FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 10 – Objektorientierung 4 | Objektorientierter Entwurf

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Objektorientierter Entwurf

Inhalt

FH MÜNSTER
University of Applied Sciences

- OO Entwurf
  - Herausforderungen der Softwareentwicklung
  - Prinzipien
- Zusammenfassung und Ausblick

Softwareentwicklung

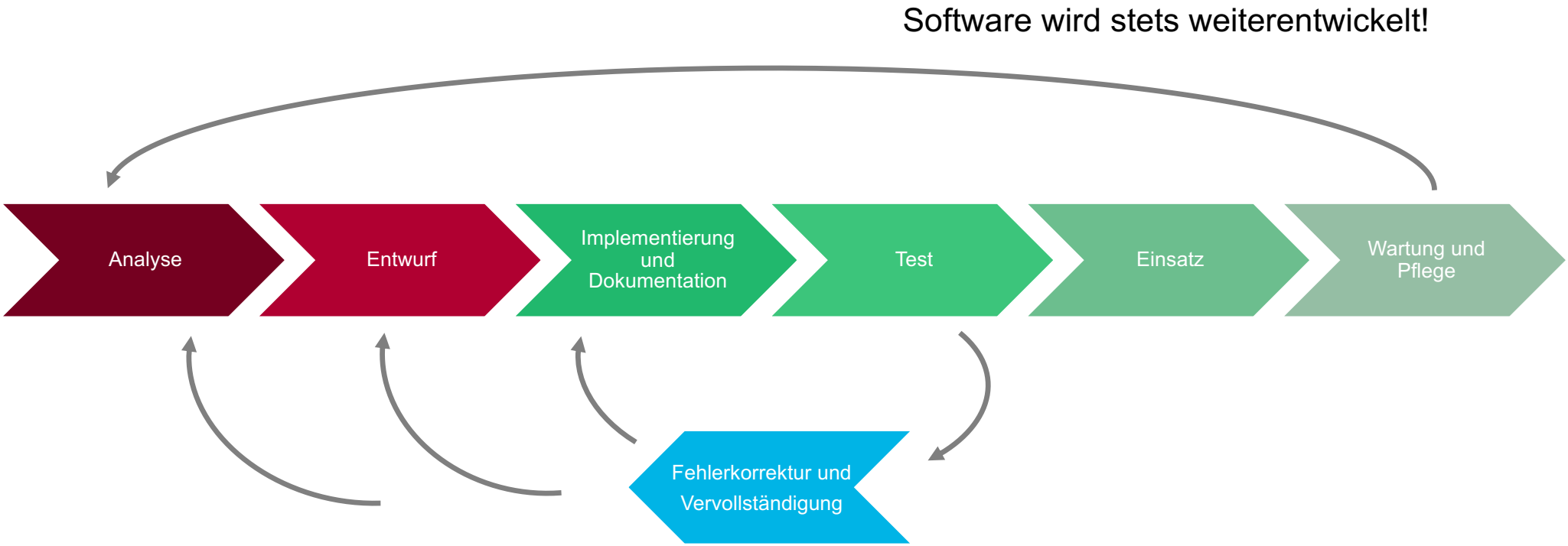
Ein komplexes Geschäft

Zwei Formen von Komplexität in der Softwareentwicklung:

- **Algorithmisch:**
  - Lösungen für komplexe Probleme finden, wie z.B. Objekte in Bildern erkennen, gesprochene Sprache analysieren, Signale filtern etc.
- **Entwurfstechnisch:**
  - Einfache Aufgaben, wie Kundendaten aus einer Datenbank auslesen in einem Programm umsetzen erfordern häufig eine große Anzahl von eher einfachen Funktionen.
  - Die Komplexität besteht darin, diese große Anzahl an Funktionen zu organisieren und trotzdem leichte Veränderbarkeit und Erweiterbarkeit zu gewährleisten.

Sofwareentwicklung

Phasen der Softwareentwicklung



Softwareentwicklung
Herausforderungen

- Arbeit in großen Teams, was eine gute Verständlichkeit und Benutzbarkeit des Quellcodes voraussetzt.
- Software wird immer wieder verändert und erweitert.

- Ziel:
  - Beherrschbarkeit der Software, um obige Herausforderungen zu meistern.

  ➢ Kann erreicht werden durch Einhaltung einiger Prinzipien

Objektorientierter Entwurf

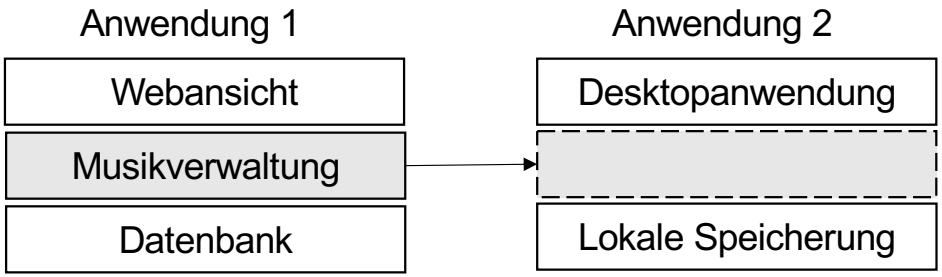
Begriffserklärung

FH MÜNSTER
University of Applied Sciences

Module

Ein Modul ist ein überschaubarer eigenständiger Teil einer Anwendung.
Ein Modul kann sein:

- eine Funktion oder Methode
- eine Klasse
- eine Gruppe von Klassen
- eine Bibliothek …



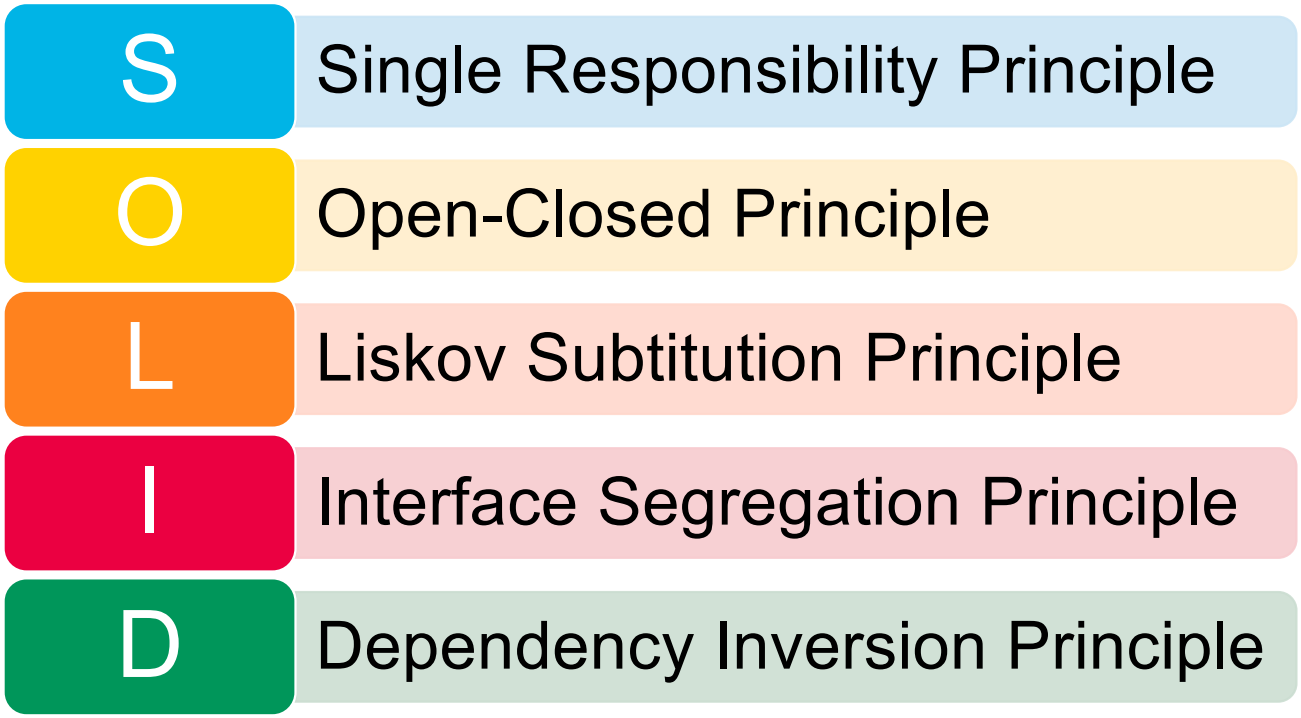
Beispiel: Modularer Aufbau einer Musiksoftware

Untermodule

Module können aus weiteren Modulen, den Untermodulen zusammengesetzt sein.

Einführung in die Objektorientierung

SOLID-Prinzip



Objektorientierter Entwurf

Single Responsibility Principle (SRP)

1. Prinzip einer einzigen Verantwortung

- Jedes Modul soll genau eine Verantwortung übernehmen.
  … d.h. Eine Funktionalität soll in der Klasse realisiert sein, die diese Funktionalität ohne weiteres Wissen bearbeiten kann.

Vorteile:

- Sollten sich Anforderungen ändern ist die Identifikation der betroffenen Module/Klassen, die geändert werden müssen, einfacher und die Anzahl der betroffenen Stellen im Quelltext geringer.
- Erhöhung der Mehrfachverwendbarkeit.

Objektorientierter Entwurf

Single Responsibility Principle (SRP)

Beispiel: Steuerung eines Lüfters

- **Schlecht:** Eine einzelne Schaltungseinheit übernimmt: Temperaturmessung, Steuerung des Lüfters, Anzeige auf einem Display.

- **Besser:**
  - **Temperaturmodul:** misst Temperatur.
  - **Steuerungsmodul:** schaltet den Lüfter je nach Temperatur.
  - **Anzeigemodul:** zeigt aktuelle Werte auf dem Display.

Objektorientierter Entwurf

Single Responsibility Principle (SRP)

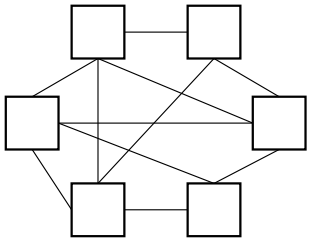
- Regeln, um das Prinzip besser einhalten zu können:

    1. Kohäsion maximieren:
       Alle Teile des Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein.

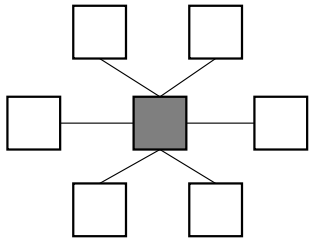
    2. Kopplung minimieren:
       Wenn für die Umsetzung einer Aufgabe viele Module zusammenarbeiten müssen, entstehen Abhängigkeiten. Diese Module haben eine starke Kopplung, welche minimiert werden sollten:

Bankkonto
- kontonummer: String - kontoinhaber: String - kontostand: float
+Bankkonto(kontonummer: String, kontoinhaber: String) +einzahlen(betrag: float) : void +auszahlen(betrag: float) : void +ausgabe_kontostand() : void

Beispiel: Klasse mit hoher Kohäsion



Beispiel: hoher Grad der Kopplung



Reduzierung durch neues Modul

Objektorientierter Entwurf

Open-Closed Principle (OCP)

2. Offen für Erweiterung, geschlossen für Änderungen

- Erweiterungen sollen leicht möglich sein.
- Der vorhandene Quelltext soll möglichst wenig verändert werden müssen.

Eine Kompaktkamera ist für einen Einsatzbereich konstruiert und optimiert. Sie ist einfach, aber nicht erweiterbar.

Eine Spiegelreflexkamera besitzt eingebaute Erweiterungspunkte, an denen man bestimmte Komponenten anschließen kann.

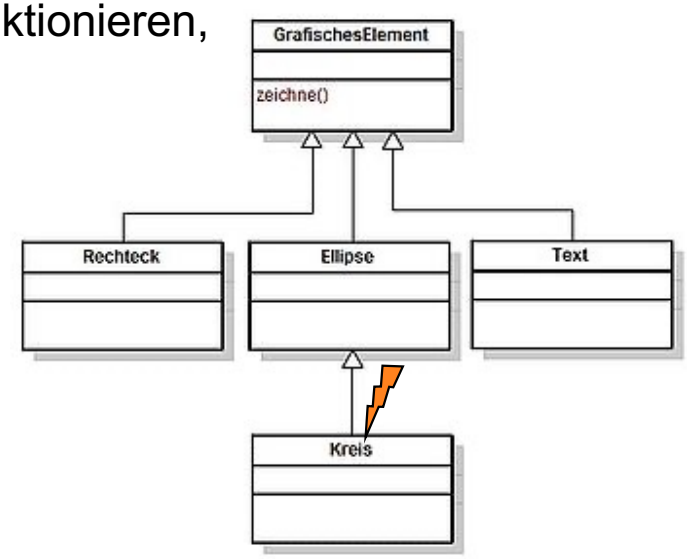
Objektorientierter Entwurf

Liskov Subtitution Principle (LSP)

3. Ersetzbarkeitsprinzip

- Das Programm soll sowohl mit Objekten der Oberklasse als auch mit Objekten deren Unterklasse korrekt funktionieren, ohne dass Änderungen notwendig sind.

Beispiel: Sollte Ellipse Methoden zur unabhängigen Skalierung in x und y-Richtung implementieren, werden diese auch an Kreis vererbt. Nach Anwendung einer solchen Skalierung wäre möglicherweise ein Kreis kein Kreis mehr!



Objektorientierter Entwurf

Interface Segregation Principle

4. Prinzip der Schnittstellentrennung

- Clients sollten nicht gezwungen sein, Interfaces zu implementieren, die sie nicht benötigen.“
- Schnittstellen sollen klein und spezifisch sein, statt viele unzusammenhängende Methoden zu erzwingen.

*Beispiel: Ein alter Drucker sollte **nicht** gezwungen sein, Methoden wie fax() oder scan() zu implementieren, wenn er nur drucken kann.*

Objektorientierter Entwurf

Dependency Inversion Principle (DIP)

5. Umkehr der Abhängigkeiten

- Ein Entwurf soll sich auf Abstraktionen stützen und nicht auf Spezialisierungen.
- Umkehr bedeutet hier dass die Abhängigkeit von der Spezialisierung gelöst wird und eine Abhängigkeit von einer Abstraktion geschaffen wird.

Vorteile:

- Abstraktionen machen Systeme flexibler.
- Abhängigkeiten zwischen Modulen werden minimiert.

Objektorientierter Entwurf

Und sonst

FH MÜNSTER
University of Applied Sciences

- Don't repeat yourself – Vermeide Wiederholungen
  - Wenn sich ein Stück Quelltext wiederholt, ist es oft ein Indiz, dafür, dass Funktionalität vorliegt, die zu einem Modul zusammengefasst werden kann.
- KISS – Keep it simple and stupid
  - Möglichst einfache und klare Implementierung bevorzugen
- YAGNI – You ain‘t gonna need it
  - Keine Verallgemeinerungen einführen für Dinge, die vielleicht in Zukunft einmal benötigt werden
  - Beispiel: Nicht von Vornherein Setter-Methoden für alle Attribute

Objektorientierter Entwurf

Unit Tests

Mach es testbar

Beim Schreiben von Software soll darauf geachtet werden, dass diese auch testbar ist.

- Bestimmte Designentscheidungen von Software unterliegen nicht nur der Verbesserung der Funktionalität, sondern auch der leichteren Ausführung von Tests auf dieser Software.
- Diese Tests können automatisiert nach jedem Build ausgeführt werden (häufig: Unit-Tests*).

Vorteil:

- Die Anforderungen an die Software können effizient getestet werden.
- Oft hat die bessere Testbarkeit Einfluss auf ein verbessertes Softwaredesign.

*__Unit-Tests__
Ein Unit-Test ist ein Stück eines Testprogramms, das die Umsetzung einer Anforderung an eine Softwarekomponente überprüft.

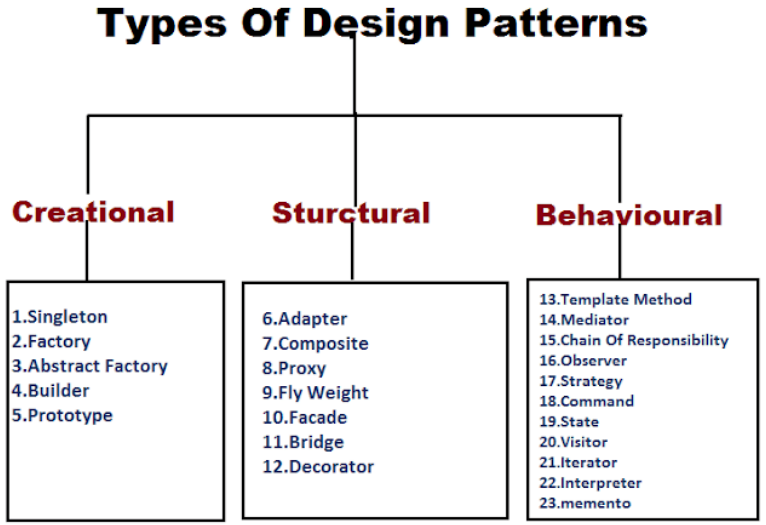
Objektorientierter Entwurf

Weitere Begriffe

FH MÜNSTER
University of Applied Sciences

- Entwurfsmuster (Design Patterns)
  - Idee: Probleme in der Softwareentwicklung wiederholen sich. Hierfür existierende und erprobte Musterlösungen nutzen.

- Clean Code:
  - Idee: Code sollte selbsterklärend und Intuitiv verständlich sein
    - Möglichst verständliche Variablen und Methode- bzw. Funktions-Namen wählen
    - Funktionen/Methoden möglichst kurz halten



https://javascript.plainenglish.io/what-are-software-design-patterns-fe8a9f9ecabb Abruf: 17.6.2025

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 11 – Zusammenfassung und Ausblick

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Zusammenfassung

Was haben wir gemacht?

- **Grundsätzlicher Aufbau von C++ Programmen**
  - Syntax und Schlüsselwörter
  - Header-, Quelltextdatei Struktur
  - Trennung von Deklaration und Implementierung
- **Objektorientierung**
  - Modellierung von Software (UML)
  - Objekte und Klassen (Datenkapselung)
  - Beziehungen zwischen Objekten und Klassen (Vererbung, Assoziationen)
  - Polymorphie
- **Werkzeuge**
  - Compiler, Debugger, Doxygen etc.
- **Ausnahmebehandlung (Exceptions)**
- **Templates** und Effiziente Programmierung in C++ mit **Referenzen** und **Zeigern**

Ausblick
Weitere Themen

Fortgeschrittene Themen:

- Lamda-Expressions
- Metaprogrammierung mit Templates
  - Metaprogrammierung: das Schreiben von Code, der selbst Programme oder Programmteile erzeugt
- Reguläre Ausdrücke <regex>
- Nebenläufigkeit mit Threads <thread>, Coroutinen
- Grafische Benutzungsschnittstellen (GUI)

Ausblick
Lamda-Expressions

FH MÜNSTER
University of Applied Sciences

Fortgeschrittene Themen:

- **Lamda-Expressions**
- Metaprogrammierung mit Templates
  - Metaprogrammierung: das Schreiben von Code, der selbst Programme oder Programmteile erzeugt
- Reguläre Ausdrücke <regex>
- Nebenläufigkeit mit Threads <thread>, Coroutinen
- Grafische Benutzungsschnittstellen (GUI)

```
std::vector<double> v{-11.0, 3.2, 4.1, -7.0, 8.0, 1.5};

std::sort(v.begin(), v.end(),
  //Lambda-Expression
  [](auto x, auto y){ return abs(x) < abs(y);}
);

std::for_each(v.begin(), v.end(),
  // Weitere Lambda-Expression
  [](const auto& e){std::print("{} ", e);}
);

// Ausgabe
// 1.5 2.7 3.2 4.1 -4.4 -7 8 -11
```

Ausblick
Nebenläufigkeit

Fortgeschrittene Themen:

- Lamda-Expressions
- Metaprogrammierung mit Templates
  - Metaprogrammierung: das Schreiben von Code, der selbst Programme oder Programmteile erzeugt
- Reguläre Ausdrücke <regex>
- **Nebenläufigkeit mit Threads <thread>, Coroutinen**
- Grafische Benutzungsschnittstellen (GUI)

```
#include <chrono>
#include <iostream>
#include <thread>
using namespace std;

void f(int t) {
  this_thread::sleep_for(chrono::seconds(t));
  cout << "Thread " << this_thread::get_id()
       << " : Funktion beendet! Laufzeit = " << t << s\n";
}

int main() {
  thread t1(f, 4);
  thread t2(f, 6);
  thread t3(f, 2);
  cout << "t1.get_id(): " << t1.get_id() << '\n';
  cout << "t2.get_id(): " << t2.get_id() << '\n';
  cout << "t3.get_id(): " << t3.get_id() << endl;
  t1.join(); // warten auf Beendigung
  cout << "t1.join() ok" << endl;
  t2.join(); // warten auf Beendigung
  cout << "t2.join() ok" << endl;
  t3.join(); // warten auf Beendigung
  cout << "t3.join() ok\n" << "t1.get_id(): "
       << t1.get_id(); // kein Thread mehr
  cout << "\nmain() ist beendet\n";
  return 0;
}
```

Ausblick
GUIs

Fortgeschrittene Themen:

- Lamda-Expressions
- Metaprogrammierung mit Templates
  - Metaprogrammierung: das Schreiben von Code, der selbst Programme oder Programmteile erzeugt
- Reguläre Ausdrücke <regex>
- Nebenläufigkeit mit Threads <thread>, Coroutinen
- **Grafische Benutzungsschnittstellen (GUI)**

Cross-Plattform

GTK
z.B. Mozilla, Libre Office

Qt Library
z.B. Photoshop Elements, Autodesk Maya, VLC Mediaplayer, KDE
Sehr umfangreich:
- 2D/3D Grafik und Multimedia
- Threads
- Datenbankanbindung
- …

Proprietär

Microsoft Foundation Class

Cocoa

Ausblick

Bibliotheken in C++

https://en.cppreference.com/w/cpp/links/libs

A list of open-source C++ libraries

< cpp | links

The objective of this page is to build a comprehensive list of open-source C++ libraries, so that when one needs an implementation of particular functionality, one needn't to waste time searching on web (DuckDuckGo, Google, Bing, etc.)

If you know a library that might be useful to others, please add a link to it here. There are no restrictions on what can be included except that the **source** of the library must be readily **available** to download.

The page is provided "as is" - with the hope of being useful, but without any warranties. Outdated, misleading or wrong links might appear here. If you've noticed one of these, it would be great if you fixed the error.

Libraries: Table Of Contents

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de