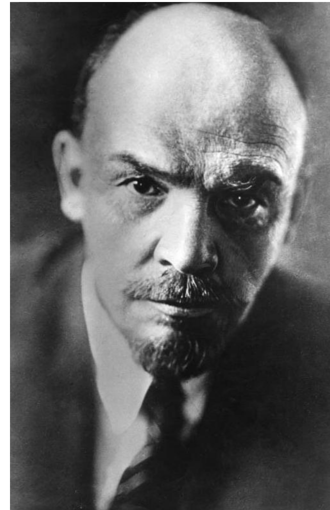
Konzepte

Robustheit

- 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

Vertrauen ist gut, Kontrolle ist besser!

Lenin (zugeschrieben)



Was sollte ggf. überprüft werden?

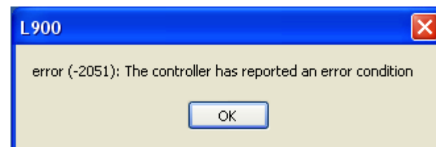
- Existenz
- Typ
- Größe
- Gültigkeit des Wertebereiches

Zwei Möglichkeiten (in Matlab)

1. manuelle Überprüfung
2. inputParser (kommt später)

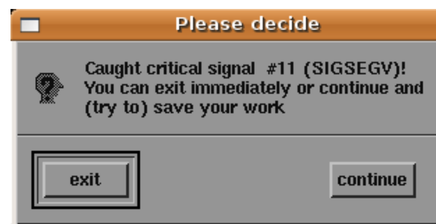
Aussagekräftige Fehlermeldungen

„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...



UNI FREIBURG

UNI FREIBURG

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
- (grafische) 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

UNI FREIBURG

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

UNI FREIBURG

Unit Tests (Modultests)

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

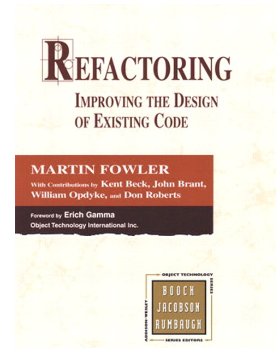
- Test des Verhaltens von Funktionen „von außen".
- Vergleich der Rückgabe oder des Verhaltens einer Funktion mit dem Erwartungswert.
- Normalerweise durch Test-Frameworks realisiert (und automatisiert).
- Voraussetzung für andere Programmierkonzepte (Refactoring, Test Driven Development)

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.

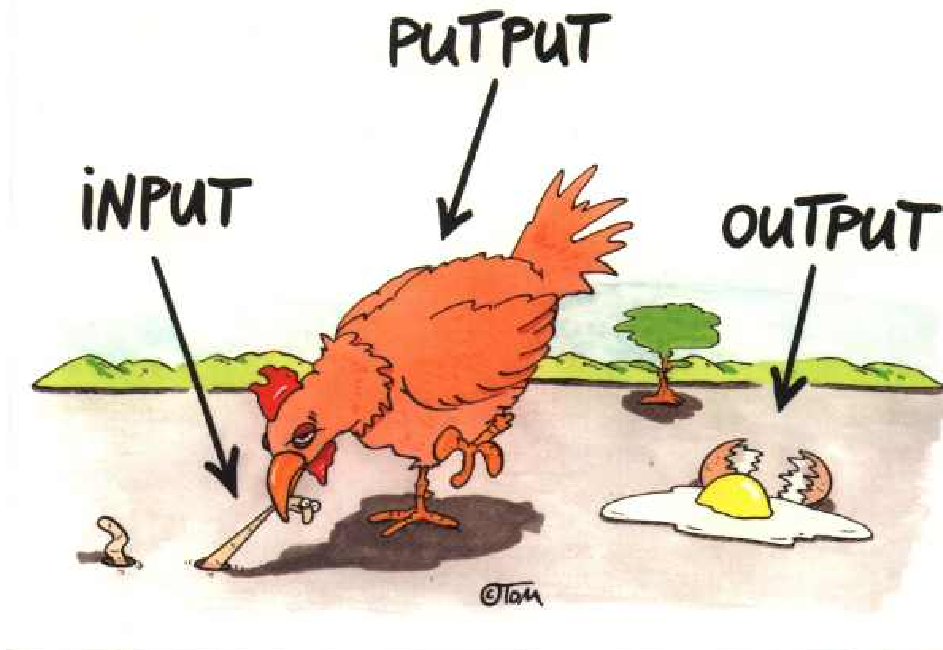
Ziel: Vereinfachung von Code

- 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



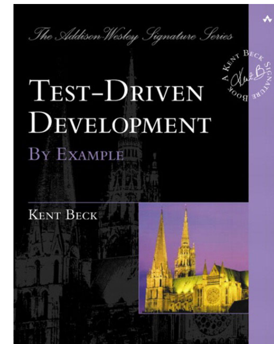
Thomas Körner, alias ©TOM

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

Konzepte

Programmierkonzepte: Vor- und Nachteile

UNI FREIBURG

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.

Datenverarbeitung

Zentraler Aspekt empirischer Wissenschaften

UNI FREIBURG

„I think you should be more explicit here in step two.“

Sydney Harris

UNI FREIBURG

Daten: Währung der empirischen Wissenschaften

- Grundlage und Ausgangspunkt empirischer Wissenschaft
  - Daten sind nicht notwendigerweise „offensichtlich"
  - Messung zur „Aufnahme" von Daten
- Daten allein sind wertlos
  - Hintergrundinformationen (Metadaten) gehören immer zwingend dazu.
  - Ein gut gepflegtes Laborbuch ist essentiell
  - Zur Datenverarbeitung ist ein Zugriff auf die Metadaten oft notwendig/hilfreich ($\Rightarrow$ maschinenlesbar ablegen)
- Daten überdauern, Interpretationen ändern sich
  - Die Verantwortung des Wissenschaftlers: saubere Datenaufnahme und -dokumentation

UNI FREIBURG

In einer idealen Welt...

- ...gibt es nur rauschfreie Daten.
- ...weiß man immer, wer was wann wie gemessen hat.
  - Experimentator, Probe, Datum, Experiment
  - Zweck des Experiments
- ...sind sämtliche Informationen zum Aufbau bekannt.
  - Bei Handaufbauten essentiell
  - Auch bei kommerziellen Geräten wichtig
  - Dokumentation austauschbarer Komponenten
- ...lassen sich alle Verarbeitungsschritte nachvollziehen.
  - Historie der Probenvorbereitung
  - Historie der Datenaufbereitung

Metadaten

Informationen über Merkmale anderer Daten,
aber nicht diese Daten selbst.

Wie lassen sich Metadaten (sinnvoll) ablegen?

- Sofort während der Messung
- Möglichst akkurat und vollständig
- In maschinenlesbarer Form
- Möglichst bei und nicht getrennt von den Daten

☛ Beispiele für Metadaten und ihre Ablage folgen

Listing 2: Beispiel für eine Infodatei

```
1  general Info file - v. 0.0.1 (2014-01-20)
2
3  GENERAL
4  Filename:                somefile
5  Date:                    2014-01-21
6  Time start:              11:05:00
7  Time end:                15:50:00
8  Operator:                Alois Kabelschacht
9  Label:                   Short and comprehensive label
10 Purpose:                 Kill time
11
12 SAMPLE
13 Name:                    Sample
14 Description:             Cool sample that doesn't show any signal
15 Preparation:             No clue, just found it lying 'round
16
17 COMMENT
18 I had a dream: Measuring this sample would solve all my problems,
19 answer all my questions, help me finishing my thesis.
20
21 Alas, it seems not to work out...
```

UNI FREIBURG

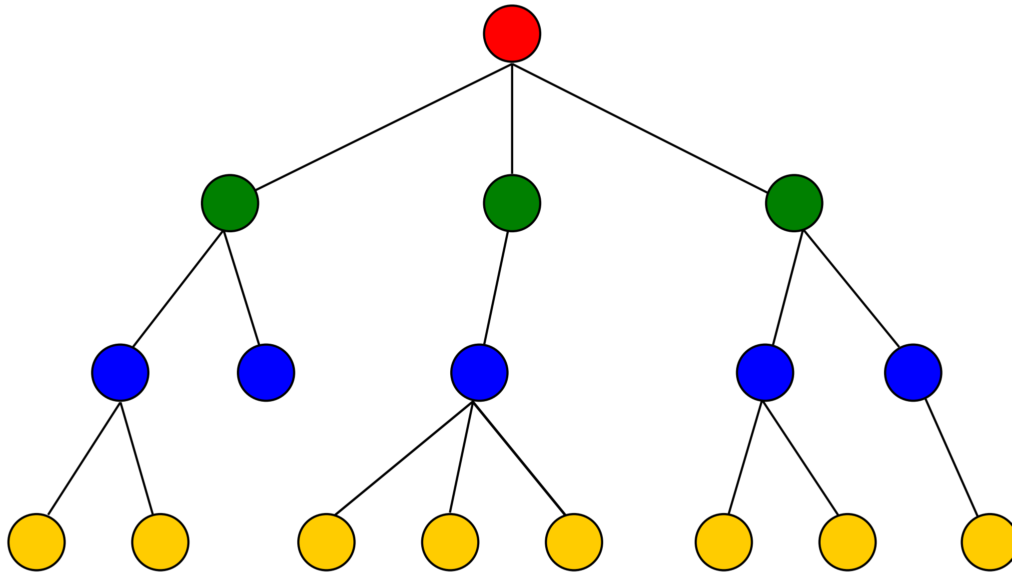
Infodatei – eine Möglichkeit der Ablage von Metadaten

- Menschen- und maschinenlesbar
  - Wird vom Experimentator erzeugt
  - Wird vom verarbeitenden Computer eingelesen
- Intuitiv aufgebaut
  - Nur einfache Dinge werden in der Praxis genutzt.
  - Der Anwender muss den Mehrwert erkennen können.
- Modular
  - Aufbauten und Anforderungen ändern sich.
  - Zukunftsfähigkeit sorgt für Akzeptanz

☛ Fokus: Einfache Benutzbarkeit
(Erfahrung: Nur was einfach und intuitiv ist, wird benutzt.)

Datenverarbeitung

Hierarchische Datenablage – Zusammenfassen, was zusammengehört



UNI FREIBURG

Hierarchische Datenablage

- Zusammenfassen, was zusammengehört
  - Blöcke zusammengehöriger Parameter
  - Sorgt für Übersichtlichkeit (und kurze Feldnamen)
- Gerichteter Graph (gewurzelter Baum)
  - Baum mit einer Wurzel
  - Knoten: Verzweigungen des Baums
  - Blätter: Enden von Knoten (Verzweigungen)
  - In der Informatik häufig als Datenstruktur verwendet
- Kriterien
  - Ein Baum hat genau eine Wurzel.
  - Blätter gehören zu genau einem Knoten.
  - Knoten können beliebig viele Blätter/Unterknoten haben.

Datenverarbeitung

Hierarchische Datenablage – Assoziative Datenfelder

UNI FREIBURG

Geordnete Listen

#	Wert
1	0.0000
2	0.0025
3	0.0050
⋮	
n-1	0.2475
n	0.2500

#	Wert
1	'Im'
2	'Anfang'
3	'war'
⋮	
n-1	'die'
n	'Tat'

Assoziative Datenfelder

Schlüssel	Wert
Name	'K. Racht'
Alter	42
Adresse	(siehe unten)
Hobbies	{'...','...'}

Adresse:

Schlüssel	Wert
Straße	'Talstraße'
Nummer	21

UNI FREIBURG

Assoziatives Datenfeld (*associative array*)

Datenstruktur, die Zeichenketten (statt Zahlen) verwendet, um die enthaltenen Elemente zu adressieren.

- Keine festgelegte Reihenfolge der Felder
- Ideal zur Ablage von Schlüssel-Wert-Paaren
- Komplexe hierarchische Datenstrukturen durch Verschachtelung möglich

☛ Tipp: Schlüsselname sollte nachvollziehbare Verbindung zum Datenwert liefern.

Standardisierte Speicherformate für hierarchische Daten

Aufgabe Speicherung verschachtelter hierarchischer Daten

Lösung XML als universelles, standardisiertes Format

XML (Extensible Markup Language)

Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien

- Vom World Wide Web Consortium (W3C) standardisiert
- reines Textformat, menschenlesbar
- Einsatz: plattform- und implementationsunabhängiger Austausch von Daten zwischen Computersystemen

UNI FREIBURG

Datensatz

Einheit von (gemessenen) Daten und zugehörigen Metadaten

- Daten und Metadaten liegen immer zusammen vor
  - Daten ohne Metadaten sind wertlos.
  - Metadaten personenunabhängig speichern
- Daten und Metadaten sind maschinenlesbar
  - Auswertesoftware ist sich der Metadaten „bewusst“
  - Automatische Auswertung abhängig von den Metadaten
- Zentrales Konzept für die Datenverarbeitung
  - Hilft, das Versprechen einzulösen, dass Toolboxen die Nachteile gegenüber Skripten ausgleichen können.

UNI FREIBURG

Ein Datensatz besteht aus mindestens drei Teilen

1. Daten
   - Eigentliche (gemessene) Daten
   - (Meist) numerisch
2. Metadaten
   - Zusätzliche Informationen zu den Messdaten
   - Z.B. aus einer Infodatei
3. Historie
   - Dokumentation aller Verarbeitungsschritte der Messdaten
   - Vollständige Nachvollziehbarkeit und Wiederholbarkeit

☛ In der Praxis ggf. noch weitere Teile

UNI FREIBURG

Historie – Dokumentation aller Verarbeitungsschritte

- Zielstellung
  - Nachvollziehbarkeit
  - Reproduzierbarkeit
- Felder für einen Eintrag
  - Name des Durchführenden
  - Datum
  - Name und Version der verarbeitenden Routine
  - Version des zugrundeliegenden Programms (z.B. Matlab)
  - Name und Version des Betriebssystems
  - ggf. sämtliche Eingabeparameter für die Routine

☛ Gewährleistet zusammen mit einer Versionsverwaltung (für den Code) die vollständige Reproduzierbarkeit.

Berichte generieren

- Motivation
  - Was sind die Charakteristika eines Datensatzes?
  - Wie wurden die Daten konkret aufgenommen?
  - Was wurde mit dem Datensatz alles gemacht?
- Inhalte
  - Zusammenfassung aller Informationen zu einem Datensatz
  - *Auf einen Blick* – charakteristische Abbildungen
- Vorteile
  - Berichte unabhängig von der Auswertungssoftware
  - Katalog vorhandener Daten einfach erstellbar
  - Vergleichbar: identische Abbildungen für jeden Datensatz

Laborinformationssystem

- Motivation
  - Übersicht über Proben, Messungen, Daten, Auswertungen
  - Einfache Durchsuchbarkeit nach unterschiedlichen Kriterien
- Voraussetzungen
  - Daten und Metadaten in maschinenlesbarer Form
  - Standardisierte Datenerfassung (Formulare, Infodateien)
- Kernaspekte der Umsetzung (Nutzersicht)
  - Hohe Verfügbarkeit (räumlich und zeitlich)
  - Einfache Bedienbarkeit
  - Plattformunabhängig
  - Einfacher Export der verfügbaren Informationen
  - Offensichtlicher Mehrgewinn bei konsequenter Nutzung

Nutzerschnittstellen

Zwei Arten von Nutzerschnittstellen

UNI FREIBURG

Textbasierte Nutzerschnittstelle (CLI)

- Menüs und Nutzereingaben in einer Textkonsole
- Vollständig deterministisch (bis auf Nutzereingaben)
- Linear: immer nur eine Entscheidungsmöglichkeit
- Strukturiert, aber mit wenig Freiheiten

Grafische Nutzerschnittstelle (GUI)

- Grafische Anordnung von Bedienelementen
- Reihenfolge der Ereignisse unvorhersehbar
- Nichtlinear: beliebige Entscheidungsmöglichkeiten
- Große Freiheit: Alles (implementierte) jederzeit möglich.

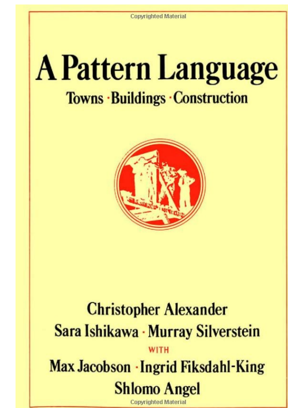
UNI FREIBURG

Gutes Design ist wichtig – und zahlt sich aus

- Nutzerschnittstellen werden (arbeits-)täglich genutzt.
  - Selbst kleine Verbesserungen zahlen sich aus.
  - Auf den Nutzer und seine Bedürfnisse hören.
- Intuitive Nutzerführung vereinfacht komplexe Abläufe.
  - Freiheit, alles zu tun, was grundsätzlich möglich ist
  - Das Naheliegende nahe liegend anordnen.
  - Reduktion einzelner Ebenen: Erfassbarkeit auf einen Blick
- Gutes Design steht am Ende eines langen Prozesses.
  - Reduktion auf das Wesentliche ist eine Kunst.
  - Dauerhaftigkeit erfordert rigoroses Durchdenken.
  - Prototypen konsequent in der Praxis testen

Bewährte Muster aus der Praxis

- Wiederkehrende Muster
  - Funktional oder „kulturell“ bedingt
  - Bewährt, aber nicht immer optimal
  - Sollten Kreativität nicht behindern
- Wiedererkennungseffekte
  - Kann die Bedienung beschleunigen
  - Ersetzt nicht die Einarbeitung
  - Gefahr: vermeintliche Vertrautheit



☛ Konzepte und Muster nicht unhinterfragt einsetzen

☛ Aber: Es lohnt nicht, das Rad immer neu zu erfinden...

Gründe für die Trennung

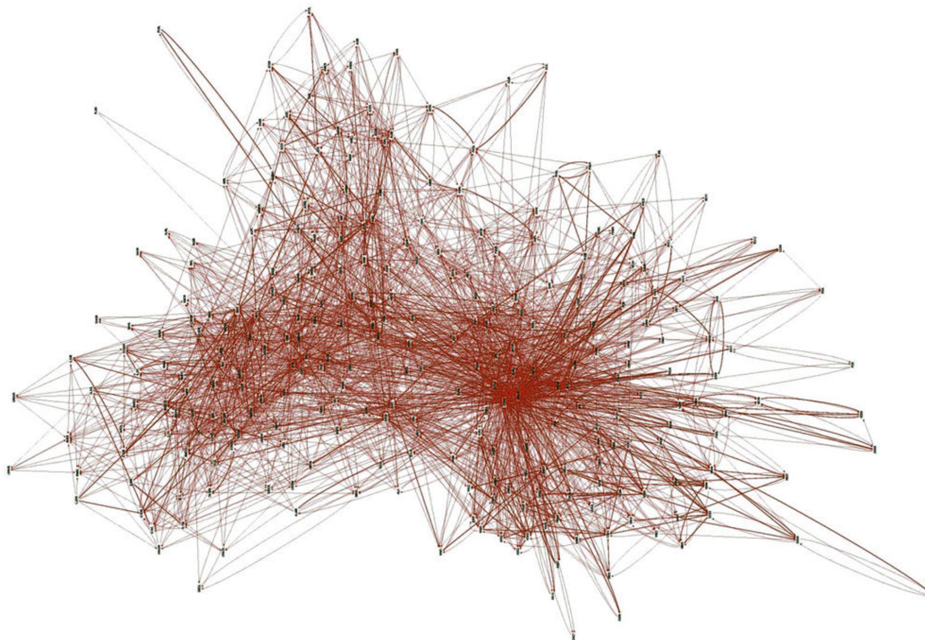
- Saubere und fehlerfreie Datenverarbeitung ist in der Wissenschaft von allergrößter Bedeutung.
  - Nachvollziehbarkeit der Prozessierung der Daten
  - Automatisierte Wiederholbarkeit einer Prozessierung
- Eine Routine für einen Schritt der Datenverarbeitung
  - Kann direkt oder von der jeweiligen Nutzerschnittstelle aufgerufen werden
  - Fehler müssen nur an *einer* Stelle behoben werden.
- Modularisierung vereinfacht die Programmierung von Nutzerschnittstellen
  - Konzentration auf die Schnittstelle
  - Die verarbeitenden Routinen sind bereits vorhanden.

Gründe für die Trennung (Fortsetzung)

- Fehler in der Nutzerschnittstelle verhindern nicht die weitere Auswertung.
  - Komplexe Matlab-GUIs sind schwer plattformunabhängig und unabhängig von der Matlab-Version lauffähig zu halten.
  - Immer auch die Möglichkeit geben, die Auswerteroutinen händisch aufzurufen.
- Freiheit und Unvorhersehbarkeit
  - Eine feste Schnittstelle (CLI und besonders GUI) schränkt den Benutzer zu stark ein.
  - Wissenschaft lebt vom frischen Blick auf alte Probleme: „Lego-Prinzip" als Erfolgsgarantie.

Gründe für die Trennung (Fortsetzung)

- Testbarkeit
  - CLIs und GUIs (fast) nicht automatisiert testbar
  - Tests der Datenverarbeitungsschritte gerade in der Naturwissenschaft von essentieller Bedeutung
  - Tests oft über Parameter für Spezialbedingungen
- Arbeitsteilung
  - Schnittstellendesign und Datenverarbeitungsroutinen erfordern vollkommen unterschiedliche Qualifikationen.
  - Model-View-Control-Ansatz (MVC) in der Praxis bewährt
- Automatisierbarkeit
  - Datenverarbeitung oft nach festem Schema
  - „Skriptbarkeit“ der Verarbeitung zur Arbeitserleichterung



Brain connectivity map of *C. elegans*

Nutzerschnittstellen
Besonderheiten von GUIs

UNI FREIBURG

- Nicht linear und nicht vorhersehbar
  - Nutzer kann jederzeit jedes implementierte Ereignis in beliebiger Reihenfolge auslösen.
- Ereignisgetrieben
  - Grundsätzlich anderes Programmier-Paradigma
- Parallel
  - Zeitlich parallele Verarbeitung mehrerer Prozesse
- Keine Nachvollziehbarkeit der Bearbeitungshistorie
  - Muss per Hand implementiert werden

☛ Direkte, tiefgreifende Auswirkung auf die Programmierung

Ereignisgetriebene Programmierung

Der Programmfluss wird durch Ereignisse (*events*) gesteuert, das Programm *nicht* linear durchlaufen.

Threadsicherheit

Eine Komponente kann gleichzeitig mehrfach ausgeführt werden, ohne sich gegenseitig zu behindern.

Wettlaufsituation (*race condition*)

Das Ergebnis einer Operation hängt vom zeitlichen Verhalten bestimmter Einzeloperationen ab.

UNI FREIBURG

UNI FREIBURG

Berücksichtigung ergonomischer Grundsätze bei Software

- Empfehlungen in der Norm DIN EN ISO 9241 geregelt

Grundsätze der Dialoggestaltung nach DIN EN ISO 9241-110

1. Aufgabenangemessenheit
2. Selbstbeschreibungsfähigkeit
3. Steuerbarkeit
4. Erwartungskonformität
5. Fehlertoleranz
6. Individualisierbarkeit
7. Lernförderlichkeit

UNI FREIBURG

Einige hilfreiche Programmierkonzepte

- There is more than one way to do it (TIMTOWTDI)
- Keep easy things easy and the hard possible
- Don’t repeat yourself (DRY)
- Keep it simple stupid (KISS)
- Convention over configuration
- Aussehen, Programmlogik und Datenverarbeitung trennen

☛ Generelle Konzepte für robusten, einfach wartbaren Code

☛ Lassen sich gewinnbringend für GUIs einsetzen

Nutzerschnittstellen

GUI-Entwicklung: Designentscheidungen

Lounge Chair & Ottoman, Charles & Ray Eames, 1956

UNI FREIBURG

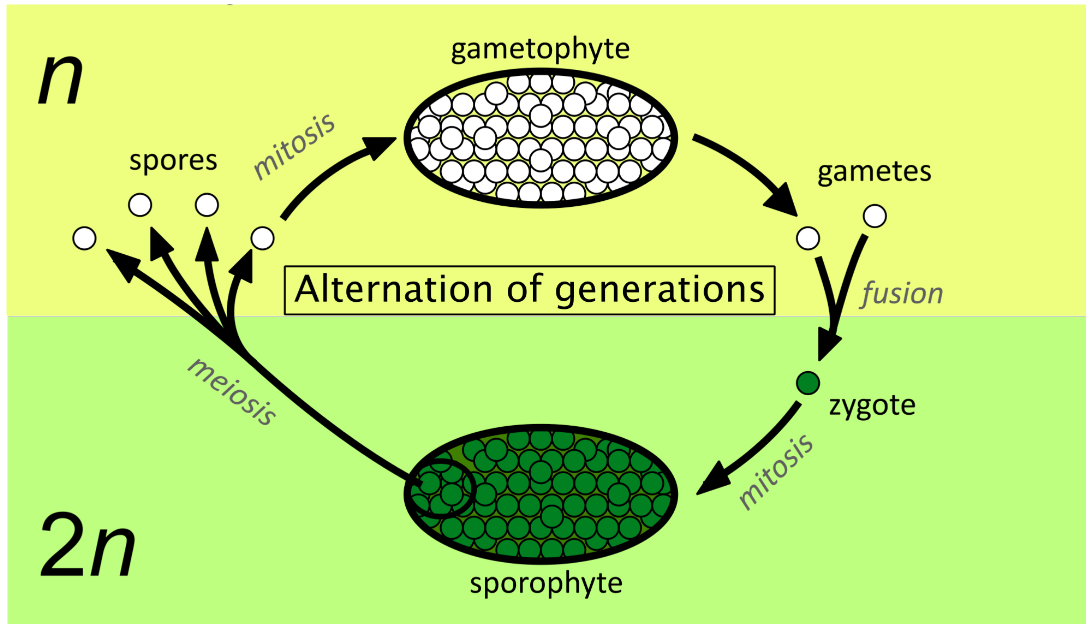
Designentscheidungen – mehrere Ebenen

- Anwendersicht
  - Sprache der Elemente
  - Anordnung von Elementen
  - GUI in der Größe veränderbar?
  - Ein oder mehrere Fenster
  - Verwendung von Menüs
  - Tastenkürzel
- Entwicklersicht
  - Entwicklung: per Hand oder mit Hilfsmitteln
  - Programmiersprache und Grafik-Bibliothek
  - Plattformunabhängige Entwicklung?

☛ Keinerlei Anspruch auf Vollständigkeit

Nutzerschnittstellen

Allgemeiner Entwicklungszyklus einer GUI



UNI FREIBURG

Allgemeiner Entwicklungszyklus einer GUI

1. Was soll die GUI können?
   - Anforderungsanalyse (so detailliert wie möglich)
   - Enge Abstimmung mit der (künftigen) Nutzergruppe
2. Abläufe überlegen
   - Erfahrungen und Konventionen berücksichtigen
3. Prototypen implementieren und optimieren
   - Prototypen verarbeiten keine Daten
   - Programmlogik möglichst vollständig implementieren
   - Abläufe aus der Praxis heraus optimieren
4. Funktionen hinterlegen
   - Idealerweise bereits vorhanden
   - Modular und von der GUI unabhängig

Tipps aus der Praxis

- Anforderungsanalyse
  - Möglichst klar formulierte Zielstellungen
  - Arbeitsabläufe aus der Praxis kommend festhalten
  - Enge Abstimmung mit (künftiger) Nutzergruppe
- GUI-Design beginnt mit Zettel und Stift.
  - Wesentlich flexibler als Prototypen
  - Schnellere Entwicklungszyklen möglich
  - Kosten-Nutzen-Rechnung
  - Voraussetzung: Vorstellungskraft, Abstraktionsvermögen
- Abläufe lassen sich oft erst in der Praxis optimieren.
  - Setzt (voll) funktionsfähige Prototypen voraus
  - Skaliert mit der eigenen Erfahrung

UNI FREIBURG

Tipps aus der Praxis (Fortsetzung)

- Auf die Nutzer hören
  - Implementierung macht oft betriebsblind.
  - Nutzer steht im Fokus, nicht die dahinterliegende Technik
- Nutzern über die Schulter schauen
  - Wir sind daran gewöhnt, „workarounds“ zu finden.
  - Nutzer verwenden GUIs häufig anders als gedacht.
  - Offenbart häufig Schwächen in der Implementierung
- GUIs als Entwickler an realen Beispielen testen
  - Wenn etwas schon den Entwickler nervt...
  - Ermöglicht deutlich kürzere Entwicklungszyklen
  - Voraussetzung: Satz an (realistischen) Testzenarien

UNI FREIBURG

Alternativen zu Matlab-GUIs

- Frei verfügbare Programmiersprachen
  - Fortran
  - C/C++
  - Python
  - ...
- Frei verfügbare Grafikbibliothek
  - wxWidgets
  - GTK+
  - Qt
  - ...

☛ Setzt Beherrschung sowohl der Programmiersprache als auch der Grafikbibliothek voraus.

UNI FREIBURG

Textbasierte Nutzerschnittstellen (CLI)

- Relativ einfach zu implementieren.
- Lineare (festgelegte) Nutzerführung
- Notwendigkeit, alles textlich zu beschreiben

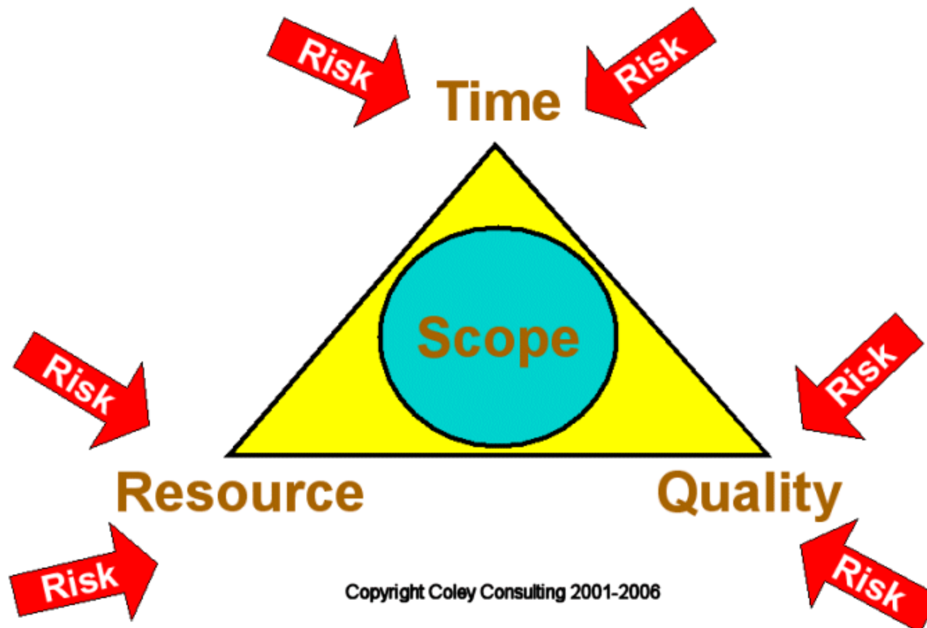
Grafische Nutzerschnittstellen (GUI)

- Oft schnellerer Zugang für den Gelegenheitsnutzer
- Wesentlich aufwendiger in der Programmierung
- Matlab: Festlegung auf ein kommerzielles Programm
- Plattformunabhängigkeit schwer zu gewährleisten

☛ Klare Abwägung: Lohnt sich der Aufwand wirklich?

Projektplanung

Meilensteine und Zeitabschätzung



Meilensteine

- Aufteilung des Projektes in sinnvolle Abschnitte
  - Programmierung häufig notwendiges Mittel zum Zweck
  - Einzelne Schritte erhöhen die Übersichtlichkeit.
  - Schneller Erfolg motiviert zum Weitermachen.
- Planung und Dokumentation einzelner Meilensteine
  - Zeit für Programmierung oft knapp bemessen
  - Effizienz durch Ausformulierung der nächsten Schritte
  - Wichtig für Arbeitsteilung bei gemeinsamen Projekten
- Umsetzung
  - Ggf. eigener Zweig/Entwicklungszweig im VCS
  - Dokumentation des aktuellen Zustandes

UNI FREIBURG

I love deadlines.
I like the whooshing sound they make as they fly by.

(Sinnvolle) Zeitabschätzung

- Verfügbarer Zeitrahmen
  - Welcher Zeitrahmen steht global zur Verfügung? (BSc-, MSc-, Doktorarbeit)
  - Wie hoch ist der Programmieranteil am Gesamtprojekt?
- Praxis
  - Realistische Zeitabschätzung Frage der Erfahrung
  - Programmiererfahrung beschleunigt die Umsetzung

☛ Viele Projekte sterben mangels realistischer Planung.

Douglas Adams

OPEN ACCESS Freely available online

PLOS | BIOLOGY

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Guy[6¤], Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11], Ethan P. White[12], Paul Wilson[13]

*“Scientists spend an increasing amount of time building and using software. However, **most scientists are never taught how to do this efficiently**. As a result, many are unaware of tools and practices that would allow them to write more reliable and maintainable code with less effort. We describe a **set of best practices for scientific software development** that have solid foundations in research and experience, and that improve scientists’ productivity and the reliability of their software.”*

PLoS Biol **12**:e1001745, 2014

Summary of Best Practices

1. Write programs for people, not computers.
2. Let the computer do the work.
3. Make incremental changes.
4. Don’t repeat yourself (or others).
5. Plan for mistakes
6. Optimize software only after it works correctly.
7. Document design and purpose, not mechanics.
8. Collaborate.

PLoS Biol **12**:e1001745, 2014

Ausblick

Das war's. Wie geht's weiter?

So long, and thanks for all the fish.

Vorschau: Eigene Projekte

- Ab jetzt sind die eigenen Projekte an der Reihe...
- Viele Dinge lernt man nur durch Ausprobieren.

Herzlichen Dank für's Mitmachen!

Douglas Adams