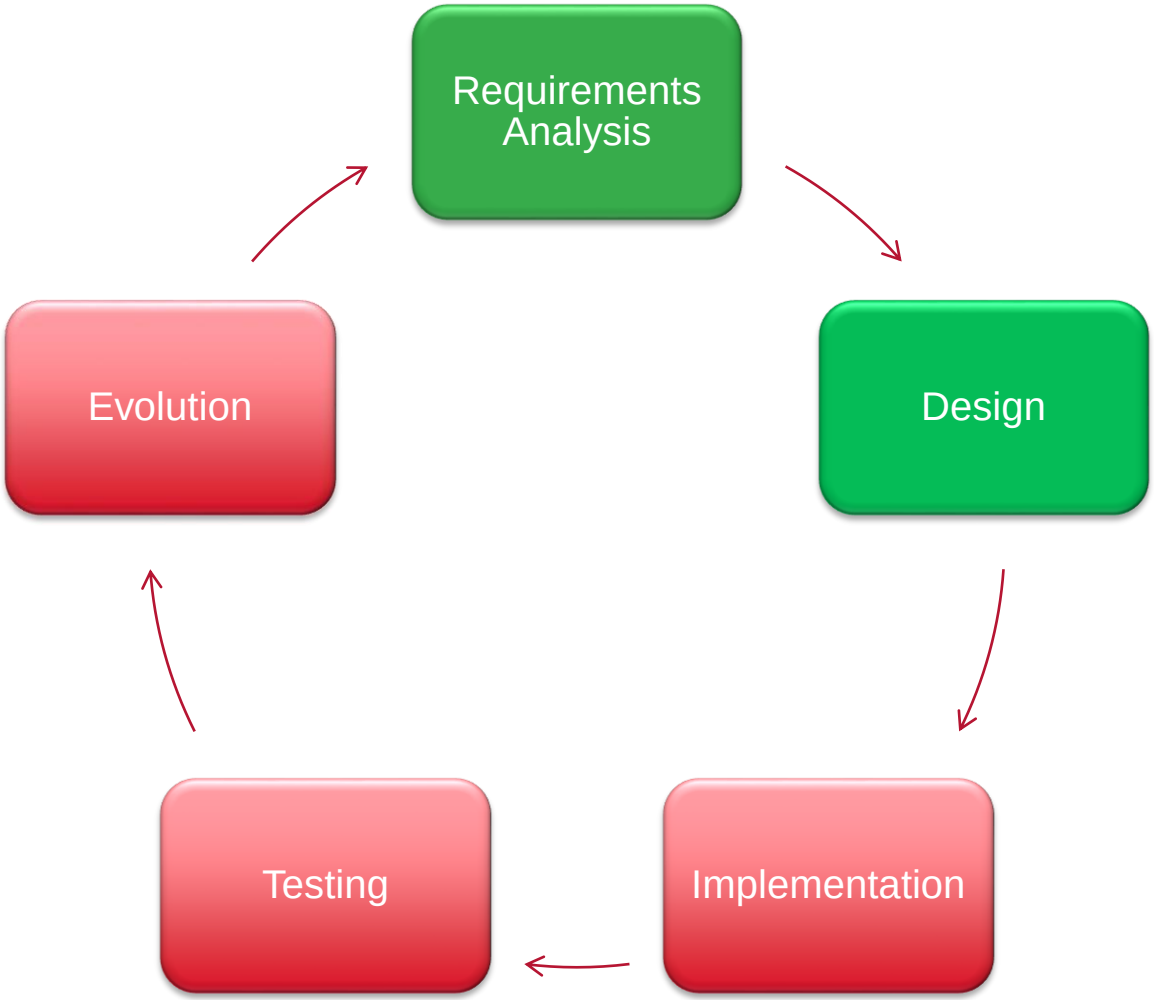
Media Engineering Programming Tools

R. Weller

University of Bremen, Germany

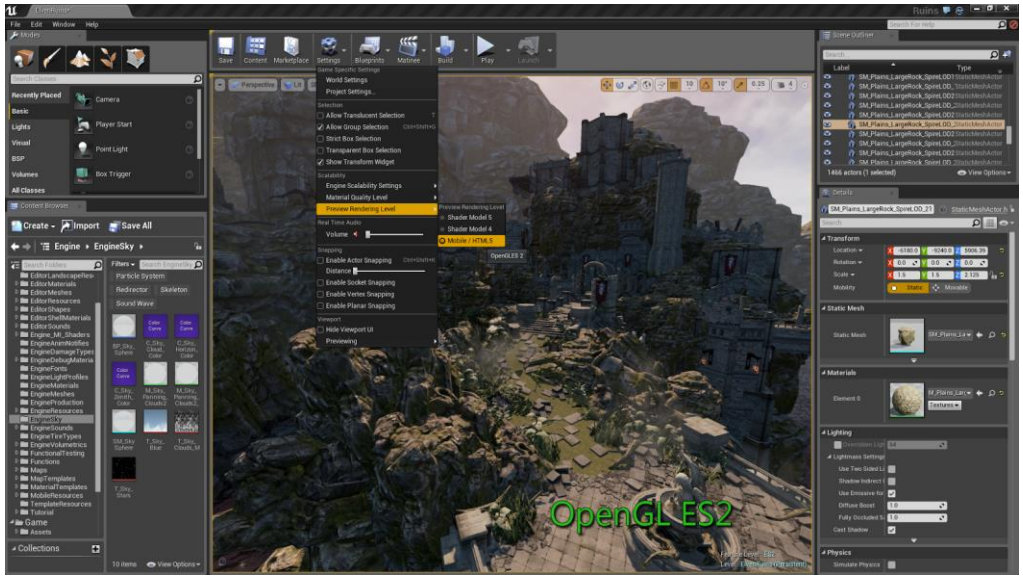
cgvr.cs.uni-bremen.de

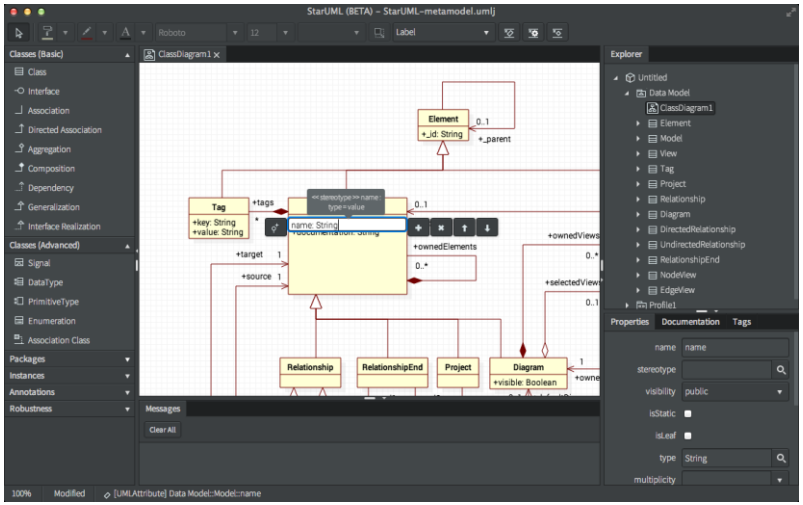
Der Software Development-Lifecycle



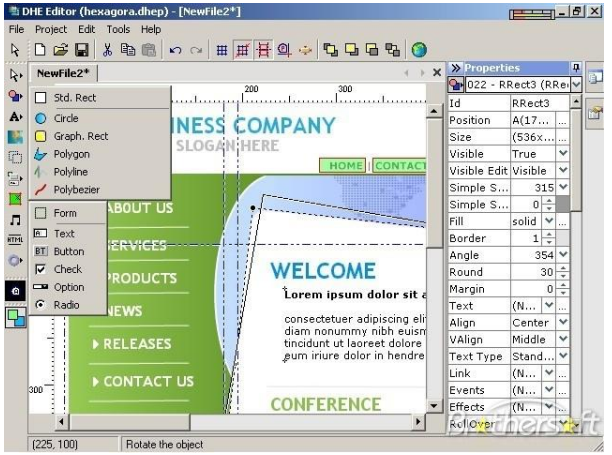
Also endlich?

Einige Tools kennen wir schon



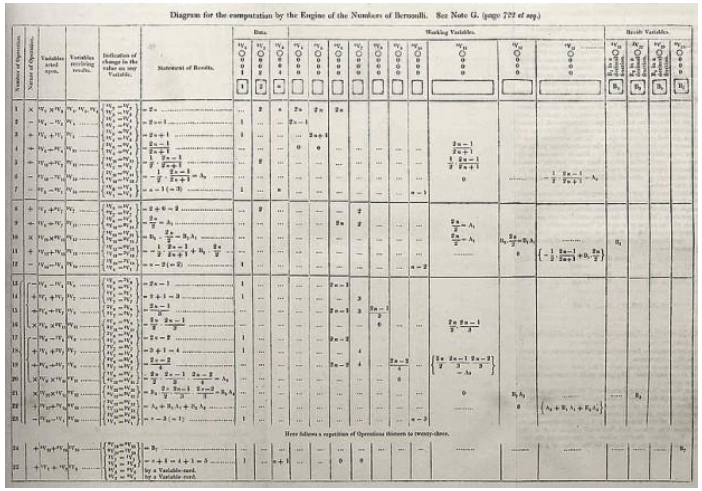


Viele Werkzeuge für digitale Medien

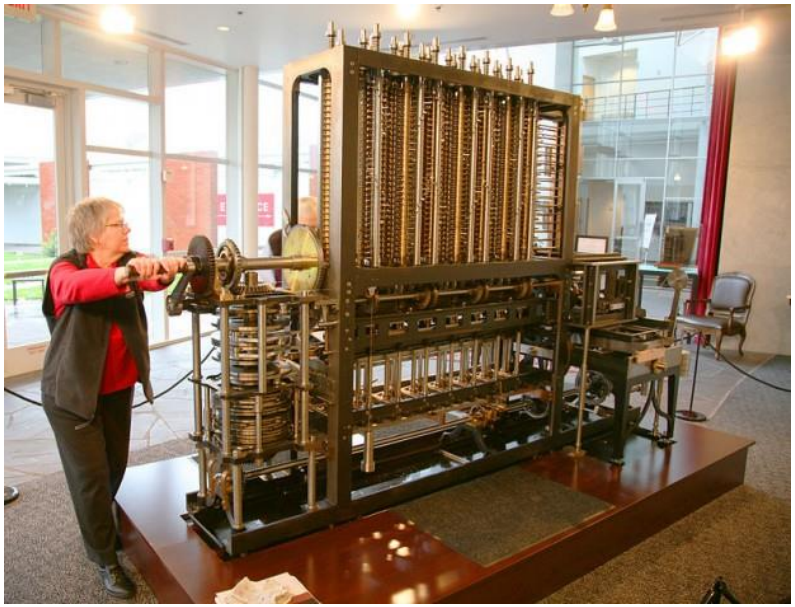


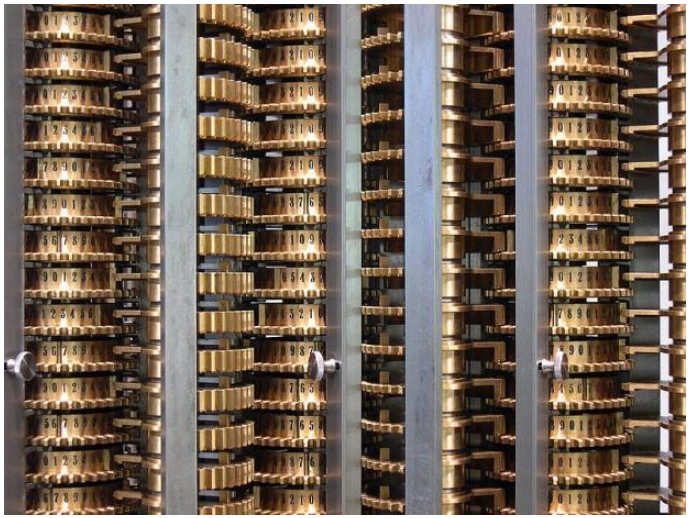
- In dieser Vorlesung beschränken wir uns auf Programmierwerkzeuge

Programmierung – Die Anfänge

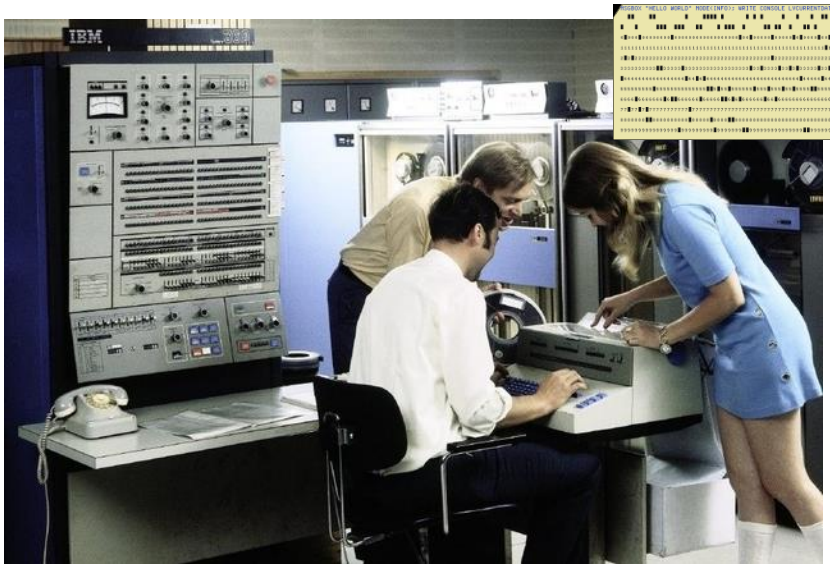


Ada Lovelace
(1815 – 1852)



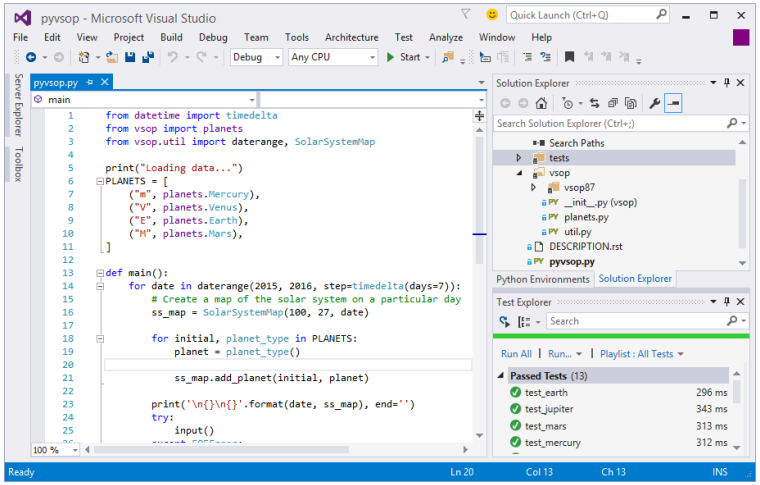


Programmierung im Wandel der Zeit

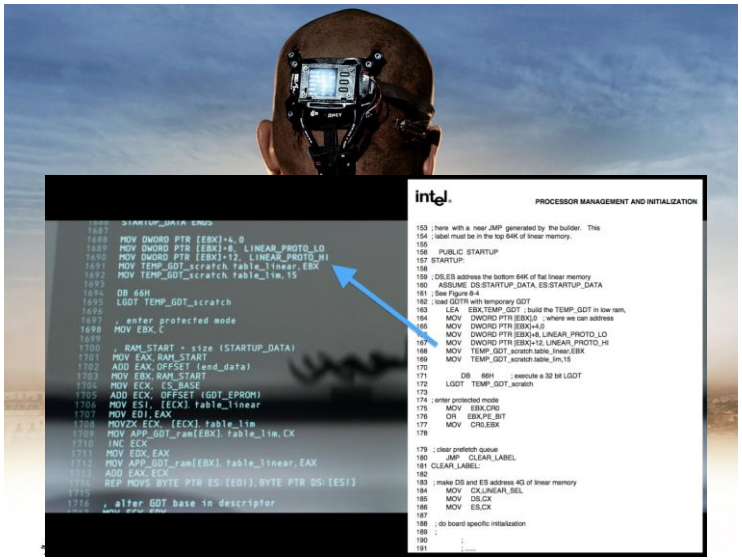


1969

1980



Heute



2154?

Weg zum Ausführbaren Programm

- Quellcode
  - In C++:
    - Header als *.h
    - Implementation als *.cpp – Dateien speichern

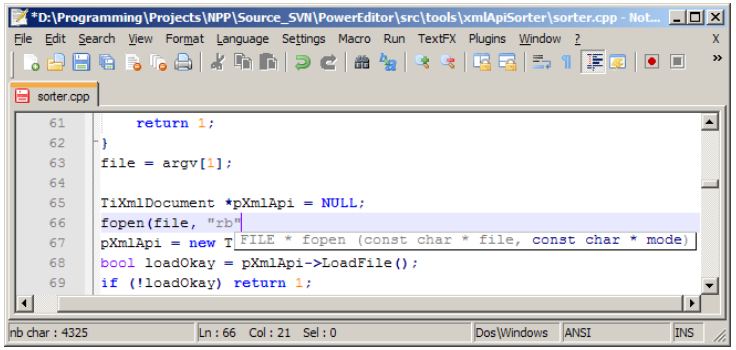
```
void CMymfc30bView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CMymfc30bDoc* pDoc = GetDocument();
    if(m_tracker.HitTest(point) == CRectTracker::hitMiddle) {
        COleDataSource* pSource = SaveDib();
        if(pSource) {
            // DoDragDrop returns only after drop is complete
            CClientDC dc(this);
            OnPrepareDC(&dc);
            CPoint topleft = m_rectTracker.TopLeft();
            dc.LPtoDP(&topleft);
            // 'point' here is not the same as the point parameter in
            //  OnDragEnter, so we use this one to compute the offset
            m_dragOffset = point - topleft;  // device coordinates
            pDoc->m_bDragHere = TRUE;
            DROPEFFECT dropEffect = pSource->DoDragDrop(
                    DROPEFFECT_MOVE|DROPEFFECT_COPY, CRect(0, 0, 0, 0));
            TRACE("after DoDragDrop -- dropEffect = %ld\n", dropEffect);
            if (dropEffect == DROPEFFECT_MOVE && pDoc->m_bDragHere) {
                pDoc->OnEditClearAll();
            }
            pDoc->m_bDragHere = FALSE;
            delete pSource;
        }
    }
    else {
        if(m_tracker.Track(this, point, FALSE, NULL)) {
            CClientDC dc(this);
            OnPrepareDC(&dc);
            // should have some way to prevent it going out of bounds
            m_rectTracker = m_tracker.m_rect;
            dc.DPtoLP(m_rectTracker); // Update logical coords
        }
    }
    Invalidate();
}
```

- Compiler
  - Übersetzt Quellcode in maschinenlesbare Sprache

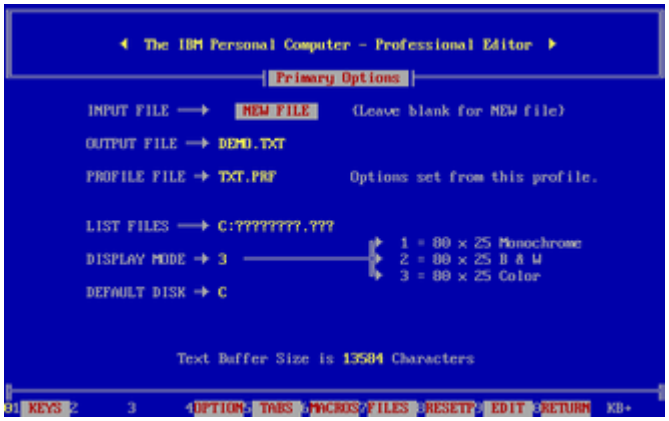
- Linker
  - Fasst mehrere kompilierte Module zu Gesamtprogramm zusammen
  - Symbolische Adressen von Funktionen oder Variablen werden in Speicheradressen umgewandelt

Quellcode Editoren

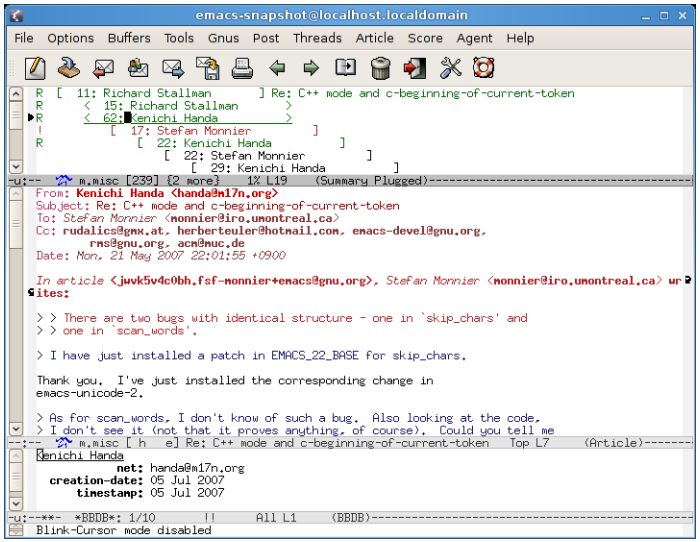
- Features
  - Syntax-Highlighting
  - Suchen und Ersetzen
  - Code-Faltung
  - Codevervollständigung
  - Calltips
  - Automatische Einrückung
  - Symbolbrowser
  - Makros



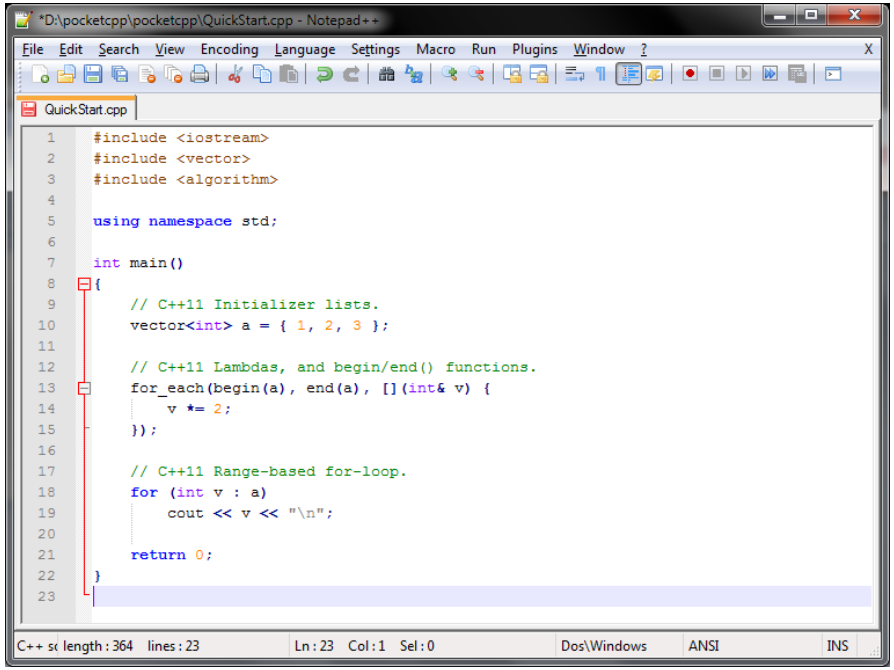
Einige Quellcode-Editoren



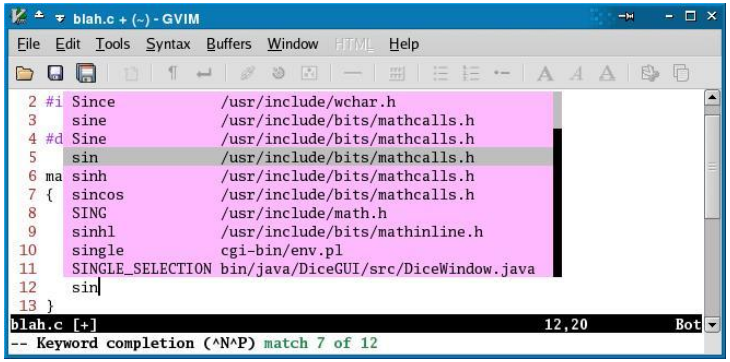
IBM Professional Editor



Emacs



Notepad++



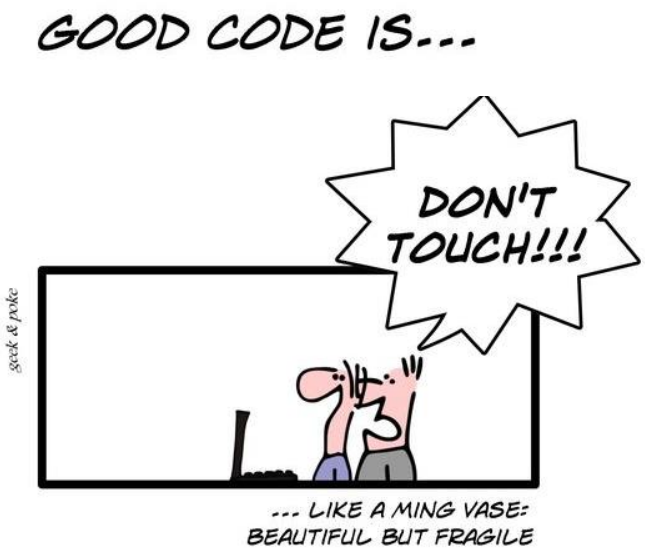
Vi

Warum braucht man schönen Code?

- „Schönen“ Code schreibt man nicht (nur) für sich selbst
  - Programmieren heute meist Teamarbeit
    - Kollegen müssen den Code auch lesen
  - Wiederverwendbarkeit
    - Eventuell will man Codeteile in Zukunft in anderen Projekten wieder verwenden
  - Wartung
    - 80% der Lebenszeit entfallen auf Wartung
    - Programmierer warten selten selbst

Wann ist Code schön?

- Ziel: Der perfekte Code ist
  - Lesbar
  - Verständlich
  - Wartbar
- Elemente guten Programmierstils
  - Festlegung von Namenskonventionen
    - Benennung von Klassen, Variablen, Funktionen
  - Code-Strukturierung
    - Einrückungen
    - Leerzeichen
    - Zeilenanzahl
  - Dokumentation
    - Kommentare



Standards?

- Einige Qualitätsnormen im Softwarebereich fordern die explizite Anwendung von Regelwerken
  - Beispiel: MISRA-C (1998)
- Tatsächlich hängen die Regeln von vielen Faktoren ab
  - Programmiersprache
  - Projektgröße
  - Historische Entwicklung
  - …
- Beispiele
  - Google Java Style (19 Seiten)
  - GNU Coding Standards (85 Seiten)
  - Linux Kernel Coding Style (3 Seiten)
  - CGVR-Programmierrichtlinien

Bekannte Namenskonventionen

- Ungarische Notation
  - Entwickelt von Charles Simonyi
  - Lange Zeit von Microsoft verwendet
  - Beispiele:
    - Variablennamen setzen sich aus (Präfix) (Datentyp) (Bezeichner) zusammen
    - ichArray, uuluwchArray
  - Namen wegen Herkunft des Erfinders und des exotischen Aussehens
  - Linus Torwalds Meinung: „Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged—the compiler knows the types anyway and can check those, and it only confuses the programmer”
- Ähnliche Konventionen
  - Reddick-Namenskonvention (Microsoft .NET)
  - Leszynski-Namenskonvention (Microsoft Access)

CGVR-Namenskonventionen

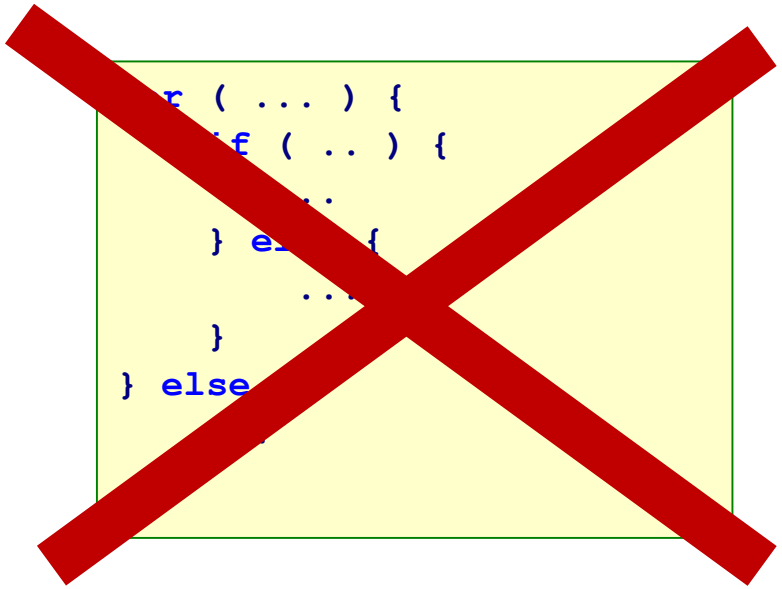
Art	Beispiel	Erläuterung
Lokale Variablen	mylocalvar my_local_var	Alles klein
Instanzvariablen	m_var	m für „member“
Klassennamen	MyCoolClass	Anfang Groß (ohne „C“ für Klasse)
Methoden	calcSomeCoolValue()	inCaps, klein anfangen
Konstanten	const int MaxNum = 10;	Anfang groß
Defines	#define MY_DEFINE	all-caps mit Unterscores
Typedef	SomeThingT	Wie Klassen mit Suffix „T“
Enum-Typen	myEnumTypeE	Wie Variablen, Suffix E
Enum-Member	COL_TRUE, COL_FALSE	Wie Defines
Template-Parameter	<MyTypeT>	Wie Klassennamen, Suffix T

CGVR-Konventionen für Einrückungen

- Allman/“East Coast“-Style
- Einrückung: 4 Leerzeichen
  - Keine Tabulatoren verwenden
- Klammern beginnen in eigener Zeile
  - Leichtere Zuordnung

```
for( ... )
{
        if( ... )
        {
                ...
        }
        else
        {
                ...
        }
}
```

- Nicht K&R-Style verwenden
  - Vorteil: Code kompakter
    - Aber bei heutiger Monitorauflösung kein Argument mehr
  - Nachteil: Öffnende Klammern werden leicht übersehen



Weitere CGVR-Konventionen

- Nicht mit Leerzeichen sparen

```
for(
      obj->M(i)=-1;
}
```

```
for( i = obj->begin(); i < obj->l() && obj->f(i) != -1; i++ )
{
      obj->f(i) = -1;
}
```

- Maximale Zeilenlänge: 80 Zeichen
  - Kürzere Zeilen leichter lesbar
    - Vergleiche mehrspaltigen Satz in Zeitungen
  - Formatierung bleibt beim Ausdrucken vorhanden
  - Bei zu großer Schachtelungstiefe eher in Funktionen auslagern
  - Verhindert zu lange Namen
    - Namenswahl extrem wichtig, lang Klassennamen deuten oft auf Designfehler hin

CGVR-Konventionen für Klassen

- Klassenlayout

```
class MyClass
{
        public:
                1. Konstanten
                2. Typen
                3. Variablen
                4. Methoden

        protected:
                ...

        private:
                ...
}
```

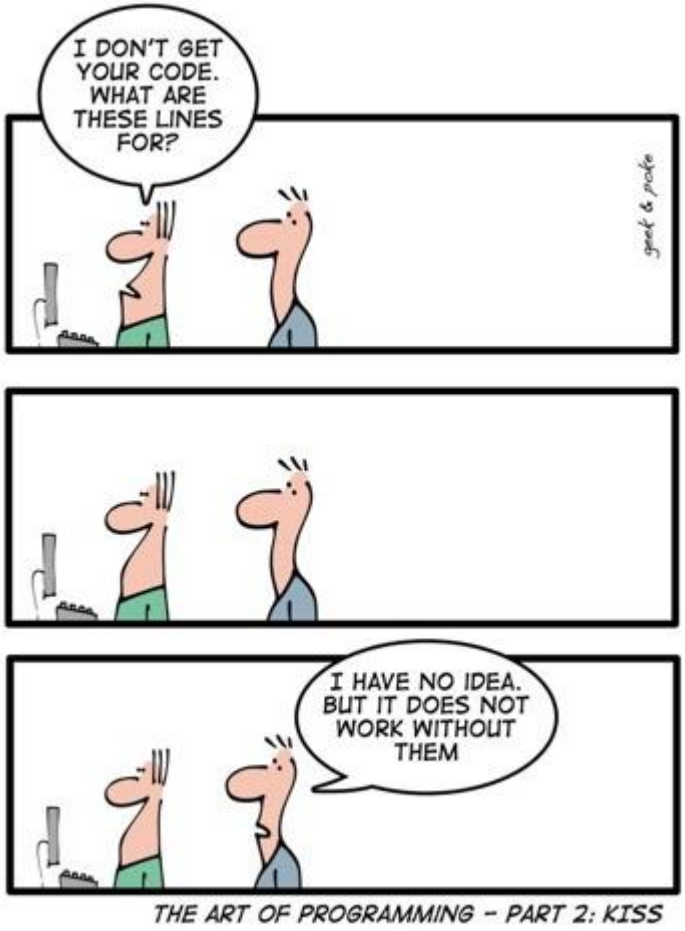
- Tabulieren von Variablen

```
int             x, y;           // dominant coord planes of polygon
int             xturns, yturns; // # turns of xslope, yslope
objPolyhedronP  p1, p2;
int             i;
```

```
int x, y;       // dominant coord planes of polygon
int xturns, yturns; // # turns of xslope, yslope
objPolyhedronP p1, p2;
int i;
```

Kommentare

- Mehrere Funktionen
  - Hilft eigene Gedanken zu ordnen (und damit sauberer zu programmieren)
  - Hilft anderen zu verstehen, was passiert
  - Code ist nicht selbsterklärend!
- Wie man es macht und wie nicht
  - „Später“ kommt nie!
  - Optimalerweise dann kommentieren, wenn Funktion „halb“-fertig ist
    - Gedanken noch frisch
    - Anschließend nochmal überprüfen



Arten von Kommentare

- Grundsätzlich 4 Arten von Kommentaren
  - Am Anfange von Klassen
    - Adressat: Verwender der Klasse
    - Big Picture
    - Besonderheiten (z.B. Compiler-Flags)
  - Vor Funktionen
    - Beschreibung
    - Parameter (besonders Ausgabeparameter kennzeichnen)
    - Pre-/Postconditions
    - Seiteneffekte
  - Vor Variablen
    - Immer vor globalen Variablen
  - Zwischen Code-Abschnitten
    - Beschreiben was gemacht wird, nicht wie

Beispiel: Kommentierung einer Funktion

- Diesen Header vor jede Funktion kopieren und ausfüllen!

```
/**  Do something (Einzeiler)
 *
 * @param param1    blubber (in)
 * @param param2    bla (out)
 *
 * @return
 *   Beschreibung der Rückgabe-Werte, z.B.
 *   -1 falls fehlgeschlagen, 0 wenn alles ok.
 *
 * Ausführliche, mehrzeilige Beschreibung, z.B.
 * Diese Funktion berechnet ...
 * Basiert auf dem Algorithmus von ...
 *
 * @throw Exception
 *   Genaue Beschreibung der Exceptions und deren Bedingungen, z.B.
 *   XOutOfMemory, falls kein Speicher mehr verfügbar.
 *   XCoffee, falls kein Kaffee mehr da.
 *
 * @warning
 *   Beschreibung von Eingabe-Parameter-Werten oder Zuständen,
 *   die zu Fehlern oder Abstürzen führen, die aber nicht abgefangen werden.
 *   (sollte nur sehr selten vorkommen!)
```

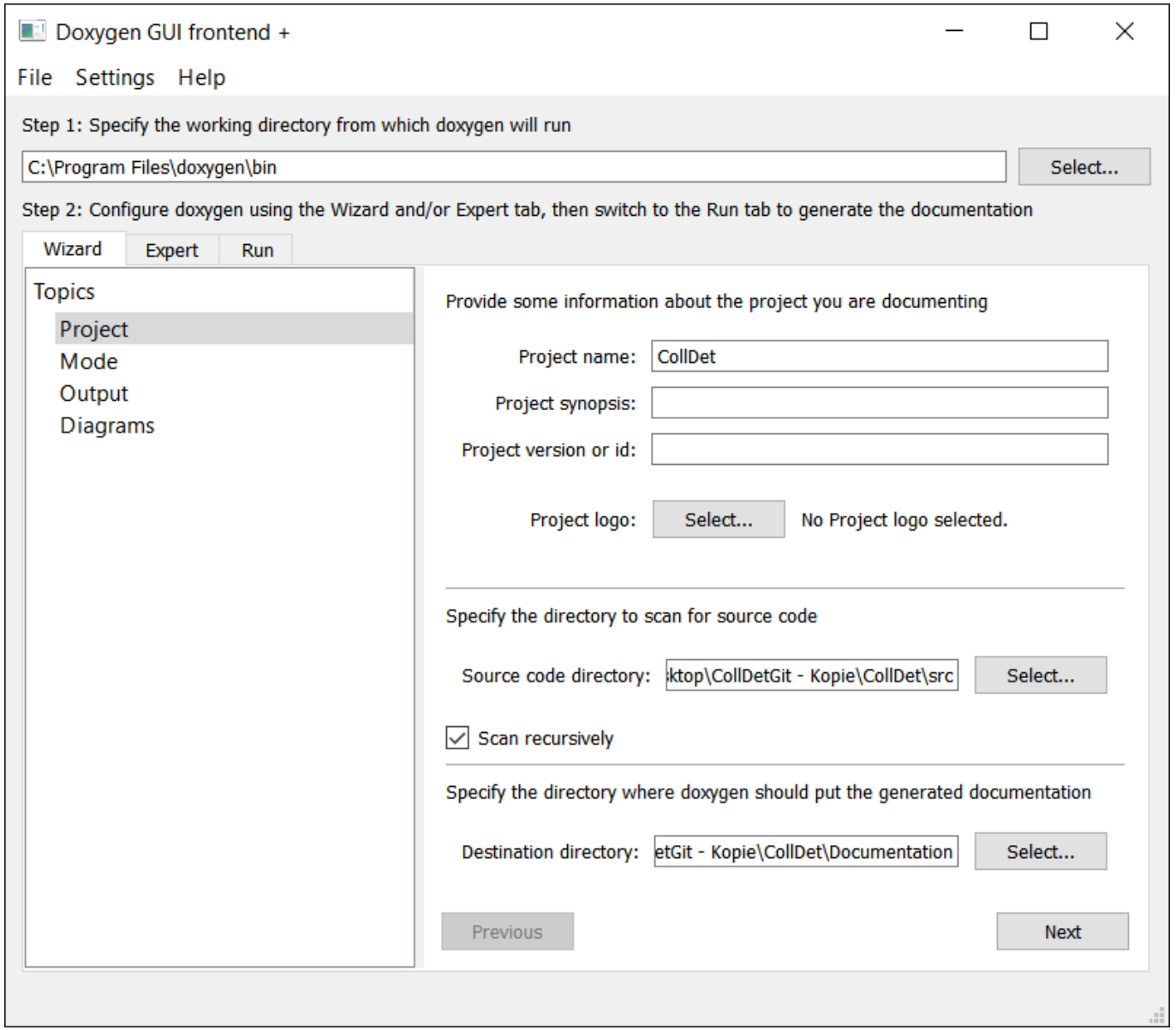
Beispiel: Kommentierung einer Funktion (cont)

```
 * @pre
 *   Genaue Beschreibung der Vorbedingungen, z.B.
 *   Erwartet, dass die Funktion init() schon aufgerufen wurde.
 *   Param1 muss von der Funktion blub() berechnet worden sein.
 *
 * @sideeffects
 *   Nebenwirkungen, globale Variablen, die verändert werden, .., z.B.
 *   @arg The global variable @c M_Interest will be modified.
 *
 * @todo
 *   Schneller machen.
 *
 * @bug
 *   Produziert einen core dump, wenn @a param1 = 0.0 ist.
 *
 * @internal
 *
 * @see
 *   Querverweise auf andere Funktionen, z.B., weil sie etwas ähnliches berechnen,
 *   oder weil es irgend eine Abhängigkeit zwischen beiden gibt.  Beispiel:
 *   eineAndereFunktion()
 *
 **/
```

Software-Dokumentationswerkzeuge

- Automatische Generierung von Dokumenten aus verschiedenen Quellen
  - Quelltext
  - UML-Diagrammen
- Export als browsbare HTML-, PDF- oder Epub-Dateien
- Beispiele:
  - Doxygen (C++)
  - Javadoc (Java)

Live-Demo Doxygen



Tools für schönen Code?

- Beautifier (auch Quelltext-Formatierer)
  - Formatieren vorhandenen Quelltext gemäß voreingestellter Regeln
  - Beispiel: Prettyprint
- Style Checker
  - Überprüft die Einhaltung eines vorher definierten Programmierstils
  - Führt im Gegensatz zum Beautifier aber meist keine Umformatierung durch
  - Unterstützen oft auch eine (sehr eingeschränkte) semantische Analyse

```
int
foo (int k)
{
  if (k < 1 || k > 2)
    {
      printf ("out of range\n");
      printf ("this function requires of 1 or 2\ ");
    }
  else
    {
      printf ("Switching\n");
      switch (k)
            {
            case 1:
              printf ("1\n");
              break;
            case 2:
              printf ("2\n");
              break;
            }
    }
}
```

```
int foo(int k){if(k<1||k>2){printf("out of range\n");
printf("this function requires a value of 1 or 2\n");}else{
printf("Switching\n");switch(k){case 1:printf("1\n");break;case 2:printf("2\n");break;}}}
```

Das Gegenteil – Code Obfuscators

- Macht das Gegenteil vom Beautifier
- Ziel: Soll Reverse-Engineering verhindern
  - Security-by-Obscurity
- Beispiel

```
void _(int __,int ___  ,int ____,int _____){((
___/__)<=_____)? _(__,___+_____,____,_____ ):!
(___ % __) ? _(__,___+_____,___ % __, _____):(
(___ % __)==(___ / __) && !____)?(printf("%d "
,(___ / __)),_(__,___+_____,____,_____)):((___
%__)>_____ &&(___        % __) < (___ / __))?_(
__,___+_____,____+!((___ / __)%(___ %    __)),
_____):(___        <__*__)?_(__,      ___+_____,
____,_____):0;} int main(void){_(50,0,0,1  );}
```

Kompilieren und Linken

- Speichern aller *.cpp und *.h-Dateien
- Aufruf des Compilers
  - Dieser erzeugt *.obj-Dateien als Zwischenergebnis
- Aufruf des Linkers
  - Zusammenfassen der *.obj-Dateien (und eventueller externer Abhängigkeiten) zu ausführbarem Programm

```
E:\Zhenya\Program\menuet\article_h11\examples\vc\hello>cl /c /O2 /nologo /GS- /G
R- hello.cpp kosFile.cpp kosSyst.cpp mcsmemm.cpp
hello.cpp
kosFile.cpp
kosSyst.cpp
mcsmemm.cpp
Generating Code...

E:\Zhenya\Program\menuet\article_h11\examples\vc\hello>link /nologo /manifest:no
 /entry:crtStartUp /subsystem:native /base:0 /fixed /align:16 /nodefaultlib hell
o.obj kosFile.obj kosSyst.obj mcsmemm.obj
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run

E:\Zhenya\Program\menuet\article_h11\examples\vc\hello>pe2kos hello.exe hello

E:\Zhenya\Program\menuet\article_h11\examples\vc\hello>
```

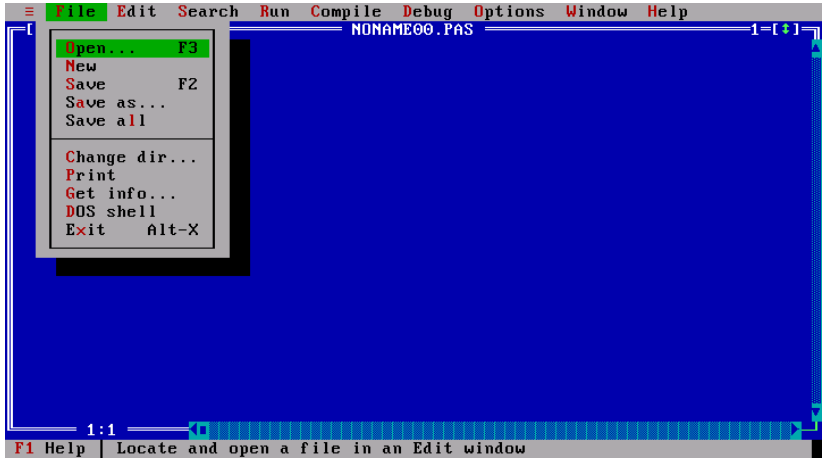
The Good News

Integrierte Entwicklungsumgebungen (IDEs)

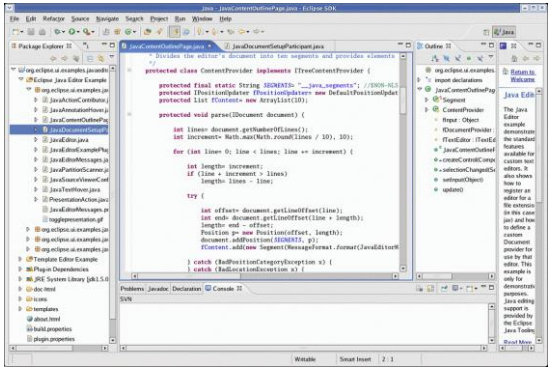
- Quellcode-Editor
- Integrierter (automatisierter) Aufruf von Compiler und Linker
  - Je nach unterstützten Sprachen auch Interpreter
- Oft noch zusätzliche Features:
  - Integrierter Debugger
  - Refactoring
  - Versionsverwaltung
  - Profiling
  - Projektmanagementtools
    - UML-Editoren
  - Werkzeuge für das GUI-Design
  - …

IDEs - Damals und Heute

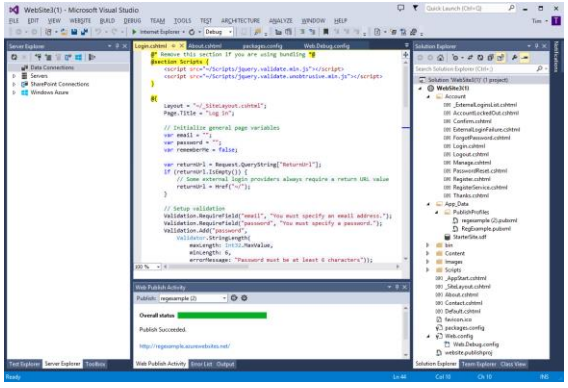
Maestro I (1975)



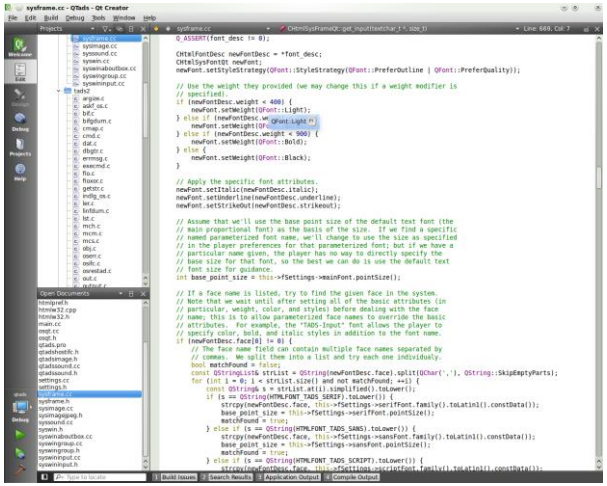
Borland Turbo Pascal (1983)



Eclipse

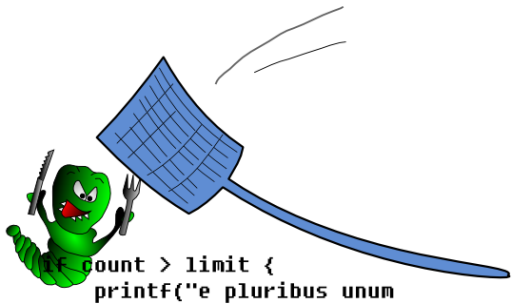


Visual Studio



Qt Creator

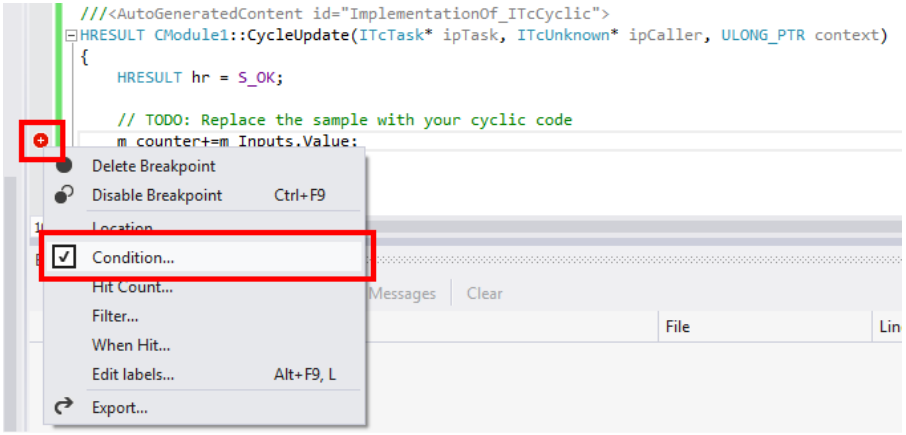
Debugger

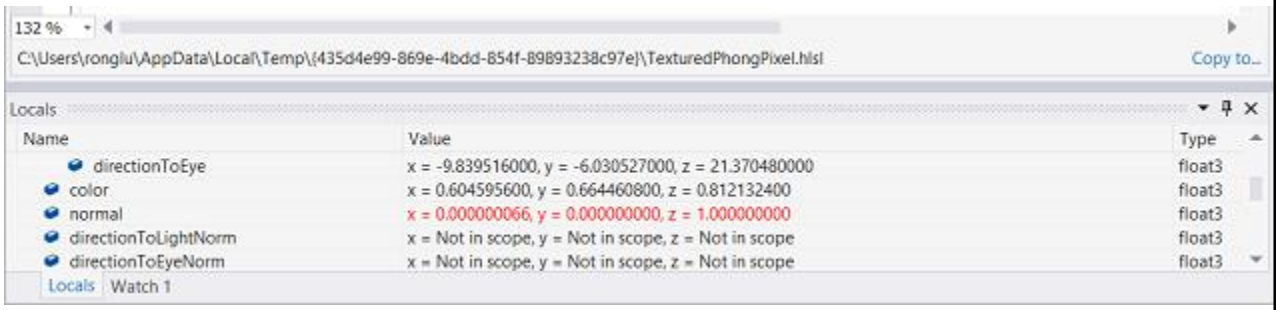


- Werkzeug zum Auffinden von Fehlern
  - Vornehmlich in Programmen
  - Gibt es aber auch für Hardware
  - Kann auch zum Reverse-Engineering eingesetzt werden
- Hauptfunktion:
  - Anhalten eines Programms zu beliebigem Zeitpunkt
  - Inspektion des Inhalts des Speichers
- Prinzipiell lässt sich jedes Programm auf Assembler-Ebene debuggen
- Debugger für Hochsprachen bieten aber viele Komfortfunktionen
  - Auflösung von Funktionen
  - Haltepunkte direkt im Quellcode-Editor
  - Ändern von Speicherinhalten

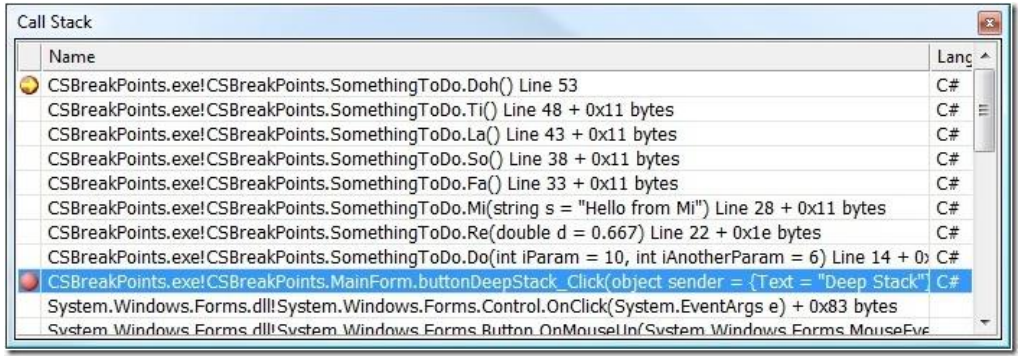
Debugger-Bestandteile

- Haltepunkte
  - Definition von Haltebedingungen
- Darstellung des Speicherinhalts
  - Globale und lokale Variablen

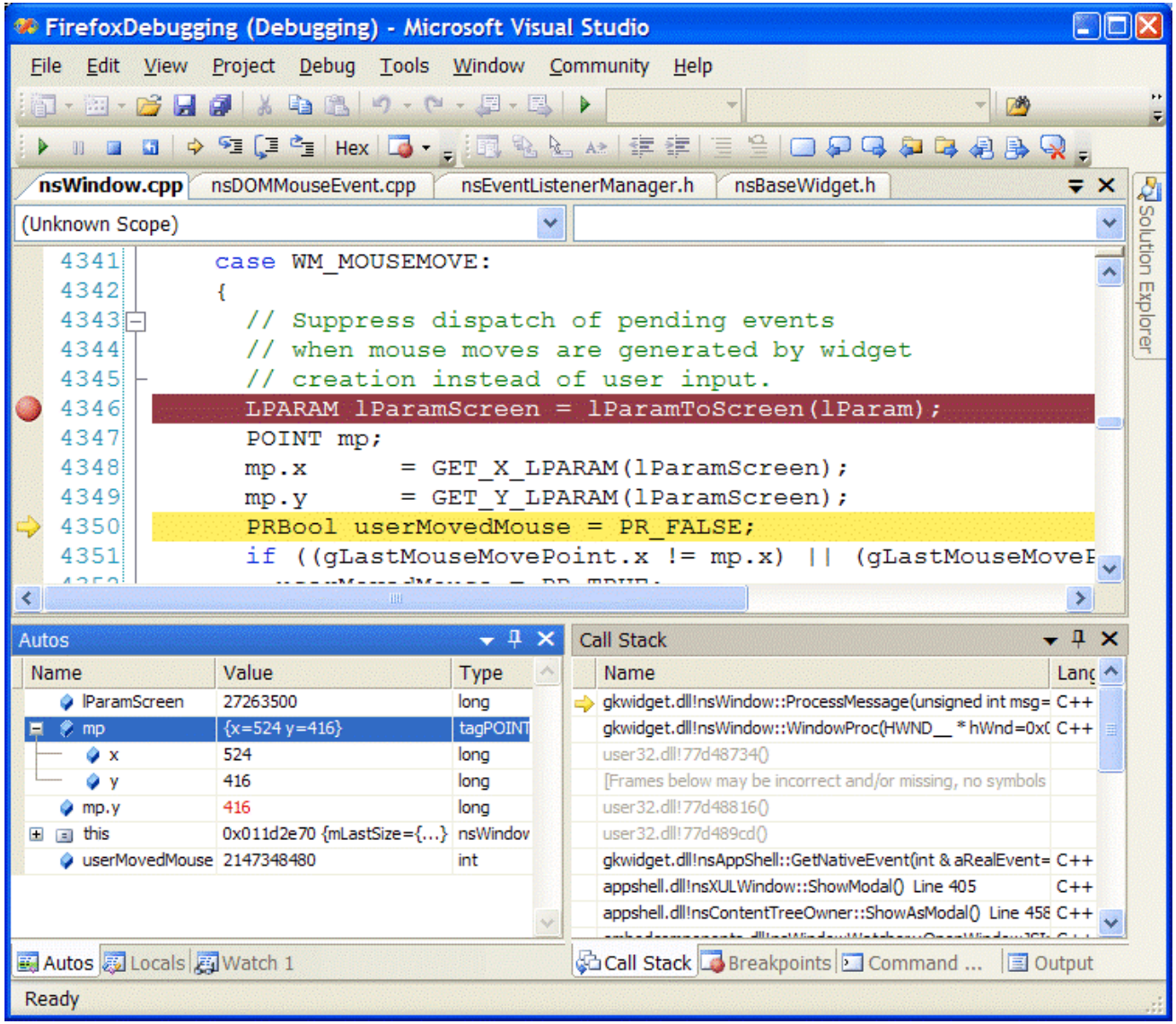




- Aufrufliste (Call-Stack)
  - Zeigt, welche Funktionen vor dem Haltepunkt aufgerufen wurden

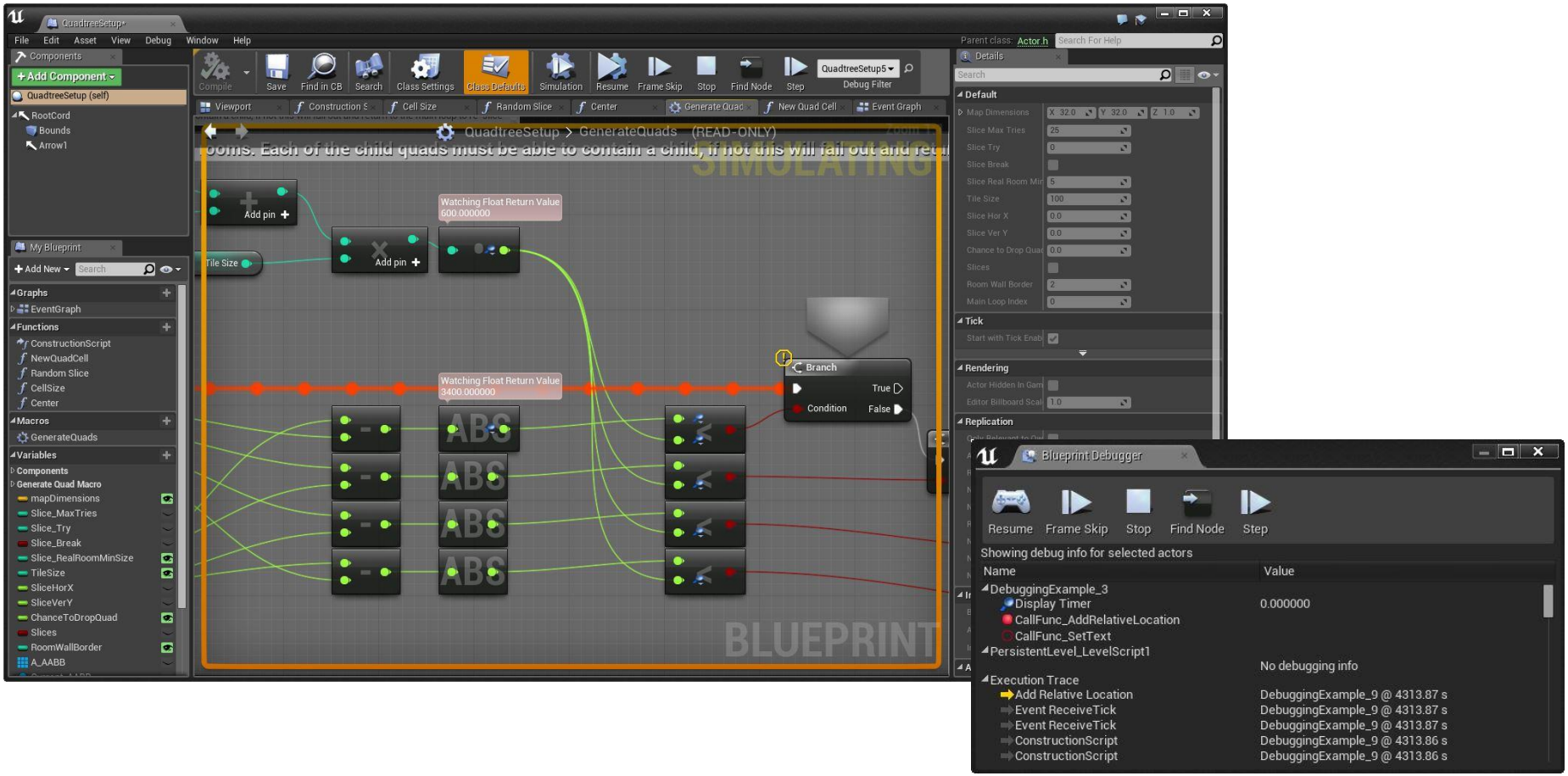


Live-Demo Visual Studio Debugger



Debugging in der Unreal-Engine

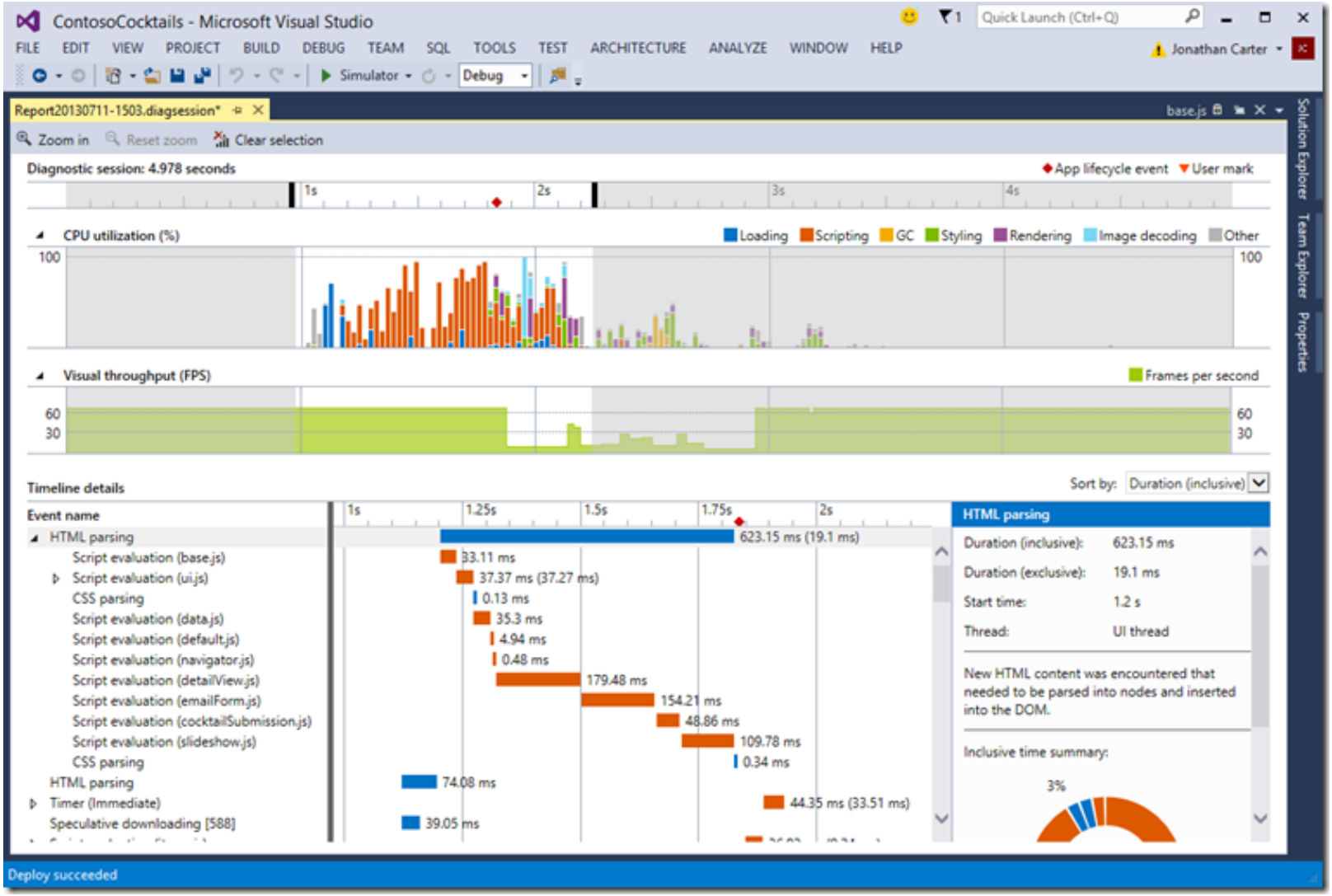
- Die Unreal-Engine bietet neben dem Debugging in der IDE auch Funktionalität für das Debugging von Blueprint



Profiler

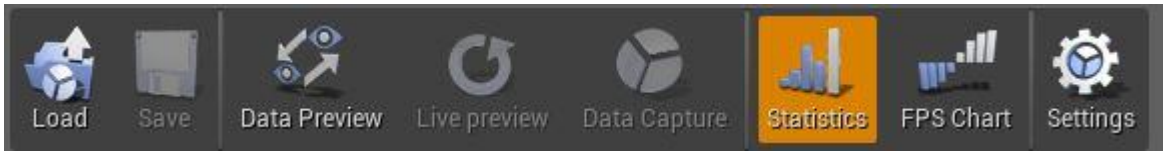
- Werkzeug zur Analyse des Laufzeitverhaltens
  - Helfen Performance-Bottlenecks zu identifizieren
  - Protokollieren während der Laufzeit verschiedene Daten
    - Speicherverbrauch
    - Funktionsaufrufe
    - Berechnungszeiten
    - Nebenläufigkeit
  - Stellen diese Daten meist grafisch dar

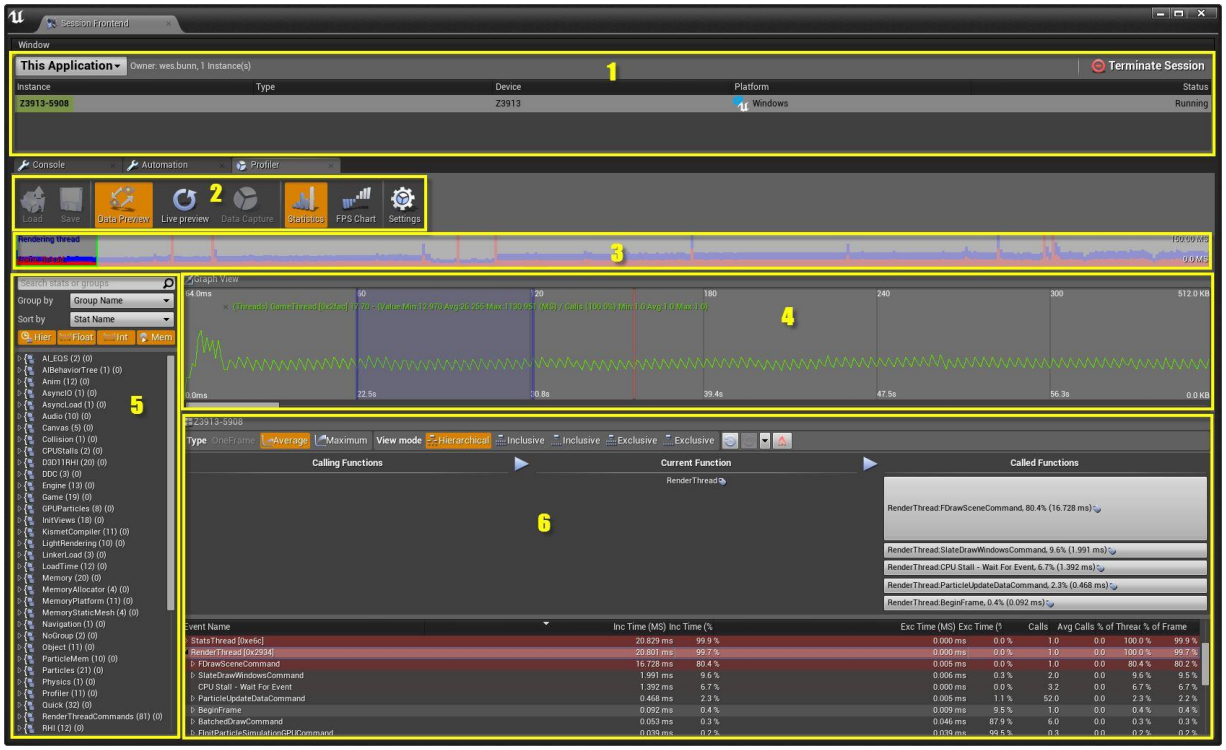
Live-Demo Visual Studio Profiler



Profiling in der Unreal-Engine

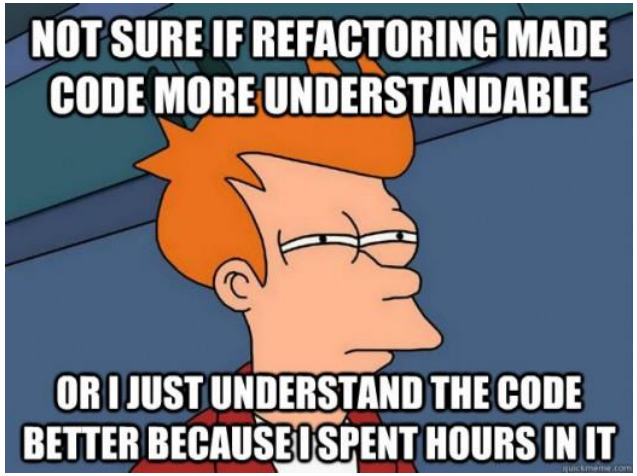
- Die Unreal-Engine bietet Datensammlung von CPU- und GPU-Daten an um Performance-Bottlenecks zu finden





Refactoring

- Werkzeug zur strukturellen Verbesserung von Quelltexten unter Beibehaltung des Verhaltens
- Ziele
  - Lesbarkeit
  - Übersichtlichkeit
  - Verständlichkeit
  - Vermeiden von Redundanzen
- Funktionalitäten
  - Umbenennung von Symbolnamen
  - Aufteilung/Zusammenfassen von Funktionseinheiten (Klassen, Methoden)
  - Verschieben von Symbolen in andere Funktionseinheiten



Refactoring Beispiel

- Schlecht: Lokale nichtssagende Variable x wird mehrfach verwendet

```
double x = 2 * (breite + hoehe);
cout << "Umfang: " << x << endl;
x = breite * hoehe;
cout << "Fläche: " << x << endl;
```

- Besser: Getrennte Variablen mit aussagekräftigem Namen

```
double umfang = 2 * (breite + hoehe);
cout << "Umfang: " << unfang << endl;
double flaeche = breite * hoehe;
cout << "Fläche: " << flaeche << endl;
```

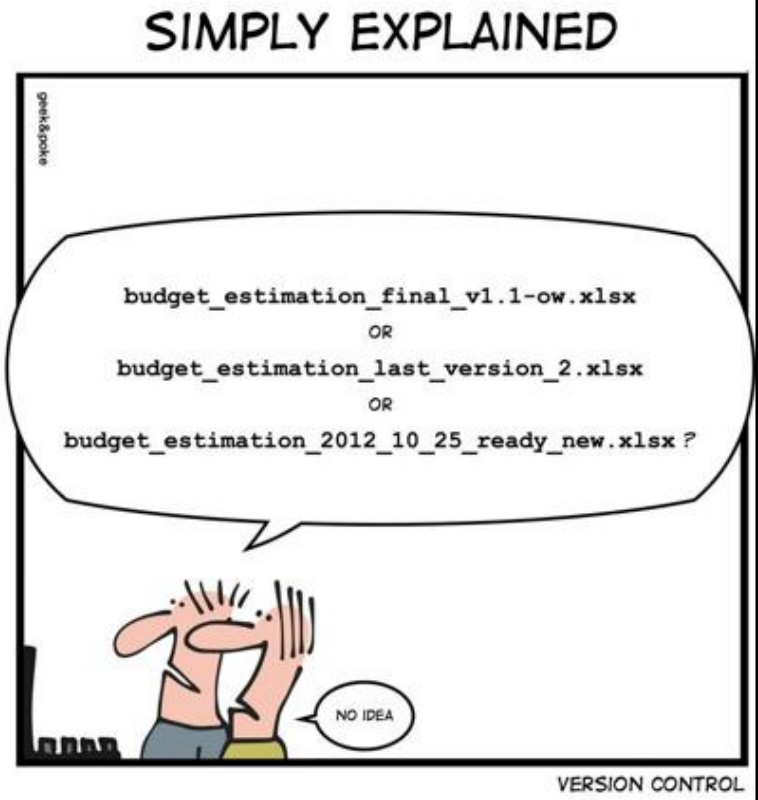
- Andere Variante: Berechnung in Funktionen der Klasse kapseln

```
Rechteck  *rect = new Rechteck(breite, hoehe);
cout << "Umfang: " << rect->getUmfang() << endl;
cout << „Flaeche: " << rect->getFlaeche() << endl;
```

  - Wiederverwendbarkeit

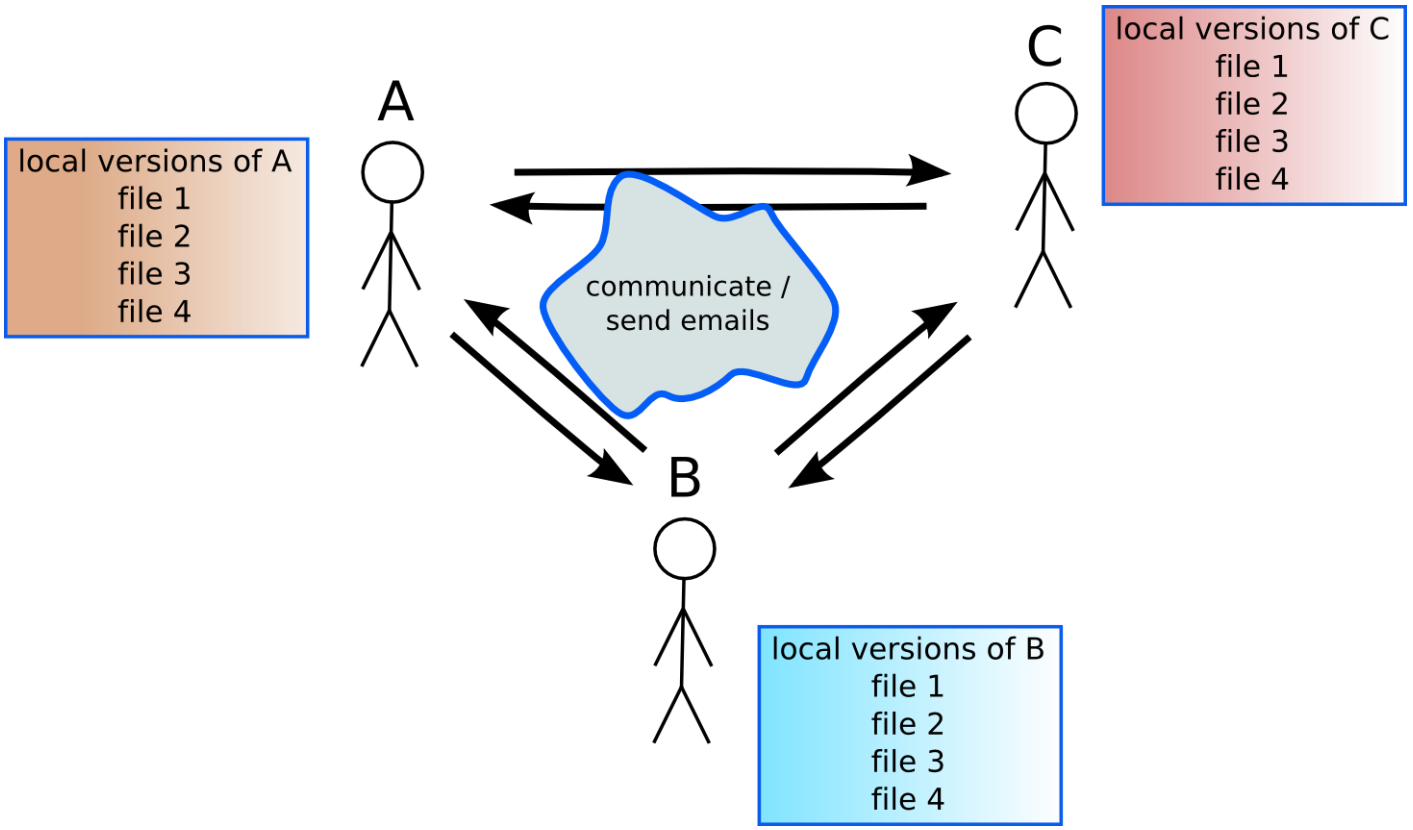
Versionsverwaltung

- Quellcode (aber auch anderen Content, wie Dokumente, Diagramme, …) zu verwalten ist relativ einfach, wenn nur eine Person am Projekt beteiligt ist
- Bei Teamarbeit treten Probleme auf
  - Die gleiche Datei kann von mehreren Personen gleichzeitig bearbeitet werden
    - Diese können in anderem Raum/Land sitzen
  - Wie garantiert man trotzdem konsistenten Stand des Quellcodes?
    - Jeder muss Änderungen bekommen/sehen
    - Es darf keine Konflikte geben



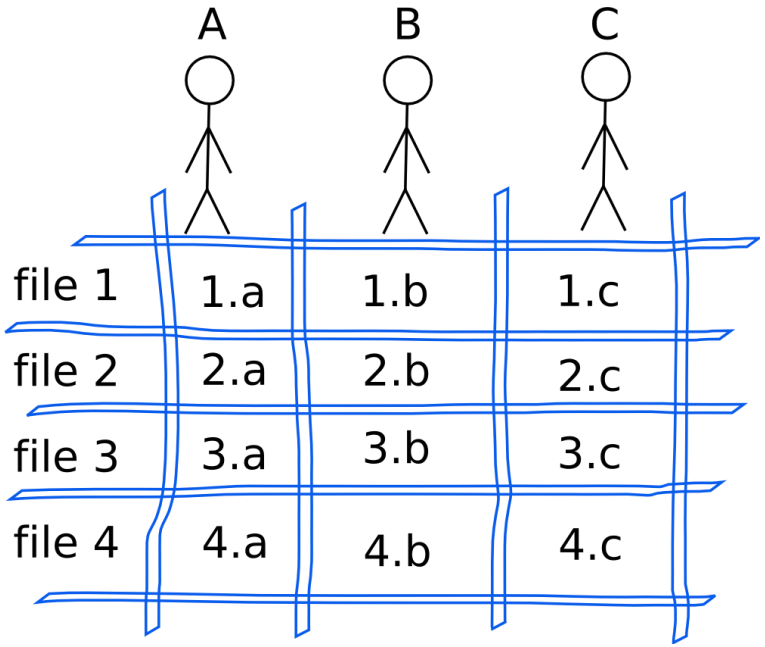
Herausforderung verteilte Softwareentwicklung

- Versuch 1: Kommunikation via Email
  - Senden der bearbeiteten Dateien an alle anderen Beteiligten
  - Beispiel: 3 Personen in einem Projekt mit 4 Dateien



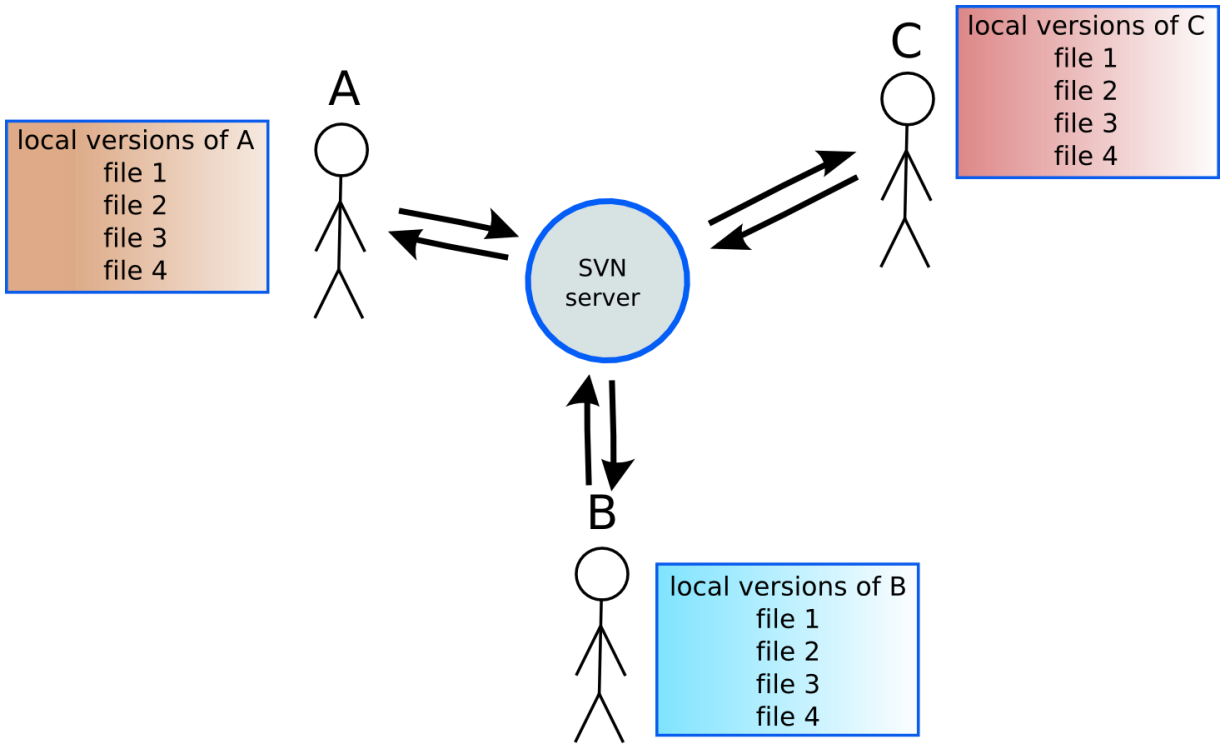
Probleme bei Email-Kommunikation

- Problem:
  - 3 unterschiedliche Versionen für je 4 Dateien
  - 4^3=64 mögliche Kombinationen aus Dateiversion und Person



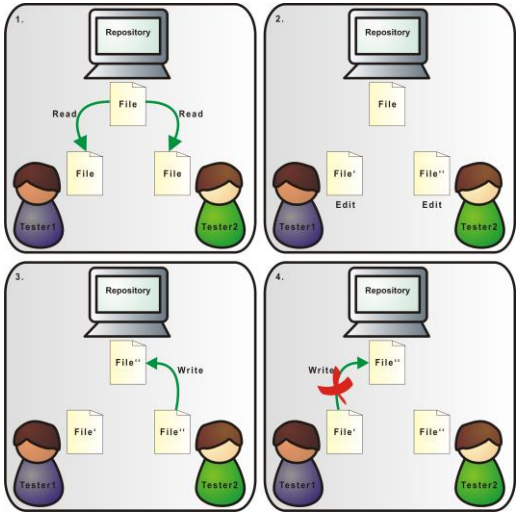
Verteilte Softwareentwicklung - Serveransatz

- Versuch 2: Speichere Dateien auf gemeinsamem lokalem Server
  - Speichere auch ältere Versionen (Historie)
  - Auflösung von Konflikten



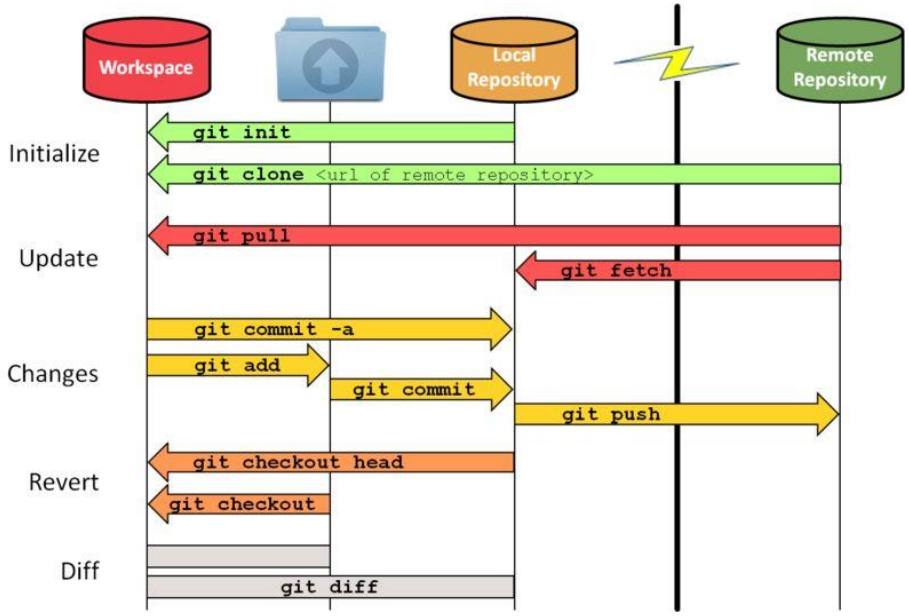
Versionsverwaltung - Varianten

- Lock-Modify-Unlock
  - Wer eine Datei bearbeiten will, sperrt sie für alle anderen Benutzer
  - Vorteil: Keine Konflikte
  - Nachteil: Eine Datei kann immer nur von einer Person gleichzeitig bearbeitet werden
  - Nur für kleine Teams
- Copy-Modify-Merge
  - Jeder holt sich eine lokale Version der Dateien
  - Alle können simultan an allen Dateien arbeiten
  - Im Fall von Konflikten: Auflösung erforderlich
  - Geeignet auch für große Projekte und Teams
  - Beispiele: CVS, SVN (SubVersioN), Mercurial, Git



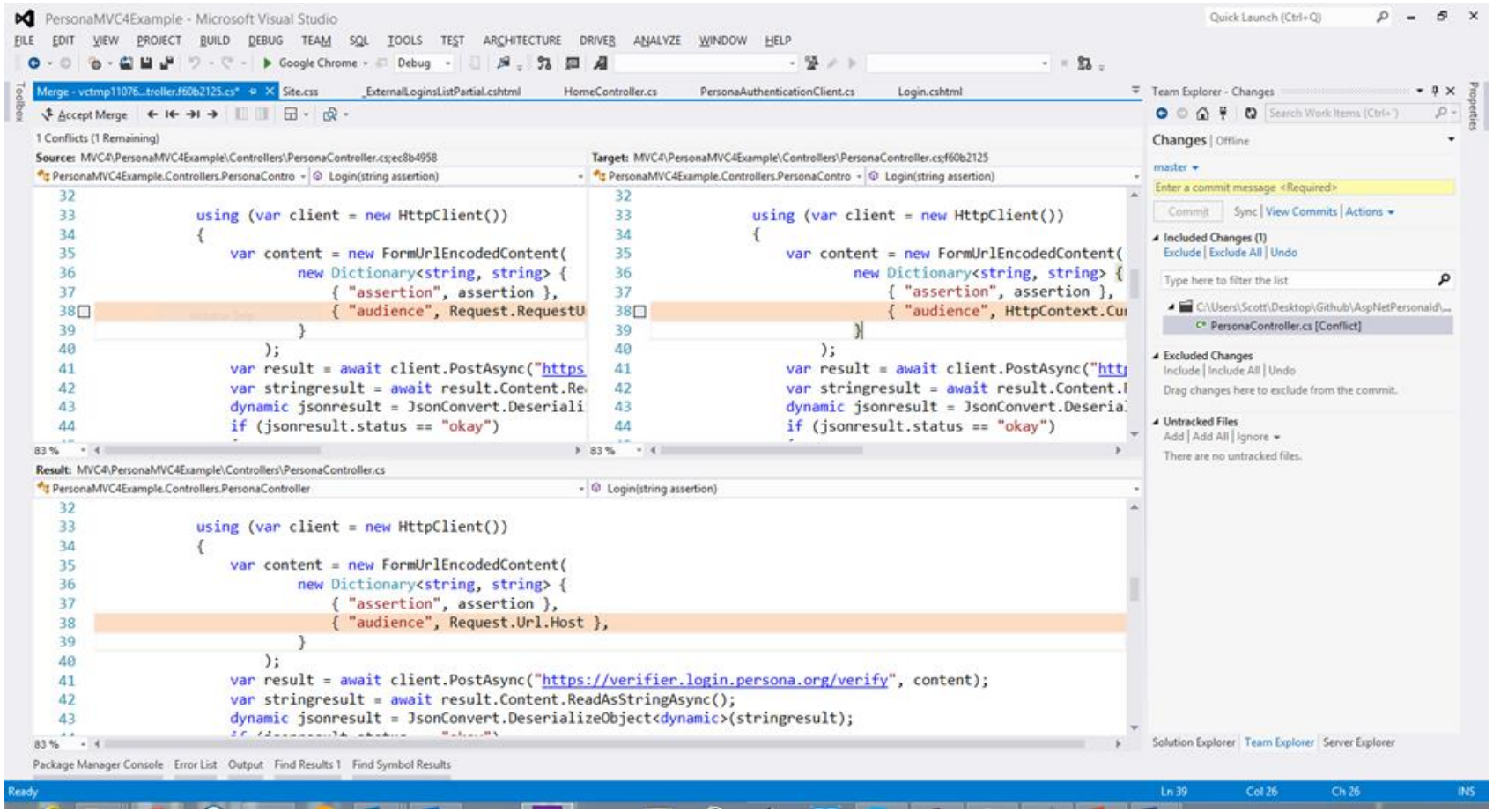
Beispiel Git

- Basisbefehle
  - clone
    - Erstellt lokale Kopie eines Git-Repositories
  - commit
    - Übernimmt Änderungen in lokale Kopie
  - push
    - Überträgt Änderungen an Server
  - pull
    - Holt Änderungen vom Server



Konfliktauflösung

- Wenn Konflikte auftreten, werden sie in einem Diff-Editor angezeigt
- In einem Merge-Schritt werden sie vom Entwickler aufgelöst

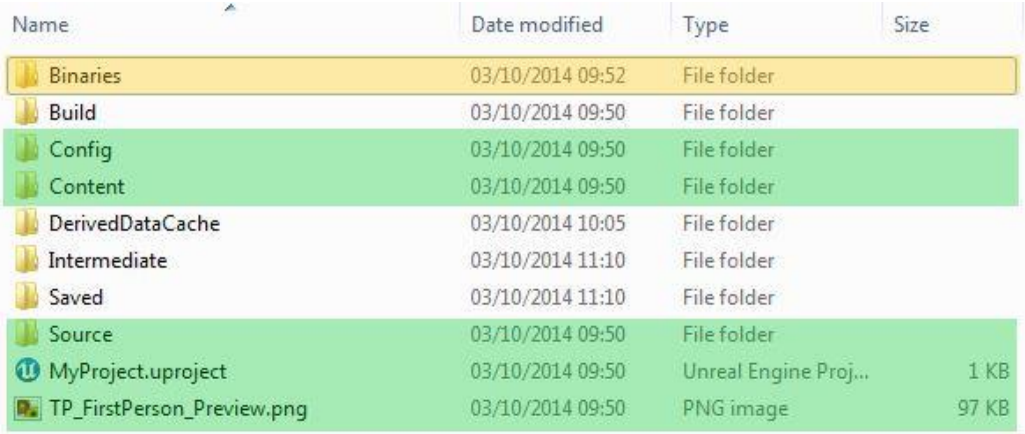


Was gehört ins Repository und was nicht?

- In der Datei .gitignore wird definiert, welche Dateien nicht ins Repository gepusht werden
- Es sollten alle Dateien enthalten sein, die man zum Erstellen des Projekts braucht
  - Sourcecode
  - Externe Libraries
    - Entweder als Sourcecode
    - Oder als Binary (*.lib,*.dll)
  - Content
  - (eventuell) Dokumentation
- Dateien die automatisch während des Builds erstellt werden gehören nicht ins Git-Repository
  - z.B. das erstellte Executable und Libraries (*.dll)
  - Debug-Informationen (*.pdb), Zwischendateien (*.obj), Logs (*.log),…

Versionskontrolle von Unreal-Projekten

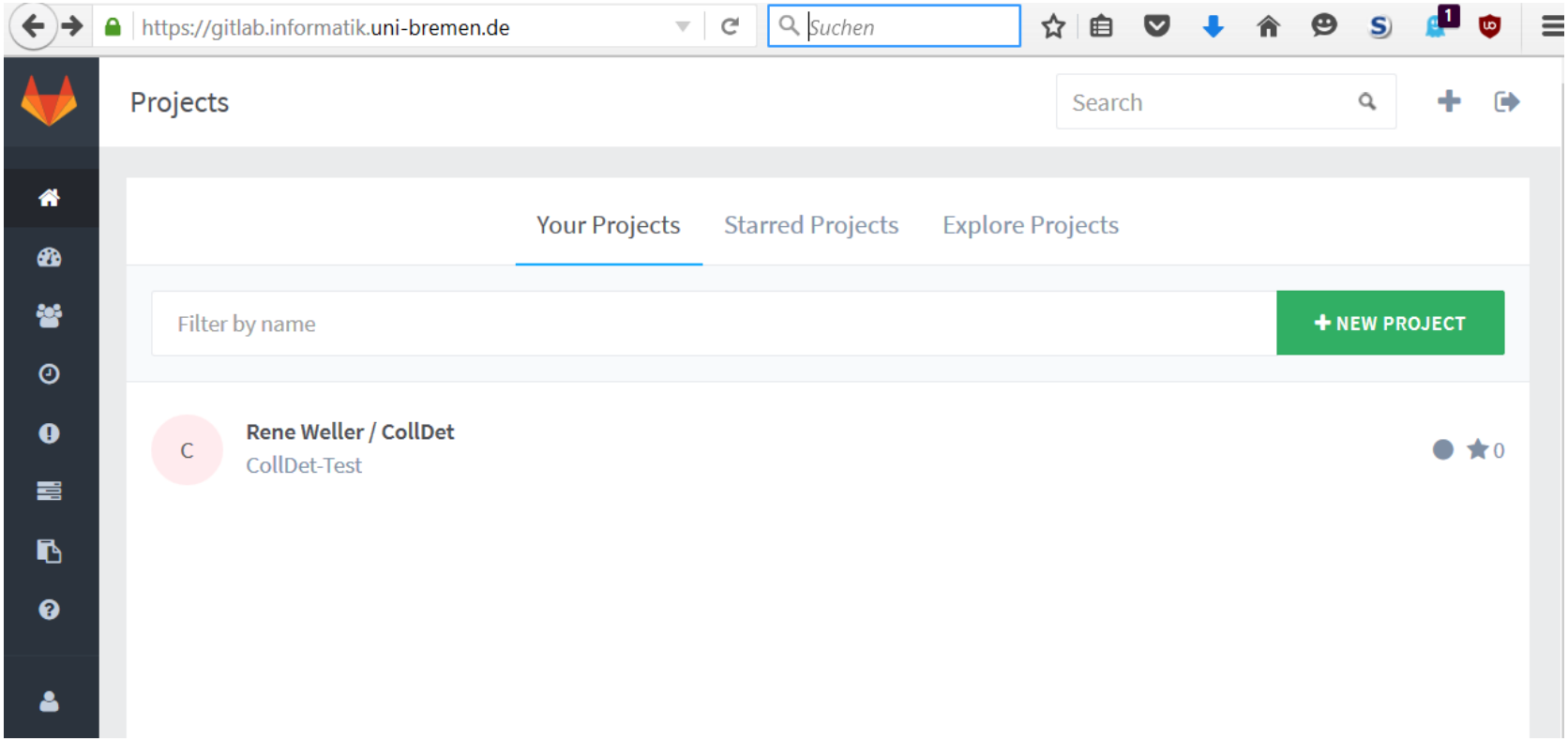
- Das gehört rein:
  - Config (Default-Konfigurationsdateien)
  - Content (3D-Objekte, Texturen, Maps,…)
  - Source (Quellcode)



- Das nicht:
  - DerivedDataCache (temporäre Dateien die während des Spiels erstellt werden)
  - Intermediate (temporäre Dateien die beim Kompilieren entstehen)
  - Saved (Lokale Konfigurationsdateien, Auto-Saves, Screenshots,…)
- Kann, muss aber nicht:
  - Build (Plattformabhängige Skripte zum Erstellen des Projekts und Abhängigkeiten)

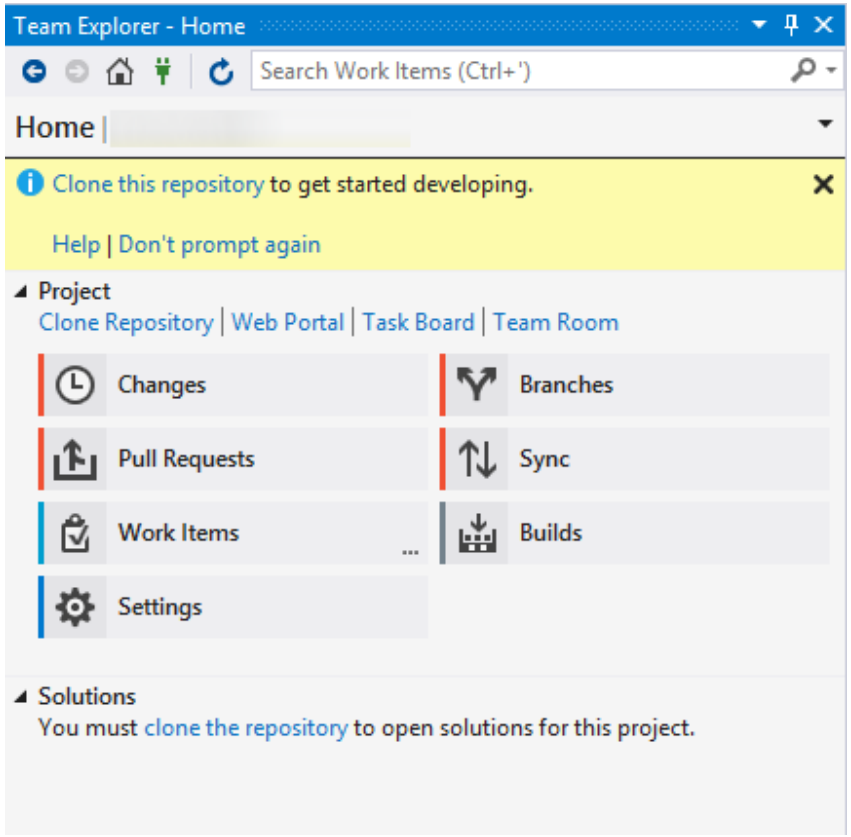
Git-Repository für die Gruppe aufsetzen

- Der FB3 der Uni Bremen stellt einen Gitlab-Server für die Benutzer der Uni bereit
  - http://www.informatik.uni-bremen.de/t/Dienste/Gitlab



Live-Demo Git in Visual Studio

- Git wird in VS 2015 nicht direkt unterstützt und benötigt ein kostenloses Plugin (direkt von Microsoft)

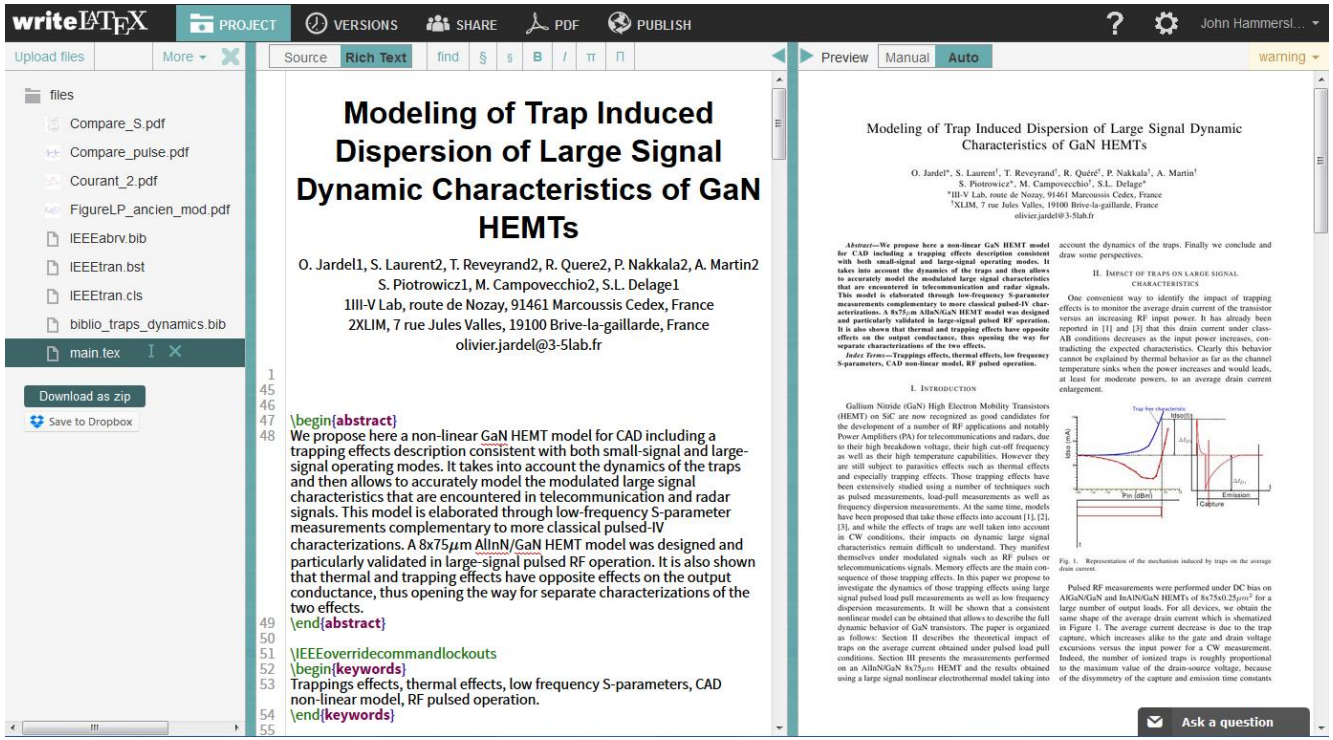


Project Hosting-Seiten

- Bieten meist erweiterte Funktionalität im Vergleich zu reinen Versionsverwaltungsservern
  - Webhosting
  - Forum, Mailinglisten
  - Build-Systeme
  - Bug-Tracking
  - Wiki
  - Statistiken
  - Blogs
  - …
- Beispiele
  - Github, Google Code, Assembla, Bitbucket, SourceForge

Etwas Off-topic aber sehr praktisch

- Overleaf.com
  - Gemeinsam Latex-Dokumente erstellen
  - Live-Vorschau
  - Arbeiten direkt im Browser



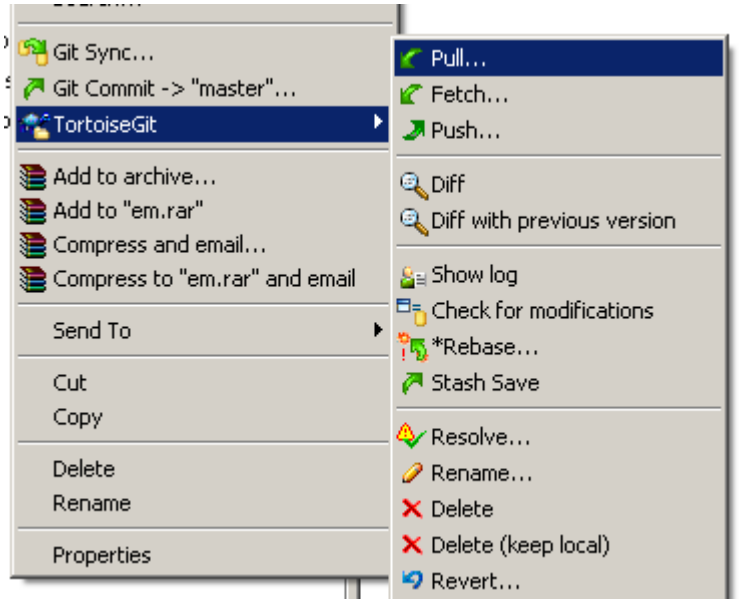
Extern oder in die IDE integriert?

- Für alle hier in die IDE vorgestellten Werkzeuge gibt es auch externe Einzelprogramme
  - Vorteile:
    - oft deutlich erhöhte Funktionalität
    - Unterstützung für Spezialhardware (z.B. die GPU)
  - Nachteile
    - Einarbeitungszeit
    - Kosten
    - Unterbrechung des Workflow

vs

Externe Tools

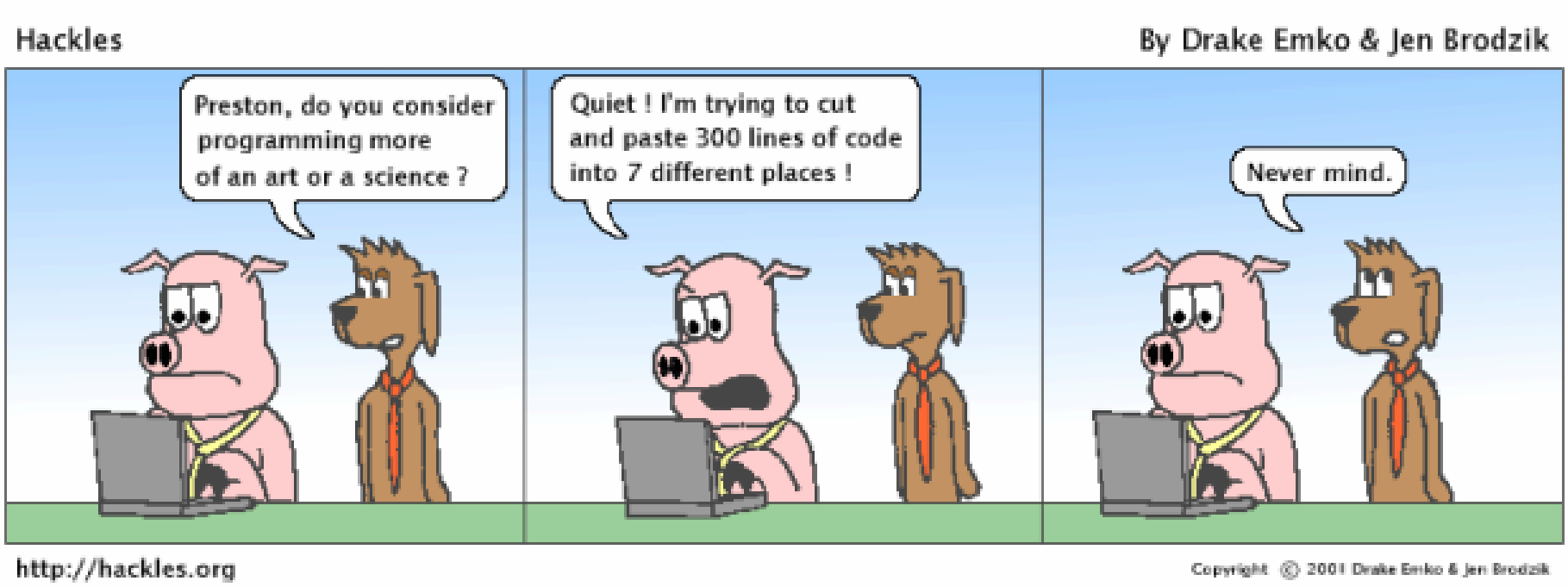
- Beispiele:
  - Editor: Hatten wir schon
    - Emacs, Vim
  - Profiler
    - Intel VTune
    - NVIDIA Nsight (für die GPU)
  - Versionskontrolle
    - TortoiseGit



- Welche man nimmt ist Geschmackssache (oder vom Auftraggeber vorgeschrieben)
- Für unser einfaches Projektchen sollten die integrierten Werkzeuge genügen

Für unser Projektchen notwendige Werkzeuge

- IDE (Visual Studio, Xcode)
  - Editor
  - Compiler/Linker
  - Debugger
  - Versionskontrolle (mit Git-Plugin)
  - Profiler
- Dokumentationswerkzeug
  - Doxygen


Hackles
By Drake Emko & Jen Brodzik
Preston, do you consider programming more of an art or a science ?
Quiet ! I'm trying to cut and paste 300 lines of code into 7 different places !
Never mind.
http://hackles.org
Copyright © 2001 Drake Emko & Jen Brodzik