Notizen Softwaretechnik

Klaus Kusche, 2010 / 2011

Inhalt

Was man machen könnte / was andere machen:

- Prozess-Modelle, Projekt-Management:
  Wie geht man bei einem SW-Projekt vor?
  Welche Phasen hat es, wie ist der organisatorische Ablauf?
  ==› V-Modell, Wasserfall-Modell
  ==› Agile Entwicklungsmodelle (Scrum, Kanban, Feature Driven Development, ...), ...
  ==› Rapid Prototyping, Test-Getriebene Entwicklung
- Anforderungs-Analyse ("Requirements Engineering"):
  ==› Use Cases, Pflichtenheft, ...
  Aufwandsermittlung
- Entwurfsmodelle ("Wie komme ich von einem Problem zu einem Entwurf?"):
  ==› Objektorientierte Analyse & Design
  ==› Modellgetriebene Software-Entwicklung
  ==› Domain Driven Design, Aspektorientierte Programmierung, ...
- Dokumentation eines SW-Projektes:
  Problembeschreibung --› Entwurf --› Code --› Test
  ==› UML (Hoff!)
- SW-Architektur im Großen, Strukturierung großer Projekte:
  ==› Model-View-Controller, komponentenbasierte Entwicklung, verteilte Systeme,...
- SW-Architektur im Kleinen:
  ==› Design Patterns ("Gang of Four"-Buch!)
  ==› Refactoring (Umstrukturierung von veraltertem Bestandscode)
- Qualitätssicherung
- Werkzeuge der Software-Entwicklung
- HCI (Mensch-Maschine-Interface), Usability, SW-Ergonomie

Was wir machen:

- Rund 75 %: Werkzeuge der Software-Entwicklung

  Schwerpunkt (mehr als die Hälfte davon):
  Werkzeuge für Debugging, Analyse, Qualitätssicherung

  ==› Keine Programmier-Veranstaltung, aber sehr *technisch orientiert!*
- Rund 20 %: Prozess-Modelle, Ablauf von SW-Projekten, Organisatorische Tipps, ...

  Nicht: Allgemeines Projektmanagement

  Sondern: Technische Aspekte, Spezifika von Software-Projekten, ...
- Rest: GUI-Design, Mensch-Maschine-Interface, ...

Unterlagen

- Diese Notizen
- Selbst-Recherche am Internet
- Online-Doku zu den Software-Werkzeugen!

Praktische Übungen

- Zu Software-Werkzeugen: Praktischer Einsatz an Beispielen

  Plattform: Linux (ausschließlich Open-Source-Werkzeuge)

  ==› Voraussetzung: Praktische Beherrschung von Linux
  (Commandline, auch für Kompilieren usw.)

  ==› Voraussetzung: C-Kenntnisse, Programmierverständnis
- Zu GUI-Design: Internet-Recherche und Auswertung

Beurteilung

- b / nb auf die Übung (wenn möglich informell / nach Mitarbeit)
- Klausur (am Papier), geteilt mit UML

Software-Engineering allgemein

Software-Engineering =
die "ingenieurmäßige" (methodische, wirtschaftliche, ...)
Herstellung von Software

Ziel: Herstellung eines **Software-Produktes**
(*mehr* als nur ein "Programm", siehe unten):

- Der Spezifikation und hoffentlich auch den realen Anforderungen (des Benutzers!) entsprechend
- Zum geplanten Termin fertig, mit vorhersagbaren Kosten und Aufwänden
- Korrekt (fehlerfrei), robust / zuverlässig,
  effizient / performant / ressourcenschonend (CPU, Speicher, ...),
  bedienungs- und benutzerfreundlich, ...
- Langlebig: Wartbar, erweiterbar, wiederverwendbar, portabel, ...
  (›› 50 % der SW-Aufwände sind Wartung!)
- Heute meist auch: Zertifizierbar!
  - Zertifizierung des allgemeinen Entwicklungs-Prozesses (==› ISO 9000)
  - Zertifizierung der Software an sich (beim Einsatz in "kritischen" Systemen)

==› setzt bestimmte Dokumentationen, Verfahren und Werkzeuge voraus!

Software-Engineering als Wissenschaft umfasst:

- Standards, Verfahren & Vorgehensmodelle, Arbeitsorganisation, ...
  (siehe Inhaltsliste oben!)
- Werkzeuge
- Programmierung an sich (auch: Algorithmen und Datenstrukturen, ...),
  Programmiersprachen, ...

Was gehört zur **Software**?

- Die eigentliche Software ("das Programm")
- Test-Software, Testdaten, ...
- Selbstentwickelte Werkzeuge, ...
- Interne Dokumentation:
  Pflichtenheft, Design-Dokumente, Projekt-Ablauf-Dokumentation, ...
  Source-Doku, Test- und Analyse-Doku, ...
- Externe Dokumentation (Handbücher, Schulungsunterlagen, ...)
  ==› Hier nicht behandelt!

... und das alles in allen Entwicklungsständen! (==› Versionsverwaltet)

Werkzeuge der Software-Entwicklung

Programmierung ist "Handwerk"
==> Beherrschung der Werkzeuge ist Teil des Könnens
(wie bei Tischler, Maschinenbauer, ...)

Richtiger Einsatz der richtigen Werkzeuge bestimmt:

- *Qualität* des Ergebnisses
- *Effizienz* der Entwicklung, Wartungs-Aufwände, ...

IDE (Integrated Development Environment)

"(GUI-basierte) Integrierte Entwicklungsumgebung"

==› vereinigt *Editor & Funktionalitäten vieler Commandline-Tools* unter einer gemeinsamen Oberfläche.

Beispiele (ohne Vollständigkeit):

- **MS Visual Studio** (C#, C / C++, Windows, kommerziell)
- **Eclipse** + CDT, NetBeans (Java-basiert, portabel, auch C/C++, open Source)
- **Kdevelop** (primär C++, Linux, open Source, mächtig)
- **Code::Blocks**, **CodeLite**, … (primär C++, kleiner, portabel, open Source)

Funktionalität / Umfang:

IDE's kombinieren im Wesentlichen…

- Editor (incl. Cross-Reference-Tool, Source Code Formatter, …).
- Compiler / Linker
- Debugger
- Make, Projektfile-Verwaltung, Projekt-Templates (ev. incl. Auto-Tools)
- Optional: GUI-Designer
- Optional: Versionsverwaltungs-Anbindung
- Optional: Anbindung zu Profiler, Memory Checking Tool (Valgrind o.ä.), …

Vorteil meist: *Komfortabler* zu bedienen als Einzeltools , weniger "Reibungsverluste".
Nachteil oft: *Weniger flexibel / leistungsfähig* als händisch aufgerufene Einzel-Tools.

Zahlreiche Tools aus dem Bereich Debugging / Analyse / Qualitätssicherung *fehlen!*

Checkliste **Editor:**

- Syntax Coloring / Highlighting
- Automatische Einrückung & Formatierung
- Block Indent / Unindent, Block Comment / Uncomment
- Folding, Klammern-Matching, Select To Brace
- Rectangular Selection
- Mehr-Dokument-Fähigkeit, Split File View (zwei Fenster in *denselben* File)
- Find All / Find & Replace in Multiple Files, Regular Expression Find
- Wechsel bzw. Suche zwischen Verwendung ‹--› Definition ‹--› Deklaration
- Context sensitive Help (Typ / Prototyp, Dokumentation), "Call tips"
- Code Completion, allgemeine Code Snippets
- Insert Comment Template (Doxygen)

- Locate Compiler Error
- Tastenkombinationen frei belegbar (==› Akzeptanz durch Entwickler!)
- Skriptfähigkeit (Makros), Plugin-Interface, "Pipe Through External Tool"
- Class Browser / Function Browser

Akzeptanzproblem / Religionskrieg:
Viele Vi / Emacs-Anhänger bei Software-Entwicklern, verweigern andere Editoren...
(und umgekehrt)
==› Vi-Mode im Editor?

Hinweise / Kriterien:

- Das Build-System muss es erlauben, *mehrere Builds desselben Projekts* gleichzeitig und unabhängig voneinander zu konfigurieren, zu bauen und auf Platte zu halten (typischerweise: *Debug Build + Release Build*, ev. noch Profiling Build usw.).

  Auch *mehrere Versionen eines Projektes* müssen gleichzeitig bearbeitbar sein.
- Auf *Integrierbarkeit von Fremd-Tools* (andere Compiler, andere Versionsverwaltung, anderes Make, andere Doku-Tools, ...) achten!
- Falls erforderlich: Auf *Multi-Plattform-Fähigkeit und Cross-Compile-Fähigkeit* achten (Projekte lassen sich auf einer Entwickler-Plattform für verschiedene Betriebssysteme / verschiedene Prozessoren bauen).

  Keine getrennten Projektdateien pro Plattform (gemeinsame Projekt-Definition für alle Plattformen)!

  Damit verbunden: *Remote-Debug-Fähigkeit über Plattform-Grenzen*.
- Auf *Portabilität* der Projekte achten!
  - Generierung *standard-konformer Makefiles*, diese sollten *die gesamte Projektinformation und alle Projekt-Einstellungen enthalten!*

    ==› Möglichst keine oder wenig Information in separaten Projektfiles!
  - Verwendbarkeit *externer Makefiles*.
  - Keine internen Projektdatenbanken oder Projekt-Files *in binären / proprietären Formaten*. (besser z.B.: Plain Text, XML, ...)
  - *Import-Möglichkeit* fremder Projekte / Makefiles.
    (was gibt es an Bestands-Projekten? An Fremd-Code?)
  - Bei lizenzpflichtiger IDE: Die gesamte Projekt-Information muss auch *nach Ablauf der Lizenz* extrahierbar sein!
- Auf *Portabilität der generierten Code-Dokumentation* (standard-konformes HTML, LaTeX, ...) achten!
- Alle Build-Tools / Build-Features müssen auch mit *skriptfähigem / commandline-fähigem Aufruf außerhalb der IDE* nutzbar sein! (d.h. ein komplettes Projekt muss sich im Batch ohne Benutzerinteraktion bauen lassen)
- Alle *Optionen der Tool-Aufrufe* müssen konfigurierbar sein und sollten in den Makefiles abgelegt werden.

- Saubere *Trennung zwischen Projektdaten und -Einstellungen*
  (gehören in die Projekt-Verzeichnishierarchie ==› werden mit eingecheckt!)
  und *Benutzer- bzw. PC-Einstellungen*
  (GUI-Vorlieben, Verzeichnisse externer Tools, …)
  (gehören in das Home-Verzeichnis des Benutzers oder
  in die Registry / das Konfigurationsverzeichnis des PC ==› werden lokal gesichert).
- *Adaptierbarkeit aller File- und Projekt-Templates*.

  Anpassbarkeit an Firmen- und Projektvorgaben:
  Source-Formatierungs-Stil, Projekt-Verzeichnis-Struktur,
  Installations-Verzeichnis-Struktur, …
- *"Server-freies" Arbeiten* mit einem Notebook sollte möglich sein
  (für den Debug-Einsatz vor Ort beim Kunden).

GUI-Designer

GUI-Designer erlauben das *Programmieren von grafischen Benutzeroberflächen*
(im besonderen Dialogen) *durch grafisches "Zusammenklicken" der Bedienelemente*
(Konfigurieren und Anordnen der Elemente mittels Maus).

Es sind *zwei Arbeitsweisen* üblich:

- Der GUI-Designer erzeugt *echten Programmcode*,
  der dann normal in das Software-Projekt hineincompiliert wird.

  Solche GUI-Designer sind Bestandteil mancher IDE's (bzw. enthalten auch ein IDE).
- Der GUI-Designer erzeugt eine *Beschreibung des GUI in einer Datei*
  (oft in XML-Form), die dann zur Laufzeit geladen und interpretiert wird
  (d.h. es gibt keinen separaten ausführbaren Code für die einzelnen Dialoge).

  In diesem Fall gehört zum GUI-Designer eine von den konkreten Dialogen
  unabhängige GUI-Engine, die als Library zum Software-Projekt gelinkt wird.

Beispiele:

- Qt Designer
- Glade (GTK)

Hinweise / Kriterien:

- **Achtung**: GUI-Designer sind meist an eine einzige GUI-Library (z.B. .net, Qt, GTK, …)
  gebunden, *der Transfer der Designs auf eine andere Library oder einen anderen GUI-Designer ist unmöglich!*
- Die Beschreibung der internen Daten soll in *Textform* (z.B. XML) gespeichert
  werden, nicht binär (u.a. wegen Checkin, Versions-Vergleich).
- Im Fall von echtem Dialog-Code: Der Code soll menschlich lesbar / modifizierbar /
  wartbar sein und Warning-frei compilieren.
- Die Hinterlegung von Funktionalität (z.B. bei Button-Klicks) geschieht meist
  durch die Anbindung von Event-Funktionen. Für diese Funktionen sollte eine
  freie Namenswahl und eine Ableitung von vorgegebenen Klassen möglich sein.
- Der GUI-Designer soll mit *Internationalisierungs-Tools* (zur mehrsprachigen

Definition von Texten) und *Testtools* (zum GUI-Scripting) verträglich sein.

RAD Tools (Rapid Application Development), Rapid Prototyping, ...

Konventionelle Softwareprojekte haben u.a. folgende Probleme:

- Die *Spezifikation* der Anforderungen und vor allem *der Benutzeroberfläche / des Bedienverhaltens* ist rein abstrakt (d.h. ohne konkretes Muster oder Anschauungsobjekt) schwierig, zumal Entwickler und Kunden oft "in verschiedenen Sprachen" reden und denken.

  Oft werden in der Spezifikation auch Anforderungen übersehen, die erst bei der praktischen Benutzung des Programms offensichtlich werden.
- Erst gegen Ende des Projektes steht eine lauffähige Version zur Verfügung, mit der getestet werden kann, ob die Entwicklung überhaupt den Vorstellungen des Kunden entspricht. Zu diesem Zeitpunkt sind aber Änderungen und Anpassungen schon sehr aufwändig und teuer.
- Bei großen Projekten, die eine Laufzeit von mehreren Jahren haben, sind Spezifikation und Anforderungen meist schon überholt, bevor das erste lauffähige Programm entstanden ist. Ein weiterer Spezifikations- und Entwicklungs-Zyklus zum Einarbeiten der Änderungswünsche dauert abermals Monate oder Jahre.

Gewünscht wäre hier ein frühes *"Spielzeugprogramm" zur Abklärungs des Sollverhaltens* und der Benutzeroberfläche mit dem Kunden bzw. zur Verringerung der Sprachbarriere und zur Vermeidung von Missverständnissen: *Haben beide Seiten dieselbe Vorstellung vom gewünschten Produkt?* In manchen Fällen ist auch eine *Machbarkeitsstudie* hilfreich.

Hier kommen Tools zum Einsatz, die es erlauben, rasch zu einem Prototypen zu kommen, mit dem der Kunde einmal experimentieren und das *Verhalten des Produktes "erfühlen"* kann.

Diese Tools umfassen primär das *GUI-Design* (siehe voriger Punkt), aber auch die Daten- bzw. *Datenbank-Modellierung* (wobei sich die Verbindung zwischen GUI und Datenmodell oft "zusammenklicken" lässt) und die eigentliche *Codierung* (meist nicht in traditionellen Sprachen, sondern skriptartig oder in Form von Software-Baukästen). Auch die Tools zur Erstellung von Web-Applikationen gehen teilweise in diese Richtung.

Für solche Tools gilt:

- **Hauptziel**: *Rasche* und einfache *Erstellung* des Prototypen, *leichte* und jederzeitige *Änderbarkeit*.
- *Aussehen* des GUI und ausgewählte *Grundfunktionalitäten* sind *wichtig*.
- *Qualität*, Robustheit und Performance sind *egal* .
- Lange Build-Phasen sind unerwünscht:
  Der Prototyp muss nach einer Änderung "auf Knopfdruck" laufen.
- Der Prototyp muss *nicht* die endgültige Datenmenge handhaben können und nicht die endgültige interne Datendarstellung und Strukturierung haben (z.B. Logik und Datenhaltung am Client im RAM statt Client-Server-Architektur mit Datenbank).

Es gibt **verschiedene Arten** des Prototyping:

- Reine *Anschauungs-Prototypen*, die nur der Spezifikation dienen bzw. Ideen

und Erkenntnisse liefern sollen und dann *verworfen* werden.

Solche Prototypen könne die *formale Spezifikation ergänzen bzw. ersetzen* (z.B. betreffend GUI) .

In diesem Fall wird das finale Produkt unabhängig vom Prototyp konventionell codiert.

- Prototypen, die *schrittweise zu einem finalen Produkt vervollständigt* werden.

Versions-Verwaltungs-Systeme

Zweck:

- **Verwaltung** / Archivierung **aller Dateien** eines Software-Produkts **in allen Ständen**
- **Buchführung** über **jede einzelne Änderung** jeder einzelnen Datei

==› **Reproduzierbarkeit & Nachvollziehbarkeit!**

Aufgaben:

- Versionsgeschichte: *Wer* (Autor) hat *wann* (Datum) *was* (Änderung) geändert? Entweder eines einzelnen Files / Verzeichnisses oder des gesamten Projektes!
- Konsistente Versions-*Nummerierung: x.y.z*
- *Labeling / Tagging* von Ständen: z.B. "Weihnachts-Beta-Release"
- *Kommentar* der Änderungen: *Warum* ==› Querverweis in den Bugtracker!
- Buchführung über zusammengehörende Änderungen in *mehreren* Dateien: "Fix *xyz* umfasst Änderungen in den Dateien *aaa, bbb, ccc*"
- *Rekonstruktion* alter Stände (nach *Datum* oder *Versionsnummer*)
- Anzeige aller *Änderungen zwischen zwei Ständen* (grafisch!)
- Verwaltung von *Branches*:
  - "Head", "Master" oder "Trunk" (aktueller Hauptentwicklungs-Branch)
  - Release- und Wartungs-Branches (nur Fixes, keine Neuentwicklungen)
  - Plattform- oder Kunden-Branches (Sonderversionen anderer Branches)
  - Feature Development Branches (experimentelle Entwicklungen)
  - Im schlimmsten Fall: Dead Head Branch (aufgegebener Head, Weiterentwicklung in anderem, älterem Stand)

  ==› Anzeige der Versionen im Idealfall als **Graph oder Baum**!
  - Automatisches Mergen von *einzelnen Änderungen* aus dem "Head"-Branch oder einem Wartungs-Branch in andere Branches (bei Wartungs-Branches auch aufwärts: Mergen eines Fixes aus dem Wartungsbranch nach "Head").
  - Vor allem bei Feature Branches: Zurück-Mergen *aller Entwicklungen* des Feature-Branches nach "Head" oder Resynchronisation mit "Head" *in beide Richtungen* (Übernahme aller Neuerungen von "Head" in den Feature-Branch).

  ==› Anzeige der *Merge-Beziehungen* im Versionsbaum
- *Sperrverwaltung*:
  - Welche Files werden aktuell gerade geändert (sind ausgecheckt), von wem?
  - *Konkurrierende Veränderungen verhindern oder mergen!*
- Automatisches Einfügen von Versionsverwaltungs-Tags (Datum, Version, Autor, …) in Source-Kommentare.

- Automatische Changelog-Erstellung
- *Optimierte Speicherung:* *Nur die Deltas* jeder Änderung, nicht jedesmal die komplette Datei, und zwar *rückwärts* (aktuelle Version im Volltext)!

Beispiele:

- Historisch: **SCCS** ("Source Code Control System", AT & T System V Unix), **RCS** ("Revision Control System", andere Unix-Dialekte) (nur lokal)
- Lange Zeit führend: **CVS** ("Concurrent Version System"), **Subversion** (beide Open Source, Remote- und Multi-User-Fähigkeit)
- Kommerziell: **Perforce**, IBM/Rational **ClearCase**, MS Visual **SourceSafe** (alt) / Team Foundation Server (neu), **BitKeeper** (verteilt)
- Aktuell im Open-Source-Bereich: **GIT** (verteilt, ursprünglich für den Linux-Kernel entwickelt, heute für zehntausende Projekte eingesetzt)
- Andere: Mercurial, Monotone (verteilt)
- Einige **Webdienste** (z.B. GitHub, Sourceforge oder berliOS) bieten über das Internet eine Versionsverwaltung für freie (und gegen Entgelt für kommerzielle) Projekte.

Terminologie:

- **Repository**: Ablage aller Files und Verwaltungsdaten
- **Checkout**: Herauskopieren einer Datei zwecks Änderung, ev. mit Sperre
- **Checkin / Commit**: Speichern einer geänderten Datei, Anlegen einer neuen Version
- **Merge**: Übernahme von Änderungen aus einem Branch in einen anderen / aus einem verteilten Repository in ein anderes

  Im Idealfall (nicht kollidierende Änderungen) vollautomatisch, sonst mit händischer Unterstützung (grafisch)

  Sonderfall **"Three Way Merge"**: Nicht zwei Files miteinander abgleichen, sondern die Änderungen zwischen zwei Files in einen dritten File einarbeiten.

Wichtige Features:

- Verwaltung mehrerer Projekte
- Datenbank (Vorteil oder Nachteil???)
- Netzwerk-Zugriff, Client-Server-Architektur
- Commandline- bzw. Batch-Zugriff (für automatische Checkouts für die Builds) sowie Zugriff mit GUI (Versionsbaum!)
- Client-Plugins für IDE's
- Web-Interface
- Remote Checkout / Checkin / Clone (über HTTPS oder anderes "firewall-gängiges" und sicheres Protokoll!)
- Rechte-Verwaltung (wer darf was ändern?)
- Verwaltung auch von Binär-Dateien
- Hooks für eigene Programme / Skripts bei Checkin und Checkout

(z.B. automatische Qualitätssicherung, Verständigung des Projektleiters, ...)

- Export- / Import-Fähigkeiten der Versionsgeschichte
- *"Blame Tool":* Anzeige von Fileinhalt mit Version, Datum, Autor neben jeder Zeile
- **Verteilte Versionsverwaltung:**
  - Kein zentrales Repository mehr, Dateien und Verwaltungsinformation (Versionsgeschichte) auf mehrere Standorte verteilt
  - Nicht jeder Standort benötigt das komplette Repository
  - Lokale Checkins und Checkouts ohne Server-Verbindung möglich
    ==› Sperren-freies Arbeiten
    ==› nachträglicher Abgleich / Merge mit anderen Standorten

Was wird eingecheckt?

- Alle **Sourcen** (incl. Makefiles, ...) (auch Icons, ...)
- Die **Entwicklungs-Tools** (Compiler, ...)
- Alle **Fremd-Libraries** usw.
- Alle **Test-Sourcen** und -Scripts, alle **Testfälle** / **Testinputs**
- Die **Dokumentation** + **Doku-Tools**

Andere Tools für Patches, Versionshandling usw.

Diff & Merge:

- Vergleicht 2 Dateien / Verzeichnisse.
- Liefert zeilenweise Unterschiede in verschiedenen Formaten.
- Gleicht die Unterschiede wenn möglich automatisch ab.
- Sowohl für die Commandline als auch mit GUI.

Three Way Merge:
Sonderform, spielt die Unterschiede zweier Dateien / Verzeichnisse
in einer dritten Datei / einem dritten Verzeichnis ein.

Kriterien grafischer Tools:

- Zeichenweise grafische Darstellung der Unterschiede in den einzelnen Zeilen.
- Geh zum nächsten / vorigen Unterschied.
- Händisches Einfügen oder Entfernen von Unterschieden per Tastatur und Maus, automatisches Mergen aller konfliktfreien Unterschiede.
- Händische Editierbarkeit der Files.
- Händisches Alignment der Differenzen in der Darstellung.
- Mehrere Vergleiche (ganzer Verzeichnisbaum) gleichzeitig.
- Ignorieren von Zwischenräumen / Leerzeilen / unterschiedlichen Zeilenenden / Groß- und Kleinschreibung.

"**patch**":

Kleines Tool, um Änderungen in Source-Code-Verzeichnissen einzuspielen.
Im wesentlichen die Merge-Seite eines 3-Way-Merge:

- Bekommt diff-Outputs (d.h. geänderte Zeilen) als Input
  und spielt die Änderungen in einem lokalen Verzeichnis ein.
- Erkennt geringfügig verschobene Änderungen automatisch.
- Legt nicht einspielbare Änderungen zum händischen Nachziehen ab.

Dokumentation

Grundsätze

- Es ist strikt zwischen *interner Doku* (Entwickler- / Tester-Doku) und *externer Doku* zu unterscheiden. Da externe Doku normalerweise von einer *eigenen Doku-Abteilung* erstellt wird, befassen wir uns hier primär mit interner Doku.
- Interne Doku ist im Normalfall vom jeweiligen *Entwickler* (bzw. dem Spezifikationsteam) zu erstellen.
- Interne Doku sollte so viel wie möglich *direkt aus dem Source* (unmittelbar bei den zu dokumentierenden Code-Elementen) kommen und *so wenig wie möglich in separaten Files* erstellt werden.

  Gründe:

  - Der Code muss in jedem Fall dokumentiert werden, unabhängig von einer eventuell vorhandenen zusätzlichen Doku.
  - Kommentare neben dem Code werden bei Code-Änderungen vom Entwickler meist gleich mitgeändert.

    Doku in externen Files wird meist vergessen und *weicht im Laufe der Zeit immer mehr von der tatsächlichen Codebasis ab* (und nichts ist schädlicher als falsche / unvollständige Dokumentation).
  - Auf den Source und die daraus erstellte Doku schaut jeder automatisch. Doku in externen Dateien ist *viel schwerer zu finden* bzw. es ist nicht auf Anhieb ersichtlich, dass es neben der Source-Doku noch weitere Information zu einem Code-Stück gibt (und wo es sie gibt).
- Office-Software (Word, Openoffice) ist *kein* sinnvolles Werkzeug zur Erstellung interner Doku.

  Gründe:

  - Doku im heute erwarteten Stil (mit *vielen anklickbaren Querverweisen* und vielen einzelnen kurzen Seiten *statt eines "Spaghetti-Textes"*) ist mit Word schwer zu erstellen.
  - Word-Doku ist online schlecht zu lesen (fixe Seitenformatierung, Hochformat, passt sich nicht der Fenstergröße an usw.).
  - Word-Dokumente können von der Source-Code-Versionsverwaltung *nicht sinnvoll versionsverwaltet* werden: Weder die Versionsverwaltung noch die üblichen Entwickler-Tools können sinnvolle (sprich zeilenweise!) Änderungen zwischen den Versionen eines Word-Dokumentes anzeigen, die Versionsverwaltung muss jedesmal die gesamte Datei statt nur die Deltas speichern.
  - Auch die üblichen Entwickler-Tools für Source- und Text-Files (Suchen, Ersetzen, ...) können mit Word nichts anfangen (und automatisierte / gescriptete Massenersetzungen kommen relativ häufig vor!), Source-Code-Indexer meiden Office-Dokumente ebenfalls.

- Source-Doku kann viele tausend Seiten lang sein. Word hat Probleme im Umgang mit solchen Datei-Größen und Probleme mit stark strukturiertem Text.
- Proprietäre Formate veraltern schneller, HTML, LaTeX und PDF sind "langzeitstabil". Interne Doku wird mit der Version archiviert und muss auch in 15 Jahren noch lesbar und bearbeitbar sein!
- Programmierer haben meist sehr individuelle Gewohnheiten betreffend Editor und sind mit "ihrem" Editor besonders produktiv
  ==› Sie wollen auch die Doku mit ihrem Programmier-Editor erstellen!
- Bei Open-Source-Projekten: Doku sollte plattformunabhängig und ohne spezielle Software lesbar sein.

Siehe dazu auch meine Unterlagen AIK / Office!

- Es gibt kein allgemein anerkanntes Doku-Format für separate Doku-Files.
  - Im universitären Bereich und in der Open-Source-Welt hat *LaTeX* eine sehr starke und seit Jahrzehnten etablierte Position, es ist auch sehr fähig und mächtig, aber gewöhnungsbedürftig.

    Für mathematische Texte (Formeln) ist es bis heute unerreicht.
  - Verwandt: *TeXinfo* (TeX-Makro-Aufsatz des GNU-Projektes). Erzeugt TeX, HTML und Info-Files (Info ist das klassische, rein Text- bzw. Terminal-basierte, interaktive Hilfetool und File-Format für GNU-Software-Doku).
  - *Docbook* (XML) ist meines Erachtens zu mühsam zu erstellen.
  - *AsciiDoc* ist ein sehr interessanter und vielversprechender, aber noch junger Ansatz.
- Sowohl die *Online-Doku zum Durchklicken als auch die gedruckte Doku* sollte *aus denselben Quellen mit demselben Tool* erzeugt werden (*inhaltsgleich* sein).

  Dieses Tool sollte ohne menschliches Zutun im Batch (von Skripts aus) laufen: Gemeinsam mit dem Nighty Build sollte auch jede Nacht die aktuelle Doku generiert werden!

Dokumentationsgeneratoren

Zweck: Erzeugung einer *kommentierten Klassenreferenz.*

- Ursprünglich entstanden als Erweiterung von `cxref` (Cross-Reference-Generator für C, d.h. ein Tool, das automatisch Listen von Funktionen usw. sowie Querverweise zwischen Deklaration, Definition und Verwendung bzw. zwischen Aufrufer und Aufgerufenem erstellt).
- Wichtige Vertreter heute: **Doxygen, Javadoc, Robodoc**…
- Fähigkeiten:
  - Extraktion von Doku *aus dem Code und speziellen Kommentaren im Source*.
  - Spezielle Formatierung von Funktionsparameter- und Returnwert-Doku.

- Neben Dokumentation von Klassen und Membern / Methoden:
  Doku der Typen, Funktionen, Konstanten und Enumerationswerte, globalen Variablen, Makros, ...
- Automatische Erzeugung von *Listen und Inhaltsverzeichnissen* (Files und Klassen eines Projektes, Elemente eines Files, Members einer Klasse, ...) mit Kurzbeschreibungen, Sortierung.

  Die Doku eines Elementes sollte sowohl über die Klasse
  als auch über den Source-File erreichbar sein,
  d.h. es sollte zwei getrennte Inhalts-Strukturen geben!
- Automatische Erzeugung von alphabetischen *Indices* aller Elemente.
- Automatische Erzeugung von *Abhängigkeits-Graphen*:
  - Call-Graphen von Funktionen
  - Ableitungs-Graphen von Klassen (die Klassenhierarchie sollte auch in textueller Form mit Einrückung bzw. aufklappbaren Unterknoten erzeugt werden)
  - Include-Graphen von Header-Files.
- Automatische Erzeugung von *Querverweisen* zum Anklicken.
- Bei großen Projekten: Strukturierungsmöglichkeit in Gruppen (z.B. bei einer Klassenbibliothek wie Qt: Grafik, Datenstrukturen, Multithreading, I/O, Datenbanken, ...).
- Optional: Pretty-Print des Sources (oder zumindest der Header) mit klickbaren Querverweisen von der Doku in den Source und umgekehrt.

- Erzeugen HTML und LaTeX bzw. PDF, teilweise auch RTF etc.

Andere Tools zur Doku und zum Verstehen von Code

- *Pretty Printer* ("Andre Simon's highlight", "GNU source highlight", "code2html", "webcpp", "chilight", ...) : Generieren *HTML* aus Sourcecode, um Sourcecode im selben Layout, aber mit verschiedenen Farben, Schriftarten, Fettdruck usw. auf Webseiten etc. besser darzustellen.
- Analoge *Pretty-Printer* gibt es auch für *gedruckte Ausgabe* (z.B. "a2ps").
- *Source Formatter* ("indent", "astyle", "uncrustify", ...) : Reformatieren Source Code nach bestimmten, einstellbaren Vorgaben (haben meist dutzende / hunderte Optionen und einige gebräuchliche Stile vordefiniert), d.h. sorgen für einheitliche Einrückung, einheitliche Zwischenräume, einheitliche Anordnung von { } und Kommentaren, Zeilenumbrüche bei zu langen Zeilen, ...

  Anwendung:
  - Zur Durchsetzung eines firmenweit einheitlichen Stils.
  - Zur Aufbereitung / Lesbarmachung fremder oder alter Sourcen.
  - Für Leute, die lieber "Freestyle" editieren und dann automatisch formatieren.
- Interessantes Hilfsmittel: "UniversalIndentGUI":

    - GUI-Aufsatz für ein knappes Dutzend Formatter
    - Erlaubt das Einstellen der wichtigsten Optionen im GUI.
    - Zeigt die Wirkung sofort an einem Source-File.
- Alte Textmode- bzw. Batch-Tools, generieren Listen in Plain-Text-Format:
    - ”cflow” (generiert Aufruf-Hierarchie: ”calls” / ”called by”)
    - ”cscope” (Querverweise Definition / Verwendung, zur textbasierten, interaktiven “Erforschung” unbekannter Sourcen, ruft gleich den Editor auf)
    - ”ctags” (generiert Liste von Definitionen / Deklarationen, oft intern in Editoren verwendet)

Make

make *automatisiert das Compilieren* (großer) Projekte:

- Es erzeugt intern einen *Abhängigkeitsgraph* der gewünschten Output-Files von den dazu notwendigen Input-Files.
- Es prüft, welche der benötigten Files überhaupt noch *fehlen*, und vergleicht das *File-Datum* aller vorhandenen direkt voneinander abhängigen Files (potentielles Problem bei Netz-Laufwerken usw. mit ungenauen Uhren!).
- Es baut *genau das neu, was notwendig ist / geändert wurde* (und nicht mehr!).
- Es kennt Abhängigkeiten (welcher File braucht zum Bauen welche Input-Files?)
  ==› Es kann voneinander unabhängige Files *parallel* compilieren! (==› schneller!)
- Es kann sich selbst rekursiv aufrufen
  ==› Es kann Unterverzeichnisse, Teilprojekte, … erzeugen.

make arbeitet *nur nach File-Datum*, aber greift *nicht auf die File-Inhalte* zu
==› Es ist nicht auf C/C++ beschränkt, sondern kann für beliebige Aufgaben verwendet werden, bei denen ein Programm einen Outputfile X aus Inputfiles Y1, … erzeugt.
==› Es wird nicht nur zum Compilieren verwendet, sondern auch zum Installieren, Aufräumen, Testen, Doku erstellen, `.tar`-Archiv erstellen, …

Gesteuert wird make vom "**`Makefile`**". Dieser wird pro Projekt
(bei größeren Projekten eventuell pro Verzeichnis) erstellt und enthält:

- Die *Abhängigkeiten ("Dependencies")*:
  `target : prerequisites` (d.h. *erzeugter File* `:` *dazu notwendige Files* ,
  z.B. `main.o : main.c mytypes.h mylib.h`)

  Das sind normalerweise explizite Filenamen, können aber auch sein:

  - Patterns, Wildcards und allgemeine Regeln für bestimmte Extensions ("wie erzeuge ich allgemein einen `.o`-File aus einem `.c`-File?")
  - *Pseudo-Targets*, die *keinem* File entsprechen:
    `all`, `clean`, `doc`, `test`, … (z.B.: `all : prog1 prog2 doc`)
- Zu jeder Abhängigkeit optional die *Befehle* (z.B. Compiler-Aufruf, Doxygen-Aufruf, Install-Befehl, bei `clean` Remove-Befehl, bei `test` Aufruf der Testprogramme, …), um aus den Prerequisites das Target zu erzeugen:
  `\t` *command* (*jede Befehlszeile muss mit einem echten Tab beginnen!*)

  command wird der jeweiligen Shell zur Ausführung übergeben.
- Eine Abhängigkeit + dazugehörige Befehle wird auch "*Regel*" ("Rule") genannt.
- *Variablen*-Definitionen und -Verwendungen:
  In vielen Fällen werden sowohl in den Abhängigkeiten als auch in Befehlen Variablen statt expliziten Filenamen und Befehlen eingesetzt:

  - Für *Gruppen von Files*, z.B. alle `.c`-Files, alle `.o`-Files, alle Executables, gemeinsame Header von allen Files, …
    Viele make-Implementierungen bieten auch Möglichkeiten, z.B. aus einer Variable mit allen `.c`-Files eine Variable mit allen `.o`-Files zu erzeugen.

    - Für die aufzurufenden *Befehle*: Compiler, Linker, Install-Programm, …
    - Für deren *Optionen*: Compiler-Optionen für Debug-Build / für Release-Build, Suchpfade für System-Header und Libraries, Liste der zu linkenden Libraries, Zielpfad, Owner und Permissions für Installs, …

  Für diese Variablen haben sich standardisierte Namen eingebürgert: CC, CFLAGS, …

  Daneben gibt es vom make-Standard vordefinierte *Pseudo-Variablen*, die in jeder Regel *automatisch* definiert sind, z.B. für den *Namen des Targets*, für die *Liste aller Prerequisites*, aller zeitlich neueren Prerequisites, oder des ersten Prerequisites, …
- Weiters erlauben modere make-Implementierungen noch andere Konstrukte, z.B. if's oder include's von anderen Makefiles.

Die meisten make-Implementierungen enthalten intern umfangreiche *vordefinierte Regeln und Variablen*: In ganz einfachen Fällen (1 C-Source, 1 Executable) reicht ein Aufruf ganz *ohne Makefile*, in vielen Fällen reicht es, *nur die Abhängigkeiten* in den Makefile zu schreiben: Für die dazugehörigen Befehle genügen die vordefinierten Regeln.

Der Makefile kann:

- *manuell* erstellt werden,
- manuell mit Hilfe von *Dependency-Generatoren* erstellt werden (siehe unten),
- *von der Entwicklungsumgebung* aus Projektinformationen und in GUI-Konfigurationsdialogen festgelegten Einstellungen automatisch generiert werden,
- oder mit Commandline-Werkzeugen automatisch generiert werden, z.B. vom configure-Befehl der *GNU autotools* (siehe unten) aus einem Template oder von cmake aus Konfigurationsdateien.

make kann explizit bzw. gescriptet (ohne Benutzer-Interaktion) aufgerufen werden und wird von fast allen IDE's / Editoren intern gestartet, wenn man "Build" aufruft. Aufruf meist nur "make", optional mit explizitem `-f` *Makefile* und explizitem Target (zu erstellendem File), sonst wird "Makefile" im aktuellen Verzeichnis verwendet und das erste in diesem File enthaltene Target gebaut. Aufruf im Batch meist im "keep going"-Modus ("mach trotz Fehlern weiter, so lange es geht"), um möglichst viele Fehler in einem Lauf zu finden.

Alternativen:

- **ninja**: Siehe unten bei **meson**.
- **jam**: Nur für C / C++:
  Scannt Inhalt der Source Files für Abhängigkeiten (#include usw.)
  ==› braucht weniger explizite Angaben.
- **ant**: Standard Java Build Tool.
- IDE-interne Project / Build Tools ohne make
  (vermeiden!!! Build *ohne* IDE im *Batch* muss möglich sein!).

Problem bei make:

- Rund 1 Dutzend verbreitete, *inkompatible* make*-Implementierungen / Erweiterungen*.
- POSIX standardisiert make,

aber der *POSIX-Standard ist für die Praxis zu wenig mächtig*.

Trotzdem:
Es hat sich relativ einheitlicher Stil eingebürgert (wie C Programmier-Richtlinien)
(siehe "GNU Makefile Conventions", "Autoconf Portable Makefile Guide", …)

Hilfsmittel: **Automatisches Erzeugen der Abhängigkeiten**: `gcc -M` oder `makedepend`

Die "Autotools"

Die Autotools sind eine Entwicklung von GNU, die helfen soll, portable Software und portable Makefiles zu erstellen.

Sie bestehen aus zwei Schritten:

- Der erste Schritt geschieht *beim Entwickler beim Erstellen der Software*: Er erstellt *Makefile-Gerüste*, die üblicherweise nur eine Auflistung der projektspezifischen Files und ihrer Abhängigkeiten enthalten, und einen *Konfigurationsfile* für die Autotools, der u.a. eine Beschreibung des Projektes und eine *Liste aller potentiell plattformabhängigen Konstrukte* in dessen Sourcen enthält (aus einem Katalog von einigen tausend Konstrukten, die die Autotools testen und konfigurieren können).

  Ein Tool hilft beim Auffinden potentiell unportabler Konstrukte in C/C++-Code, alle diese Code-Stellen werden mit entsprechenden *Makros* versehen.

  Die Autotools erstellen daraus einerseits *das* `configure`*-Script* für den zweiten Schritt (das genau die für dieses Softwarepaket benötigten Tests enthält), und andererseits die *Templates*, die `configure` zur Erzeugung der endgültigen Makefiles und Headerfiles braucht.
- Der zweite Schritt besteht im Wesentlichen aus dem Programm `configure`. Es wird auf der *Zielplattform* als Vorstufe zum Bauen der Software ausgeführt.
  - Es erlaubt einerseits, die Software und ihre Installation durch Optionen beim `configure`-Aufruf anzupassen (Inkludieren oder Weglassen optionaler Programm-Komponenten, Angabe des Installationsverzeichnisses, Konfiguration für Cross-Compile-Builds, Angabe speziller Compiler-Flags, …).
  - Andererseits führt es jede Menge *automatischer Tests* durch (z.B. durch Kompilieren und Ausführen kleiner Testprogramme), um die *Eigenheiten des Zielsystems* zu ermitteln (Verhalten kritischer Library-Funktionen, Ort und Inhalt von Header-Files, vorhandene System-Tools und deren Optionen, Länge und Byte Order von Datentypen, …).

  Aus beidem generiert es dann
  - einen *Makefile* pro Verzeichnis
  - sowie einen *C-Headerfile* mit allen plattformabhängigen Defines.

  Als Vorlage für diese Files dienen die im Schritt 1 erstellten Templates.
- Im dritten Schritt wird wie bisher mittels `make` der eigentliche Code übersetzt.

  Der C-Code sollte dabei für alle potentiell inkompatiblen / plattformabhängigen Dinge die Makros und Funktionen aus den von den Autotools generierten Files (im Besonderen aus dem von `configure` erstellten Header) verwenden.

Außerdem umfassen die Autotools `libtool` (ein Tool, das das Erzeugen und Linken von Shared Libraries auf verschiedenen Plattformen vereinheitlichen soll, indem es zu jeder Library einen Beschreibungs-File anlegt und alle Compiler- und Linker-Aufrufe in ein Skript verpackt) und `gettext` (ein System zum Internationalisieren aller Texte in einem Programm).

cmake

Ein alternatives Tool, das immer mehr an Bedeutung gewinnt, um aus Plattform-unabhängigen Konfigurationsdateien Makefiles und Header für verschiedene Plattformen zu generieren, ist `cmake`. Es generiert auch Makefiles für automatische Tests des Kompilats und die Installation.

Alternativ kann `cmake` statt `make` auch `ninja` als Backend verwenden.

meson / ninja

Baut Projekte in verschiedenen Sprachen aus Projektbeschreibungs-Files
ohne Verwendung von `make` oder Makefiles:

- **`meson`** ist das für den Menschen sichtbare Tool,
  das aus der High-Level-Definition des Projektes
  die Low-Level-Build-Beschreibung generiert.
- **`ninja`** ist ein auf Effizienz / Performance optimiertes Build-Tool ähnlich `make`,
  dessen Inputfile-Syntax aber nicht dafür gedacht ist,
  dass der Mensch direkt `ninja`-Files schreibt,
  sondern optimiert ist für von anderen Tools automatisch generierte Build-Files
  (`meson`, `cmake` usw.).

Compiler (incl. Preprozessor, Assembler, Linker, Hilfstools)

Ein C/C++-Compiler sollte u.a. folgende *Möglichkeiten* bieten:

- Preprozessor: Ausgabe der Dependencies (für Makefiles).
- Preprozessor, Linker: Wegdefinieren / Umdefinieren der System-Header-Verzeichnisse / System-Library-Verzeichnisse / System-Startup-Codes.
- Preprozessor: Ausgabe der C-Sourcen nach dem Preprozessor (zum Finden von Problemen in Makros oder in den Header-Files), dabei Erhaltung der Formatierung und der Kommentare und Einfügung der ursprünglichen File-Positionen.
- Preprozessor: Listing aller definierten Makros.
- Wählbares temporäres Verzeichnis (aus Platz- und aus Sicherheitsgründen).
- Festlegen des Source-Zeichensatzes (Unicode, ISO Latin, ...).
- Optionale Ausgabe möglichst vieler Warnings für dubiose Programmkonstrukte, Qualität und Verständlichkeit von Fehlermeldungen und Warnings (z.B `gcc`/`llvm`).
- Ausgabe von Warnings für alle nicht hundertprozentig Standard-konformen Konstrukte (der anzuwendende Standard sollte wählbar sein!).
- Wählbarer `char`-Typ (man sollte beides (`signed` / `unsigned`) testen können!).
- Ausgabe der Assembler-Sourcen mit Querverweisen zum C-Source (wertvoll zur Analyse der Performance und zum Nachweis von Compiler-Fehlern).
- Debug-Output auch für optimierten Code (um Fehler zu analysieren, die nur bei Optimierung auftreten: "Heisenbugs").
- Separierte Debug-Information (nicht im Executable, sondern in getrenntem File) (bei `gcc`: Durch separate Tools).
- Erzeugung zusätzlicher Laufzeit-Checks (Integer Overflow, Stack Check, ...).
- Strict inlining Mode (genau das, was im Programm steht), bzw. gar kein Inlining.
- Reduzierter C++-Runtime-Code (keine Exceptions, keine RTTI, ...) für kleine Systeme.
- Genaue Festlegung der Ziel-Hardware (betreffend Befehlssatz und Optimierung).
- Unterstützung von Profiling und Coverage-Analyse (siehe folgende Kapitel).
- Unterstützung von Feedback-Optimierung ("Profile Guided Optimization").
- Unterstützung von Code-Optimierung beim Linken (damit über File-Grenzen hinweg optimiert werden kann) (`.o`-Files und Lib's enthalten dann interne Compiler-Datenstrukturen!)
- Linker: Strip (entfernen der Debug-Information).

Pragmatische Hinweise:

- Es spart viel Mühe, wenn der produktive Code *für alle Plattformen vom selben Compiler* (meist `gcc`) erzeugt wird.
- Auch wenn jeder Compiler behauptet, hundertprozentig ABI-kompatibel zu sein: *Das Zusammenlinken von Compilaten mehrerer Compiler ist riskant!*

- Es ist generell sinnvoll, den Compiler mit möglichst *hohem Warning-Level* laufen zu lassen! **Der Compiler ist die wichtigste Hilfe gegen Banal-Fehler, und Warning-freier Code ist der erste Schritt zu guter Code-Qualität!**

  Umgekehrt: In einem Programm, das hunderte Warnings erzeugt, fällt eine (vielleicht entscheidende) mehr oder weniger nicht mehr auf ==› wird übersehen!
- Die meisten Compiler finden mit Optimierung mehr (und wichtigere!) Warnings (z.B. wegen globaler Datenfluss-Analyse: Lesen uninitialisierter Variablen)!
- Es ist der Codequalität zuträglich, den Source zum Test von *mehreren* Compilern übersetzen zu lassen (sie finden oft unterschiedliche Probleme: 4-Augen-Prinzip). Das vermeidet auch das versehentliche Verwenden Compiler-spezifischer Sprach-Erweiterungen.
- *Precompiled Header* beschleunigen zwar das Compilieren großer Projekte, aber machen teilweise noch viel Ärger wegen unausgereifter Implementierungen…

Tools im Compiler-Umfeld

- **Compiler-Compiler**:

  Erzeugen Code für lexikalische Analysatoren / Syntax-Analysatoren aus Syntax-Beschreibungen mit Code-Annotationen.

  Beispiel: `lex`, `yacc` (siehe 3. Semester)
- **Compiler-Caches**:

  Diese sind meist Wrapper-Skripts um den Compiler und speichern bei jedem Compiler-Lauf alle Optionen sowie Prüfsummen aller Input-Files sowie alle Listings und Output-Files auf der Platte. Wird der Compiler ein zweites Mal mit identen Inputs aufgerufen, starten sie nicht nochmals den Compiler, sondern liefern direkt den gecachten Output.

  Beispiel: `ccache`
- Tools zum **verteilten Compilieren**:

  Für große Software-Pakete ist es nicht nur sinnvoll, mehrere Compiler-Aufrufe parallel zu starten, sondern auch, die Compiler-Aufrufe auf mehrere / viele Server zu verteilen.

  Arbeitsweise: Entweder wird nur der Compile-Step verteilt (nicht der Preprozessor oder Linker), dann sind keine gemeinsamen Filesysteme nötig: Es muss nur der aktuelle C-Source auf den Server geschickt werden. Oder es gibt gemeinsame Filesysteme, dann können auch die Präprozessor- / Linker-Schritte verteilt werden.

  Beispiel: `distcc`

Tools für Objects, Libraries und Executables

- Löschen aller nicht benötigten Symbole und Debug-Infos (`strip`).
- Anzeigen der Code- und Daten-Segmente und ihrer Größe.
- Anzeigen und Manipulieren der einzelnen Objekte in einer Library (`ar`).

- Anzeigen der Symbole und deren Adresse (Link Map) / Anzeige der undefinierten Symbole in einem `.o`-File / Anzeige einer Link Cross Reference (`ld`, `nm`).
- Decodierung des C++ Name Manglings (Typcodierung in Methodennamen).
- Probe-Loaden: Welche Shared Libraries werden benötigt / geladen?
  Welche Objekte fehlen in Shared Libraries? (`ldd`)
- Durchsuchen von Binärfiles nach Strings (`strings`).

Unter Linux sind diese Tools (zusammen mit Assembler und Linker) im Paket `binutils` (und teilweise `elfutils`, …) zusammengefasst.

- Decompiler: Versucht, aus Binaries (und Debug-Info) wieder compile-fähigen Sourcecode zu generieren. Anwendungen: Computer-Forensik (Analyse von Schadcode), Fehlersuche in fremdem Code.
  Bei C/C++ ohne Debug-Info eher unbefriedigend, bei Java besser…

Sonderfall Cross-Compiler

Ein Cross-Compiler erzeugt Code für eine *andere Prozessor-Plattform* oder eine *andere Betriebssystem-Plattform* als diejenige, auf der er selbst ausgeführt wird (Beispielsweise Entwicklung für einen ARM-Prozessor mit Echtzeit-Betriebssystem auf einem x86-Windows-PC-Arbeitsplatz).

Neben dem Compiler, Assembler und Linker müssen auch die anderen Binär-Tools (`strip`, `nm`, …) in Cross-Versionen für die Zielplattform vorliegen, ebenso der Debugger. Diese Tools müssen statt auf die System-Header und -Libaries des Host-Systems auf *lokale Kopien der Header und Libraries des Target-Systems* zugreifen.

Im Zusammenhang mit `make` und den Autotools ergeben sich weitere Komplikationen:

- In den Makefiles müssen *getrennte Tools und getrennte Variablen für Compiler-Flags, zu linkende Libraries usw.* verwendet werden, je nachdem, ob im jeweiligen Schritt Binärfiles für das Zielsystem oder Binärfiles für lokal laufende Hilfsprogramme erzeugt werden sollen.
- Auch in den Autotools muss diese Trennung zwischen Tools und Settings für lokale Compilierung und solche für Cross-Compilierung berücksichtigt werden, typischerweise müssen zwei getrennte Sätze von `#define`'s generiert werden.
- Weiters kann "`configure`" seine Hauptaufgabe, nämlich Features und Eigenheiten des Zielsystems betreffend Header-Files, Libraries, Datentypen (z.B. Pointer-Länge und Byte Order) usw. automatisch durch Ausführen kurzer Testprogramme zu ermitteln, nur mehr stark eingeschränkt erfüllen: Das Ergebnis lokal laufender Testprogramme entspricht nicht notwendigerweise den Verhältnissen am Zielsystem.

Es gibt vor allem im Linux-Bereich fertige Software-Systeme zur Cross-Generierung kompletter Linux-Systeme incl. aller Anwendungen (meist basierend auf sehr komplexen Makefiles, Beispiel "`buildroot`" oder "OpenEmbedded"), die alle zur Cross-Entwickung nötigen Schritte automatisieren:

- Sie laden die Sourcen aller benötigten Komponenten aus dem Internet.
- Sie erstellen zuerst einmal einen Cross-Compiler und alle notwendigen Tools.

- Sie konfigurieren und compilieren alle für das Zielsystem nötigen Software-Komponenten, vom Kernel über die Libraries bis zu den Anwender-Programmen.
- Sie generieren ein bootfähiges Filesystem-Image, das auf die Platte oder den Flash-Speicher des Zielsystems (oder z.B. einen USB-Stick) kopiert werden kann.

Fehlersuche und Analyse des Programm-Verhaltens

Der Debugger

Ein Debugger kann ein laufendes Programm anhalten, seinen Zustand analysieren, und es *schrittweise* weiterlaufen lassen.

Dabei bietet ein Debugger normalerweise folgende *Ansichten*:

- Aktuelle Position im *Source*, aktueller Assembler-Code.
- Aktueller *Call-Stack*
  (Funktionen incl. Argument-Werten und Zeilennummern der Aufrufe).
- Werte aller lokalen und globalen *Variablen*
  (lokale Variablen für alle Ebenen im Call-Stack).
- Registerinhalte, "rohe" *Speicherinhalte* (in Hex / in Dezimal / als Text).

Wenn der Debugger symbolische Informationen (Variablen- und Funktions-Namen, Zeilennummern) liefern und die Typen von Variablen und Funktionsparametern kennen und richtig anzeigen soll, muss der Code mit Debug-Info compiliert worden sein (`gcc`: Option `-g`).

Da bei Optimierungen Code verdoppelt, verschoben, inline eingesetzt usw. wird und ev. sogar Schleifenvariablen wertmäßig transformiert werden, entspricht der Ablauf in einem optimierten Programm nicht mehr dem Source. Trotzdem ist es ein wichtiges Feature, dass auch bei voller Optimierung noch verwendbare Debug-Info erzeugt wird!

Die *Fähigkeiten* der meisten Debugger sind heute sehr ähnlich:

- *Breakpoints*, Continue to next Breakpoint, Run to Cursor.
- Single *Stepping* (eine Source-Zeile, ein Assembler-Befehl),
  Step Over (eine Zeile, wenn sie Funktionen aufruft bis nach dem Call),
  Step Into (eine Zeile, wenn sie Funktionen aufruft in jeden Call hinein),
  Step Out (bis zum Return der aktuellen Funktion).
- *Watchpoints* (Programm läuft, bis eine bestimmte Variable sich ändert oder einen bestimmten Wert annimmt).
- *Ändern* von Variablen und Speicherbereichen.

Qualitative Unterschiede bestehen vor allem in der *Datendarstellung*:

- Werden Variablen *im richtigen Typ* angezeigt, werden die Inhalte von Arrays und Structs / Objekten richtig dargestellt?
- Bekommt man *im Source* die aktuellen Variablen-Werte als *Tooltip*?
- Kann man *Substrukturen von Daten mit Mausklick öffnen* und wegklappen,
  kann man *Pointer mit Mausklick verfolgen*, auch über mehrere verkettete Elemente?
- Wie werden verkettete Datenstrukturen dargestellt?
- Kann man beliebige C-Ausdrücke symbolisch berechnen?
  Können anzuzeigende Variablen und Watchpoints komplexe Ausdrücke (mit Indices und Indirektionen) sein?

- Kann man *Speicherinhalte symbolisch ändern*
  (d.h. mittels normaler C-Zuweisung mit beliebigen Lvalues)?

Ein weiteres Differenzierungsmerkmal ist die Fähigkeit, *Multithreaded-Programme* zu debuggen (das ist noch ein ziemlich unbefriedigendes Gebiet).

Der Debugger kann ein alleinstehendes Programm oder Bestandteil der IDE sein.
Unter Linux wird als Debugger-Engine fast ausschließlich der `gdb` verwendet,
der einerseits für sich allein mittels Commandline-Interface bedient werden kann
und andererseits eine Aufruf-Schnittstelle für GUI-Oberflächen bietet,
die von zahlreichen IDE's für integriertes Debugging genutzt wird.

Gestartet kann der Debugger normalerweise auf 3 Arten werden:

- Das zu testende Programm wird im und *vom Debugger selbst gestartet*.
- Der Debugger *verbindet sich zu einem bereits laufenden Prozess*.
- Der Debugger *analysiert ein Speicherabbild (Core Dump)*, den das Betriebssystem beim Absturz oder gezielten Abschuss eines Programmes erstellt hat.

  Das nennt sich "Post Mortem Debugging" (Sezieren der Leiche).

  Das Erstellen von Core Dumps ist heute bei den meisten Linux-Distributionen per Default abgedreht und muss erst mit `ulimit -c ...` *freigegeben* werden.

  Ctrl-\ ist im Terminal normalerweise auf `quit` gebunden und bricht das gerade laufende Programm mit den Signal `SIGQUIT` und einem Core Dump ab.
  Andere *Signale*, die normalerweise auf einen *Programmfehler* hindeuten, produzieren per default ebenfalls Dumps (`SIGILL`, `SIGSEGV`, `SIGBUS`, `SIGFPE`).
  Auch die *C-Standard-Funktionen* `abort()` *und* `assert(…)` sollten einen Core Dump produzieren, wenn sie aufgerufen werden und Cores nicht abgedreht sind.

  Wenn es nur darum geht, die Quellcode-Zeile eines `SIGSEGV` ("Urwald-Pointer" usw.) zu finden, ist es einfacher, das Programm einfach mit dem Tool `catchsegv` zu starten: Es liefert grundlegende Debug-Info im Absturz-Moment, u.a. den Call Stack.

Sonderformen des Debuggings sind

- **Remote Debugging:**

  Hier läuft der zu debuggende Code und das Debugger-Benutzerinterface auf *zwei verschiedenen Maschinen*, oft sogar auf zwei verschiedenen Hardware- oder Betriebssystem-Plattformen. Dies wird vor allem im Bereich der Embedded-Systeme und Steuerungen eingesetzt.

  Der Debugger ist in diesem Fall in zwei mittels Netzwerk, serieller Schnittstelle oder USB kommunizierende Hälften geteilt:

  - Der eigentlichen Debug-Logik, die das zu testende Programm am Zielsystem kontrolliert ("`gdbserver`").
  - Dem Debugger-Benutzerinterface am Entwickler-System.

- **Kernel Debugging:**

  Beim Debugging des Kernels wird es sich fast immer um *Remote Debugging* handeln, da das Zielsystem ja handlungsunfähig ist, wenn der Kernel angehalten

wird, und daher auch kein benutzbares Debugger-Benutzerinterface bieten kann.

Meist sind zum Kernel Debugging neben dem Debugger auch noch *eigene Device-Treiber* erforderlich, die die Schnittstelle zum Entwickler-System auch ohne laufenden Kernel bedienen können.

Alternativ wird eine *hardware-basierte Debug-Schnittstelle wie* **JTAG** verwendet (kann CPU und Speicher autonom, d.h. ohne Zutun des Prozessors, bedienen).

`strace` und `ltrace`

`strace` und `ltrace` sind nicht-interaktive Tools zum *Verfolgen des Programmablaufes*:

- `strace` hängt sich in die Schnittstelle *zwischen Programm und Kernel* und zeichnet alle (oder bestimmte) *System Calls* incl. Argumenten und Returnwert auf. Weiters zeichnet es alle *Signale* auf, die der Prozess vom Betriebssystem bekommt.
- `ltrace` hängt sich in den *Dynamic-Linking-Mechanismus* und zeichnet alle Calls von *Funktionen in allen oder bestimmten Shared Libraries* (im Besonderen alle Calls von Standard-C-Library-Funktionen) auf.

Beide liefern Logfiles, die nachher mit einem Editor durchsucht werden können.

Sie können wahlweise mit dem auszuführenden Programm aufgerufen werden (und starten das Executable dann selbst) oder sich mit einem laufenden Prozess verbinden.

Diese beiden Tools haben andere Schwerpunkte, Einsatzbereiche und Stärken als ein Debugger:

- Sie brauchen *keinerlei Debug-Information* in den Executables und liefern sogar bei Executables, zu denen man den Source nicht hat, verwertbare Informationen.

  Da `strace` die Parametertypen jedes einzelnen System Calls und deren Bedeutung fix eingebaut hat, kann es die Argumente trotzdem schön symbolisch in der richtigen Formatierung anzeigen.
- Sie liefern die komplette *zeitliche Geschichte* vom Programmstart bis zum Fehler, nicht einen Snapshot des aktuellen Status wie beim Core Dump.
- Dafür liefern sie keinen Stacktrace zu jedem Aufruf (sie können allerdings die Adresse des aufrufenden Befehls ausgeben, zu dem man dann mit Debug-Information und `addr2line` die Stelle des Aufrufs im Source ermitteln kann).
- Sie helfen oft auch dann, wenn man keine Ahnung hat, wo das Problem sitzt, und sich daher schwer tut, sinnvolle Breakpoints zu setzen. Weiters helfen sie bei Programmen, die mit einer Fehlermeldung, aber nicht mit einem Core Dump abbrechen.
- Sie können (genügend Plattenplatz für die Logs vorausgesetzt) zur Suche von Fehlern, die nach Stunden oder Tagen Programmlaufzeit oder nur sporadisch auftreten, einfach mitlaufen, ohne dass jemand ständig ein Auge darauf haben muss.
- Sie arbeiten *auch mit Multithreaded-Programmen* und Programmen, die Subprozesse erzeugen oder andere Programme aufrufen.

  Dabei können sie die Logs aller Prozesse entweder unter Erhaltung der zeitlichen Reihenfolge in einen einzigen Logfile vereinen oder einen separaten Logfile pro

Prozess anlegen.

Das hilft u.a., Race Conditions und Verklemmungen zwischen Prozessen zu finden.

Typische Beispiele für Vorgehensweisen:

- Im Log die Ausgabe der Fehlermeldung (`write`) suchen, von dort rückwärts lesen.
- Im Log Funktionsaufrufe suchen (meist von unten nach oben!), deren Returnwert einen Fehler anzeigt. Bei System Calls: Returnwert `-1`, `errno`-Wert steht daneben.
- Zugriffe auf eine bestimmte Datei suchen.

Außer zur Fehlersuche lassen sie sich auch als erste *Performance-Analyse* einsetzen:

- Sie liefern die *Dauer* jedes einzelnen Aufrufes, die *Abstände* zwischen zwei Aufrufen, oder exakte *Zeitstempel* zu jedem Aufruf.
- Sie können am Ende *Statistiken* ausgeben (Anzahl und Gesamt-Zeitverbrauch jedes Aufrufes).

`sotruss` ist ein weiteres Tool, das ähnlich `ltrace` arbeitet.

lsof

`lsof` listet alle Files, die ein Prozess (oder alle Prozesse) augenblicklich offen hat. Dazu gehören:

- Das aktuelle *Arbeitsverzeichnis*.
- Das aktuelle *Root-Verzeichnis*.
- Der *Executable File* selbst.
- Alle *zum normalen I/O geöffneten Files*, incl. Directories, *Pipes*, Devices, *Sockets*, Netzwerk-Verbindungen, … (remote NFS-Files und AFS-Files werden als solche erkannt und angezeigt, remote SMB-Files nicht).
- Alle mit `mmap` geladenen Files. Das umfasst u.a. *alle Shared Libraries*, die das Programm geladen hat, und *seine Shared-Memory-Segmente*.

`lsof` bietet zahlreiche Optionen zur gezielten Selektion bestimmter Files und Prozesse.

/proc

`/proc` ist (ebenso wie `/sys`) ein *Pseudo-Filesystem*, d.h. diese Files existieren nicht auf Platte, sondern werden vom Kernel "vorgetäuscht". In diesem Filesystem liefert der Kernel umfangreiche Informationen zum aktuellen System-Zustand. Normalerweise greift man allerdings nicht direkt auf die Information in `/proc` zu, sondern über Befehle wie `ps` (oder einen grafischen Task-Manager), `lsof` usw..

Systemweit lassen sich dort z.B. die aktuelle CPU- und Disk-Aktivität, die aktuelle Speicherauslastung und Statistiken zu den Interrupts finden.

Weiters gibt es dort ein *Unterverzeichnis pro Prozess*. In diesem Unterverzeichnis finden sich z.B. die offenen Files und vieles andere mehr. Hilfreich ist ev. die Information `maps` bzw. `smaps` (aktuelle *Belegung des Adressraumes* des Prozesses) und `status` (*allgemeine Information, u.a. Signalmaske, Memory-Statistiken und Anzahl der Threads*).

Letzte Hilfe: Magic SysRq

Falls der Linux-Kernel entsprechend konfiguriert ist, lassen sich im Notfall (wenn das System sonst nicht mehr reagiert oder kontrollierbar ist) über *Alt-SysRq-Tastenkombinationen* bestimmte System-Aktionen ausführen, u.a.

- Abschießen des Programms, das gerade den Bildschirm belegt (e.v. nützlich bei "wildgewordenen" Grafik- und Video-Programmen oder hängendem Xserver) bzw. Umschalten auf die Text-Konsole.
- Abschießen des Speicher-gefräßigsten Programms, einbremsen hochpriorer Prozesse.
- Anzeigen der aktuellen Prozesse und Speicherauslastung, der aktuellen CPU-Register, …
- Sync der Diskpuffer, unmount der Filesysteme, und Reboot.

Alt-SysRq-Tastenkombinationen werden direkt vom Kernel erkannt und bearbeitet, sie benötigen kein separates Programm.

Debugging von Speicherfehlern

Speicherfehler sind das häufigste Problem in C / C++, das zur Compilezeit unentdeckt bleibt und erst zur Laufzeit mehr oder weniger zufällig bzw. sporadisch auftritt. Sie gehören auch zu den am schwersten reproduzierbaren und analysierbaren Fehlern und sie sind für den Großteil aller Sicherheitslücken verantwortlich.

Im Wesentlichen gibt es 3 Klassen von Speicherfehlern:

- *Echte Fehlzugriffe*. Dazu gehört u.a.
    - Das Lesen und Schreiben von NULL-Pointern und uninitialisierten Pointern.
    - Das Lesen und Schreiben über Arraygrenzen hinaus ("off by one"-Fehler, unterminierte Strings, ungeprüfter Input bzw. zu klein angelegte Arrays, Überlauf bei Index-Arithmetik, …).
    - Der Zugriff auf Pointer, die auf lokale Daten zeigen, nach dem Ende der entsprechenden Funktion.
    - Mit dem Umstieg auf 64-bit-Systeme kamen auch zahlreiche Probleme durch Casts zwischen Pointern und `int`'s in Bestandscode dazu (diesbezügliche Warnings des Compilers sollten aktiviert sein und beachtet werden!).
    - Miss-Interpretation von Daten als Pointer, z.B. in Unions oder bei variabel vielen Argumenten.
    - Und unabhängig von Pointern: Lesen von Daten, die nie initialisiert wurden.
- Fehler im Umgang mit der *dynamischen Speicherverwaltung*. Dazu gehört u.a.
    - Unzulässige Pointerzugriffe nach dem `free` des Speicherbereiches.
    - Doppelte `free`-Aufrufe mit demselben Pointer.
    - `free`-Aufrufe mit Pointern, die gar nicht von `malloc` stammen (Pointer auf lokale oder globale Daten, Pointer von `alloca`, uninitialisierte Pointer),

sowie `free` von Pointern, die mitten in einen `malloc`-Block zeigen statt auf dessen Anfang.

- Weiters in C/C++: Vermischung der 3 Arten von Allokation: `malloc` / `free`, non-Array-`new` / `delete`, Array-`new` / `delete []` .

Da unmittelbar vor und nach einem `malloc`-Block `malloc`-interne Verwaltungsdaten (Längeninformation, Pointer-Verkettungen benachbarter freier Blöcke, …) stehen, sind `malloc`-Blöcke auch besonders sensibel gegen Schreibzugriffe über die Grenzen hinaus: Der nächste `malloc`- oder `free`-Aufruf läuft wegen korrupter interner Daten ins Ungewisse…

- *Memory Leaks*: Es werden nicht alle dynamisch angelegten und nicht mehr benötigten Speicherbereiche freigegeben ==› der Speicherbedarf des Programms wächst ständig, bis zum Programmabbruch wegen Speichermangels.

  Bei Speicherverwaltungen, die auf Reference Counting basieren, gehören dazu auch isolierte zyklische Pointerstrukturen.

- *Memory Fragmentation*: Das ist kein echter Programmfehler, sondern eine "ungünstige" Nutzung der Speicherverwaltung: Durch ungünstige Abfolgen verschieden großer `malloc`- und `free`-Aufrufe zerfällt der freie Speicher in so viele kleine, isolierte Bereiche, dass ein `malloc` eines größeren Blockes nicht mehr möglich ist.

Unter Linux kommt noch das Problem des "Memory Overcommitments" dazu. Da das ein Betriebssystem-Thema ist, gegen das das Anwender-Programm machtlos ist, gehen wir hier nicht weiter darauf ein.

Tools für Speicherfehler

1.) Compiler-*Erweiterungen:*

Die Idee ist, dass schon der Compiler zusätzlichen Code erzeugt, der zur Laufzeit die Gültigkeit jedes einzelnen Speicherzugriffes prüft.

Dieses Konzept erkennt einen Großteil aller Speicherfehler, aber natürlich nur innerhalb des mit Prüfungen übersetzten Codes: Ruft das Programm irgendwelche System-Library-Funktionen mit ungültigen Pointern auf und stürzt dort ab, so wird das nicht erkannt (weil im Library-Code ja keine Prüfungen enthalten sind). Weiters erkennen alle diese Systeme keine Verwendung uninitialisierter Variablen.

- Mudflap für `gcc`
  (bis Version 4.8 im Standard-`gcc` enthalten, Optionen `-fmudflap` und `-lmudflap`, nicht mehr weiterentwickelt: Abgelöst durch Address Sanitizer):

  Mudflap führt so wie die folgenden Tools Buch über jeden gültigen Speicherblock (es klinkt sich in `malloc` / `free` und in das Anlegen und Freigeben lokaler Variablen ein) und prüft jeden Pointerzugriff, hat aber keine fixe Zuordnung zwischen Pointer und Speicherblock, in den er zeigt, merkt also nicht, wenn ein Pointer von einem gültigen Bereich in einen anderen wechselt (es legt auch keine "verbotenen" Schutzwörter rund um jeden Block an), und erkennt auch manche Fälle von use-after-return nicht. Zusätzlich kann am Programm-Ende ein vollständiges Listing aller Memory-Leaks generiert werden.

Typische Programme werden um einen Faktor 5-10 langsamer.

- Jones / Kelly / Brugge / Bader Bounds Checking Patch für gcc:

  Die Idee ist folgende:

  - Es wird über jeden gültigen Speicherblock (lokale, globale und dynamische Arrays) einzeln Buch geführt, u.a. über Anfangs- und End-Adresse.
  - Jeder Pointer ist fix mit einem bestimmten Speicherblock verbunden.
  - Bei jedem Pointer-basierten Zugriff und auch schon bei jeder Pointer-Arithmetik wird geprüft, ob der Pointer innerhalb der Grenzen des Speicherblockes liegt, zu dem der Pointer ursprünglich gehört hat.

  Der Nutzen ist zweierlei:

  - Der Schutz vor Speicherfehlern ist praktisch hundertprozentig.

    Das Konzept fängt auch Pointer, die versehentlich von einem gültigen Speicherblock zum nächsten gültigen Block wandern, und es fängt sie schon beim Berechnen der falschen Pointer, noch bevor wirklich zugegriffen wird.
  - Am Schluss des Programms steht eine Liste aller noch belegten Speicherblöcke zur Verfügung.

  Die Nachteile des Konzeptes sind:

  - Der Overhead ist sehr hoch (Faktor 3 – 30 in der Laufzeit, ebenfalls Faktoren im Speicherbedarf).
  - Das Konzept wurde nur für C (nicht für C++) implementiert.
  - Diese Entwicklung wird seit 2004 nicht mehr gepflegt und ist nur für sehr alte gcc-Versionen verfügbar (2013: Funktioniert nicht mehr).
- Miro für gcc:

  Das ist ein Rework des vorigen Konzeptes für etwas aktuellere gcc-Versionen (2008). Die Memory-Leak-Funktionalität wurde entfernt, dafür funktioniert das Speicher-Checking jetzt auch für C++.
- Address Sanitizer für `llvm` / `clang` (und seit neuestem auch für gcc):

  Address Sanitizer prüft C und C++, indem es von jedem Byte im gesamten Adressraum (Heap, Stack, globale Daten, …) Buch führt, ob es gültig oder ungültig ist, und jeden Pointer-Zugriff prüft. Es ist also etwas schwächer als der Bounds-Checking-gcc: In "fremde" gültige Objekte zeigende Pointer werden nicht erkannt. Allerding legt `llvm` um jedes Speicher-Objekt ungültige Schutzzonen an, erkennt dadurch also deutlich mehr als `mudflap`. Durch eine andere Implementierung der Prüfung auf gültige Adressen (direkte Abbildung aller Adressen auf eine Byte-Weise Lookup-Tabelle statt Liste aller gültigen Blöcke) ist er aber viel schneller (Slowdown um einen Faktor 2 oder weniger statt um einen Faktor über 10), auf Kosten eines extrem hohen (allerdings nur virtuellen) Speicherbedarfs.

  Address Sanitizer ist standardmäßig in `llvm` und ab Version 4.8 auch in gcc integriert, wird also vermutlich der neue Standard (zumal Google es massiv entwickelt und einsetzt).

Im Umfeld des Address Sanitizers wurden weitere Tools entwickelt:

- Der *Memory Sanitizer* erkennt Lesezugriffe auf Speicherbereiche, die zuvor nie geschrieben bzw. initialisiert wurden.
- Der *Leak Sanitizer* liefert am Ende des Programms eine Liste aller dynamisch angelegten und nicht mehr freigegeben Speicherbereiche. Sein Vorteil ist der minimale Overhead.
- Der *Thread Sanitizer* erkennt Data Races, d.h. Zugriffe aus mehreren Threads auf gemeinsamen Speicher, deren Reihenfolge vom Zufall bzw. vom Scheduling der Threads abhängt.

- SafeCode für `llvm` / `clang`:

  SafeCode ist (war?) ein Forschungsprojekt, das (nur für C) ebenfalls ungefähr dieselben Fähigkeiten wie `mudflap` implementiert (also jeden Pointer-Zugriff auf Gültigkeit prüft), aber versucht, so viele redundante Prüfungen (d.h. Prüfungen, deren Ausgang sich aus dem vorherigen Code logisch herleiten lässt, bzw. mehrfache aufeinanderfolgende Prüfungen desselben Pointers) wie möglich zur Compilezeit wegzuoptimieren. Es soll den Slowdown ebenfalls auf rund 2 reduzieren.

2.) *Stackguard und ProPolice Stack Smashing Protector:*

Auch hier handelt es sich um eine `gcc`-Erweiterung, die inzwischen standardmäßig eingebaut wurde (`-fstack-protector`), allerdings mit einem ganz anderen Ziel: Es geht nur darum, Fehler und vor allem böswillige Angriffe abzufangen, die über lokale Arrays (z.B. Strings) am Stack hinaus schreiben und das Programm durch eine zerstörte oder bewusst manipulierte Return-Adresse am Stack zum Absturz oder zur Ausführung fremden Codes bringen.

Die Methode ist einfach: Zwischen den lokalen Arrays am Stack und der Return-Adresse werden bei jedem Aufruf einer Funktion Wörter mit bestimmten Bitmustern eingeschoben. Diese werden unmittelbar vor dem Return auf Unversehrtheit geprüft.

Außerdem werden Variablen umsortiert: Einzelne Variablen werden vor Arrays angelegt, um sie aus dem Gefahrenbereich zu bringen.

Die Performance ist (außer bei sehr kurzen Funktionen) gut (ein paar Prozent Overhead), der Schutz schon vom Design her sehr gering. Auch gegen Memory Leaks hilft der Ansatz nicht.

3.) *Valgrind:*

Valgrind verfolgt einen komplett anderen Ansatz:

- Es verarbeitet unmodifizierten Code.
- Allerdings wird der Code nicht direkt am Prozessor ausgeführt, sondern interpretiert. Valgrind ist also eine Art x86-Emulator.

  Dadurch erkennt Valgrind jeden Speicherzugriff mit seiner tatsächlichen Adresse.
- Es bietet die Möglichkeit, durch Plugins zusätzlichen Code vor oder nach jedem einzelnen interpretierten Maschinenbefehl auszuführen.

Eines dieser Plugins ist memcheck:

- Es ersetzt malloc und free durch Versionen, die über jeden aktuell allokierten Speicherbereich Buch führen. Es weiß also, welche Speicherbereiche derzeit gültig sind und welche nicht.
- Weiters führt es Buch über jede einzelne Speicheradresse: Ist sie überhaupt gültig? Wurde sich schon jemals geschrieben (initialisiert)? Ist es ein Datenwert oder z.B. eine Return-Adresse?

  Daher kann es jeden Zugriff prüfen, sowohl auf Gültigkeit, als auch, ob die Daten initialisiert oder uninitialisiert sind.
- Schließlich fängt es alle System Calls ab und überprüft die Gültigkeit der Pointer-Parameter.

Valgrind mit memcheck erkennt einen wesentlichen Teil aller Speicherzugriffsfehler und ist das häufigste Tool, um die Verwendung uninitialisierter Variablen zu finden. Da der gesamte ausgeführte Code von Valgrind interpretiert wird (inclusive aller Libraries), werden auch solche Fehler gefunden, die erst innerhalb von Libraries zu Abstürzen führen.

memcheck kümmert sich allerdings nur um dynamisch angelegte Daten: Über lokale und globale Daten wird nicht im Detail Buch geführt. Valgrind speichert auch keine Zuordnung von Pointern zu bestimmten Speicherblöcken: Es erkennt nicht, wenn ein Pointer plötzlich und unbeabsichtigt von einem gültigen in einen anderen gültigen Speicherbereich wechselt – gültig ist gültig.

Der Overhead ist sehr hoch, sowohl betreffend Laufzeit (Faktor 10 bis 30), als auch betreffend Speicherbedarf. Dafür kann Valgrind auch Code prüfen, den man nicht mit Pointerchecks neu compilieren kann oder will (weil es z.B. die System-Libraries sind, oder weil man den Source nicht hat).

Es gibt ein anderes Valgrind-Plugin sgcheck, das mit demselben Konzept wie der Bounds-Checking gcc nur die Pointerzugriffe auf lokale und globale Daten prüft (d.h. sich zu jedem einzelnen Pointer merkt, zu welchem Speicherblock er gehört und was die Grenzen dieses Blocks sind), um den Preis eines noch höheren Overheads.

4.) *Purify:*

Purify ist das bedeutendste kommerzielle Produkt. Es ist für viele Compiler und Betriebssysteme verfügbar. Die genaue Arbeitsweise ist nicht dokumentiert, aber im wesentlichen wird der Code im fertigen Compilat und in allen Libraries vor dem Ausführen analysiert und modifiziert bzw. instrumentiert (d.h. Purify versucht, im Code alle Pointer- und Array-Operationen zu finden und um Checks zu erweitern oder durch Calls in seine eigene Runtime-Library zu ersetzen, und erzeugt in eigenen Verzeichnissen neue Versionen aller exe's und dll's).

Dann macht Purify zur Laufzeit das, was auch Valgrind macht: Es führt über jedes einzelne Byte im Speicher Buch (ungültig, gültig aber uninitialisiert, gültig und initialisiert) und prüft jeden Pointer- oder Array-Zugriff.

Die Erkennungsrate ist ähnlich gut wie Valgrind, mit denselben Vor- und Nachteilen: Purify funktioniert auch auf Code ohne Source, aber es hat keine Zuordnung zwischen Pointern und den Speicherblöcken, auf die sie zeigen. Die

Performance ähnlich schlecht wie bei Valgrind.

5.) Andere Produkte:

- Fail-Safe C: Speicher-prüfender (instrumentierender) Open-Source C-Compiler.
- Parasoft Insure++:
  Ältester / mächtigster kommerzieller instrumentierender C-Compiler.
- Micro Focus BoundsChecker:
  Visual Studio Plugin für Speicherfehler-Erkennung.
- DrMemory: Valgrind-ähnliches Open-Source-Tool.

6.) Diverse Malloc-Libraries:

Es gibt Dutzende Ersatz-Libraries, die das Standard-`malloc` bzw. `new` ersetzen.

"In" sind gerade die Google Perftools, traditionelle Libraries (für Linux) sind Dmalloc, MemProf, Mpatrol, Electric Fence bzw. DUMA und einige mehr.

Auch die normale Glibc enthält (sehr rudimentäre) derartige Funktionen, die aber ausreichen, um Memory Leaks zu finden.

Diese Libraries haben vor allem zwei Ziele:

- Buch zu führen über alle Allokationen und Deallokationen, um am Ende des Programms *alle nicht freigegebenen Speicherblöcke und die Programmstelle, wo sie angelegt wurden, auflisten* zu können.

  Das geschieht meist dadurch, dass bei jedem `malloc` der Call Stack des Aufrufers (nur die Programm-Adressen) gespeichert wird, wodurch man mit Hilfe der Debug-Information die Zeile Code ermitteln kann, in der das `malloc` erfolgte.

- *Fehler* im Zusammenhang mit `malloc` und `free` zu finden.

  Das geschieht dadurch, dass beim `free` mehr Konsistenzprüfungen gemacht werden (also ein Tradeoff zwischen Laufzeit und Sicherheit), um beispielsweise doppelte `free` zu erkennen. Weiters werden rund um die `malloc`-Blöcke oft Schutzwörter angelegt und beim `free` geprüft, um Speicherüberschreiber zu erkennen (das erkennt allerdings nur Schreibzugriffe, keine Lese-Fehlzugriffe).

  Der Schutz ist allerdings einem prüfenden Compiler oder Valgrind bei weitem unterlegen: Er findet nur einen Bruchteil aller Fehler und schützt bei weitem nicht hundertprozentig vor korrupter Speicherverwaltung oder Abstürzen.

In C++ gibt es auch die Möglichkeit, die Speicherverwaltung von `new` und `delete` pro Klasse mit eigenen Funktionen zu überschreiben. Das kann beispielsweise wie folgt verwendet werden:

- Freigegebene Objekte werden nicht mehr an die C++-Heapverwaltung zurückgegeben, sondern pro Klasse in eine separate Freiliste eingehängt, aus der sie das `new` dieser Klasse (und nur dieser Klasse) wieder entnimmt. Damit erreicht man, dass einmal vom Heap angeforderte Blöcke nie mehr ihre Größe ändern und immer ein Objekt ein und desselben Typs enthalten. Dadurch zeigen "vergessene" Pointer zwar auf ein nicht mehr gültiges Objekt, aber wenigstens auf ein Objekt vom richtigen Typ und von der richtigen Größe.

Weiters kann man die Freiliste First-In-First-Out verwalten, sodass freigegebene Objekte so spät wie möglich wiederverwendet werden.

- Man kann freigegebene Objekte mit bestimmten Default-Werten füllen. Einerseits kann man dadurch im normalen Programmablauf erkennen, wenn man versehentlich auf ein schon freigegebenes Objekt zugreift, und andererseits kann man im `new` prüfen, ob das Objekt seit dem `delete` verbotenerweise nochmals geschrieben wurde.

Eine solche Speicherverwaltung hat auch den Vorteil, normalerweise zeitlich und logisch wesentlich deterministischer zu arbeiten als der programmweite C++-Heap, was die Reproduktion, Eingrenzung und Analyse von Fehlern erleichtert.

7.) Electric Fence:

Electric Fence ist ein Spezialfall einer Malloc-Library, die das Betriebssystem nutzt, um Speicher-Fehlzugriffe am Heap zu erkennen: Jeder Malloc-Block wird auf eine eigene Page der virtuellen Speicherverwaltung gelegt, und zwischen zwei gültigen Pages wird immer eine ungültige Page angelegt. Läuft ein Pointer über die gültige Page hinaus, wird vom Betriebssystem ein Seitenfehler ausgelöst.

Electric Fence bietet nur einen sehr groben Schutz, und das nur für Heap-Objekte. Es hat geringen Laufzeit-Overhead, aber extrem hohen Speicher-Overhead.

8.) Statische Analyse:

Manche Fehler (sowohl Memory Leaks als auch Speicherüberschreiber, Zugriff auf uninitialisierte Variablen oder Pointer usw.) lassen sich auch zur Compile-Zeit finden, indem ein Compiler (oder ein speziell für diesen Zweck entwickeltes Tool) genau Buch führt, welche Variable an welcher Stelle im Programm welche Werte annehmen kann, welche Funktion wo mit welchen Argumenten aufgerufen wird usw.. Weiters führen diese Tools Buch darüber, wie viele Pointer an welchen Stellen des Programms auf welche dynamischen Objekte zeigen. Dadurch wird erkannt, wenn Objekte nicht freigegeben werden, obwohl nie wieder auf sie zugegriffen wird bzw. kein Pointer mehr auf sie zeigt, oder wenn Objekte freigegeben werden, obwohl noch andere Pointer als der zum Freigegeben benutzte auf sie zeigen.

Weiters suchen diese Tools gezielt nach Aufrufen "gefährlicher" Library-Funktionen wie z.B. `strcpy` statt `strncpy`, `sprintf` usw.

Das nennt man statische Programm-Analyse, weil das Programm als Text analysiert wird, ohne es auszuführen. Ihre "Treffermenge" weicht normalerweise in beide Richtungen von der dynamischer Tools ab:

- Einerseits gibt es Fehler, die sich rein aus dem Programmtext heraus mit vertretbarem Aufwand nicht finden lassen, weil man eben nicht vollständig vorhersagen kann, welche Wege ein Programm beschreitet und welche Werte es annimmt, ohne es auszuführen oder zu simulieren.
- Andererseits gibt es Fehler, die durch Glück oder unzureichende Tests in der Praxis nicht aufgetreten sind, aber bei einer statischen Analyse auffallen.

Bekannte derartige Tools sind

- PC-lint (ein kommerzieller Programm-Prüfer, besprechen wir später).

- `splint`, uno, `cppcheck`, Clang Static Analyzer, … (Open-Source-Tools).
- `sparse` speziell für spezifische Fehlermöglichkeiten des Linux-Kernels.

Profiling und Tracing

Bei beidem geht es darum, Aussagen über das *zeitliche Programmverhalten* zu gewinnen.

Trotzdem sind die Ziele verschieden:

- Beim **Profiling** geht es darum, *Statistiken* über den Programmablauf zu gewinnen:
  - Welche Funktion wurde wie oft aufgerufen?
  - Wieviel Prozent des Programmcodes wurden durchlaufen?
  - In welchen Teilen des Codes hat das Programm wieviel Prozent seiner Zeit verbracht?
  - Welche Funktionen, `if`-Zweige oder Schleifen wurden nie ausgeführt? Welcher Zweig eines `if` ist häufiger?
  - Wie viel Zeit hat der Aufruf einer bestimmten Funktion im Schnitt / maximal gedauert?

  Genutzt bzw. gebraucht werden diese Statistiken zu folgenden Zwecken:

  - **Analyse der Laufzeitverteilung:**

    "Hot Spots" (d.h. besonders stark frequentierte bzw. zeitintensive Codestücke) in den Programmen zu finden, wo sich eine Umprogrammierung oder algorithmische Verbesserung besonders lohnt, und verschiedene Programm- bzw. Optimierungsvarianten miteinander zu vergleichen.

    Programmteile zu finden, in denen viel mehr Zeit verbraucht wird als erwartet (Denkfehler? Schlampige Programmierung? Falsche Annahmen über den typischen Input oder die Datenmenge?).
  - Dem Compiler **Feedback für seine Optimierung** zu geben (Für welchen Zweig des if's soll der bedingte Sprung springen und für welchen nicht? Bei welchen Funktionen zahlt sich Inlining aus? Welche Schleifen sollen wie oft unrolled werden? Welche Variablen gehören bevorzugt in Register? …)
  - Dem Tester bzw. der Qualitätssicherung Feedback zu geben, welche Teile des Codes von den Tests wie **vollständig abgedeckt** werden und welche Codestücke vom Test nicht erfasst werden.

  Dabei gibt es zwei grundsätzlich verschiedene *Arten von Profilern*:

  - **Instrumentierende Profiler:**

    Hier wird jeder Call und jedes Return (oder im Extremfall jeder Basic Block, d.h. jede lineare Folge von Instruktionen ohne Sprünge hinein und heraus) instrumentiert, d.h. mit zusätzlichem Code versehen, der *jeden Aufruf bzw. Durchlauf mitzählt und die Zeiten aufakkumuliert*. Dies geschieht entweder schon *durch den Compiler* selbst (im Fall der `gcc`-basierten freien Tools **`gprof`** und **`gcov`**) oder nachträglich *durch Analyse und Modifikation des Binärcodes* (im Fall des kommerziellen Marktführers **"Quantify"** von IBM/Rational).

Dadurch bekommt man **exakte Zahlen**, um den Preis eines **sehr hohen Overheads** (meist Faktor 10 aufwärts). Außerdem wird der Programmablauf verzerrt, weil kurze Funktionen durch den zusätzlichen Overhead in Relation zu ihrer Laufzeit viel mehr verlängert werden als lange.

- **Sampling-Profiler (statistische Profiler):**

  Hier bleibt das Programm unverändert und wird "ganz normal" ausgeführt. Das Profiling-Tool *unterbricht periodisch* (typischerweise 50-5000 mal pro Sekunde) den Programmablauf und zeichnet jedesmal den aktuellen Program Counter oder den ganzen Callstack auf.

  Dabei ist darauf zu achten, dass der Profiling-Takt *unabhängig vom Systemtakt und allen Timern oder Hardware-Synchronisationsquellen* des Programms ist, damit alle Programmteile mit gleicher Wahrscheinlichkeit getroffen werden und nicht immer z.B. ein und derselbe Timer-Handler (auf einem PC ist der *CMOS-Timer-Interrupt* meist eine gute Profiling-Taktquelle, da er sehr selten für andere Zwecke verwendet wird und in Relation zu allen anderen PC-Timern eine "krumme" Frequenz hat).

  Nach dem Programmlauf werden die Samples mit Hilfe der Debug-Information den einzelnen Funktionen zugeordnet, statistisch ausgewertet und interaktiv grafisch dargestellt: Wie viel Prozent aller Samples lagen in welcher Funktion? Wie verteilten sich die Samples einer Funktion auf ihre verschiedenen Aufrufer?

  Der Vorteil ist ein **geringerer** (einige Prozent bis Faktor 2 oder 3) und vor allem fixer (vom Programmablauf unabhängiger) **Overhead**, der fair auf den gesamten Code verteilt wird (Code, der viel rechnet, wird auch exakt proportional öfter vom Overhead getroffen) und der geringere Arbeitsaufwand (es kann ohne Vorbereitung *direkt am Produktivcode gemessen* werden).

  Der Nachteil ist eine wesentlich **geringere Genauigkeit**: Prinzipbedingt kann nur eine ungefähre Verteilung der Laufzeit auf den Code und keine exakte Anzahl von Aufrufen oder Schleifendurchläufen ermittelt werden. Außerdem braucht man eine gewisse **Mindestlaufzeit** des Programms, um zu brauchbaren Ergebnissen zu kommen. Wurde ein Codestück kein einziges Mal getroffen, lässt sich nicht sagen, ob das nur "Pech" war oder der Code wirklich nie ausgeführt wird.

  Weiters sind Sampling Profiler "blind" für Codestellen, die mit höherer Priorität als der Profiling-Code arbeiten (Interrupt-Handler, Codestellen mit gesperrten Interrupts).

  Die wichtigsten Vertreter unter Linux sind **"Oprofile"**, **"Sysprof"**, die **Google Perftools** sowie "qprof". Für Windows steht **"Very Sleepy"** kostenlos zur Verfügung.

  Der GNU Profiler **`gprof`** ist eine Mixtur: Die *zeitliche Verteilung* der Laufzeit im Code ermittelt er wie die anderen hier genannten Tools *mittels Sampling*, die *Aufrufszahlen* der Funktionen ermittelt er e*xakt durch Code-Instrumentierung* (die vom `gcc` mit der Option `-pg` schon beim Kompilieren mit eingebaut werden muss). `gprof` ist ein Commandline-Programm, Tools wie `gprof2dot` erzeugen daraus gewichtete Call-Graphen.

Neben diesen beiden Varianten von Software-Profiling enthält jeder moderne Prozessor intern Zähler, die Performance-relevante Ereignisse (z.B. Cache Misses) ohne Software-Overhead hardwaremäßig mitzählen. Auf x86 heißen diese **"Performance Counter"** und werden beispielsweise vom Tool "perfmon" genutzt.

- Beim **Tracing** geht es meist darum, *ganz exakte Aussagen* für *einzelne Ereignisse* zu erhalten:
    - Warum trat ein *Deadlock* oder ein anderes Synchronisations- oder Kommunikationsproblem zwischen Threads auf?
    - Warum wurde ein einem ganz bestimmten Zyklus eine *Echtzeit-Anforderung* nicht eingehalten?
    - Welche zeitliche Folge von Ereignissen führte dazu, dass ein bestimmtes Codestück ausgeführt wurde oder ein bestimmter *Fehler* auftrat?
    - Warum war in einem bestimmten Moment der *Interrupt so lange blockiert* oder wurde nicht bedient?

  Dazu zeichnet ein Tracer nicht statistische Zähler auf, sondern speichert den *gesamten Log einzelner Ereignisse mit ihren exakten Zeitpunkten*, damit sie nachher dargestellt werden können (üblicherweise grafisch als Zeitdiagramm: In der X-Achse läuft die Zeit, in Y-Richtung werden alle beteiligten Threads oder Prozesse angeordnet, und für jedes aufgezeichnete Ereignis bekommt der betreffende Thread zum jeweiligen Zeitpunkt eine Markierung, deren Anklicken Details über das Ereignis liefert. Dadurch erhält man ein schönes Bild der zeitlichen Verzahnung paralleler Aktivitäten).

  Relevante Ereignisse sind beispielsweise:

    - *Interrupts*, Software-Interrupts, Pagefaults, *Signale* (samt dem Ende von Interrupt- oder Signal-Handlern)
    - *Task- und Threadwechsel*, Preemptions, Erzeugen und Beenden von Threads
    - *System Calls*, I/O, *Timer*-Aktivitäten
    - *Interprozess-Kommunikation, Locking*
    - Vom Benutzer im Programm oder Betriebssystem *explizit gesetzte Trace-Punkte* ("schreib jedesmal Trace-Ereignis 12345 in den Log, wenn du hier vorbeikommst")

  Tracer schreiben die Ereignisse typischerweise in einen *großen zyklischen Puffer* im Hauptspeicher (entweder ein Puffer pro Thread, oder ein globaler Puffer mit einer parallelitäts- und interruptfesten Zugriffsfunktion). In Echtzeit-Systemen wird der Puffer normalerweise nur Post-Mortem abgespeichert (weil ein Speichern im laufenden Betrieb das Systemverhalten zu sehr beeinflussen würde), in nicht zeitkritischen Systemen kann er auch *im Hintergrund laufend geflusht* werden.

  Unter Linux ist der bedeutendste Tracer "**LTTng**", mit den meisten Entwicklungsumgebungen für Echtzeit-Systeme werden proprietäre Tracer mitgeliefert.

Andere Tools

- Die Glibc (Linux Standard C Library) enthält einen *Heap-Profiler*: Mechanismen und Hilfsprogramme, um die Heap-Aktivität (`malloc` und `free`) eines Programms statistisch auszuwerten: Siehe `memusage` und `memusagestat`.
- Die *Google Perftools* enthalten neben einem Cpu-Profiler auch einen *Heap-Profiler,* der in periodischen Abständen Statistiken über den aktuellen Heap-Inhalt und die Heap-Aktivität speichert.
- **`ltrace`** und **`strace`** liefern rudimentäre Aufrufszahlen und Zeit-Statistiken.
- *Valgrind* ist ein universeller Binärcode-Interpreter, der mit einer Vielzahl von Plugins versehen werden kann. Derzeit stehen zur Verfügung:
    - Die oben beschriebenen Plugins “memcheck” und “sgcheck”.
    - Ein *Heap Profiler,* der über die Heap-Belegung Buch führt und u.a. mitschreibt, welche Teile des Programms wie oft Heap-Allokationen und Heap-Zugriffe ausführen.

      Ein weiteres Heap-Tool führt Buch, wie oft und mit welchen Offsets auf welche Objekte im Heap zugegriffen wurde.
    - Zwei Tools, die *Synchronsiationsprobleme* (potentielle Race Conditions oder Deadlocks usw.) zwischen Threads in parallelen Programmen finden.
    - Einen Profiler, der die *Cache-Zugriffe und -Trefferquoten* ermittelt.
    - Einen Profiler, der die *Trefferquote der Branch Prediction* misst.
    - Einen Profiler, der so wie ein Instrumenting Profiler *Call-Graphs und Call-Statistiken* liefert, und einen weiteren, der auf *Basic-Block-Ebene* arbeitet.
    - Ein Tool, das Statistiken über die *einzelnen Arten von Instruktionen und Speicherzugriffen* führt.

  Gemeinsam ist allen diesen Tools leider der *sehr hohe Laufzeit-Overhead* (meist weit über einem Faktor 10).
- Analog zum Address Sanitizer hat *Google* auch einen *Thread Sanitizer* entwickelt, der in Programme mit mehreren parallelen Threads potentielle Race Conditions findet. Auch er ist signifikant schneller als `valgrind` und in `llvm`/`clang` sowie `gcc` integriert.

Qualitätssicherung

Was gehört zur Qualität?

- Erfüllung der **geforderten Funktionalität**
  (Pflichtenheft / Produktspezifikation / Anforderungs-Dokument)
- **Fehlerfreiheit**
- **Robustheit**:
  - Erkennen *falscher (zu großer, …) Eingaben / Daten*, geordnete Reaktion!
  - Geordnete Reaktion auf *Systemfehler*
    (Speicher voll, I/O-Fehler, Netzwerkprobleme, …)
  - Sauberes *Wiederaufsetzen* nach Stromausfall / Abschuss / Absturz
    mit minimalen Datenverlusten
  - Sicherheit gegen *Angriffe*
- **Stabilität**:
  - *Hochlastverhalten* (viele Benutzer-Requests, viele Steuerungs-Events, …),
    Verhalten bei mutwilliger Überlast ("Denial of Service"-Angriffe)
  - *Langzeitverhalten* (Memory Leaks, "Alterung" von Datenstrukturen)
  - Verhalten bei *großen Datenmengen*
    (Test des vorgegebenen Mengengerüstes)
- Einhaltung relevanter **Standards** (Netz-Protokolle, Dokumente, SQL, …)

  Integration in andere Software
- Erfüllung **rechtlicher Vorgaben**:
  - Datenschutz und *Datensicherheit*
  - *Protokollierungs- und Archivierungspflichten*
  - Sichere *Benutzeridentifikation*
- **Bedienbarkeit**:
  - *Optik*
  - Intuitives *Verhalten*
  - *Verständlichkeit* & sprachliche Fehlerfreiheit aller Texte und Meldungen
  - Vollständigkeit und Korrektheit der *Übersetzungen*
- **Administrierbarkeit**:
  - Installation
  - Upgrade von Vorversionen, De-Installation, Übersiedlung auf andere Platte /
    anderen Rechner, Backup (on the fly?) …
  - Funktion der mitgelieferten Tools
- **Dokumentation & Hilfe**:

    - Nutzbringender *Informationsgehalt*
      (Erklärung von Hintergrund-Info, nicht von offensichtlicher GUI-Bedienung)
    - *Verständlichkeit* & sprachliche Korrektheit
    - *Vollständigkeit* (z.B. Dialog-Tooltips)
    - Vollständigkeit und Korrektheit der *Übersetzungen*
    - Admin-Doku: Installationshandbuch usw.
- **Effizienz, Performance, Ressourcen-Verbrauch** (Skalierbarkeit!)

  ==› Dimensionierungs-Whitepaper
- **Zertifizierung**

  ==› Testprotokolle usw.

Grundlegendes

- Fehler pro Lines of Code sind historisch gesehen ziemlich konstant ,
  Softwaregröße wächst expotentiell!
- n Module mit Korrektheitswahrscheinlichkeit p
  ==› Gesamtkorrektheits-Wahrscheinlichkeit p^n

  (100 Module mit 95% der Zeit / aller Inputs Korrektheit
  ==› 0.6% Gesamt-Korrektheit !)
- **"Je später der Fehler erkannt wird, umso teurer!"**

  Kosten für Fehlerbericht-Verwaltung, für neuen Build oder neue Release-Version samt Test, für Austausch der SW beim Kunden, für Standzeiten, …

  ==› *Wenn möglich Fehler gar nicht entstehen lassen / schon in der Entwicklung erkennen.*
- Vorstufen in der **Entwicklung**:
    - Testbares *Design* (Modularisierung im Hinblick auf Modultest, Einplanung von Test-Schnittstellen, …)
    - *Programmierhandbuch* (Verbot fehleranfälliger Programmkonstrukte, Formatierungs- und Kommentierungsregelungen, …)
    - *Qualitäts-Tools* (siehe unten)
    - *"Defensive"* Programmierung (Eingabe- und Parameter-Prüfungen, Asserts, …)
    - *Code Review* (4 bis 6-Augen-Prinzip)
- Alle Qualitäts-Aktivitäten gehören **dokumentiert**,
  müssen **wiederholbar / reproduzierbar** sein.
- Alle Tools, Testdaten, Testdokumente… gehören mit dem Produkt **eingecheckt** (versioniert).
- **Maßzahl "Code Coverage"**:
  "Wie viel Prozent des Codes werden durch die Tests erfasst / abgedeckt?"
    - Bei if's: Test für beide Zweige

- Bei Schleifen: Test für keinen / einen / viele Durchläufe

Test versus Verifikation

"Tests können nur die Anwesenheit von Fehlern beweisen, nie deren Abwesenheit!"

Für absolute Korrektheit ist ein mathematischer Beweis notwendig:
Verifikation = formaler Beweis der Programm-Korrektheit.

Dieser liefert meist auch sehr viel neue Einsicht in das Problem.

Grundsätzliches Vorgehen:

Gegeben ist:

- Formale Spezifikation des *Inputs* / Anfangszustandes:
  Welche Bedingungen erfüllt der Input?
- Formale Spezifikation des *Outputs* / Endzustandes:
  Welche Bedingungen soll der Output erfüllen? Wie hängt er vom Input ab?
- Formale Spezifikation aller *Grundfunktionen,* die das Programm verwendet.

Vorgehensweise:

- Beweise ausgehend vom Anfangszustand Schritt für Schritt, *welche Bedingungen* für alle Daten (Variablen) nach jedem einzelnen Befehl / jeder Zeile gelten.
- Für jede Schleife muss eine sogenannte *"Schleifeninvariante"* gefunden werden: Das ist eine Bedingung, die vor und nach jedem einzelnen Schleifenumlauf gilt (also auch vor dem ersten und nach dem letzten). Dann muss man
  - von den Bedingungen vor der Schleife auf die Invariante mit dem Anfangswert kommen,
  - von der Invariante und der Endbedingung auf die Bedingung nach der Schleife kommen,
  - und mit den Befehlen in der Schleife von der Invariante auf die Invariante mit den Werten des nächsten Durchlaufes kommen.
- Bei größeren Programmen: Schreib eine Spezifikation für *jede Funktion für sich* und beweise sie! (wenn sie bewiesen ist, kann sie im Beweis anderer Funktionen verwendet werden.)
- Nach dem letzten Befehl sollten die sich ergebenden Bedingungen die spezifizierten Endbedingungen sein…

Aber:

- Programmverifikation erfordert sehr *hohen Aufwand* und sehr hohe Qualifikation (theoretisch-wissenschaftliche Ausbildung ab Doktorat erforderlich).
- Die Tools zur Unterstützung der Verifikation sind noch sehr dürftig.
- Verifikation befasst sich oft *nur mit den spezifikationskonformen Fällen* des Programmaufrufs (d.h. mit dem Programmverhalten bei korrektem Input), nicht mit der Reaktion auf falschen Input, Fehlbedienung usw.
- Programmverifikation ist oft blind gegen *"externe" Ereignisse* (I/O-Fehler, Speichermangel, …) und gegen *nicht spezifikationskonforme Grundoperationen*

(Hardware-Bugs in der FPU, fehlerhaft implementierte Betriebssystem-Aufrufe, …).

- Realitätskonforme Verifikation numerischer Berechnungen erfordert komplexe *Behandlung der Rundung bzw. der endlichen Genauigkeit*.

==› Die Verifikation kann den Test nicht ersetzen!

In der Praxis:

- Die Korrektheit im Rahmen von Forschungstätigkeiten neu entwickelter Algorithmen ist als *Teil der wissenschaftlichen Arbei*t formal zu beweisen.
- In der praktischen Entwicklung:
  - Verifikation dort, wo es *vorgeschrieben* ist.
  - Eventuell Verifikation eng begrenzter Codestücke mit besonders *komplexen Ideen und Algorithmen*.
  - Eventuell Verifikation nur für **Teilaspekte** der Gesamt-Korrektheit (z.B. nur für **Deadlock-Freiheit** oder **Race-Condition-Freiheit** in Multithreaded-Programmen, nur für Termination wenn sie nicht offensichtlich ist, …).

Black Box Test versus White Box Test

Black Box Test:

- *Keine Kennntnis des internen Aufbaus, kein Zugriff auf den Source*

  ==› stets Gesamtsystem-Test

  ==› Vorteil:
  Tester testet das, was der Kunde bekommt, und *verhält sich wie ein Kunde*.

  ==› Nachteil: Kein gezielter Test bestimmter Code-Stellen möglich, man kennt den Code ja gar nicht.

  ==› Nachteil: Testabdeckung meist eher gering: Viele Codestrecken vor allem unterer Programmmodule können von der Oberfläche aus nicht oder nur sehr kompliziert angesprochen werden.

- *Programm ist im Auslieferungszustand, unmodifiziert*

  ==› Vorteil: keine Gefahr von Fehlern, die durch die Testumgebung oder die Isolation einzelner Teile neu eingebaut oder zum Verschwinden gebracht werden.

  ==› Nachteil: Keine zusätzlichen Debug-Info's, Eingrenzung / Lokalisierung des Fehlers oft schwierig, hilft dem Entwickler nicht bei der Behebung.

White Box Test:

- *Test in Kenntnis des Source*

  ==› Vorteil: Modul-weiser Test möglich

  ==› Vorteil: Gezielter Test einzelner Codestrecken durch eigens dafür konstruierten Input möglich

  ==› Vorteil: Testabdeckung prüfbar

  ==› Nachteil: Einarbeiten in den Source kostet Zeit!

==› Nachteil: Tester kennt den Source, "erbt" damit die Gedanken des Entwicklers (vor allem bei gut dokumentiertem Code)

- Tester macht dieselben Denkfehler wie der Entwickler (und testet sie nicht).
- Tester übersieht dieselben Fälle wie der Entwickler.

Persönliche Einschätzung:

- Bei knapper Zeit / knappem Geld ist der Black-Box-Test wichtiger und effizienter! (findet die im praktischen Einsatz "relevanten" Fehler!)
  ==› zuerst Black-Box-Test!
- White Box Tests sollten auf die automatisierten Tests und auf die Analyse einzelner Spezialprobleme beschränkt bleiben
  ==› im Regelfall kennt der händische Tester den Source nicht (braucht ihn auch nicht zu kennen)!

Modultest versus Gesamtsystemtest

Modultest = isolierter Test einer Komponente

Extremfall **Unit-Test**: Test einer einzelnen Funktionalität von einer einzelnen Funktion
==› Gezielter Aufruf nur dieser einen Funktion aus einem eigens geschriebenen Testprogramm heraus.
==› Viele tausende einzelne Testfälle mit je einem Testprogramm.

2 Varianten:

- **Bottom-Up-Test:**
  - Test der Module / der Funktionen **"von unten nach oben"**.
  - Das zu testende Modul / die zu testende Funktion ruft *nur schon getestete (korrekte) Module bzw. Funktionen* aus dem realen Programm auf.
  - Nur das Testprogramm zum Aufruf des Moduls wird eigens geschrieben.
- **Test mit Stubs:**
  - Test immer nur einer **einzigen isolierten Funktion** aus dem zu testenden Programm.
  - Alle von der getesteten Funktion aufgerufenen Funktionen werden im Test durch *"Stubs"* ersetzt: Kurze, eigens für den Test entwickelte Hilfsfunktionen, die die Funktionalität der aufgerufenen Funktion im Originalprogramm nur für die im Test benötigten Fälle minimal nachbilden / simulieren (ev. tabellengesteuert / aus einem Testdaten-File).

In großen Firmen / bei großen Softwareprojekten:
Aufteilung der Tests ‹==› organisatorische Aufteilung:

- Jede Abteilung testet die von ihr entwickelten Komponenten lokal einzeln, bevor sie an die Produktintegrationsabteilung ausgeliefert werden.
- Die Produktintegrationsabteilung macht nur mehr Gesamttests.

Persönliche Einschätzung:

- Gestubte Unit-Tests sind für Basismodule (ohne Abhängigkeiten von anderen

Modulen) machbar und sehr sinnvoll.

- Für höhere Module sind gestubte Unit-Tests in der Praxis viel zu aufwändig (Schreiben der Stubs dupliziert im schlimmsten Fall einen Großteil der Codelogik!) und auch von der Trefferquote nicht besonders wirkungsvoll
  ==› das Geld ist in andere Tests meist viel besser investiert!
- Die Frage "Bottom-Up-Modultests versus Gesamtsystemtest" hängt sehr individuell davon ab, wie weit die einzelnen Module isolierbar und testbar sind (das beginnt schon beim Design!).
  Wenn Modultests relativ leicht realisierbar sind ==› machen!
  Wenn hoher Aufwand nötig wäre ==› lieber in andere Tests investieren!

Händische versus automatisierte Tests

Je nach Test-Art, siehe folgendes Kapitel!

Automatisiert:

- Automatische Ausführung und Auswertung.
- Eintragung der Ergebnisse meist in eine Datenbank.
- Oft eingebunden in Checkin / in Nighty Build.
- Können vom Umfang her viel mehr testen als es händisch in vernünftiger Zeit möglich wäre.
- Machen keine Fehler und Schlampereien…

Händische Tests:

- Menschliche Kreativität ist auch beim Test durch nichts zu ersetzen!
  (z.B. bei Sicherheitslücken, Ausfall-Szenarien, "kreativer" Fehlbedienung, …)
- Manche Dinge sind automatisiert schwer zu prüfen!
  (Optik, Sprachliches, Installation, …)

  Oft: Neues das erste Mal manuell testen, dann automatische Tests auf Abweichungen vom Resultat des manuellen Tests.
- Automatisierte Tests berücksichtigen die (Un-) Fähigkeiten der realen Benutzer zu wenig…
- Überspitzt formuliert: *Im händischen Test sollten der intelligenteste und der unfähigste Mitarbeiter der Firma eingesetzt werden!*

Persönliche Einschätzung:

- *Beides* hat seine Berechtigung / ist notwendig.
- Automatisierte Tests haben einen *sehr hohen Anfangsaufwand* (Entwickeln des Test-Mechanismus, Schreiben der Tests, …), aber *geringen laufenden Aufwand*.
  ==› schreckt vor allem kleine Firmen und Anfänger ab!

  Manuelle Tests haben *konstant hohen Dauer-Aufwand* (bei jeder Version wieder)
  ==› manuelle Tests für Dinge, die sich automatisiert testen lassen (Routine-Tests) sind auf Dauer zu teuer!
- *Große Firmen überbewerten meist die Aussagekraft und Wichtigkeit automatischer*

Tests & richten zu wenig Augenmerk auf manuelle Tests.

Test-Checkliste

- **Funktionalitäts-Test:** 100 % der spezifizierten Funktionalität testen
    - Bei GUI-Programmen:
      Jeder Button, jeder Input, jeder Menüpunkt, jedes Auswahl, …
      Jede Konfigurations-Option in den Systemeinstellungen
    - Bei Commandline-Programmen:
      Jede Commandline-Option
- **Randfälle, Stabilität, Robustheit:**
    - Leerer Input, maximal großer Input, um 1 zu großer Input, …
      Sonderfälle und Randzahlen , falsche Filenamen, …

      Bei Grenzen und magischen Zahlen:
      Jeweils genau und 1 darunter / darüber testen!
    - Random Input ("Fuzzer Tool", wahllos herumklicken, …),
      Fehlbedienungen, Hack-Versuche, "Dümmstmöglicher Benutzer", …
    - Ressourcenprobleme: Speicher voll, Platte voll, Benutzer-Überlast, …
    - Externe Störungen: Strom weg, Netzwerk weg, harter Programmabbruch, …
    - Cluster- und Storage-Failover
- **GUI-Kontrolle:**

  Siehe GUI Design Guidelines der jeweiligen Plattform & SW-Ergonomie-Normen!
    - *Mausfreie Bedienbarkeit*:
      Shortcut-Tasten für alle Menüs und Buttons (Standards beachtet?!)
      Sinnvoller Default-Button, sinnvolles Default-Inputfeld
      Intuitive Tabreihenfolge
    - Sinnvolle *Vorbelegungen / Voreinstellungen* für alle Text- und Auswahlfelder?
    - Vorgeschriebene *Standard*-Buttons (Ok, Cancel, …) vorhanden und in üblicher Position angeordnet? (Ok links oder rechts?)

      Standard-Menüeinträge vorhanden und in der üblichen Reihenfolge im richtigen Menü? (File-, Edit- und Help-Menü!)

      Standard-System-Dialoge verwendet (Öffnen / Speichern, Drucken, …)

      Standard-Konfigurationsmöglichkeiten implementiert?
      (Font & Schriftgröße, Toolbars, …)
    - Optische *Schönheitskontrolle*, Layout-Kontrolle:
      Verdeckte / sich überlappende Elemente
      Abgeschnittene Texte (Achtung auf Übersetzungen!)
      Zu kleine Text-Eingabe-Felder
      Unintuitive Zusammenfassung / Gruppierung / Anordnung von Elementen
      Zu große ungenutze Flächen
      Nicht erkennbare Icons (zu klein, nicht intuitiv, zu kontrastarm, …)

Korrektes Re-Layouting beim Resize eines Dialogs

- Alle Dialoge kleiner als spezifizierte *Minimal-Bildschirm-Größe*?
  Test mit verschiedenen extremen Bildschirm-Größen und -Auflösungen, extrem flachen Seitenverhältnissen!
- Werden *GUI-Systemeinstellungen* beachtet?
  GUI Style bzw. Theme ("High Contrast", "Inverse", ...), System Fonts, *Schriftgröße* (Wie sehen Dialoge bei großen Schriften aus? Wächst das Layout in beide Richtungen mit?)
- Werden *Ländereinstellungen* beachtet (Datums- / Uhrzeit-Format, Geld- und Kommazahlen-Format, Umlaute und Sortierreihenfolge, ...)
- Verhalten bei Schriftzeichen-Sprachen, rechts-links-Sprachen, ...
- Steckt Information *nur* in Farben (verboten! ==› Farbenblinde?)
- Alle *Tooltips* vorhanden?

  Kontext-bezogene Hilfe bringt bei jedem GUI-Element das richtige Thema?

- **Textkontrolle:**

  Alle Texte und Fehlermeldungen lokalisierbar? (keine fix eincodierten Texte!)
  Rechtschreibung? In allen Sprachen definiert? Sinnvoll (incl. Übersetzungen!)?

- **Echtdatentest:** Mit realen kompletten Daten vom Kunden oder Referenzdaten.

- **Lasttests:**
  Mindestens mit der spezifizierten Datenmenge / Benutzerlast (ist die realistisch?) auf der dafür spezifizierten Rechnergröße!

  Verhalten bei Überlast.

  Dauerlauf-Tests (==› Memory- und Performance-Leaks?)

- **Regressionstests:**
  Alle schon korrekt gelaufenen Tests mit jeder Version wieder machen (großteils automatisch!)
  ==› Frische Fehler eingebaut?
  (Regression = Rückschritt = schon einmal Gelaufenes funktioniert nicht mehr)

  Im Besonderen:
  Tests mit allen konkreten Inputs mit Fehlverhalten aus bisherigen Fehlerberichten (beste Quelle für Testfälle!!!)

  - ==› Fehler wirklich ausgebaut?
  - ==› Fehler später nicht wieder eingebaut?
  - ==› Folgefehler, neues Fehlverhalten, Nebenwirkungen?
    (60 % aller Fehlerbehebungen sind im ersten Anlauf falsch / mangelhaft)

  ==› Alle Tests aufheben, nicht nach Fehlerbehebung wegwerfen!

- **Doku & Hilfe:** Probelesen (von "durchschnittlichem Benutzer"!)
  ==› Dem Niveau / den Vorkenntnissen der Benutzer / Administratoren angepasst?
  ==› Kann ein *Ferialpraktikant* nur auf Basis der Doku mit dem Produkt arbeiten?

- **Installations-, Upgrade- und Administrationstests:**

- Auf *“sauberer”* Maschine! (mit genau der spezifizierten Software)
  ==› Läuft die Software auf allen spezifizierten Plattformen mit den geforderten Systemversionen / Länderversionen und den geforderten Basis-Software-Versionen?
- Von ”durchschnittlichem Admin” ohne Vorkenntnisse, nur nach Handbuch! (==› Ferialpraktikant?!)
- Mit allen Kombinationen von Installations-Optionen
  (bei Komponentenauswahl: Komponentenabhängigkeiten geprüft?)
- Platztest und Vorbedingungstest ok?
  (wird zu wenig Platz / fehlende erforderliche Software *rechtzeitig* erkannt?)
- Upgrade-Test mit *großen* Realdaten!
- System-Vergleich nach Deinstallation ==› Reste?

Generell gilt für Test-Maschinen noch mehr wie für Build-Maschinen:

- Auf “sauberer” Maschine:
  Keine Entwicklermaschine, keine Zusatz-Software, keine Reste vom vorigen Test, …
- Mit genau der Software-Umgebung (in genau den Versionen) und den Systemeinstellungen, die für das Produkt spezifiziert sind, und sonst nichts.

Und wenn ein Fehler durchgerutscht ist: **Analyse warum!** (==› neuer / geänderter Test?!)

Tools für die Qualitätssicherung

Statische Analyse-Tools

Hier handelt es sich um Tools, die *nur den Quell-Code* analysieren, nicht die Programmausführung.

Es gibt mehrere Arten:

- **Tools, die Programmierrichtlinien überprüfen:**

  Diese Tools prüfen z.B. die Einhaltung der Formatierungsvorschriften, das Vorhandensein vorgeschriebener (Doxygen-) Kommentare, die `goto`-Freiheit, die Abwesenheit globaler Variablen oder öffentlicher Member-Variablen, die vorschriftskonforme Benennung von Methoden und Membervariablen usw.

- **Tools, die potentiell fehlerhafte / gefährliche Programmstellen suchen:**

  Hier gibt es zwei Ansätze, die ineinander übergehen:

  - Tools, die im Source mittels *Pattern Matching* nach einer großen Anzahl von bekannt anfälligen Konstrukten suchen, z.B. nach `strcpy`, `sprintf` oder anderen Funktionen ohne Längenprüfung.
  - Die schon bei der Fehlersuche erwähnten statischen Analysatoren, die durch inhaltliche Analyse des Codes versuchen, an jeder Stelle des Programms für *jede Variable ihren Wertebereich* (noch undefiniert, potentiell `NULL` oder nicht, zeigt auf gültiges / freigegebenes dynamisches Objekt, Integer von-bis, …) zu ermitteln, diesen mit der Verwendung zu vergleichen, und daraus potentielle Fehler (Zugriffe auf `NULL`-Pointer, Operationen mit undefinierten

Werten, Divisionen durch 0, Off-By-One-Fehler bei Arrayzugriffen, fehlende oder doppelte `free`, zu kurz deklarierte Strings, …) zu erkennen.

Ein Sonderfall dieser Analysen ist die Suche nach potentiellen *Race Conditions* in parallelen Programmen.

Der Marktführer für diesen und den vorigen Punkt ist das kommerzielle Produkt "pcLint". Freie (allerdings bei weitem nicht so leistungsfähige) Tools sind "uno", "splint", "RATS", "cppcheck" und "flawfinder" sowie die Warnings des jeweiligen Compilers. Während "pcLint" sehr breit arbeitet, haben die meisten der freien Tools ihren Schwerpunkt in der Erkennung sicherheitsrelevanter Konstrukte (potentiell ausnutzbarer Lücken).

- **Tools, die Code-Metriken ermitteln:**

  Diese Tools liefern *statistische Maßzahlen*, die etwas über die *Code-Qualität* aussagen sollen, z.B. Kommentare pro Lines of Code, durchschnittliche / maximale Länge von Funktionen, durchschnittliche / maximale Schachtelungstiefe von `if`'s und Schleifen, …

  Manche Tools suchen auch nach *doppelten bzw. sehr ähnlichen Codestellen* als Zeichen schlechter Strukturierung bzw. gedankenlosem Copy/Paste oder beispielsweise nach totem (nicht erreichbarem bzw. nie aufgerufenem) Code.

- **Tools zur globalen Struktur-Analyse:**

  Diese Tools prüfen beispielsweise, ob das Zusammenspiel der Module eines großen Programmpaketes dem vorgegebenen Design entspricht oder davon abweicht, also ob es beispielsweise `#include`-Befehle entgegen der Modulhierarchie "von unten nach oben" gibt oder zyklische *Abhängigkeiten zwischen Modulen*, oder ob eigentlich plattform-unabhängige Module direkt plattform-abhängige System-Header inkludieren.

Frameworks für Unit-Tests

Für fast jede Programmiersprache gibt es Frameworks, die das Schreiben und Auswerten einer großen Anzahl automatischer Unit-Tests erleichtern sollen: Cunit, CppUnit, CppTest, GoogleTest, …

Im Wesentlichen bieten derartige Frameworks folgende Funktionen:

1. **Test-Makros:**

   Diese Makros werden beim *Schreiben der einzelnen Tests* verwendet, jeder Aufruf eines solchen Makros ist *ein Testfall*. Es

   - **ruft** die zu testende Funktion auf,
   - **prüft** deren Ergebnis
   - und **protokolliert** den Testfall und sein Resultat.

   Dazu wird im Makro üblicherweise der Name bzw. eine Kurzbeschreibung des Tests angegeben.

   Solche Makros existieren meist für

   - die Prüfung eines *Ergebnisses* auf `true` oder `false`,

- den Vergleich eines *Ergebnisses* mit dem erwarteten Wert (unterscheidet sich vom vorigen Fall durch ein detaillierteres Protokoll),
- die Prüfung, ob der zu testende Code *Exception-frei* läuft oder eine bestimmte Exception (im Fall von C eventuell: bestimmte Signale) auslöst,
- und in manchen Systemen (z.B. GoogleTest) auch die Prüfung auf einen bestimmten *Programm-Output* oder ein bestimmtes *Programm-Ende* (Exit-Code) bzw. auf einen harten Absturz des Programms.

2. **Test-Setup und -Teardown:**

   Das Grundprinzip von Unit-Tests ist, pro Test immer nur

   einen einzigen Fall (Codepfad) einer einzigen zu testenden Funktion in einer "jungfräulichen" Umgebung

   zu testen (d.h. mit "*frischen*", von der Funktion des zu testenden Codes unabhängigen *Eingabedaten*). Mehrere Tests nacheinander auf denselben Daten widersprechen daher diesem Prinzip, denn wenn eine zu testende Funktion die Testdaten in anderer als der erwarteten Weise modifiziert, würde das die nachfolgenden Tests beeinflussen.

   Das Test-Framework muss daher die Möglichkeit bieten, vor dem eigentlichen Testfall dessen *Testumgebung* (auch "*Fixture*" genannt) mit *Testdaten zu füllen* (Variablen und Objekte anlegen, Files öffnen, Datenstrukturen füllen, ...) und nachher wenn nötig wieder *aufzuräumen*. Da diese Testdaten oft für viele Testfälle gleich oder ähnlich sind, sollte dieser Setup- und Teardown-Code für mehrere Testfälle (z.B. für alle Tests einer Suite) nur einmal geschrieben werden müssen.

3. **Testverwaltung und -Ausführung:**

   Mit einem einzigen Start des Tests, d.h. aus einem einzigen Hauptprogramm heraus, werden typischerweise *tausende Einzeltests* automatisch ausgeführt.

   Jedes Unittest-Framework bietet daher die Möglichkeit, einzelne *Testfälle* in irgendeiner Form *im Framework zu "registrieren"*, damit sie ausgeführt werden (das Hauptprogramm selbst bleibt dabei unsichtbar, es ist interner Bestandteil des Frameworks).

   Üblicherweise geschieht dies *in mehreren Hierarchie-Ebenen*: Ein Test besteht aus einer oder mehreren Test-Suites (typischerweise pro Softwarekomponente), eine Test-Suite besteht entweder aus Unter-Suites (z.B. pro Klasse oder Datentyp) oder aus einzelnen Testfällen.

4. **Test-Auswertung:**

   Die Test-Frameworks generieren automatisch ein *Protokoll* des Tests: Anzahl der ausgeführten und erfolgreichen Tests, Auflistung aller fehlgeschlagenen Tests, ...

   Dieses Protokoll kann als Text oder als *HTML*-Seiten geschrieben, in einem interaktiven *GUI* angezeigt oder in einer *Datenbank* abgelegt werden (sinnvollerweise bietet das Framework mehrere *Output-Plugins* für dieselben internen Daten).

   Weiters ist eine laufende Fortschrittsanzeige sinnvoll, da Tests ja Stunden laufen bzw. bei Fehlern im zu testenden Code auch in Endlosschleifen geraten können.

Daneben gibt es auch Forschungsprojekte, die Tools zum Ziel haben, die aus dem Source-Code eines Programmes automatisch Test-Programme für dieses Programm generieren (z.B. LLVM Klee).

Coverage-Analyse-Tools

Die "Coverage" ist die *Codeabdeckung*, hier die Codeabdeckung von Tests: Wie viele Prozent aller Funktionen, wie viele Prozent aller `if`-Zweige, wie viele Prozent aller Schleifen werden beim Test wirklich durchlaufen? Welche Stellen im Programm werden nie durchlaufen?

Diese Tools werden daher begleitend bei anderen Tests eingesetzt und beruhen auf Code-Instrumentierung, entweder durch den Compiler (`gcc` / `gcov`) oder nachträglich auf den Binaries (IBM/Rational Quantify). Auch für Valgrind gibt es Plugins in diese Richtung.

Bedeutung haben diese Tools insoferne, als bei Code mit erhöhten Qualitätsanforderungen eine bestimmte Coverage der Tests zur Zertifizierung vorgeschrieben ist und durch Testprotokolle belegt werden muss.

Fuzzer

Fuzzer sind Tools, die das zu testende Programm mit nach gewissen Vorgaben *zufällig generiertem Input* füttern, um ihre *Robustheit und Stabilität* zu testen, seien es Inputdaten (Stdin, Files, Netzwerk, Befehlszeile) oder Benutzer-Interaktionen am GUI.

Frameworks für GUI-Test-Scripting

Diese Tools haben meist 2 Komponenten:

- Die Tools *simulieren skriptgesteuert Benutzerinteraktionen* (Mausklicks, Tastatureingaben, Scrollrad, ...). Die Skripts können entweder per Editor erstellt oder *per Recorder aufgezeichnet* werden.

  Man unterscheidet Frameworks, die dabei ohne Kenntnis des zu testenden Programmes mit *Pixelkoordinaten* arbeiten, und solche, bei denen die *GUI-Elemente logisch addressiert* werden ("Wähle Menüpunkt *xxx* aus", "Klicke auf Button *yyy*", "Schreib *blabla* in das Textfeld *xy*", ...). Beides hat Vor- und Nachteile:

  - Pixelbasierte Skripts müssen bei jeder optischen Veränderung im GUI (neuer Menüpunkt eingefügt, Dialog-Layout geändert, ...) neu aufgenommen werden.

    Pixelbasierte Systeme injizieren meist Input-Events auf einer Ebene "unterhalb" der Anwendung (zwischen Betriebssystem und Anwendung), die Anwendung ist unmodifiziert und merkt keinen Unterschied zwischen echtem und simuliertem Input.
  - Inhaltsbasierte Skripts "übersehen" oft neue optische Unschönheiten oder versehentliche Änderungen in der Menü-Reihenfolge oder dem Dialog-Layout, d.h. sie funktionieren korrekt, auch wenn die Darstellung defekt ist.

    Zur Funktion solcher Systeme ist es notwendig, dass die GUI-Library entsprechende Schnittstellen bereitstellt, um die angezeigten GUI-Elemente auslesen und per Namen ansprechen bzw. betätigen zu können. Das Test-

Tool ist also in irgendeiner Form mit der Anwendung verlinkt, und der Test-Input erreicht die Anwendung auf anderem Weg als "normaler" Input.

- Die Tools prüfen die *Reaktion des Programms* auf den simulierten Input. Auch das kann wieder auf zwei Arten (mit ähnlichen Vor- und Nachteilen) geschehen:
    - Durch *pixelweisen Vergleich* mit beim Recording angefertigten *Referenz-Screenshots*.
    - Durch *Prüfen* der erwarteten *logischen Reaktion des Programms*, d.h. durch Auslesen der angezeigten *GUI-Elemente* und ihres Zustandes bzw. Inhalts ("Ist ein Dialogfenster mit Namen *"xy"* auf- oder zugegangen?", "Ist der Button / der Menüpunkt *"abc"* enabled / disabled?", "Enthält ein Textfeld den erwarteten Text?", ...).

Leider ist die Entwicklung dieser Tools oft plattformabhängig und schwierig: Sie erfordern in vielen Fällen entweder Eingriffe in das Betriebssystem oder Eingriffe in den zu testenden Code (z.B. Austausch der GUI-Library oder zusätzliche Hooks im Code), was im Hinblick auf die Aussagekraft der Tests eigentlich vermieden werden sollte.

Benötigt werden solche Tools u.a. für folgende 3 Aufgaben:

- Automatisierter Test des *Programmes* selbst.
- Automatisiertes *Ausführen* bzw. automatisierter Test des *Installationsprogrammes*.
- Bei Client-Server-Applikationen usw.: *Automatisierte Lasterzeugung* auf den Clients (Simulation vieler gleichzeitiger Benutzer) für *Belastungs- und Performance-Tests* des Servers.

Natürlich stellt sich die Frage der Input-Simulation und des Output-Vergleiches auch bei nicht-GUI-Programmen, hier ist sie aber mit den üblichen Tools (diff, Shellscripting usw.) relativ leicht zu lösen.

Tools zur Programmverifikation

Diese Tools sollen bei der mathematischen Verifikation von Programmen die Routine-Schritte übernehmen.

Bugtracker

Die zentrale Anwendung des Qualitätswesens ist die **Fehlerverwaltung**. Hierfür gibt es sowohl mächtige *kommerzielle Produkte* (z.B. IBM ClearQuest) als auch zahlreiche freie Entwicklungen (z.B. Mozilla's Bugzilla). Weiters bieten zahlreiche *Web-Plattformen* für Open-Source-Projekte (z.B. SourceForge) den Projekten auch die Nutzung einer Fehlerverwaltung an.

Die Fehlerverwaltung enthält für jeden einzelnen **Fehler** und jeden **Änderungswunsch** bzw. Entwicklungs-Auftrag die gesamte *Beschreibung* und *Geschichte* von der *Erkennung bis zur Behebung* (alle derartigen Systeme nutzen dafür eine *Datenbank*):

- *Nummer, Kurztitel, Langtext.*
- Welche *Version* ist fehlerhaft
  (allgemein oder auf welcher einzelnen *Plattform* / Betriebssystemversion / ...)?
- Klassifizierung und Zuordnung des Fehlers (*Fehler-Art, Schwere, Priorität, betroffene*

Komponente, …).

- *Fehlergeschichte* und *Fehlerstatus*: **Wer hat wann was gemacht** (Fehler gemeldet / analysiert / behoben / getestet / dokumentiert)? Wann hat sich der Fehlerstatus (oder andere Attribute des Fehlers) geändert? Für welche Version oder welches Datum ist die *Behebung eingeplant*, mit welcher Version ist sie *wirklich erfolgt*?
- *Verantwortlichkeiten* (zuständiger Einplaner, *Entwickler* und Tester, ev. auch *Ansprechpartner* beim betroffenen Kunden)?

Features, die ein Bugtracker bieten sollte:

- *File-Attachments:* Das können Informationen zum *Fehler* (Logs, Coredumps, Traces, Konfig-Files, Testprotokolle), zum *Code* (Verbesserungsvorschläge) oder zum *Test* (Testdaten zur Reproduktion) sein sowie für den Fehler relevante *Dokumente* (Analysedokumente, Dokumentation / Release Notes).
- Querverbindung zur *Versionsverwaltung*: Aus der Versionsverwaltung sollte man sich vom Behebungs-Checkin zum Fehler durchklicken können, aus dem Bugtracker zu allen dadurch hervorgerufenen Code-Änderungen (im kommerziellen Bereich z.B. IBM ClearCase / ClearQuest, im freien Bereich "trac").

  Motto: *"Keine Änderung des Codes ohne dazugehörigen Fehlerbericht oder Änderungs- / Erweiterungsauftrag!"*

  Ebenso sind Querverweise in die *Testsoftware* sinnvoll: Welche Testfälle wurden für diesen Fehler geschrieben, welche Protokolle haben diese Tests geliefert?
- Ein konfigurierbares *Web-Interface* (nur lesend oder Vollzugriff).
- Umfangreiche *Suchfunktionen* (nach Kombinationen beliebiger Attribute und mittels Volltextsuche im Langtext).

  Fehlende / schlecht handhabbare / wenig zielgenaue Suchfunktionen führen normalerweise zur massenhaften Doppelerfassung von Fehlern ==› zusätzlicher Aufwand!
- Auswertungen für das *Projektmanagement* bzw. Anbindung an die *Ressourcenplanung* (dafür notwendig: Einschätzung des voraussichtlichen und Erfassung des tatsächlichen Arbeitsaufwandes für jeden Fehler): Offene Fehler pro Komponente / pro Entwickler / pro Priorität bzw. Fehlerklasse, bis zu einer Version oder einem Milestone noch zu behebende Fehler, pro Version / pro Zeitraum neu entstandene und behobene Fehler, durchschnittliche Dauer der Fehlerbehebung, durchschnittlicher Aufwand der Fehlerbehebung und des Tests, Anzahl der Iterationen bei der Fehlerbehebung, Anzahl der Reopens (Fehlbehebungen), …

Projektmanagement

Allgemeines Projektmanagement ==› siehe eigene Lehrveranstaltung

Vorstufen (Bedarfserhebung, Strategie- bzw. Ziel-Festlegung, …) ==› fehlen hier

Unterscheidung *Projekt / Produkt* ==› ignorieren wir hier
(ganz grob: 1 Release eines Produktes entspricht etwa einem Projekt)

Ablauf eines SW-Projektes ==› nächste Kapitel

Abteilungen rund um die SW-Entwicklung

- **Das Produktmanagement:**
  ==› Definiert die *Strategie*, die Ziele, die Schwerpunkte
  ==› Legt die *Eigenschaften* des Produktes fest (das "Was")
  ==› Legt Zeitvorgaben und verfügbare *Ressourcen* fest
  ==› Setzt *Prioritäten* bei Termin- und Ressourcenkonflikten, bei kritischen Fehlern
  ==› Trifft die *Release-Freigabe-Entscheidung*
- **Die Entwicklung:**

  … besteht aus **Entwurf/Design** und **Implementierung**

  ==› *Entwickelt* das *Produkt* samt *interner Dokumentation*
  ==› Setzt *vorbeugend qualitätssichernde Maßnahmen* um
  ==› Je nach Struktur: Entwickelt Teile der Produkttests
  (kann auch in der Qualitätssicherung sein)
- **Produktintegration, Build- und Release-Management:**
  ==› *Integriert* die Komponenten und Dokumentation zum Gesamtprodukt
  ==› Ist verantwortlich für die *Versionsverwaltung*,
  legt die Tags und Branches im Versionsbaum an
  ==› Ist verantwortlich für die *Build-Tools und -Skripts*
  ==› Erstellt den *Nightly Build*, erstellt Builds für beliebige Versionsstände,
  baut und archiviert jegliche Software, die das Haus verlässt
  (Releases, Bugfixes, Kunden-Sonderversionen…)
  Kein Bit direkt aus dem Engineering verlässt das Haus!
  ==› Verwaltet alle eingesetzten *Software-Werkzeuge* und deren Versionsstände
- **Qualitätssicherung:**
  ==› *Prüft* das Produkt (siehe oben), analysiert dessen Performance
  ==› Je nach Struktur: Entwickelt *Produkttests*, *Testdaten*, *Testtools*, …
  ==› *Unterstützt* den Support und die Beratung:
  Stellt *Kunden-Szenarien und Kunden-Fehler* nach, baut Kunden-Prototypen
  (im Extremfall: Hält Kopien aller wichtigen Kunden-Installationen bereit)

  Sollte **getrennt** von der Entwicklung sein!
- **Lokalisierung, Dokumentation und Schulung:**
  ==› *Dokumentiert* das Produkt (nur *externe* Doku: gedruckte Doku, Online-Hilfe)
  ==› *Lokalisiert* das Produkt (erstellt die Länder-Versionen)
  ==› Unterstützt das *Marketing* (Verpackung, Produkt-Flyer, Online-Produktinfo's,
  Screenshots und Videos, …)

==› Erstellt *Schulungsunterlagen* und *Trainings-Software*, hält Schulungen, …

- **Verkaufsunterstützung, Beratung, Support:**
  Berät und unterstützt den Kunden…

  … *vor* der Kaufentscheidung (Optimaler Einsatz des Produktes, Integration in die Kundenlandschaft, Dimensionierung der Hardware, Klärung technischer Fragen)

  … bei der *Einführung* des Produktes (Installation & Konfiguration, Datenübernahme, Customizing, Tuning, Schulung, …)

  … bei *Fragen* und *Problemen* (Hotline, …)

  … bei *kritischen Operationen* (Upgrade, Datenmigrationen, Desaster Recovery)

  … bei der *Betriebsüberwachung* (Problemfrüherkennung, Tuning, 7*24-Überwachung, …)

  ==› *Ist die wichtigste Quelle für Feedback und Inputs!*

- **Patent- und Lizenzmanagement:**
  ==› Schützt (patentiert/registriert) und verwertet (verkauft/lizensiert) *eigenes Know-How* und eigene Software
  ==› Arbeitet die eigenen *Lizenztexte* aus (eines der wesentlichen Ziele: Schutz vor Haftungsansprüchen!)
  ==› Verwaltet Verträge, Lizenzschlüssel, …
  ==› Prüft eigene Entwicklungen auf *Verletzung fremder Rechte*
  ==› Prüft *rechtliche Unbedenklichkeit* verwendeter Fremdsoftware (z.B. im Hinblick auf "infektiöse" Open-Source-Lizenzen)
  ==› *Lizensiert* fremdes Know-How (fremde Patente) und fremde Software

Der ewige Konflikt…

zwischen *Manager* der Entwicklungsabteilung (Testabteilung, …) und *Senior Programmer* (= erfahrenster / bestqualifizierter Entwickler):

Der **Senior Programmer**
… möchte nicht und/oder kann nicht managen und Personal führen.
… ist bei Entscheidungen zu technisch orientiert, hat kein Gefühl für Zeit und Geld.
… verabscheut nicht-technische Planung, nicht-technische Dokumente.
… sollte die zentrale Rolle im technischen Design des Produktes spielen.
… ist (sofern er motiviert ist) unersetzlich für die technische Überlegenheit des Produktes und die Lösung kniffliger Probleme (komplexe Algorithmen, Fehlersuche, …).
… sollte seine Weisheit und Erfahrung an möglichst viele junge Entwickler weitergeben.
… erwartet viel Freiheit.
… verdient entweder für seine Qualität zu wenig (==› Motivation?), oder ist der Firma in einer nicht-leitenden Position in Relation zu teuer.

Typischerweise ist ein Senior Programmer daher in einer leitenden Position falsch eingesetzt, sowohl aus fachlicher Sicht als auch im Hinblick auf seine persönliche Job-Zufriedenheit: Er sollte an der technischen Basis bleiben, hier bringt er den größten Nutzen für die Firma.

Der **Manager**
… spricht eine andere Sprache, hat eine andere Kultur und hat für seine Mitarbeiter

oft nicht nachvollziehbare Werte bzw. Vorgaben von oben.
… muss seine Verantwortlichkeiten gegenüber der Geschäftsleitung einhalten.
… hat meist zu wenig technische Kompetenz
für technisch fundierte Entscheidungen und für das Produktdesign.
… ist zu lang / zu weit weg von der Basis, macht keine technische Tagesarbeit mehr,
hat keinen Einblick in technische Details mehr.
… steht technischen Akut-Problemen oft hilflos gegenüber.
… wird deshalb von den meisten Entwicklern gering geschätzt und nicht respektiert.

Gerade mit den "alten Weisen" der Entwicklung ist der Konflikt und das beiderseitige seelische Unglück daher vorprogrammiert. Hier ist die Kunst der Balance gefragt.

Nach meiner persönlichen Erfahrung hat sich Folgendes bewährt:

- Die *untersten Teamleiter* sollten rein *technische Mitarbeiter* sein und zu ›› 50 % ihrer Zeit noch *technische Arbeiten im Tagesgeschäft* ausführen.
- Die wirklich *guten Senior Programmer* sollten keine leitende Funktion haben (außer vielleicht einem oder zwei persönlichen Mitarbeitern ihres Vertrauens), aber *etwas außerhalb der Hierarchie stehen* (nicht auf unterster Ebene unter einem normalen Teamleiter) und *nicht zu 100 % ihrer Zeit vorab fix verplant* werden.
- Ab der 2. Ebene von unten sollten die verantwortlichen Leiter zwar eher Management-orientiert sein, aber einen Senior Programmer oder einen "*Rat der (technischen) Weisen"* zur Beratung in technischen Dingen zur Seite gestellt bekommen, wobei die menschliche Kompatibilität ein wesentlicher Faktor ist.

  Der Senior Programmer bzw. der Rat der Weisen sollte dann auch die "technische" Kommunikation nach unten in beiden Richtungen übernehmen. Auf der anderen Seite sollte er das Recht haben, Dinge nach oben zu eskalieren, wenn er von seinem Manager übergangen wird.

Tools

- **Projekt- und Ressourcen-Management-Tools** (z.B. MS Projekt):
  - Verwalten den *Zeitplan & Ablauf* des Projektes:

    Dauer der einzelnen Teilschritte +
    Abhängigkeiten zwischen den Teilschritten =
    Zeitpunkt der Meilensteine

  - Verwalten die *Ressourcen-Belegung*
    (Welcher Mitarbeiter / welche Abteilung macht wann was?)

  ==› *Kopplung mit Bugtracker!*

  Sollte den Projektablauf nicht nur voraus planen,
  sondern auch *laufend prüfen und rückwirkend dokumentieren* **("Projekt-Tagebuch")**:

  - Wird der Projektplan eingehalten?
    (Abschluss- oder Testdokumente zu jedem Meilenstein pünktlich?
    Mitarbeiter wie geplant eingesetzt?
    Fehler und Änderungswünsche zum geplanten Zeitpunkt abgeschlossen?)
  - ==› Kostenrechnung!

    - ==› Evaluierung des Erfolges, Erkennen von Fehleinschätzungen, Schwachpunkten, …
    - ==› Lerneffekt, Feedback für zukünftige Projekte, organisatorische Änderungen
- **Mindmapping- und Brainstorming-Tools**, …

Die Spezifikation eines SW-Projektes

Unterscheide:

- **Anforderungsdefinition, "Lastenheft" und "Pflichtenheft"**:

  Was soll die Software leisten?

  "Externe" Spezifikation
  ==› Grundlage des *Tests*

  bei Projekten:
  ==› Technischer Teil des *Vertrags mit dem Kunden*!

- **Design- und Entwurfsdokumente, Modulspezifikationen**, …:

  Wie ist die Software aufgebaut?

  "Interne" Spezifikation
  ==› Grundlage der *Implementierung*

- Daneben: *Organisatorische Spezifikation:*
  Zeitpläne, Organisationsstruktur und Verantwortlichkeiten, finanzielle Mittel, …

Das Lastenheft und das Pflichtenheft

- Das **Lastenheft** (Anforderungskatalog, Requirements Specification)
  legt die *Anforderungen aus der Sicht des Kunden* fest:
  Was soll die Software können, welche Aufgaben soll sie erfüllen, …

  Bei Vergabe eines Software-Projektes mittels Ausschreibung:
  Das Lastenheft wird vom *Ausschreibenden (Auftraggeber)* erstellt
  und ist der technische Teil der *Ausschreibung*, d.h. legt fest,
  welche Anforderungen die Angebote erfüllen müssen.

- Das **Pflichtenheft** (Systemspezifikation, Feature Specification) beschreibt,
  welche *Fähigkeiten und Eigenschaften* der *Hersteller* in das Produkt einbauen will,
  um den Anforderungen des Lastenheftes zu entsprechen.

  Das Pflichtenheft beschreibt nur die *von außen* sicht- bzw. prüfbaren
  Eigenschaften und Fähigkeiten des Produktes, nicht seinen internen Aufbau.

  Bei Vergabe eines Software-Projektes mittels Ausschreibung:
  Das Pflichtenheft wird vom *Anbietenden (Auftragnehmer)* erstellt
  und ist der technische Teil des *Angebotes*, d.h. beschreibt,
  wie der Auftragnehmer die im Lastenheft festgelegten Anforderungen
  realisieren bzw. erfüllen will.

Beide können u.a. Angaben zu folgenden Punkten enthalten:

- **Funktionale Anforderungen:**

  Was soll das Programm können / tun / berechnen?

  Auch: Wie soll es auf Fehler, falsche Eingaben usw. reagieren?

- **Benutzerschnittstelle:**

  Aussehen und Verhalten des **GUI**
  (ev. durch nicht-funktionalen Prototyp spezifiziert)

  Wie weit muss das GUI behinderten-gerecht, konfigurierbar, … sein?

  Welche Sprachen / Zeichensätze sind zu unterstützen?
  (sind rechts-nach-links-Lokalisierungen usw. gefordert?)

  Programmaufruf, Optionen, …

- **Hardware-Umfeld:**

  Welche Prozessor-Plattformen sind zu unterstützen?

  Welche minimale und maximale Hardware-Konfiguration ist zu unterstützen?
  (Anzahl und Geschwindigkeit CPU, RAM, Plattenplatz, …)

  Welche Bildschirme (Größe / Auflösung!), Drucker (Protokoll, Farbe / SW) und andere Peripherie-Geräte (Scanner, Cardreader, …) sollen bedient werden?

  Welche Eigenschaften (Geschwindigkeit, Latenz) haben die Netzwerk-Verbindungen? Sind drahtlose Clients, Heimarbeitsplätze usw. zu bedienen?

- **Software- und Daten-Umfeld:**

  Welche Betriebssysteme sind zu unterstützen?
  (32 / 64 Bit? Welche Versionen? Welche Lokalisierungen?)

  Sind bestimmte Programmiersprachen oder GUI-Bibliotheken vorgeschrieben?

  Welche Fremdsoftware ist anzubinden und wie?
  Welches Datenbank-Produkt in welcher Version, welcher Webserver, welches Backup-Produkt, welche Betriebsüberwachungs-Software ist zu verwenden?
  Gibt es eine zentrale Benutzer- und Passwort-Verwaltung (“Single Login”)?

  Bei Web-Software:
  Welche Browser in welchen Versionen sind zu unterstützen?
  Welche aktiven Inhalte sind zulässig und welche nicht?
  Welche Web-Standards sind einzuhalten?

  Was soll wo in der Verzeichnis-Struktur stehen
  (z.B. eigene Bereiche für Massendaten, Trennung SW / Konfig / Daten / Doku)?
  Welche Dateien sollen gelesen / geschrieben werden, in welchem Format?
  Welche Import- / Export-Schnittstellen sind vorzusehen, welche Datenformate sind zu unterstützen (z.B. Druck / Archivierung: PDF)?

  Welche Altdaten sind zu übernehmen, in welchem Format liegen sie vor?

  Welche Kompatibilität zu Vor- und Folgeversionen des eigenen Produktes ist gefordert? (Extremfall *“Rolling Upgrade”* ohne Standzeit)
  Ist ein gemischter Betrieb verschiedener Client- und Server-Versionen gefordert?

- **Quantitative Anforderungen:**

Mengengerüst aller Daten, zulässige Begrenzungen für Daten

Anzahl und Aktivität der Benutzer

Durchschnittliche / maximale Antwort-Zeiten,
Laufzeiten nicht-interaktiver Operationen (Backup, Auswertungen, …)

==> *"Service Level Agreement"*

- **Verfügbarkeit:**

  Welche Verfügbarkeit (Ausfallzeit pro Jahr) ist gefordert?

  Welche Hochlaufzeiten bzw. Restore-/Recovery-Zeiten nach hartem Stop / Datenverlust sind zulässig?

  Muss Backup im laufenden Betrieb möglich sein? Upgrade im laufenden Betrieb?

  ==> *"Service Level Agreement"*

- **Cluster-Fähigkeit:**
  [Ein Cluster ist ein Verbund von zwei oder mehr Rechnern und zwei oder mehr Storage-Systemen, meist verteilt auf zwei geographisch getrennte Rechenzentren, zur Ausfallsicherheit bzw. zur Erhöhung der Verfügbarkeit: Jede Seite hat eine komplette Kopie der Daten. Fällt ein Rechenzentrum aus, übernehmen die Rechner des anderen Rechenzentrums deren Aufgaben, möglichst automatisiert und ohne dass die Anwender etwas merken.]
  Mit welchen Umschaltzeiten und Umschaltverlusten (z.B. neu Anmelden) läuft die Anwendung im Cluster?

  Welche Cluster-Konfiguration wird unterstützt?

  - *Aktiv / Cold Standby:* Die Reserverechner sind zwar gebootet, aber die Anwendung wird auf der Sekundärseite erst dann gestartet, wenn die Primärseite ausfällt.
  - *Aktiv / Hot Standby:* Die Anwendung auf der Sekundärseite ist gestartet und zieht ihren internen Status und Datenbestand ständig von der Primärseite nach (im Fall von Datenbanken oft mittels "Log Shipping"), bedient aber keine Benutzer, solange die Primärseite läuft.
  - *Aktiv / Aktiv:* Beide Seiten laufen gleichberechtigt im Normalbetrieb, teilen sich die Benutzerlast, und halten ihren internen Status und Datenbestand laufend gegenseitig synchron.

- **Administrative Handhabung:**

  Wie sieht die Installation / das Upgrade aus?

  Welche Tools sind mitzuliefert?

  Welche Arten von Backup (voll / inkrementell / mit Einzel-Restore-Möglichkeit) und Archivierung sind gefordert?

  Welche Features betreffend Logging und Betriebsüberwachung sind gefordert?

- **Datenschutz / Datensicherheit, …:**

  Geforderte Authentifizierung der Benutzer (Smartcard usw.)?

  Besondere Sicherheits- oder Datenschutz-Anforderungen,

Protokollierungs- und Archivierungspflichten?

Geforderte Zertifizierungen? (*Wer trägt die Kosten?*)

- **Lieferumfang:**

  In welcher Form wird das Ergebnis ausgeliefert?

  Welche Dokumentation / Hilfe / Schulungsunterlagen in welchen Sprachen sind mitzuliefern?

- **Abnahmekriterien:**

  Wie wird festgestellt, dass das Projekt fertig / der Vertrag erfüllt ist,
  d.h. unter welche Kriterien zahlt und übernimmt der Kunde das Produkt
  und entlässt den Hersteller aus seinen Pflichten?

  Welche Nachbesserungs- und Supportverpflichtungen
  (mit welchen *Reaktionszeiten*) gelten danach?

Tools:

MS Visio (für Graphen, Diagramme, ...) o.ä. hilft oft!
("Ein Bild sagt mehr als 1000 Worte.")

Weitergehende Tools zur technischen Unterstützung der Spezifikation (**UML-Tools**, ...!) bzw. *automatischen Generierung* von Spezifikations-Dokumenten sind Geschmackssache!

Der Entwurf / die Implementierungs-Spezifikation

Früher typischerweise Top-Down, heute meist *Komponenten- und / oder Daten-orientiert*.

Erste Frage daher:

- Welche *Daten* (Datenbanken) verwaltet die Anwendung?
- In welche *Komponenten oder Services* zerfällt die Anwendung?

==› Spezifikation des *Datenmodells* in der Datenbank / der Datenformate in Files.

==› Spezifikation der (Netzwerk-) *Protokolle / Schnittstellen* zwischen Komponenten / zwischen Client und Server.

Anwendungen sind heute oft verteilt, müssen **Multiprozessor-Systeme** oder (zwecks Skalierbarkeit) **mehrere Rechner** nutzen können.

Nächster Schritt in diesem Fall:

- Auf *welche Maschinen / welche Prozesse / welche Threads* wird die Anwendung verteilt?
- Wie geschieht das?
  Funktional (fixe Anzahl von Threads mit verschiedenen Aufgaben)?
  Master / Worker? (idente Threads,
  entweder fixe Thread-Anzahl mit Load Balancing / Request Queue
  oder ein Thread pro Request / pro Connect, ... ==› variable Thread-Anzahl)
- ==› *IPC-Design*: Welche Kommunikations- und Synchronisations-Mechanismen nutzen die Prozesse und Threads? Was gibt es an Shared Memories, Pipes, ...?

Letzte Schritte:

- Klassisches *Modul- und Klassen-Design*.
- Festlegung der *Algorithmen und Datenstrukturen*.
- ==> *Header-File* jeder Klasse als Klassen-Spezifikation.

Davon möglichst unabhängig:

- *GUI-Design*

Tool: UML ???

Grundprinzip:

MVC: Model View Controller (Modell-Präsentation-Steuerung)

- Saubere Aufteilung in drei Bereiche mit wohldefinierten (und möglichst schlanken) Schnittstellen.
- Jeder Bereich sollte austauschbar sein (z.B. WebGui statt eigenes GUI-Programm, In-Memory-Datenhaltung oder Cloud Storage statt On-Disk-Datenbank, ...) ohne die anderen Bereiche ändern zu müssen.
- **Model:** Das Datenmodell / der Datenspeicher (*Datenbank*, Datenstrukturen im Hauptspeicher, ...) samt *Zugriffsfunktionen*: Der einzige Bereich, der *persistente Anwendungsdaten* enthält! Enthält oft auch die unmittelbar mit den Daten verbundene Programmlogik (die Operationen auf den Daten, z.B. Suchen).
- **View:** Die Präsentation / das *GUI*: Enthält *keine Daten* und *keine Logik*, stellt nur dar und nimmt Benutzer-*Eingaben* entgegen (und validiert sie eventuell gleich).
- **Controller:** Enthält die globale *Ablauflogik* (nicht Daten-Logik) und *koordiniert* die beiden anderen Bereiche: Löst bei Benutzereingaben die entsprechenden Aktionen auf den Daten aus.

Typisch für viele heutige Anwendungen ist eine **Drei-Ebenen-Architektur**, die grob dem MVC-Modell entspricht:

- **Client** (heute oft: Web-Client, clientseitige Scripts) am Arbeitsplatz des Benutzers: Darstellung, Validierung der Inputs, *keine* persistente Datenspeicherung.
- **Frontend-Server** / Applikationsserver (heute oft: Webserver + serverseitige Scripts): Programmlogik, Sitzungsdaten, ...
- **Backend-Server** / Datenbankserver: Datenhaltung.

Software-Entwicklungs-Modelle

Jahrzehntelang galt ausschließlich das klassische "**Wasserfallmodell**", es orientierte sich am Top-Down-Entwurf:

- Anforderungsanalyse / Pflichtenheft / Projektplan
- Design / Implementierungsspezifikation
- Implementierung und Unit-Tests
- Integration und Systemtests
- Auslieferung / Abnahmetest / Wartung

Diese Schritte werden nacheinander *sequentiell* für das gesamte Projekt ausgeführt (jede

Phase beginnt erst, wenn die davor abgeschlossen ist, jede Phase endet mit *Meilensteinen und verpflichtenden Dokumenten*). Ergibt sich in einem Schritt ein Problem, muss das ganze Projekt (mindestens) einen Schritt *zurückgesetzt* werden.

Die Problematik ist die *lange, starre Laufzeit*, in der es nicht möglich ist, auf geänderte Anforderungen und Prioritäten zu reagieren, und die fehlende Interaktion mit dem Kunden während der Projektlaufzeit (nur ganz am Anfang / ganz am Ende!). Auch der erste lauffähige Code entsteht relativ spät, Prototypen sind nicht vorgesehen.

Weiters bewirken die zentrale, hierarchische Projektorganisation und die umfangreichen Dokumente mit einer frühzeitig vorgegebenen Aufteilung in Teilaufgaben oft ein Kommunikationsdefizit zwischen den einzelnen Abteilungen bzw. zwischen den Entwicklern der einzelnen Komponenten.

Das Ergebnis sind daher oft an der Realität / am Markt vorbei entwickelte oder bei Fertigstellung schon veralterte Software-Projekte und Produkte (und grob fehleingeschätzte Projekt-Laufzeiten und -Aufwände).

Das **V-Modell** ist eine *Vorschrift der bundesdeutschen Behörden*, wie Softwareentwicklung abzulaufen hat. Es ist eine Erweiterung des Wasserfall-Modells: Die Idee ist, das der *Zerlegung in immer kleinere Details* in der Analyse-, Entwurfs- und Spezifikations-Phase des Wasserfall-Modells spiegelbildlich ein *Zusammensetzen und Testen immer größerer Teile* nach der Implementierung gegenübersteht: Trägt man auf der x-Achse die Zeit und auf der y-Achse den Detailgrad (oben Gesamtsystem, unten Einzelfunktion) auf, ergibt sich ein V mit der eigentlichen Implementierung unten in der Mitte (Details siehe z.B. Wikipedia).

Zu jedem Schritt vor der Implementierung auf der linken Seite gibt es auf gleicher Ebene einen Schritt nach der Implementierung auf der rechten Seite, in dem getestet wird, ob das entwickelte Produkt den Dokumenten und Spezifikationen entspricht, die auf derselben Ebene vor der Implementierung festgelegt wurden.

Das V-Modell verträgt sich gut mit klassischen Entwicklungsmodellen und Top-Down-Entwurf, aber schlecht mit agiler Softwareentwicklung.

Eine Weiterentwicklung des Wasserfall-Modells ist das **Spiralmodell**, das die Phasen

- *Zielfestlegung*
- *Analyse*, Abwägung von Alternativen und Risiken, *Prototyp*
- *Entwicklung* und *Test*
- *Planung* der nächsten Runde

mehrmals iteriert (für *immer vollständigere bzw. detailreichere Prototypen* bzw. dann für das endgültige Produkt), um die **Laufzeit eines Zyklus zu verkürzen** und dazwischen immer wieder **Korrekturmöglichkeiten und Abstimmungen mit dem Kunden** zu erlauben.

Ebenfalls auf diesem klassischen, zyklischen Modell baut **RUP** (*"Rational Unified Process"*) auf, ein Konzept und Produkt (in Form von unterstützenden Tools) der Firma IBM / Rational. Die Gründer dieses Modells waren zugleich die *Erfinder von UML*, das Konzept und die Tools bilden daher alle Schritte eines Softwareprojektes in UML nach, UML ist das zentrale Spezifikations- und Entwurfsmittel.

Erstes wesentliches Gegenkonzept war dann "**Extreme Programming**" seit 1995:

- Auch *zyklisch* in Iterationen, aber *vielen und kurzen*: Alle 1-4 Wochen ein Zyklus.
- Möglichst *wenig Formalismus*, dafür in jeder Phase *viel informelle Kommunikation* (auch mit dem *Kunden*, der ja bei klassischen Modellen nur ganz am Anfang und Ende eingebunden ist!).
- Teamwork bzw. *Pair Programming* (zu *zweit* an einem Rechner / *einem Stück Code*).
- Statt kompletter Spezifikation und Projektplanung vorab:

  Prototyp bauen,
  Akzeptanz durch den Kunden testen,
  Akzeptiertes sofort implementieren und auch gleich testen,
  Prototyp erweitern,
  wieder Akzeptanz testen usw.,
  bis der Kunde zufrieden ist.

  ==› Nichts Unnötiges implementieren, "Featuritis" vermeiden.
  ==› Aktuelle / geänderte Anforderungen des Kunden sofort berücksichtigen.
  ==› Enger Zusammenhang mit "Rapid Prototyping".
  ==› Das Produkt entsteht inkrementell!

"Extreme Programming" ist ein Beispiel des Konzeptes **agiler Softwareentwicklung**, inzwischen haben sich mehrere herausgebildet ("Feature Driven Development" ist z.B. sehr ähnlich). Grundsätze sind **"einfach"** ("Keep it small and simple"), **kundennah**, **flexibel** und **teamorientiert** (keine fixe Zuteilung von Aufgaben an Einzelpersonen für die Dauer des Projekts - *jeder Code gehört jedem* im Team).

Das aktuell am meisten eingesetzte derartige Modell ist **Scrum**, diese Methode hat im Vergleich zu Extreme Programming klare Organisations- und Ablaufvorgaben (und ist daher erwas besser mit den Denkmodellen und Werten klassischer Manager vereinbar):

- Auch hier sind einzelne, **kleine Teilaufgaben** das zentrale Planungs- und Ablaufmodell.
- Sie werden in **täglichen Meetings** besprochen, priorisiert und zugeteilt, womit sich ein **ein-Tages-Arbeitszyklus** ergibt.
- Über dem 1-Tages-Zyklus steht ein **1-Monats-Zyklus**, in dem das Produkt um **eine Version** und **vorab ausgemachte Features** weiterentwickelt wird. Nach jedem dieser Zyklen erfolgt ein **Abgleich mit dem Kunden** und den Anforderungen und eine **Auswahl und Priorisierung der Features** für das nächste Monat.
- Auch hier sind die **Teams klein** (Richtwert 7 Personen).

Nachteil dieser Modelle ist der Widerspruch zu klassischem Management mit der Unmöglichkeit einer frühzeitigen Zeit- Kosten- Und Aufwandsplanung, das Fehlen rechtlich belastbarer formaler Dokumente, und die beschränkte Team- bzw. Projektgröße.

Ein weiterer Extremfall ist die **testgetriebene Entwicklung**: Hier werden die *Tests vor dem eigentlichen Code* geschrieben und stellen *zugleich die Spezifikation* der Funktionalität dar. Auch dieses Modell geschieht in *ganz kleinen Iterationen*: Tests für ein Feature schreiben, gleich danach das Feature implementieren, dann Tests für das nächste Feature festlegen usw..

Wichtige Zeitpunkte eines Software-Projektes

Typischerweise hat ein klassisches Software-Projekt folgende wichtige Zeitpunkte, deren Ziel-Datum und Erreichungskriterien vorher festzulegen sind (diese Zeitpunkte decken sich üblicherweise mit wichtigen Meilensteinen bzw. dem Beginn der nächsten Phase im Projektplan):

- **Feature Freeze** (Achtung, verschiedene Bedeutung, siehe unten):

  Zu diesem Zeitpunkt stehen die Features des Produktes (das "Was") fest, d.h. ab diesem Zeitpunkt werden *keine neuen Wünsche und Anforderungen* für die aktuelle Version mehr berücksichtigt.

- **Interface Freeze** (unter verschiedenen Namen bekannt):

  Zu diesem Zeitpunkt steht das *Design, d.h. die Schnittstellen* zwischen den Services / Modulen / Klassen fest. De facto sollten zu diesem Zeitpunkt alle *Header-Files* im finalen Zustand sein.

- **Code Freeze** (Achtung, verschiedene Bedeutung, siehe unten):

  Ab diesem Zeitpunkt dürfen keine neuen Features mehr implementiert und keine Design- oder Implementierungsänderungen bzw. Erweiterungen / Optimierungen mehr am Code vorgenommen werden, sondern *nur mehr Fehlerbehebungen*.

  Spätestens ab diesem Zeitpunkt (oft sogar schon beim Interface Freeze) wird in der Versionsverwaltung ein *eigener Branch* für dieses Release angelegt: Echte Weiterentwicklungen erfolgen im Head und betreffen diesen Branch nicht mehr.

- **Integrations- und Testfreigabe**:

  Zu diesem Zeitpunkt sollten die einzelnen Komponenten soweit fertig und fehlerfrei sein (Unittests laufen ohne grobe Probleme), dass mit dem *Build und den Tests des Gesamtproduktes* begonnen werden kann.

- **Produktionsfreigabe**:

  Code, Dokumentation usw. sind fertig: Die *Erstellung der finalen CD's* beginnt. Ab diesem Zeitpunkt können Änderungen nur mehr in Form eines *späteren separaten Wartungsreleases* erfolgen.

Oft wird auch folgende Bedeutung von Feature Freeze und Code Freeze verwendet:

- Der Feature Freeze entspricht dem oben dargestellten Code Freeze: Ab diesem Zeitpunkt werden *nur mehr Fehlerbehebungen* (und ev. funktions- und *schnittstellenneutrale Verbesserungen bestehenden Codes*) vorgenommen, und zwar vom Programmierer in Eigenverantwortung.
- Der Code Freeze wird *restriktiver* ausgelegt: Ab diesem Zeitpunkt darf der Programmierer nicht mehr nach eigenem Dafürhalten Fehler beheben, sondern nur mehr jene Behebungen machen, die *von der Projektleitung ausdrücklich angeordnet* werden. Verbunden ist das üblicherweise mit einer *Sperre der Versionsverwaltung*, sodass jeder einzelne Checkin im Release-Branch nur mehr nach individueller Freigabe durch die Projektleitung möglich ist.

Für jedes Wartungs-Release ("Service Pack") werden dieselben Schritte wieder durchlaufen.

Bei Produkten und Projekten, die mehrere Plattformen unterstützen, kommt ein weiterer Schritt dazu:

- **Portierungs-Freigabe**:

  Der Code auf der Primärplattform (Entwicklungs-Plattform) ist so weit fertig und stabil, dass mit der *Portierung* und den plattform-spezifischen Anpassungen für die *Sekunddärplattformen (Portierungsplattformen) begonnen* werden kann.

- Die weiteren Schritte (Code Freeze, Integrationsfreigabe, Produktionsfreigabe) erfolgen auf den Sekundärplattformen üblicherweise ein paar Tage oder Wochen *nach* der Primärplattform.

Mit den oben genannten Schritten gehen folgende Produkt-Vorab-Releases einher:

- **Alpha-Release**:

  Das Alpha-Release ist eine *Vorschau* auf das kommende Produkt, ein Prototyp, der in allen wesentlichen Punkten funktional und final sein sollte: Es vermittelt einen ersten Eindruck von den wichtigsten neuen Features und dem zu erwartenden GUI, ist aber üblicherweise noch bei weitem *nicht Feature Complete* und auch *nicht qualitätsgesichert*.

  Zweck des Alpha-Releases ist es üblicherweise, *Feedback* zur Akzeptanz und dem Nutzen geplanter neuer Features oder größerer GUI-Redesigns zu bekommen:

  Das Alpha-Release ist meist so früh im Produktzyklus angesiedelt, dass dabei gewonnene Erkenntnisse noch in die Spezifikation und das Design des Produktes bzw. der Benutzeroberfläche einfließen können.

- **Beta-Release**:

  Das Beta-Release ist normalerweise *Feature Complete* (d.h. entspricht in der Funktionalität dem finalen Produkt) und soweit fehlerfrei, dass man damit im wesentlichen arbeiten kann.

  Zweck des Beta-Releases ist es, das fast fertige Produkt *einem breiten Test* (in verschiedensten Umgebungen und Anwendungs-Szenarien) zu unterziehen und *Rückmeldungen betreffend Fehlern und Inkompatibilitäten* zu bekommen.

  Diese Beta-Tester sind meist *ausgewählte Kunden* mit überdurchschnittlichem technischen Background.

Maßnahmen und Werkzeuge der Leitung

Durch folgende Vorgaben kann und soll die Leitung einer Entwicklungsabteilung Einfluß auf die Entwicklung (und damit auf Produktivität und Code-Qualität) nehmen:

- Auswahl des *Software-Entwicklungs-Modells*, Vorgabe der *Organisation* und der Abläufe (siehe oben), klare *Verantwortlichkeiten*
- Zu erstellende *Dokumente und Protokolle*
- **Programmier-Richtlinien** (==› automatische Prüfung!)
  & klare Regelung für Ausnahmen:
  - Formatierung (==› IDE-Einstellungen!), Klammer-Politik
  - Kommentierung, Kommentare für Dokumentationstool

==› Richtlinien für interne Klassen-Dokumentation

- Namens-Richtlinien (Stil, Prefixes, Groß/Klein?), benannte Konstanten
- Modularisierungs- und Schnittstellen-Vorgaben
- Aufteilung in Files: Filenamen, Directory-Hierarchie
- Reihenfolge der Konstrukte in einem File
- Verbotene Konstrukte und Funktionen (Goto? strcpy/strcat?)
- Regeln betreffend Compiler und Lint:
  ==› Welche Warning-*Optionen* sind zu verwenden (automatisch!)
  ==› Welche Warnings sind akzeptabel, welche nicht (auch automatisch!)
- Limits (Codelänge pro Funktion, Schachtelungstiefe, …)
- Vorgaben betreffend Input- und Parameter-Prüfung, Fehlerbehandlung, Asserts, …
- Handhabung von Plattform- und Compiler-Abhängigkeiten (#if oder nicht?)
- GUI Design Guide
- Datenbank-Vorgaben (Tabellennamen, erlaubte SQL-Konstrukte, …)
- Textkonventionen (einheitliche Terminologie, einheitliche Benutzer-sichtbare Texte, …)

- *Einsatz von Werkzeugen*:
  - Richtlinien betreffend *Formatierungs- und Dokumentationswerkzeugen*
  - Build-Prozess
  - Einsatz der Versionsverwaltung, *Branch- und Release-Konzept, Verbindung Bugtracker / Versionsverwaltung* ("Kein Check-In ohne dazugehörigen Bug oder Feature-Request"), Check-In des "Drumherums" (Tests, Tools, Doku, …)
- Qualität:
  - Umfang und Methodik der *Tests*, zu erstellende Unit-Tests
  - Automatische Tests (beim Checkin / beim Nightly Build)
  - *Code Reviews* (siehe unten)
  - Geforderte Protokolle, *geforderte Testabdeckung*, geforderte Fehlerfreiheit
  - ***Qualität ist vor allem eine Frage des Stellenwertes, der ihr von der Leitung demonstrativ gegeben (oder nicht gegeben) wird!***
- Und die führungspsychologischen "Musts":
  - Selbst eine technische Ahnung haben
  - Wissen, was läuft: Kenntnis der Probleme
  - Reden, reden, reden…

Ein oft vergessener Grundsatz der Softwareentwicklung:

Es gibt kein "später" ("später" = "nie"),
keinen Nachbearbeitungs-, Doku- oder Reinschrift-Schritt

==› *gleich richtig, sauber und ordentlich machen (lassen)!*

Code Reviews

Ein ganz wesentliches, aber in den meisten Firmen stark unterentwickeltes oder falsch eingesetztes Werkzeug zur *Erhöhung der Code-Qualität* (sowohl im Sinne von *Fehlerfreiheit* als auch im Sinne von *technisch hochwertigem Design und gutem Stil*) sind **Code Reviews**: Das **Lesen von Programmcode durch andere als den Autor**, primär durch erfahrene Senior Programmer, und zwar sowohl im Hinblick auf *formale Vorgaben* und Stil, als auch im Hinblick auf *technische Qualität* (geeignete Algorithmen und Datenstrukturen, Effizienz, Robustheit, Wart- und Erweiterbarkeit, …).

Der willkommene Nebeneffekt ist, dass Code Reviews auch einen großen Beitrag zur *Weiterentwicklung bzw. Fortbildung junger Entwickler* leisten (einerseits, weil ihre eigene Arbeit diskutiert und verbessert wird, und andererseits, weil sie fremden Code sehen und seine Stärken und Schwächen besprochen werden) und die Kenntnis und das *Verständnis des Codes auf einen größeren Mitarbeiter-Kreis erweitern* (weil ja bei den klassischen Softwareentwicklungs-Modellen normalerweise jeder nur mit Scheuklappen seinen eigenen Code kennt und sonst nichts, was ein großes Risiko für die Firma darstellt).

Wie bzw. wann kann Code Review stattfinden?

- Schon beim Schreiben des Codes, indem Code *von vornherein zu zweit entwickelt* wird: “Peer programming” oder “pair programming”.

  Diese Methode hat zwar den größten Lerneffekt und liefert oft die technisch besten Lösungen, kann aber *menschlich kompliziert* sein und hat den Nachteil, dass oft *beide Autoren gegenüber gemeinsam entwickelten Fehlern blind sind* (und gerade solche dem Autor entgangenen eigenen Fehler sollen ja durch Code Review gefunden werden).

- Beim Einchecken: *Jedes Checkin muss von zwei Personen durchgeführt werden:* Dem Autor und einem zweiten Programmierer (Senior Programmer, Bürokollege, Gruppenleiter), der ebenfalls in der Versionsverwaltung aufscheint und damit die *Verantwortung* übernimmt, den Code durchgesehen zu haben.

  Diese Variante ist normalerweise der beste Kompromiss aus Effizienz bzw. Aufwand, Wirkung und menschlichen Faktoren (vor allem dann, wenn der Autor seinen Reviewer wählen kann und eine gute Gesprächsbasis zu ihm hat).

- *Nachträglich*, d.h. nach dem Checkin.

  Nachträgliche Code Reviews erfolgen meist nicht flächendeckend bzw. einzeln für jedes aktuelle Checkin: Typischerweise wird stattdessen stichprobenartig eine Komponente oder ein Modul ausgewählt, das dann *komplett analysiert* wird.

  Natürlich ist ein nachträgliches Review auch formlos und spontan durch einen einzelnen Mitarbeiter möglich (z.B. durch den Gruppenleiter, einen Senior Programmer oder einen QA-Mitarbeiter), aber üblicherweise werden nachträgliche Reviews *sehr formal* von eigens dafür gebildeten größeren Teams in mehreren eigens angesetzten Sitzungsterminen bzw. Hearings des Autors und mit formalen, schriftlichen Review-Protokollen bzw. Änderungsaufträgen durchgeführt:

  - Präsentation des betreffenden Codes durch den Autor in einem Vortrag.

- Verteilung des Codes auf die Reviewer.
- Review (jeder Review-Teilnehmer für sich im stillen Kämmerlein), jeder verfasst einen schriftlichen Review-Bericht.
- Sitzung der Review-Kommission, Verfassen einer schriftlichen Code-Beurteilung an den Autor.
- Stellungnahme des Autors zum Bericht (schriftlich / in einem Vortrag), Diskussion über die vorzunehmenden Änderungen, schriftlicher Änderungsauftrag.
- Durchführung der beschlossenen Änderungen.
- Nochmaliger Review (Kontrolle der vorgenommenen Änderungen), Iteration des Vorganges oder Abschlussbericht.

Aus dem letzten Absatz ergeben sich auch die Gründe für den s**chlechten Ruf des Code Reviews** gerade in großen und traditionell streng strukturierten Unternehmen:

- Die Prüfung ist normalerweise *alles andere als zeitnah* und findet nur mit Glück aktuell dazugekommene Fehler.

  Da die *Fehler meist viel zu spät gefunden werden* (in fortgeschrittenen Stadien des Projektes bzw. überhaupt erst nach der Auslieferung der Version), ist eine Korrektur bzw. eine Verbesserung des Designs dann im Normalfall kaum noch möglich bzw. sehr aufwändig.

  Oft passiert es sogar, dass der für den untersuchten Code zuständige Entwickler sich *nicht mehr im Detail an diese Arbeit erinnern* kann, bzw. dass der Code überhaupt schon komplett "verwaist" ist.

- Der Aufwand ist sehr hoch, die Effizienz (aus Kostensicht) ist schlecht: Das liegt primär am *hohen formalen Overhead*: Mit vertretbarem Aufwand kann *nur ein Bruchteil des gesamten Codes* in diesem formalen Rahmen mit Sitzungen und Protokollen reviewed werden, vieles bleibt ungeprüft.

- Weiters liegt das an der Tatsache, dass bei der Gesamt-Analyse eines Moduls meist *sehr viel alter Code mitreviewed* wird, der ohnehin schon seit langem durch die QA gegangen und praxiserprobt ist (und zwar vielleicht nicht optimal, aber wohl schon *frei von signifikanten Fehlern*).

  Eine *Konzentration nur auf neue, fehlerträchtige Codestückchen* (nur aktuelle Checkins) wäre im Hinblick auf die *Ausbeute an gefundenen Problemen wesentlich effizienter* (außer es ist das explizite Ziel, "Altlasten" zu beurteilen).

- Der Review wird vom Betroffenen mehr *als Strafgericht denn als anregende Diskussion empfunden* (zumal die Protokolle ja teilweise sogar aufbewahrt und zur *Personalbeurteilung* herangezogen werden), vor allem, wenn das Ergebnis nur schriftlich ergeht und *keine Diskussion* möglich ist.

  Die psychologische Wirkung ist höchst negativ, weil gerade in derart konservativen Unternehmen die "mein Code gehört mir"-Mentalität weit verbreitet ist und die Mitarbeiter schon ein grundsätzliches Problem mit der Diskussion ihrer Werke haben.

  Der *Weiterbildungseffekt und die anderen positiven Nebenwirkungen eines Code*

Reviews bleiben damit weitgehend aus und werden vom Management auch von vornherein gar nicht beabsichtigt oder berücksichtigt.

- Der Review wird von allen als *sinnlose zeitliche Belastung* empfunden: Vom Geprüften, weil er üblicherweise seinen Code in einem Vortrag präsentieren und später eine Antwort auf das Review-Protokoll verfassen muss, und von den Prüfern, weil mit der Review-Tätigkeit viele Sitzungen, Protokolle usw. verbunden sind.

  Grundsätzlich überwiegen bei dieser Methode zeitlich gesehen *formale Aktivitäten und unproduktive Sitzungen* oft den technischen Anteil, was für die beteiligten meist hochkarätigen Techniker frustrierend ist. Technische Diskussionen kommen oft zu kurz.

  Außerdem ist es praktisch unzumutbar, die *üblicherweise großen Brocken* eines formalen Modul-Reviews *konzentriert durchzuarbeiten*: Das konzentrierte Lesen hunderter oder tausender fremder Code-Zeilen in einem Rutsch (über Stunden) überfordert den besten Programmierer…

Daher ergeben sich folgende Empfehlungen für einen erfolgreichen Code Review:

- Möglichst zeitnah (am besten *vor* der Qualitätssicherung!), auf neue Codestücke und Änderungen konzentriert.
- Laufend, in kleinen Portionen.
- Flächendeckend (zumindest für alle Neuentwicklungen / Änderungen).
- Informell, mit hohem Diskussionsanteil und wenig Overhead.
- In kleinen, menschlich kompatiblen Teams, in positivem Klima und mit positiven Zielen (außer in hartnäckigen Fällen ohne Archivierung oder Rückmeldung an die Personalabteilung / den Vorgesetzten).

Demgegenüber spielt Code Review in Open-Source-Projekten seit jeher eine wesentliche Rolle und ist dort auch besser in der sozialen Kultur verankert.

Software-Entwicklung am Beispiel des Linux-Kernels

Der Linux-Kernel ist ein für seine Größe (über 20 Mio. Codezeilen) und seine sowohl organisatorische (hunderte Entwickler auf der ganzen Welt verteilt) als auch technische Komplexität sehr gut und effizient entwickeltes und sehr fehlerfreies Softwareprojekt!

Folgende Kernpunkte zeichnen die Linux-Kernel-Entwicklung aus:

- Klare *Programmier-Richtlinien*, die informell, aber zumindest in Kernbereichen bei Bedarf doch hartnäckig durchgesetzt werden (und Tools, die zumindest grobe Schnitzer erkennen).
- Sehr viel (öffentliche) *Kommunikation und technische Diskussion*
  (==› Linux Kernel Mailing List + Listen für jedes Sub-Projekt).
- Entwicklung ist getragen von erfahrenen *"Senior Programmers"* ("kennen jede Zeile Code"), die das letzte Wort haben: Sehr technisch orientiert, kein Management-Einfluß.
- Extrem gründlicher *Code Review*:

    - Jede Änderung / Erweiterung muss *vor* der Aufnahme in den Kernel auf irgendeiner für das betroffene Subsystem relevanten Mailing-Liste oder der Haupt-Mailing-Liste *veröffentlicht und diskutiert* worden sein.
    - Jede Änderung muss neben dem Autor selbst von einem (bei zentralen Änderungen von mehreren) anerkannten *Senior Programmern geprüft und unterschrieben werden*. Diese "Signed off by ..." sind dokumentiert und nachvollziehbar (in der Versionsverwaltung).

  Und das alles, *bevor* eine Änderung in den Stamm-Kernel eingecheckt wird.
- "Politik der kleinen Schritte": *Patches müssen klein und überschaubar sein* (große Patches sind auf viele kleine Patches aufzuspalten), jeder Patch sollte *nur eine gedankliche Änderung umfassen* und nicht mehrere Dinge, jeder Patch muss eine Erklärung / einen Kommentar beinhalten.
- Klarer und *kurzer Versionszyklus* mit vergleichsweise *langer Stabilisierungsphase*:
    - Unmittelbar nach dem Release einer neuen Version *2 Wochen "Merge Window"*, in dem die *Änderungen* für die nächste Version aus den Subsystem-Entwickler-Repositories in den Hauptzweig eingebracht werden.

      Sonst muss eine Änderung meist auf das nächste Release warten...
    - Danach *6 - 10 Wochen Stabilisierungsphase* bis zum neuen Release: Nur *Bugfixes* und ev. kleine Änderungen / Erweiterungen im Hauptzweig.

      Die Freigabe richtet sich nach der Qualität, nicht nach einem fixen Datum.
- "Release often, release early":
    - *Wöchentliche öffentliche Beta-Releases* ("-rc") zum Testen unmittelbar vom Ende des Merge-Windows bis zum Release, *tägliche Snapshots*.
    - Auch nach dem Release: Bei Bedarf kurzfristige, im Schnitt ein- bis zweiwöchentliche *Pflege-Releases* (unabhängig vom neuen Hauptzweig, nur wesentliche Bugfixes) bis zum nächsten Release, bei ausgewählten Versionen auch länger ("LTS-Releases", min. 2 Jahre unterstützt).
    - Eigener Bereich ("staging"), wo unreife / experimentelle Treiber und Code-Teile reifen können ==> werden immer mitgebaut und mitveröffentlicht, damit man sie testen kann, aber sind klar als "nicht produktiv" gekennzeichnet.
- Informelle, aber *sehr breite Test-Basis*, laufende *tägliche Tests* (manuell und automatisiert, auch betreffend Performance).
- Nutzung zahlreicher *Tools*, vor allem Nutzung einer eigenen, sehr ausgeklügelten und für das Linux-Entwicklungsmodell optimierten *Versionsverwaltung*:
    - *Verteilte Versionsverwaltung* mit vielen (hunderten) *lokalen Repositories*, in denen *vorab entwickelt und getestet wird*, und einem *zentralen Hauptzweig*.
    - Unabhängig von Hauptzweig und Entwickler-Repositories gibt es einen zentralen Experimental-Zweig ("linux-next"), in dem nichttriviale Neuerungen längere Zeit ausprobiert werden können, bevor sie in den Hauptzweig kommen.
    - Häufige, einfache und performante Synchronisation der Stände in beide

Richtungen (alle oder ausgewählte Änderungen).

- *"Bisect-Tool"* zum raschen Lokalisieren des schuldigen Patches bei neuen Fehlern.
- Absolute *Fälschungssicherheit der Versionsgeschichte*.
- Eigenes *Tool*, das über *Patch-Postings auf den Mailing-Listen* und deren Status, Diskussion und Revisionen *Buch führt* und die gemailten Patches mit tatsächlich in der Versionsverwaltung eingecheckten Patches abgleicht.

  Automatische Übernahme des Code-Textes einer Patch-Mail als Patch in die Versionsverwaltung, automatische Übernahme des normalen Mail-Textes als Patch-Kommentar, des Betreffs als Checkin-Titel, automatische Übernahme von Autor usw..

Weitere Tools: in *semantischer, statischer Checker* ("Sparse") und einige *kleinere Prüf-Skripts* für zahlreiche typische, vor allem Kernel-spezifische *C-Probleme*.

Benutzeroberflächen

Ziele

- *Intuitiv bedienbar:*
    - Menüpunkte in gewohnter Anordnung, mit üblichen Shortkeys, mit gewohnter Funktion, …
    - Sinngemäß dasselbe für Dialog-Buttons und Toolbar-Elemente (z.B. Funktion / Anordnung / Reihenfolge von Ok / Apply / Cancel).
    - In allen Dialogen: Korrekter Default-Button, korrektes vorselektiertes Dialog-Element, korrekte (und effiziente!) Tab-Reihenfolge.
    - Verwendung von System-Standard-Dialogen (File Open, Print, Farb-Wahl,…).
    - Verwendung von System-Standard-Icons.
- *Effizient bedienbar:*
    - Optimiert für *häufige* Operationen (Toolbar-Icon, Shortkey, …).
    - History der zuletzt geöffneten Files, der zuletzt gesuchten Wörter, der zuletzt selektierten Fonts, … bzw. Vorbelegung der Dialoge mit den zuletzt verwendeten Werten.
    - "Save Session": Ansicht, Fenstergröße, offene Dokumente, …
    - Automatische Vervollständigung, andere Input-bezogene Unterstützung (Tooltips, …).
    - Funktionalitäten an den aktuellen Kontext anpassen: Rechtsklick-Menü, kontext-sensitive Hilfe, …
- *Barrierefrei:*
    - Konfigurierbar: Übergroße Fonts (für alles, d.h. incl. Menüzeile, Tooltips, …), übergroße Icons, übergroßer Cursor, …
    - "High Contrast"-Theme (schwarz auf weiss) und "Inverse"-Theme (weiss auf schwarz) muss funktionieren.
    - "Maus-freie" Bedienung muss möglich sein: Shortkeys für alle Menüs, Dialog-Buttons, …
    - Keine zu komplexen Tastenkombinationen bzw. Umschalt-Möglichkeit auf sequentielle Kombinationen (Kombinationen aus nacheinanderfolgenden statt gleichzeitigen Tasten).
    - Eignung für "Grobmotoriker":
        - Geschachtelte Menüs sind *sehr* ungünstig!
        - Keine zu kleinen Klick-Ziele in Dialogen.
        - Alternativen zu Drag-and-Drop mittels Maus.
    - Farben dürfen nicht allein zur Darstellung wesentlicher Information benutzt werden (Farbenblindheit!).

    - Sounds dürfen nur unterstützende, keine wesentliche Funktion haben (Gehörlose).
    - Keine permanent blinkenden Anzeigen (Epileptiker!).
- *Flexibel betr. Anzeige:*
    - Alle in Frage kommenden Bildschirm-Größen und -Auflösungen müssen gut bedienbar sein:
        - Achtung auf zu große Dialoge bei kleinen Auflösungen!
          Ev. manuelles / automatisches Dialog-Resize,
          im äußersten Notfall *scrollbare* Dialoge.
        - Achtung auf zu kleine Fonts, Icons, … bei extrem hochauflösenden Schirmen.
    - Flexibilität betr. Bildschirm-Format (von 16:9 quer bis 16:9 hoch).
    - (Früher: Eignung für Schwarzweiss-Anzeige.)
- *Flexibel betr. Eingabe:*
    - Je nach Plattform sind nur 1 oder 2 Maustasten garantiert, nicht 3 oder mehr! Auch das Scrollrad ist optional.
    - Eignung für langsame oder sehr ungenaue Zeige-Geräte (Touchscreen, Touchstick, Touchpad) ==› Große Zielflächen!

      Besonders bei Industrie-Steuerungen: Handschuhe, Kälte, …
    - Maus-Zieh-Operationen und Gestures nur optional,
      komplexe Klick-Folgen nur optional.
    - Keine Abhängigkeit von nicht garantiert vorhandenen Tasten
      (==› Netbook- bzw. Notebook-Tastatur: Kein numerischer Tastenblock, keine rechte Window- und Ctrl-Taste, …).
- *Systemangepasst:*
    - Beachtung der System-Einstellungen betr. Schriftart und -größe, Theme oder Farben, Icon-Theme, Sound, …

      Achtung: Dialoge müssen mit der Schriftgröße *wachsen*!
    - Beachtung der System-Einstellungen betr. Keyboard (Autorepeat, Shift Lock oder Caps Lock, …) und Maus (vertauschte Button-Reihenfolge, Doubleklick-Intervall und -Radius, DnD-Radius, …).
    - Beachtung der systemweiten *Ländereinstellungen*.
- *Robust gegen unbeabsichtigte Aktionen:*
    - Rückfrage bei allen heiklen Operationen (Abschaltbar!).
    - Keine “gefährlichen” Default-Buttons.
    - Wo immer möglich / sinnvoll: Undo / Redo!
- *Nicht arbeits-störend:*
    - Keine sich dauernd bewegenden Elemente ==› Lenkt ab, macht nervös (Negativbeispiel: “Karl Klammer”).

Auch blinkender Cursor abschaltbar!

- Tooltips und andere "on the fly" eingeblendete Informationen / Auswahldialoge abschaltbar.
- Keine unerwartet aufgehenden Dialoge, die sich in den *Vordergrund drängen*
  Problem: Bekommen gerade getippten Input, der für ein ganz anderes Fenster gedacht war (*"Focus Stealing"*).
  Besser: Notification-Möglichkeit des Systems nutzen!
- Niemals System-modale Dialoge (Problem: Stört extrem, wenn man andere Anwendungen bedienen möchte), nach Möglichkeit auch keine Anwendungs-modalen Dialoge (Problem: Verhindert z.B. Resize des Hauptfensters).
- Wenn Sounds, dann abschaltbar (z.B. Großraumbüros!).
- "Selbst-umordnende" (häufig benutzte Einträge oben) und "selbst-ausblendende" (selten benutzte Einträge ausgeblendet / erst nach Klick auf "…") Menüs werden auch oft als störend empfunden
  ==› zumindest abschaltbar!
- Design-Grundprinzip wie so oft:
  "KISS: Keep it small and simple"
  "Weniger ist mehr"

  Konzentration der Haupt-Benutzeroberfläche auf das Wesentliche / auf die oft benutzten Optionen und Features: Selten benutzte Dinge in Subdialogen unterbringen / optional machen / …

- *Feedback gebend:*
  - *Jede* Benutzer-Interaktion (Tastendruck, Klick, …) muss *sofort* eine erkennbare und sinnvolle Reaktion des Programmes auslösen (damit der Benutzer weiß, dass er erfolgreich geklickt usw. hat)!
  - Wenn die Reaktion nicht sofort möglich ist (Berechnung dauert, …): Zumindest sofort den Cursor auf Sanduhr umstellen!
  - Wenn die Operation lange dauert:
    Fortschrittsbalken oder Restzeit anzeigen, ev. Abbruch-Möglichkeit bieten
  - Wenn der Programmstart länger dauert: Splash Screen vorsehen!
- *Konfigurierbar:*
  - Konfigurierbare Toolbars, ev. konfigurierbare Menüs.
    Achtung: Toolbars sind optional, deren Funktion muss auch über Menü's erreichbar sein!
  - Konfigurierbare Shortkeys.
  - Wahl zwischen integrierten Toolbars (oben / seitlich) oder Subwindow-Toolbars, vor allem bei großen Toolbars.
  - Wahl zwischen Tabbed Single Window und Multi Window.
- *Individuell erkennbar, "Corporate Identity"*

Vorgehensmodell

- *Ziele* in Einklang bringen:
    - Was will (nicht soll!) der User mit dem Produkt machen?
      Wie will der Benutzer mit dem Produkt arbeiten?
    - Was soll das Produkt können?
- Analyse der *Fähigkeiten* des Users:
    - Was hat er für Vorkenntnisse? Was hat er für technisches Verständnis?
    - Welches Vorprodukt ist er gewohnt?
    - Ist er explorativ? Umstellungs- bzw. lernwillig? Neuem aufgeschlossen?
    - Was sind seine kognitiven / sensorischen / körperlichen Fähigkeiten?
- Analyse der *Erwartung* des Users:
    - Wie erwartet der User, dass sich das System verhält?
    - **Welche Bedien-Schritte möchte der User der Reihe nach machen, um seine Arbeit zu erledigen?**

==› Ev. *verschiedene GUI's pro Benutzer-Rolle:*
Normaler User, Power-User, Administrator, …

- *Blick* zur Konkurrenz / auf bestehende Lösungen!
  (**Achtung:** Viele GUI-Ideen sind patentiert, Bedienkonzepte und User Interfaces sind im Unterschied zu reiner Software patentierbar!)
- *Prüfen*: Passt das GUI-Konzept zur Hardware (Bildschirmgröße, Grafik-Leistungsfähigkeit, …) und Software (Programmiersprache, unterstützte Web-Standards, …)?

Quellen

- ISO 9241 (im Besonderen: Teil -110, früher -10)
- *User-Interface-Guidelines* der betreffenden Plattform
- Vorschriften betreffend Barrierefreiheit ("BITV")

Methoden und Werkzeuge

- "Einem typischen Benutzer über die Schulter schauen" (z.B. bei Schulungen):
    - Wo hat er Nachdenk-Pausen, wo sucht er, wo braucht er die Hilfe?
    - Vor allem: Was macht er immer wieder falsch?
    - Welche Features findet bzw. nutzt er nicht?

  ==› Rapid Prototyping! (je früher umso besser!)
- Professionelle GUI-Analyse:
  Automatisierte Aufzeichnung und Auswertung des Benutzerverhaltens:
    - Nachdenkpausen (Zeiten zwischen Keyboard- und Maus-Events), Warten bis sich der Tooltip öffnet, Hilfeaufrufe

- Aufzeichnung vergeblich angeschauter Menüs und vergeblich aufgerufener Dialoge.
- Benutzungshäufigkeit bestimmter Features, Shortkeys, ...
- Analyse und Auswertung des Mausweges
  (gezielt oder wirr suchend? Immer wieder die falsche Stelle ansteuernd?)
- Eye Tracking (Wohin schaut der Benutzer? Schaut er konzentriert auf eine Stelle oder oft suchend / abgelenkt hin und her?)
- Stress-Analyse
- Oft propagiert: “Think aloud”

  Funktioniert nicht gut: Gerade in den kritischen Phasen (spontan irritiert oder konzentriert denkend) vergisst man auf das Sprechen ...