

1 Libraries & Frameworks

1.1 Library

Eine Library (engl. für Bibliothek) stellt bereits implementierte und (getestete) Funktionalitäten (Code) zur Verfügung. Sie enthält statische/konstante Daten und sowie Programmcode. Der Code der Source Files (*.c, *.cpp) werden in die Library gepackt und sind dann nicht mehr sichtbar. Somit müssen nur die Header-Files und Library Files mit dem Executable verlinkt werden. Libraries können über API genutzt werden. Sie ermöglichen den Aufruf von Methoden sowie den Zugriff auf Daten. Es gibt 2 Arten von Libraries: Static und Shared/Dynamic Libraries.

1.1.1 Static Library

Library Inhalt wird komplett zur ausführbaren Applikation hinzugefügt. Der Inhalt der Library ist Teil der ausführbaren Applikation.

Vorteil: Library kann nicht vergessen gehen

Nachteil: Library wird bei jedem Linken im Speicher abgelegt

Typische Dateiendungen: Linux .a / Windows .lib

1.1.2 Shared/Dynamic Libraries

Shared/Dynamic Libraries werden zur Laufzeit des Programms geladen. Sie kann während dem Aufstarten oder irgendwann zur Laufzeit (load/unload) erfolgen. Typischerweise werden die Libraries während dem Aufstarten geladen. Dies bedingt, dass die Library schon während dem Linkvorgang bekannt gemacht wird. **Das Laden der Libraries irgendwann zur Laufzeit (load/unload) wird hier nicht genauer betrachtet.**

Vorteil: Library wird nur einmal gespeichert für mehrere Applikationen

Nachteil: Applikation kann nicht gestartet werden, falls die Library nicht gefunden wird. Typische Dateiendungen: Linux .so (shared object) / Windows .dll (dynamic link library)

1.2 Framework

Begriff: engl. framework: Gerüst, Rahmen

Ein Framework (engl. für Gerüst, Rahmen) ist gemäss macOS eine Sammlung von Library Versionen. Nebst dem Aufrufen von Methoden (gleich wie Library) kann ein Framework auch informieren. Es kann auf Ereignisse (Events) registriert werden und gibt so eine Art Skelett vor. Dabei wird das Hollywood Prinzip angewendet (Don't call us, we will call you). Das Framework beeinflusst somit die Struktur der eigentlichen Anwendung.

Bekannte Frameworks: Qt, .Net, EswRobot (Roboter in ProgC Praktika)

1.2.1 Library erstellen (-i Prakti 1)

static

- LibraryDemoStatic Folder
 - gcc -Wall -c a.c b.c
 - ar r libTest.a a.o b.o
- ExeDemoStatic Folder
 - gcc -Wall -o Test main.c \ -I../LibraryDemoStatic \ -L../LibraryDemoStatic -lTest
- Flags
 - ar r, r for replace existing
 - -IPathToIncludeFolder
 - -LPathToLib
 - -lLibName (ohne lib prefix und .a)

shared

- LibraryDemoShared Folder
 - gcc -Wall -c -fpic a.c b.c
 - gcc -shared -o libTest.so a.o b.o
- ExeDemoShared
 - gcc -o Test main.c \ -I../LibraryDemoShared \ -Wl,-rpath,../LibraryDemoShared \ -L../LibraryDemoShared \ -lTest
- Flags
 - -fpic → position-independent code
 - -IPathToIncludeFolder
 - -LPathToLib
 - -lLibName (ohne lib prefix und .so)

1.2.2 Projektstruktur (-i Prakti 1)

Für Executable

build: Alle object-Files und executable Files (*.o*)

libs: Alle nicht System-Libraries (*.a, *.so, *.h)

src: Source Code der Applikationen (*.c, *.cpp, *.h)

Für Libraries

include: Extern verfügbare Header der Library, API (*.h)

lib: Library file der zu buildenden library (*.a, *.so)

src: Source code der Library inclusive private Header (*.cpp, *.c, *.h)

2 Qt Grundlagen

2.1 Grundlagen

Zweck: Erstellung von User Interfaces (im speziellen für GUIs) und plattformunabhängige Anwendungen (im Bereich Desktop, Embedded und Mobile)

Anwendungen: Skype, Google Earth, Wolfram Research und viele mehr

2.1.1 Qt Oberflächen

Qt Desktop (Qt Widgets)

Qt Oberflächen basieren auf der Qt C++ Klassenbibliothek, welche für die GUI-Elemente ("Widgets") entsprechende Klassen wie QLabel, QPushButton etc. bereitstellt. Das GUI wird unter Verwendung dieser Klassenbibliothek als C++-Programm geschrieben und häufig für die Maus- und Tastaturbedienung genutzt. Neben **Qt Widget** gibt es noch **Qt-Quick (QML)** und **WebEngine**. Der Fokus der Vorlesung liegt auf **Qt Desktop**.

Qt-Quick

Qt-Quick basiert auf QML (= Qt Meta Language oder Qt Modelling Language), einer speziellen Sprache, welche das Aussehen des Programms festlegt. Verfügt über eine "Deklative" Festlegung der Darstellung und des Verhaltens, ähnlich wie bei HTML mit Javascript. Qt-Quick wird vielfach für mobile Applikationen mit Touchscreen Bedienung genutzt.

WebEngine

Web Engine wird für die Darstellung von Web Content mit HTML/CSS/JS verwendet.

Namenskonventionen

- "Qt QtCore - Hauptmodul (wird von allen Modulen genutzt); Weitere Module: QtGui, QtWidgets, QtNetwork, QSql, QTest
- "Q Klasse wie QObject, QApplication, QWidget, QDialog, QString

- ”q globale Variable / Funktionsname wie qApp, qTranslator, qDebug(), qWarning(), qFatal()
- ”Q_GROSS Qt-Makro-Namen wie Q_OBJECT, QCOMPARE, QEXPECT_FAIL, QFETCH
- #include †Qt-Modul- oder Klassenname” wie #include †QtGui, #include †QObject oder #include †QString

2.2 Meta Object System

Dieses System bietet zusätzliche Funktionenitäten zu C++. Es unterstützt das Hollywood Prinzip mittels **Signals and Slots**. Es basiert auf QObject Klassen, Q_OBJECT Makro und einem Meta-Object Compiler (moc).

2.2.1 QObject

Aus der QObject Klasse wurden weitere Qt Klassen abgeleitet wie z.B. QPushButton, QLabel, QGridLayout, QTimer und QThread. Die QObject Klasse wird zu einem header file hinzugefügt mit **#include †QObject;**. QObjects können mit einer Parent-Child-Beziehung verknüpft werden, woraus ein ganzer QObject Tree entstehen kann. Ein Object Tree macht es einfacher eine grosse Anzahl an Dokumenten, die miteinander verbunden sind, auf einen Schlag zu löschen. Das Eltern-QObject wird im C'tor des Child-QObjects wie folgt angegeben:

QLabel * child = new QLabel("0000", &parent);

Alternativ kann setParent() verwendet werden:

child->setParent(parent) Dies kann jedoch bezüglich der D'tor Reihenfolge problematisch sein (Siehe Prakti 2).

Ein Parent-Object kann beliebig viele child-Objects haben, während ein child-Object nur ein parent-Object hat. Die Löschung eines parent objects hat die automatische Löschung sämtlicher child objects zur Folge. Die Löschung eines child objects zerstört einfach die Beziehung zum parent object.

2.2.2 Meta Object Compiler (moc) & qmake

Der Meta Object Compiler (moc) durchsucht das header file nach Q_OBJECT Makros und generiert ein moc_XXX.cpp File mit dem meta-object relevanten Code erstellt. Diese Dateien müssen bei der Ausführung zur Anwendung hinzugelinkt werden. Für die Verlinkung und Ausführung von Meta Object Compiler (moc), User Interface Compiler (uic) und Resource Compiler (Rcc) wird qmake verwendet, welche diese Qt Compiler automatisch einbindet.

Qmake erzeugt aus einer plattformunabhängigen Projektbeschreibung, dem Projektfile (.pro-Datei) ein plattformspezifisches 'makefile'. Diese makefile wird dann von den platformspezifischen Tools wie make dazu verwendet, um die Source-Files zu verarbeiten (compilieren, etc.). Dabei ist qmake selbst ein plattformunabhängiger make-file-Generator.

Der Befehl lautet: **qmake [project file name].pro**

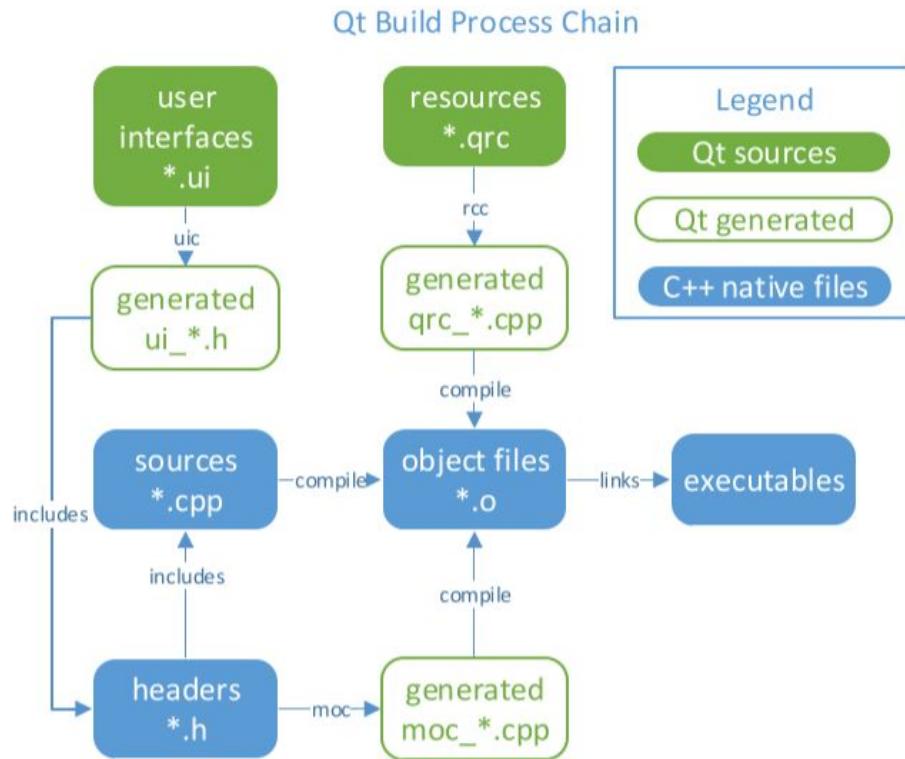


Abbildung 2.1: Qt Build Prozessablauf

2.2.3 Projektbeschreibung (*.pro)

Keywords:

- QT: Liste von verwendeten Qt Modulen
- Target: Name der Applikation (Output File)
- TEMPLATE: Projekt Typ: app oder lib
- CONFIG: z.B. CONFIG += C++14
- windows/linux/macx: Plattform spezifische Compilierungsregeln
- SOURCES/HEADER/FORMS: Source-, Header- oder Formdateien die dem Projekt hinzugefügt werden sollen

```
QT      += core gui charts
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = ch02-sysinfo
TEMPLATE = app

SOURCES += main.cpp \
    MainWindow.cpp \
    SysInfo.cpp \
    CpuWidget.cpp \
    MemoryWidget.cpp \
    SysInfoWidget.cpp

HEADERS += MainWindow.h \
    SysInfo.h \
    CpuWidget.h \
    MemoryWidget.h \
    SysInfoWidget.h

windows {
    SOURCES += SysInfoWindowsImpl.cpp
    HEADERS += SysInfoWindowsImpl.h
}

linux {
    SOURCES += SysInfoLinuxImpl.cpp
    HEADERS += SysInfoLinuxImpl.h
}
FORMS   += MainWindow.ui
```

2.3 Qt Hallo Welt Beispiel

```
#include <QtWidgets>
int main(int argc, char *argv[])
QApplication app(argc, argv);
QLabel label ("Hello Qt World!"); % Erstellen eines Textfelds mit dem Text Hello Qt World!
label.setAlignment(Qt::AlignCenter); % Textfeld in der Mitte ausrichten
label.resize(250, 150); %Breite & Höhe des Testfelds
label.setWindowTitle("My first Qt-Program"); %Titel des Popup Fenster
label.show(); %Befehl zum Anzeigen des Textfelds
return app.exec();
```

2.3.1 QApplication Klasse

Die QApplication Klasse ist pro Qt Applikation nur einmal vorhanden. Sie ist zuständig für das Event handling und weitere Qt Framework spezifische Funktionalitäten. Eine der Methoden heisst exec(). In dieser Methode verbirgt sich die while(1) Schleife. QApplication Klasse ist zudem für die Initialisierung der Applikation mit optionalen Funktionen wie beispielsweise font() und doubleClickInterval() zuständig.

3 QWidgets & Layout

3.1 QtWidgets

Applikation mit GUI = Problem-Domain + Darstellung + Interaktion

Bsp.: Counter = Funktionalität (inkrementieren und dekrementieren) + Fenster mit inc & dec Buttons und der Anzeige des aktuellen Counter-Stands + Inkrementieren, wenn inc-Button gedrückt. **Allgemeine Konzepte von GUI Frameworks**

Ansicht festlegen: Visuelle Elemente in der Benutzeroberfläche werden mittels Klassen vom GUI Framework zur Verfügung gestellt.

Interaktion mit dem Programmbenutzer -; Signals & Slots

3.1.1 QWidget Klasse

QWidget (Widget = "Window Gadget") ist eine Basisklasse für alle anderen Qt Widgets wie bspw. QLabel, QLineEdit, QPushButton. QWidget übernimmt als Parent-Widget folgende Aufgaben: - Anzeigen, verstecken, freigeben, sperren -; wird auch auf die Child-Windows angewendet

- Weiterleiten von Ereignissen an die betroffenen Child-Windows
- Memory Management

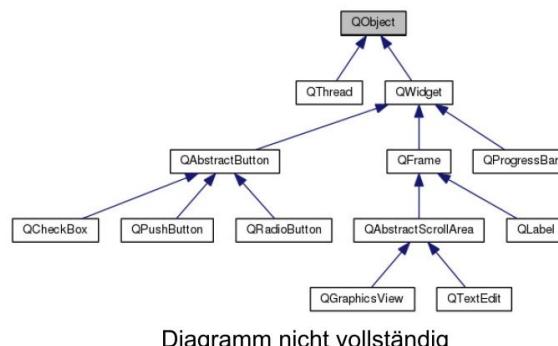


Diagramm nicht vollständig

Abbildung 3.1: Qt Vererbungen von QObject und QWidget

3.2 Layout

Art	Selbst codiertes GUI	GUI-Designer (Qt Creator)
Beschreibung	C++ Code für GUI Beschreibung	Interaktives Tool
Vorteil	kann sich zur Laufzeit ändern	schnell zusammen "geklickt"
Nachteil	aufwändig, Code selber schreiben	Nur statische Design möglich

Bei der Erstellung eines GUIs mittels WYSIWYG Editor – GUI-Designers (Qt-Creator) erfolgt die Anordnung der Widgets innerhalb eines sogenannten Formulars mit Hilfe eines interaktiven Tools. Es erfolgt eine automatische Umwandlung der Formulardaten (*.ui) in entsprechenden Programmcode mittels uic.

3.2.1 Manuelles Layout

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDialog mainWindow;
    mainWindow.resize(230, 75); // w, h
    QLabel * label = new QLabel("0000", &mainWindow);
    label->setGeometry(20, 20, 100, 35); // x, y, w, h
    QPushButton * incButton = new QPushButton ("Inc");
    incButton->setParent( &mainWindow );
    incButton->setGeometry(150, 10, 55, 25); // x, y, w, h
    QPushButton * decButton = new QPushButton ("Dec", &mainWindow);
    decButton->move(150, 40); // x, y (statt 'setGeometry()')
    decButton->resize(55, 25); // w, h (statt 'setGeometry()')
    // .....
    mainWindow.show();
    return app.exec();
}
```

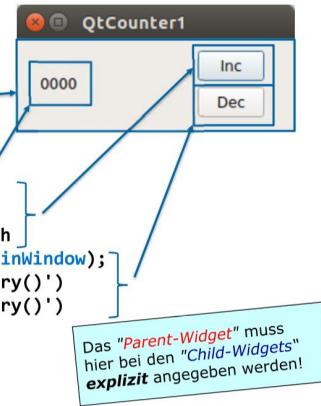


Abbildung 3.2: Beispiel eines manuellen Layouts

Die Positionierung erfolgt mittels eines Koordinatensystems bei dem der Ursprung (0,0) in der linken oberen Ecke ist. Die Position und Grösse wird in Pixel angegeben. Die x-Werte erhöhen sich nach rechts, während die y-Werte nach unten ansteigen. (x, y) sind die Koordinaten der linken oberen Ecke eines Widgets innerhalb eines anderen, des äusseren Widgets. w, h sind Breite und Höhe des inneren Widgets.

Methoden zur Festlegung von Position und/oder Grösse **void QWidget::setGeometry(int x, int y, int w, int h);**
void QWidget::resize (int w, int h);
void QWidget::setFixedSize(int width, int height);
void QWidget::move (int x, int y);

3.3 Qt Layout Manager

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDialog mainWindow;
    QBoxLayout* layout = new QHBoxLayout;
    mainWindow.setLayout(layout);
    QLabel * label = new QLabel("0000");
    layout->addWidget(label);
    QPushButton* incButton= new QPushButton("Inc");
    layout->addWidget(incButton);
    QPushButton* decButton= new QPushButton("Dec");
    layout->addWidget(decButton);

    mainWindow.show();
    return app.exec();
}
```

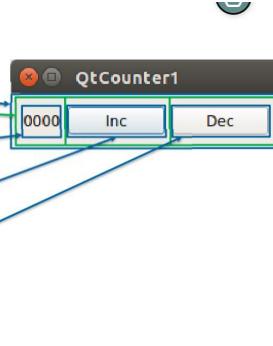


Abbildung 3.3: Qt Layout Manager

Klassen, welche von QLayout ableiten:

- QHBoxLayout: Ordnet Elemente horizontal an
- QVBoxLayout: Ordnet Elemente vertikal an
- QGridLayout: Ordnet Elemente in einem zweidimensionalen Gitters an.
- QFormLayout: Ordnet Elemente in Form von Zeilen an Zeile = QLabel gefolgt von anderem QWidget (z.B. QLineEdit)
- QStackedLayout: „Äufeinandergelegte“ Widgets (Stapel), nur immer ein Widget ist sichtbar

Das QLayout-Objekt ist bei einem Parent-Widget zuständig für das Layout (Anordnung) von dessen Child-Widgets:

QDialog mainWindow;

QVBoxLayout* mainLayout = new QVBoxLayout;

mainWindow.setLayout(mainLayout); Wird ein child-widget mit addWidget(label) hinzugefügt, dann wird es automatisch mit dem parent Widget verknüpft. Das QLayout-Objekt ist selbst ein "Kind" vom "Parent-Widget". Es ist zweckmäßig, sich das QLayout-Objekt als ein spezielles Geschwister vorzustellen, das als Kindermädchen ("Nanny") der "Child-Widgets" wirkt. QLayout-Objekte haben nie QWidget-Objekte als eigene Kinder.

Das QLayout-Objekt informiert das "Parent-Widget" automatisch über die Existenz der "Child-Widgets". Das geschieht mittels der addWidget Methode des Layout Managers:

QLabel * label = new QLabel("0000"); mainLayout->addWidget(label); Das "Parent-Widget" muss bei den "Child-Widgets" nicht explizit angegeben werden!

Layouts können auch verschachtelt werden: **QVBoxLayout* mainLayout = new QVBoxLayout; QHBoxLayout* bottomLayout = new QHBoxLayout; mainLayout->addLayout(bottomLayout);**

3.4 Qt Layout Manager

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDialog mainWindow;
    QVBoxLayout* mainLayout = new QVBoxLayout;
    mainWindow.setLayout(mainLayout);
    QLabel * label = new QLabel("0000");
    mainLayout->addWidget(label);
    QHBoxLayout* bottomLayout = new QHBoxLayout;
    mainLayout->addLayout(bottomLayout);
    QPushButton* incButton= new QPushButton("Inc");
    bottomLayout->addWidget(incButton);
    QPushButton* decButton= new QPushButton("Dec");
    bottomLayout->addWidget(decButton);
    mainWindow.show();
    return app.exec();
}
```

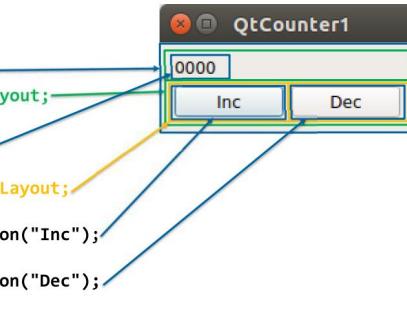


Abbildung 3.4: Qt Layout: Beispiel mit Verschachtelung

3.4.1 GUI Erstellung mittels Qt-Creator

Der Zugriff auf die Widgets: Widget::Widget(QWidget *parent) : QWidget(parent), **ui(new Ui::Widget)** -> Erstellt ein Layout **ui->setupUi(this);** **ui->lineEdit->setText("Test");** -> Setzt den Text in lineEdit Widget:: Widget() **delete ui;** -> Löscht das Layout **ui->lineEdit->setText("Test");**

3.4.2 Top-Level-Window/Widget

Definition: Widgets auf der obersten Hierarchiestufe des Widget-Trees. Widgets ohne Eltern Jedes Widget kann ein "Top-Level-Window" sein. Der Titelbalken ("TitleBar") gehört nicht zu Qt (!), sondern wird vom Betriebssystem erzeugt. Übliche Widgets für "Top-Level-Windows/Widget":

QMainWindow: Typisch als Applikations-Window (Hauptfenster). Es hat Toolbars und eine Statusbar.

QDialog "Popup-Window": typisch für Abfragen wie "Soll dies Datei wirklich gelöscht werden? Okay", "Abbrechen".

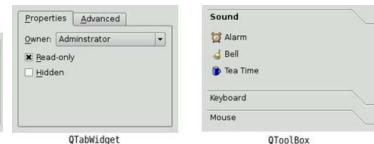
Häufig modal, d.h. die Hauptanwendung bleibt blockiert, bis der Dialog beendet ist. Die Standarddialog-Klasse von Qt ist QMessageBox **QWidget:** Einfaches Fenster, üblicherweise "non-modal" (d.h. nicht blockierend). In den weiteren Beispielen wird meistens QWidget als Top-Level Widget verwendet

3.5 Qt Widget Collection für Darstellungen (Selbststudium)

Single Page Container



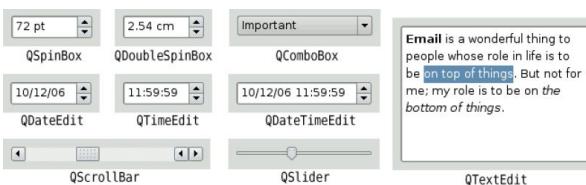
Multi Page Container



Button widgets



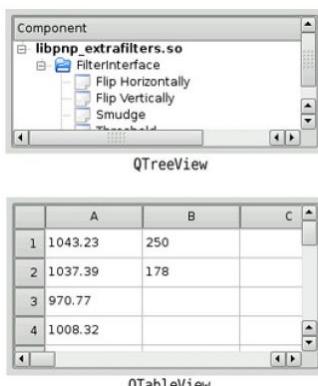
Input Widget



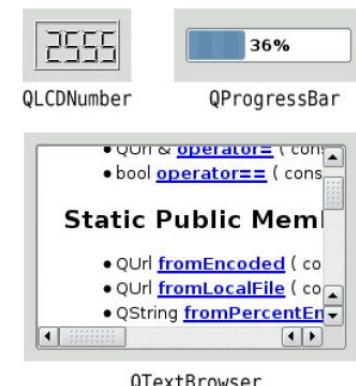
Color and Font Dialog



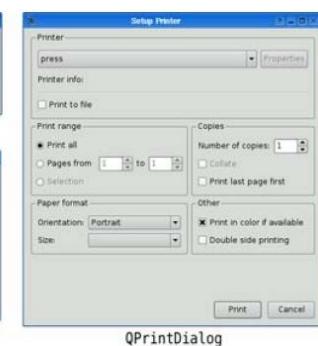
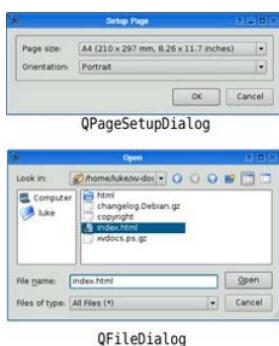
Item View Widgets



Display Widgets



File und Print Dialogs



Feedback Dialogs



4 Callback & Signals und Slots - Interaktion

4.1 Reaktive Systeme

Reaktive Systeme (reactive systems) reagieren auf (oft externe) Ereignisse wie digitale Inputs, die Erreichung von analogen Schwellenwerten, Timer, Buttonclicks oder ähnliches in GUI, etc. Dabei bestehen häufig auch Echtzeitanforderung an das System. Embedded Systems sind häufig auch reaktive Systeme. Ereignisse sind per Definition asynchron, d.h. sie treten zu einem beliebigen Zeitpunkt ein, während ein "normales" Programm immer synchron (zuerst das, dann das, dann ...) ist.

4.1.1 Pooling

Ereignisse können von synchronen Programmen durch Polling verarbeitet werden, d.h. das Programm fragt periodisch oder dauernd ab, ob irgendein Ereignis eingetreten ist. Polling ist sehr einfach zu implementieren. Dabei entstehen jedoch üblicherweise sehr viele Leerabfragen, d.h. bei vielen Abfragen ist nichts eingetreten, was eine unnötige Prozessorbelastung bewirkt. Sie können durch periodische Abfragen (Kopplung an Timer) reduziert aber nicht vermieden werden. Die maximale Reaktionszeit wird durch die Abfrageperiode und die Anzahl Abfragen (Stichwort: Looptime bei SPS) definiert.

4.1.2 C Callbacks

Callbacks stellen bidirektionale Verbindungen von Modulen mit C dar.

Beispiel einer bidirektionalen Verbindung zwischen A und B

Dabei kennt Klasse A, B UND Klasse B kennt ebenfalls die Funktion von Klasse A. C-File von A:

```
void aInit(void)
{ bRegisterFoo(&aFooY); /*Eine Funktion von B wird in A aufgerufen. Der Parameter der
Funktion ist eine Funktion der Klasse A. */ }
```

```
void aFooY(void)
{ printf("aFooY called\n"); }
```

C-File von B:

```
static void (*regFct)(void); % Funktionspointer
void bFooX(void)
{ if (0 != regFct)
  regFct(); }
void bRegisterFoo(void (*fct)(void))
{ regFct = fct; /* Die von der Klasse A aufgerufene Funktion bRegisterFoo hat als Parameter
  eine Funktion aus der Klasse A, welche dem Funktionspointer regFct zugewiesen wird und
  wieder auf die Funktion aFooY in der Klasse A zeigt. */ }
```

Beispiel einer unidirektionalen Verbindung zwischen Klasse A und B

Dabei hat die Klasse A eine Verbindung zu Klasse B, aber nicht umgekehrt.

Das File der Kasse A sieht folgendermassen aus:

```
#include "B.h" % Das Header File von B wird eingefügt class A
public:
A() b.foo(); % Verknüpfung mit B
void fooY()
private:
B b; % Ein Objekt der Klasse B kann somit in A aufgerufen werden
;
```

Bidirektionale Verbindung von Klassen

Klasse A kennt Klasse B UND Klasse B kennt Klasse A -> Das funktioniert so nicht, da es sich um ein circular include handelt. Eine Ausnahme stellt ein Class Forwarding dar.

Klasse A.h <pre>#include "B.h" class A { public: A() { b.foo(); } void fooY() {} private: B b; };</pre>		Klasse B.h <pre>#include "A.h" class B { public: void foo() { a.fooY(); } private: A a; };</pre>
---	---	--

Funktioniert so nicht → circular include (Ausnahme: class forwarding)

Abbildung 4.1: Keine gültige Bidirektionale Verbindungen von A und B

Die nachfolgenden Versionen einer bidirektionalen Verbindung:

V1: Klasse A kennt Klasse B UND Klasse B kennt Objekt von Klasse A Funktioniert, da in Header-File B.h nur Pointer auf A, der Rest ist im B.cpp File. Unschön: B muss immer noch A kennen (class forwarding erforderlich). Besser: A sollte unabhängig von B sein

V2: Klasse A kennt Klasse B UND Klasse B kennt Objekt von Klasse C, Klasse A ist eine Klasse C. Bessere Variante als V1: B kennt nicht mehr A direkt

V3: Klasse A kennt Klasse B UND Klasse B kennt Objekt von Klasse C, Klasse A ist eine Klasse C. Besser Variante als V2: der Funktionspointer wird weggelassen, da überflüssig

```
#include "B.h"
class A //Observer
{
public:
    A(B& b) : b(b)
    {
        b.registerFoo(*this, &A::fooY);
    }
    void fooY();
private:
    B& b;
};

int main()
{
    B b; A a(b);
    b.fooX(); // fire event
    return 0;
}

#include "B.h"
class A : public C //Observer
{
public:
    A(B& b) : b(b)
    {
        b.registerFoo(*this, &C::fooY);
    }
    virtual void fooY();
private:
    B& b;
};

int main()
{
    B b;
    A a(b);
    b.fooX(); // fire event
    return 0;
}

#include "B.h"
class A : public C //Observer
{
public:
    A(B& b) : b(b)
    {
        b.registerFoo(this);
    }
    virtual void fooY();
private:
    B& b;
};

int main()
{
    B b; A a(b);
    b.fooX(); // fire event
    return 0;
}

class A;
class B //Subject
{
public:
    ...
    void fooX()
    {
        if (0 != regFct)
            (regContext->*regFct)();
    }
    void registerFoo(A& c, void (A::*fct)())
    {
        regFct = fct;
        regContext = &c;
    }
private:
    void (A::*regFct)(); //Member Funktionspointer
    A* regContext;
};

class C {
    ...
    virtual void fooY() = 0;
};

class B //Subject {
public:
    ...
    void fooX()
    {
        if (0 != regFct)
            (regContext->*regFct)();
    }
    void registerFoo(C& c, void (C::*fct)())
    {
        regFct = fct;
        regContext = &c;
    }
private:
    void (C::*regFct)(); //Member Funktionspointer
    C* regContext;
};

class C {
    ...
    virtual void fooY() = 0;
};

class B //Subject {
public:
    ...
    void fooX()
    {
        if (0 != regContext)
            regContext->fooY();
    }
    void registerFoo(C& c)
    {
        regContext = &c;
    }
private:
    C* regContext;
};
```

Abbildung 4.2: Bidirektionale Verbindungen: V1-V3

Abstraktere C++ Umsetzungen folgen im Modul OOAD: Observer Pattern. Die kennengelernten Callback und Observer Methoden können für native C++ verwendet werden. In Qt können Signal/Slots verwendet werden.

4.2 Qt Signals and Slots: Qt Interaktion zwischen QObjects

Qt verwendet anstatt Callbacks ein Signal-Slot Prinzip. Analogie zu Callbacks: Das zu informierende Objekt (Observer) hat einen slot, Das informierende Objekte (Subject) hat ein signal, die Registrierung erfolgt über eine connect Funktion.

Es wird der Funktionspointer von C++-Member verwendet. Signals und Slots werden mit dem QObject::connect() verbunden.

Form: connect(SenderObj* (=Subject), SenderMethode* (signal1), EmpfängerObj* (=Observer), EmpfängerMethode* (slot2))

Bsp.: connect(&b, &B::fooX, &a, &A::fooY);

Beispiel:

```
class A : public QObject //Observer
{
    Q_OBJECT

public slots:
    void fooY()
};

class B : public QObject //Subject
{
    Q_OBJECT
signals:
    void fooX();
};

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    A a;
    B b;
    QObject::connect(&b, &B::fooX, &a, &A::fooY);
    // fire event
    emit b.fooX();           Sender      (signal)
    return app.exec();       (slot)      Empfänger
}
```

Abbildung 4.3: Beispiel Signal & Slots

4.2.1 Signals - Signal-Memberfunktion

Signals werden nur deklariert und nicht implementiert. Es werden keine Zugriffsrechte wie private public oder protect verwendet. Der Rückgabetyp ist immer void. Ein Signalaufruf erfolgt mit **emit method(value)**. Die C++ Definition erfolgt mit **#define signals public**. Ein Signal kann mit mehreren Slots oder anderen Signalen verbunden werden. Mehrere Signal können mit demselben Slot verbunden werden. Es kann einen Übergabeparameter haben.

4.2.2 Slots

Slots sind immer public (by default). Für private slots muss das Keyword slots weggelassen werden. Die C++ Definition lautet `#define slots`. Slots können einen Übergabeparameter haben.

4.2.3 Signals & Slots: Connect

Der Parameter Typ und Anzahl muss stimmen Signal: Slot: void clicked() -> void onClicked()
void indexChanged(int idx) -> void onIndexChanged(int idx)

Ab Qt5 - mit Funktionspointer

```
connect(senderObj, &SenderClass::senderMethod  
revObj, &RevClass::revMethod);  
+ weniger Klammern  
+ keine Datentypen für Parameter; aber Compiler meldet Fehler
```

Alte Notation

```
connect(button, SIGNAL(clicked()))  
count, SLOT(incValue()));  
Keine Compilerfehlermeldungen, falls falsch -> wird zur Laufzeit geprüft
```

Globale Funktionen als Slots (ab Qt5)

```
void printValue(int x)  
{ std::cout << "Value=" << x << std::endl; }  
connect(button, &QPushButton::clicked, &printValue);
```

Überladene Methoden

```
void QLabel::setNum(int num)  
void QLabel::setNum(double num)  
connect(..., static_cast<void (QLabel::*)(int)>(&QLabel::setNum), ..., ...);
```

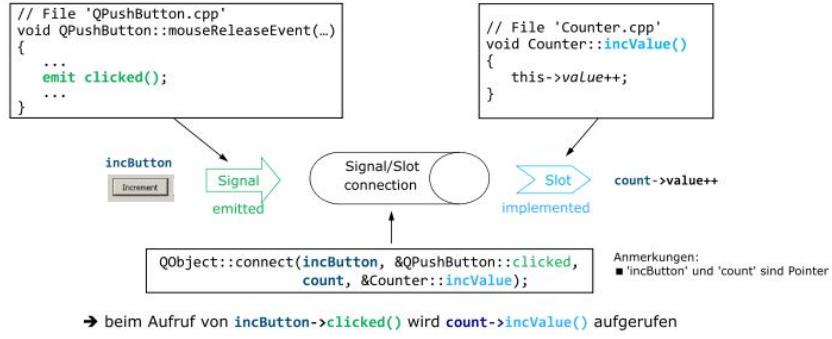


Abbildung 4.4: Beispiel mit Signal aus Framework

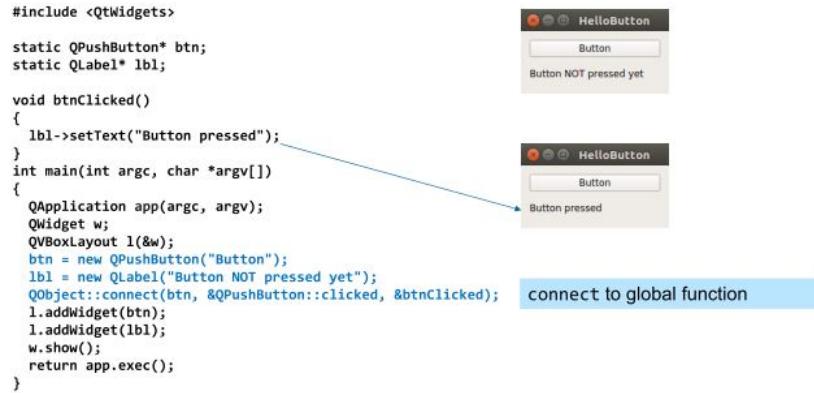


Abbildung 4.5: "Hello Button" GUI Programm

```

// File 'Counter.h'
#ifndef _COUNTER_H_
#define _COUNTER_H_
#include <QObject>
class Counter : public QObject
{
    Q_OBJECT
private:
    int value;
public:
    Counter(QObject *parent);
    int getValue() const
    {
        return value;
    }
    void setValue(int value);
signals:
    void valueChanged(int newValue);
};

// File 'Counter.cpp'
#include "counter.h"
Counter::Counter(QObject *parent)
: QObject(parent)
{
    value = 0;
}
void Counter::setValue(int value)
{
    if (value != this->value)
    {
        // wichtige Abfrage, ansonsten infinite loop
        this->value = value;
        emit valueChanged(value);
    }
}

// 'main.cpp':
...
Counter a;
Counter b;
QObject::connect(&a, &Counter::valueChanged, &b, &Counter::setValue);
QObject::connect(&b, &Counter::valueChanged, &a, &Counter::setValue);
a.setValue(12); // a.getValue() = ? , b.getValue() = ?
b.setValue(48); // a.getValue() = ? , b.getValue() = ?
...

```

Abbildung 4.6: Counter Beispiel: Connect mit sich selbst

5 Projekt Management

5.1 Management von Softwareprojekten

Eine **Organisation** kann definiert werden über ihre Struktur, ihre Funktion oder die Institution, welche sie darstellt.

Das **Management** übernimmt Leitungsaufgaben in Projekten und Unternehmen. Es handelt sich um eine Gruppe von Personen welche sich mit typischen Aufgaben wie Planung, Delegation, Organisation sowie Führung und Erfolgskontrolle.

Ein **Artefakt** ist ein primäres oder sekundäres Arbeitsergebnis, welches durch Projektaktivitäten erstellt wird. Beispiele: Pflichtenheft, Architekturentwurf, Testfallbeschreibung.

Ein **Projekt** ist komplex, neuartig, zeitlich befristet und weist begrenzte Ressourcen / Budget, Teamarbeit und messbare Ziele und Ergebnisse auf. Projektbeispiele sind ein neu entwickeltes, lauffähiges Softwaresystem, Teilergebnisse für die Entwicklung von Software, die Migration einer Software auf eine andere Hardwareplattform. Keine Projekte sind Routinetätigkeiten, Vorhaben des "daily business" wie beispielsweise Archivverwaltung und Kantinenbewirtschaftung. Ein Grenzbereich könnte die Pflege der Datenverarbeitung darstellen.

Softwareentwicklungsprojekte unterscheiden sich nicht grundlegend zu anderen Entwicklungsprojekten wie Elektrotechnische Erzeugnisse, Maschinen und Bauwerke. Es gibt jedoch einige signifikante Unterschiede durch die **Immateriellität** (Abstraktes Artefakt, bei SW nur Entwicklungsprozesse und Inbetriebnahme und keinen wirklichen Produktions- bzw. Fertigungsprozess). Die Folge davon ist, dass Fehler bei materiellen Produkten in der Regel noch während Planung oder Durchführung der Fertigung erkannt werden.

Projektstatistiken

Gemäss der CHAOS 2004 Umfrage kam es 2005 zu 15.52 % und 2007 zu 11.54 % Projektabbrüchen. Die Statistiken zeigen einen Trend, sollten jedoch nicht absolut verglichen werden. Gemäss der Standish Group sind die Zahlen nicht glaubhaft («Nur weil jemand eine Frage stellt, bedeutet das nicht, dass wir antworten. Tatsächlich antworten wir eher nicht»)

Gescheiterte Software Projekte

Der Ariane 5 Fehlstart Am 4. Juni 1996 startete die Ariane 5 Rakete zu ihrem Erstflug. Nach

genau 36,7 Sekunden sprengte sich die Rakete selbst inklusive vier Satelliten.

Kosten: 500 Mio Dollar

Ursache: 64Bit zu 16Bit-Wandlung

5.1.1 Ziele des Managements

Die Ziele des Software Engineering sind ein Produkt kostengünstig, termingerecht und in angemessener Qualität anzuliefern. Teilziele sind die Beherrschung von Zeit und Kosten. Kann ein unterschiedlicher Fokus gelegt werden: kostenfixiert, terminfixiert, leistungsfixiert, mit dem Ziel der Erreichung von Qualitätszielen, der Sicherung von Investitionen oder der Einhaltung von Rahmenbedingungen.

Folgender Zusammenhang gilt zwischen den Fehlerkosten von Softwareprojekten:

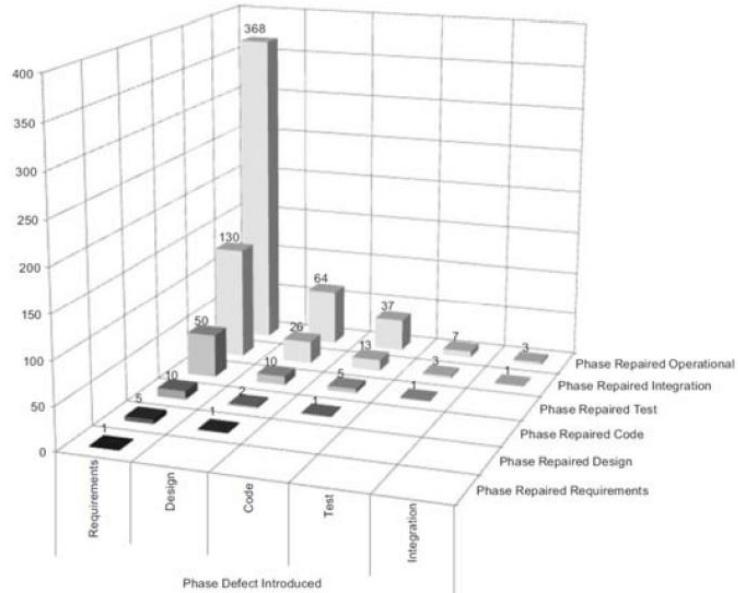


Abbildung 5.1: Relative Bugfixing Kosten

Faktoren für erfolgreiche Projekte sind:

- Angestellte: Hohe Motivation der Mitarbeiter / Erfahrene Projektmanager und Entwickler
- Kundenorientierung
- Management: Klare Projektziele / Realistische Projektpläne / Klare Verantwortungsstrukturen / Offene Kommunikationskultur
- Prozessorientierung
- Dokumentation und Artefakte
- Modularisierung und Wiederverwendung

5.2 Produkt-, Projekt- und Softwarelebenszyklus

Der Softwarelebenszyklus ist als Spezialisierung des Produktlebenszyklus anzusehen.

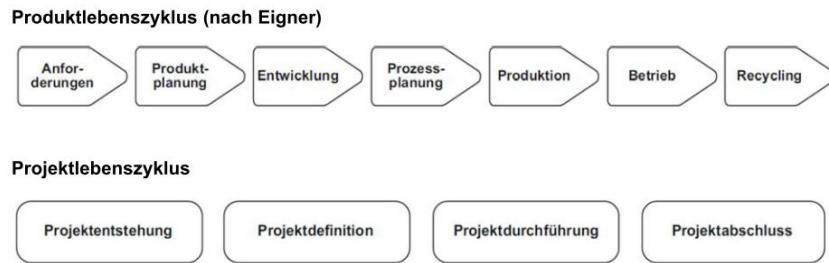


Abbildung 5.2: Produkt- & Projektlebenszyklus

5.2.1 Softwarelebenszyklus

- Analyse: Anforderungsfestlegung
- Design: Grob- & Feinentwurf (Architektur, Programmstruktur)
- Implementierung
- Test und Integration
- Betrieb & Wartung: Erprobung und Inbetriebnahme sowie Wartung und Weiterentwicklung
- Ausserbetriebnahme

Der Softwarelebenszyklus muss nicht sequentiell ablaufen. Zudem kann der Prozess iterativ mehrmals durchlaufen werden.

5.2.2 Phasen im Projektlebenszyklus

- Projektentstehung
 1. Projektidee entwickeln: Ziele, Bedarf, Chancen festlegen & ggf. Projektvorstudie durchführen – **M1: Projektskizze**
 2. Aufwand schätzen sowie Anforderungen und Ziele grob festlegen: Anforderungen notieren, grober Plan definieren, Schätzung durchführen und in einem Business Case zusammenfassen – **M2: Projektauftrag**

3. Angebots- und Vertragswesen: Lastenheft und Angebot festlegen und Gegenpartei übergeben – **M3: Vertrag / Projektvereinbarung**

- Projektdefinition

1. Projekt strukturieren und Projektteam organisieren
2. Definition eines Projektmanagementverfahrens und Erstellen eines Projekthandbuchs
3. Definition eines QS-Verfahrens und Erstellen eines QS-Handbuch
4. Aufbau einer Projektinfrastruktur
5. Projekt planen bzw. Erstellen eines Projektplan

- Projektdurchführung

Durchführung eines iterativen Entwicklungsprozess gemäss den Phasen des V-Modells. Der Input entspricht dem Pflichtenheft und die Lieferung dem Output. Zeitgleich werden die iterativen Phasen von Planung (Soll-Vorhaben) –> Kontrolle (Messung von Kennzahlen und Soll-/Ist-Vergleich) –> Steuerung (Steuerungsmassnahmen hinsichtlich QS oder Projektmanagement) –> Anpassung des Vorhabens durchlaufen.

- Erfassung und Verfeinerung der Anforderungen
- Systementwurf
- Implementierung, Verifikation und Test
- Test und Integration
- Erprobung und Übergabe

- Projektabschluss

1. Ergebnisse der Entwicklung: Code, Systemarchitektur, Tests, Dokumentation, System
2. Lieferung abnehmen und System in Betrieb setzen (inkl. Freigaben)
3. Projekt abschliessen beinhaltet die Sicherung der gemachten Erfahrungen, die Auflösung des Projektteams und die Projektabrechnung –> Projektabschlussbericht

Mx: stehen für die erreichten Meilensteine innerhalb einer Phase

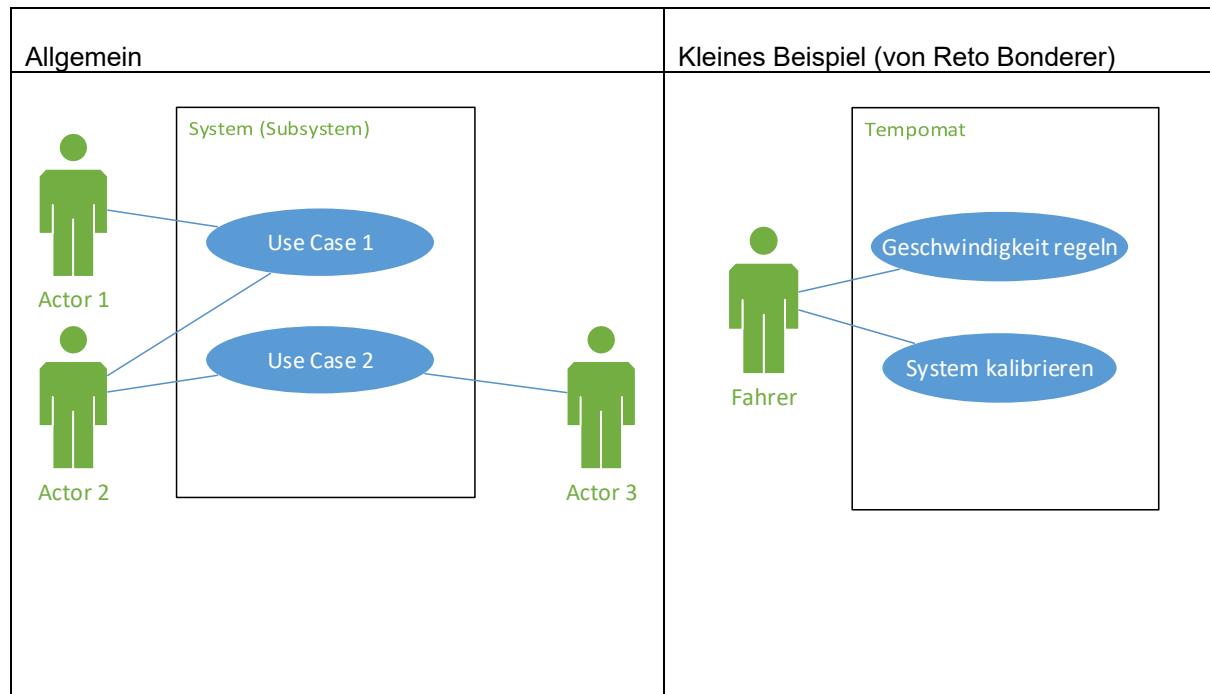
Use-Case Diagramm

Dieses Diagramm hat den Vorteil, dass nicht nur die Funktionalitäten (Use-Cases) beschrieben werden, sondern auch die Systemgrenze und die dazugehörigen Akteure. Es beschreibt somit die Funktionalität des Systems und in welchem Umfeld sich das System befindet. Das Zeichnen eines Use-Case Diagramms kann umgekehrt auch zeigen, ob das Ziel des Produkts auf beiden Seiten (auch Auftraggeber!) verstanden wurde.

Ein einfaches Use-Case Diagramm besteht aus folgenden Teilen:

Use-Case	Bezeichnen Funktionen, die entwickelt werden müssen. Die Funktionen werden von den Anforderungen abgeleitet.
Actor	Actors sind ausserhalb des betrachteten Systems. Sie interagieren resp. kommunizieren mit dem System resp. Use-Cases (siehe Use-Case/Actor Verbindung).
Use-Case/Actor Verbindung	Eine Verbindung zwischen einem Use-Case und einem Actor zeigt, dass eine Kommunikation zwischen der entsprechenden Funktionalität des Systems und dem Actor stattfindet.
Betrachtetes System	Umschliesst alle Use-Cases für ein Produkt. Zeigt die Systemgrenze des betrachteten Systems auf.

Die folgende Tabelle zeigt die grafische Darstellung des Use-Case Diagramms nach UML:



6 Vorgehensmodelle

6.1 Vorgehensmodelle in der Softwareentwicklung

Vorgehensmodelle beantworten die wesentlichen Fragen der Organisation. Sie beschreibt das systematische, ingeniermässig und quantifizierbare Vorgehen, um Aufgaben einer bestimmten Art wiederholbar zu lösen. Vorgehensmodelle werden hauptsächlich während Projekt-durchführungsphase verwendet. Sie bilden eine Grundlage für die weitere Planung, die Projektüberwachung und die Projektsteuerung.

Sie bilden ein Instrument zur Integration von Methoden wie RE (Requirements Engineering) und Testen. Sie können anhand der W-Fragen strukturiert werden: Wer (-i Rollenmodell) erarbeitet Womit (-i Werkzeug & Standards) Was (-i Artefakt-/Produktmodell) bis Wann (-i Ablaufmodell: Phase) und Wie (-i Aktivitätsmodell) geht er dabei vor?

6.2 Grundsätzliche Vorgehensmodelle

Für Softwareprojekte stehen eine Reihe grundsätzlicher Vorgehensmodelle zur Verfügung. Die Modelle sollten auf das Projekt zugeschnitten werden (tailoring). Die Wahl für ein Modell erfolgt aufgrund des Projektcharakters.

- Phasenmodell
- Spiralmodell
- Prototyping
- Agile Methoden

6.2.1 Phasenmodell

Das Phasenmodell ist ein klassischer Ansatz für die Organisation der Software- und Hardware-Entwicklung. Jede Phase erzeugt ein Ergebnis / Meilenstein, auf dem die nächste Phase aufbaut.

6.2.2 Wasserfall-Modell

Philosophie: Aufteilung anhand Schwerpunkten der Entwicklungstätigkeiten

Phasen: Analyse und Anforderung, Architekturentwurf, Implementation, Verifikation und Integration sowie Betrieb und Wartung

Vorteile:

- Einfache, klare Struktur
- Weitgehende Übereinstimmung des Artefakt- und Prozessmodells –; Vereinfacht scheinbar Planung, Organisation und Steuerung

Nachteile:

- Planungs- und Entwicklungsfehler werden spät bemerkt
- Risiken werden (zu) spät erkannt
- Umplanungen stehen im Gegensatz zur Philosophie und sind kostspielig
- Eine Rückkopplung aus der Implementationsphase findet sehr spät statt - Enorme Erfahrung ist/wäre notwendig Iterativer Wasserfall: Kaskade; Erweiterung des Wasserfalls um Korrektur-

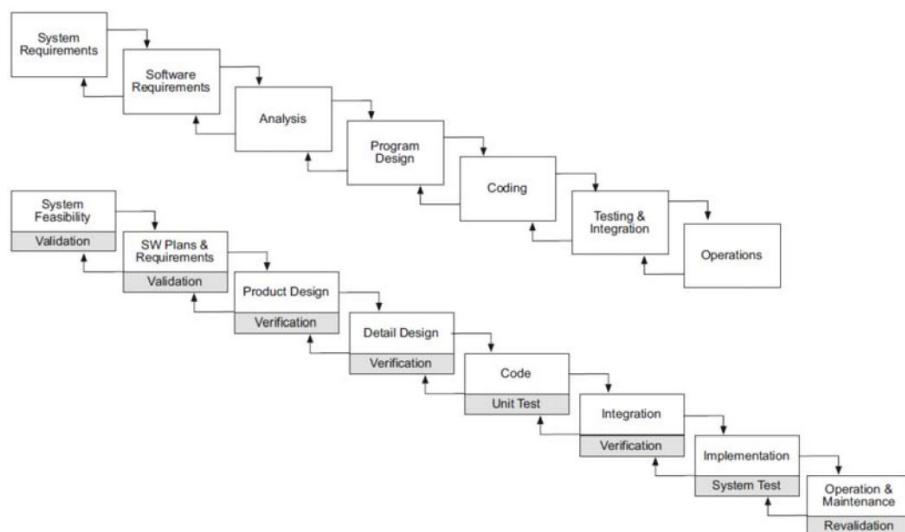


Abbildung 6.1: Wasserfall - klassisch & iterativ

schleifen; In der Realität durchführbar (nicht wie der reine Wasserfall). Man kann immer nur eine Phase zurück

6.2.3 V-Modell

Grundidee: korrespondierende Tests

Erweiterung des Wasserfallmodells –; Ermahnt zum Testen

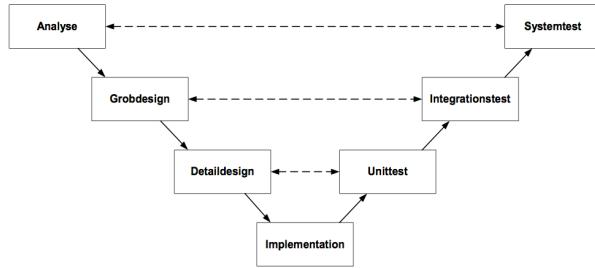


Abbildung 6.2: V-Modell

Vorteile:

- Starke Einbindung der Test zu jeder Phase
- Klare, einfache Struktur
- Enorme Erfahrung aufgrund vorhandener Rückkopplung

Nachteile:

- kann zu inhaltlichen Problemen führen

6.2.4 Spiralmodell

Wiederholtes Durchlaufen der klassischen Entwicklungsphasen (Analyse, Evaluierung, Realisierung und Planung). Kontinuierliche Bereitstellung von Prototypen. Jede Iteration bringt den Prototypen näher an das endgültige Produkt.

Philosophie Erstellung von kontinuierlichen Prototypen und kontinuierliche Prüfung des Systems. Dies erlaubt kontinuierliche Lernkurven. Zudem werden die Ergebnisse der Zyklen wie Anforderungen oder Architekturen bei jeder Iteration verfeinert. Das weitere Vorgehen wird im 3. Schritt neu gewählt. Es ist kein Vorgehensmodell an sich, denn es sollte durch andere Vorgehensmodelle ausgestaltet werden. Im 4. Schritt kann allfällig eine Aufspaltung in Teilprojekte geschehen.

Inkrementelles Vorgehen

Die Funktionalität wird schrittweise entwickelt bis das angestrebte System komplett ausgebaut ist. Iteratives Vorgehen ist zwingend nötig.

Vorteil:

- Benötigt weniger Erfahrung als Wasserfallmodell, aufgrund der zugelassenen Lernkurven
- Risiken können frühzeitig erkannt werden

6.2.5 Prototyping

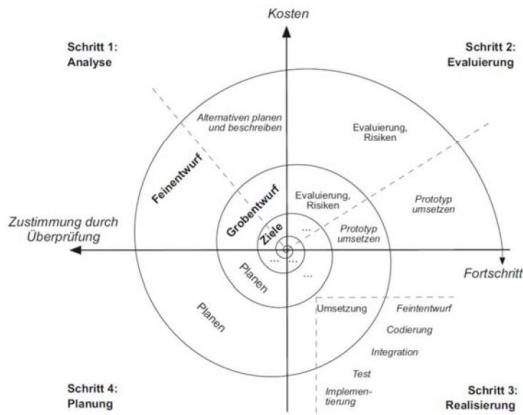
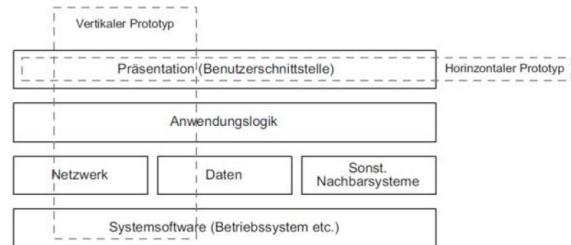


Abbildung 6.3: Spiralmodell

Prototyping beschreibt die frühe Realisierung ausgewählter oder kritischer Funktionen. Dabei geht es darum die Realisierung unter realitätsnahen Bedingungen zu zeigen. Es wird zur Prüfung der Machbarkeit, bei fehlender Erfahrung oder bei unklar definierten Anforderungen durchgeführt.

Anwendungen für Prototyping: bei User Interfaces, Smartphone Apps und Datenbanken
Es wird zwischen einem horizontalen (Fo-

kus auf eine Architekturebene) und vertikalen Prototyp (Fokus auf ein Funktionalität über alle Architekturebenen) unterschieden.



6.2.6 Agile Methoden

- Gegensatz zu „schwergewichtigen“ Vorgehensweisen mit hohem Regelungs- und Organisationaufwand – hohe Flexibilität, leichtgewichtiger Ansatz
- Agile = flink, beweglich / „Codezentriertes Vorgehen“

Agiles Manifest (2001) - Individuen und Interaktionen mehr als Prozesse und Werkzeuge

- Funktionierende Software mehr als umfassende Dokumentation
- Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung
- Reagieren auf Veränderung mehr als das Befolgen eines Plans

Prominente Agile Methoden umfassen Refactoring, Pair Programming, Test-driven Development, Continuous Integration, Planning Game / Poker

Vorteil:

- Vorgehen kann für einen Hardware unabhängigen Teil der Software sinnvoll sein – Stichwort

”Hybrides Modell”

- Wertvolle Methoden: Refactoring / Test-Driven Development –> v.a. für erfahrene Programmierer

Nachteil:

Schwierig bei Hardwareentwicklung, da Anforderungen nicht zwingend von Anfang definiert

6.3 Konkrete Vorgehensmodelle

Die hier vorgestellten Modelle basieren auf den bereits gezeigten grundlegenden Philosophien. Die dabei grob beschriebenen Vorgehensmodelle eignen sich nur bedingt für die unmittelbare Anwendung in Projekten.

6.3.1 Scrum

Bei Scrum organisiert sich das Team weitgehend selbst. Die Grundannahme ist, dass Projekte komplex sind und somit nicht von Anfang an detailliert planbar sind. Am Anfang wird ein grober Projektrahmen definiert, indem sich das Team selbstorganisierend bewegt.

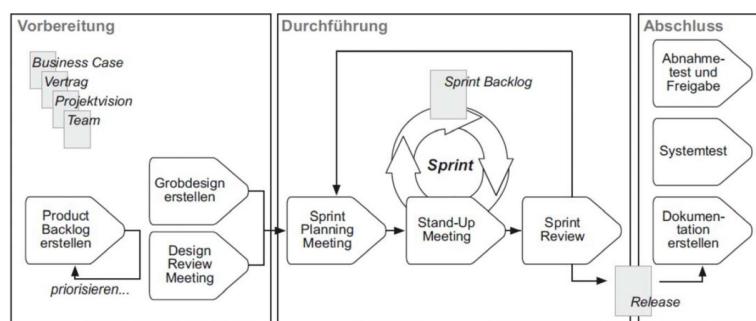


Abbildung 6.4: Scrum Prozess

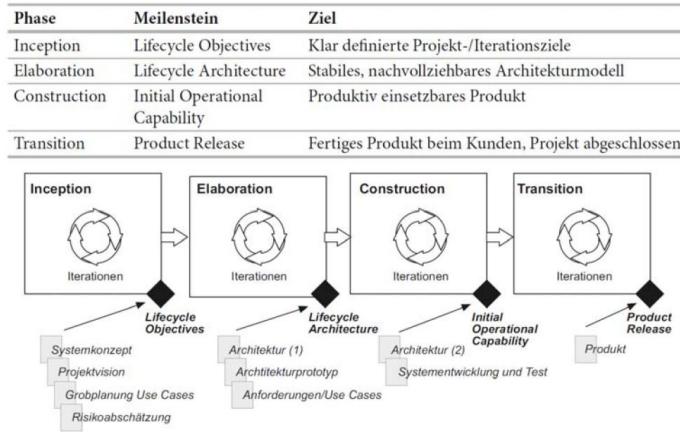
Basis: Sprints von 15-30 Tagen

6.3.2 Rational Unified Process (RUP)

RUP wurde von IBM Rational erfunden und wird kommerziell vertrieben. Es handelt sich um ein objektorientiertes, aktivitätsgtriebenes Vorgehensmodell, welches stark auf UML ausgerichtet ist. Grundprinzipien:

- Anwendungsfallgetrieben
- Die Architektur steht im Zentrum der Planung

- Das Vorgehen zur Entwicklung ist inkrementell/iterativ



6.3.3 Diskussion der Vorgehensmodelle

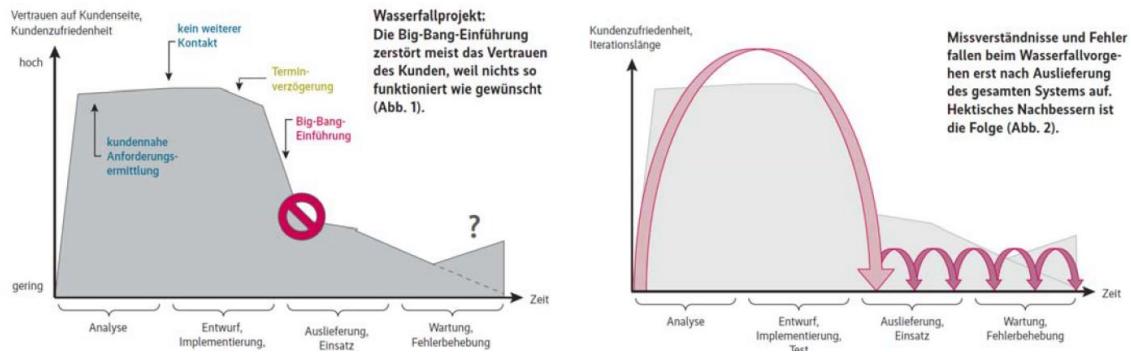


Abbildung 6.5: Diskussion der Vorgehensmodelle

Brook'sche Regel

Adding manpower to a late project makes it later

Problem: Kommunikationsaufwand wird massiv erhöht.

7 Aufwandsschätzung

7.1 Grundlagen

Die Aufwandsschätzung ist ein kritischer, schwieriger und wichtiger Punkt in der Projektvorbereitung, -vereinbarung und -planung. Sie wird üblicherweise von sämtlichen Vertragsparteien unabhängig durchgeführt. Die Aufwandsschätzung bildet eine Grundlage für die Wirtschaftlichkeit, die Planung von Zeit und Projektmitteln und die Festlegung des erforderlichen Budgets. Der Projekterfolg hängt massgeblich von der Zuverlässigkeit der jeweiligen Schätzungen ab. Das Ziel von Schätzungen ist eine realistische Angabe über den erwartenden Aufwand und das erforderliche Budget zu machen.

Pessimistische Schätzungen sind oftmals wirtschaftlich uninteressant, da das Angebot nicht konkurrenzfähig ist. Sie beinhalten die Befürchtungen, dass Mittel und Freiräume dennoch ausgeschöpft werden (?).

Optimistische Schätzungen beinhalten ein grosses Risiko, da das Projekt meist nicht planmäßig durchgeführt werden kann. Sie können zu wirtschaftlichen Verlusten führen.

Aufwand- und Kostenabschätzung sollten sich am Umfang und der Besonderheiten des zu entwickelnden Ergebnisses orientieren. Dies kann die Software selbst, Dokumentation sowie Schulungsunterlagen sowie Dienstleistungen für Schulungen und Unterstützung bei der Einführung sein.

Prinzipien bei Schätzungen

- Bildung von Schätzpaketen
- Aufwandseinheiten
- Einbindung der Mitarbeiter
- Dokumentation
- Erfahrungswerte und Aufschläge
- Kontinuierliche Kontrolle

7.1.1 Schätzungsansätze

Aktivitätsorientierte Schätzung

Nr.	Bezeichnung	Abhängig von	Aufwand (in PT)
1.	Durchführung von Interviews (Zahl, Aufwand)	...	15
2.	Ermittlung der Nutzungsfälle	...	5
3.	Erarbeitung Datenmodell	...	20
4.	Analys Altystem	...	15
...
12.	Erarbeitung Anforderungsdokument	(alle)	45
	Summe:		197

Artefaktbasierte Schätzung

Nr.	Bezeichnung	Komplexität	Aufwand (in PM)
1.	Komponente A	einfach	15
2.	Komponente B	einfach	5
3.	Funktion X	mittel	10
4.	Anwenderdokumentation	einfach	25
...
25.	Komponente Y	schwer	25
	Summe:		237

7.1.2 Streuung der Schätzung

Einflussfaktoren

Einflussfaktor	Aufwandsverhältnis
schwierige/einfache Benutzerschnittstelle	4 : 2
mit/ohne Benutzermitwirkung	2 : 2
mit/ohne Leistungsforderungen	2 : 2
mit/ohne Entwicklungsmethode	1 : 2
höhere/niedrigere Qualifikation	1 : 3
mit/ohne parallele Geräteentwicklung	2 : 1
mit/ohne CASE-Tools	1 : 7

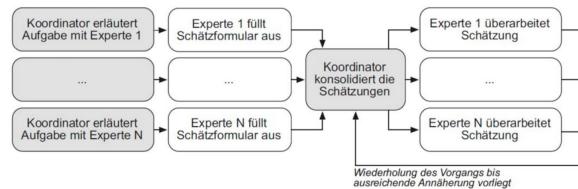
Leistungen in Abhängigkeit des Einsatzes

Aspekt	Verhältnis Schlechtes : Gutes Ergebnis
Testzeit	26 : 1
Rechenzeit für Test	11 : 1
Codierzeit	25 : 1
Programmgröße	5 : 1
Laufzeit des Programms	13 : 1

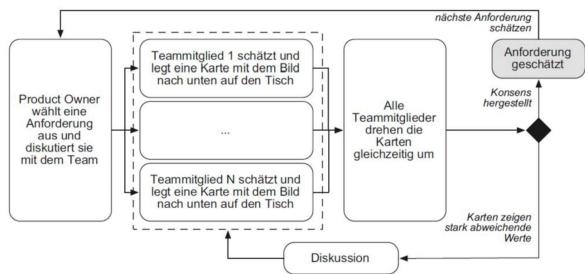
7.2 Expertenschätzungen

Die Expertenschätzungen finden an einer Schätklausur statt. Dabei führen alle Beteiligten die Schätzung gemeinsam durch. Das Ziel ist dabei realistische Schätzergebnisse mit hoher Genauigkeit unter Berücksichtigung aller relevanten Faktoren zu erzielen. Ausserdem verspricht man sich dabei eine hohe Akzeptanz bei den späteren Projektmitarbeitern.

Delphi Methode



Planning Poker



7.2.1 Drei-Zeiten-Methode (nach PERT) zur Gewichtung von Schätzungen

Aufwandsmittelwert für Schätzobjekt i: $A_i = \frac{bc_i + 4lc_i + wc}{6}$

Standardabweichung für Schätzobjekt i: $S_i = \frac{wc_i - bc_i}{6}$

bc: optimistische Schätzung (best case)

wc: pessimistische Schätzung (worst case)

lc: wahrscheinliche Schätzung (most likely case)

7.3 Algorithmische Schätzverfahren

- Aron-Modell: (Selbststudium) Broy Kap. 6.3.3.1
- COCOMO: (Selbststudium) Broy Kap. 6.3.3.2, Hummel Kap. COCOMO S. 70 ff
- COCOMO II: (Selbststudium) Broy Kap. 6.3.3.2, Hummel Kap. COCOMO S. 70 ff
- Lines of Code (LOC)
- Function Points
- Speziell für Embedded: 3D Function Points oder COSMIC Full Function Points

7.3.1 Function Points

Function Points dienen der Schätzung des Umfangs der geforderten Funktionalität. Die zu schätzende Software wird als Black Box betrachtet, weshalb die Schätzung über die Schnittstellen, mit denen das System mit seiner Umwelt interagiert, erfolgt.

Datenelemente

- (ILF): Internal Logical Files
- (EIF): External Interface Files
- Data Element Types (DET): für den Systembenutzer sichtbare Datenfelder (z.B. Attribute von Klassen)
- Record Element Types (RET): Mehrere logisch zusammenhängende DETs ergeben einen RET

DETs und RETs sind verfeinernde Elemente für ILFs und EIFs.

Transaktionselemente:

- External Input (EI): von dem Benutzer oder einer umgebenden Applikation an die zu schätzende Applikation
- External Output (EO): von der zu schätzenden Applikation an den Benutzer oder eine umgebende Applikation
- External Query (EQ): Abfrage zwischen Benutzer oder umgebende Applikation und der zu schätzenden Applikation

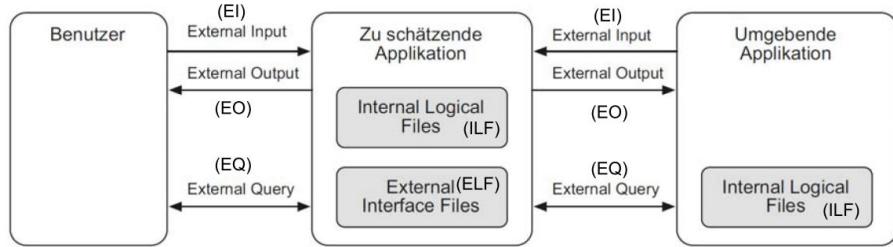


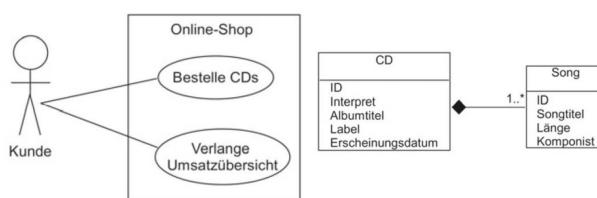
Abbildung 7.1: Daten- und Transaktionselemente

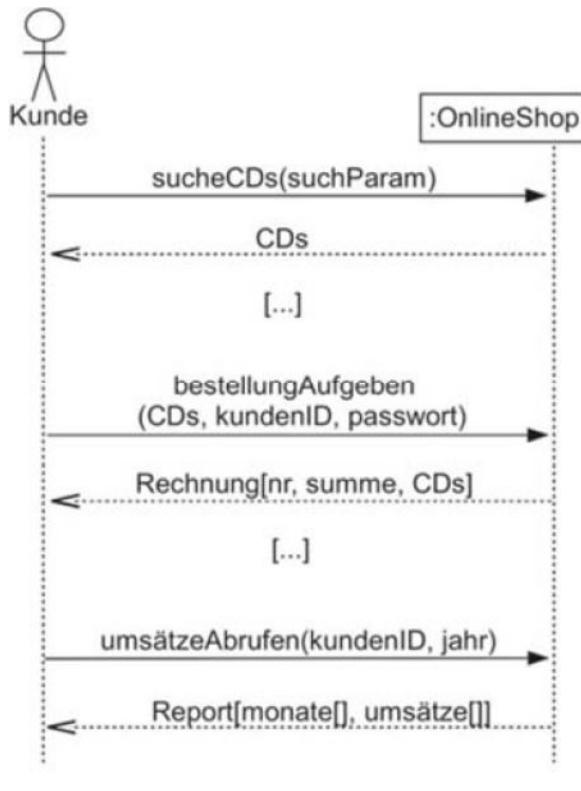
Vorgehen

Voraussetzung: Die Anforderungen müssen bekannt sein (unabhängig vom Detaillierungsgrad)

1. Use-Case Diagramm erstellen: Festlegen der Systemgrenze
2. Identifikation der Datenelemente (ILF, EIF)
3. Identifikation der Transaktionselemente (EI, EO, EQ)
4. Berechnung der Unadjusted Function Points

Beispiel zu den Function Pointers





Datenelemente

Internal Logical File (ILF):

CD; 2 RETs (CD+Song)

- CD hat 5 DETs (Attribute von CD)

- Song hat 4 DETs (Attribute von Song)

DET total: 5+4 DETs (CD + Song)

Function Point Rating (FTRs): gering

Transaktionselemente

- `sucheCDs` (EQ): 1 FTR (CD), 7 DETs (alle CD Attribute + PushBtn click + Parameter)

=_i geringe Komplexität = 3 FP

- `bestellungAufgeben` (EI): 4 FTR (Kunde, CD, Bestellung, Rechnung), 6 DETs (CD-ID, KundenID, Passwort, PushBtn click, RechnungsNr, Rechnungssumme) =_i hohe Komplexität: 6 FP

- `umsätzeAbrufen` (EO): 2 FTR(Kunde, Rechnung), 5 DETs(KundenID, Jahr, Monate, Umsätze, PushBtn click) =_i geringe K.: 4 FP

7.3.2 Bewertungslisten

Funktionpoints pro Komplexitätsstufe

Komplexität	ILF	EIF	EI	EO	EQ
Gering	7	5	3	4	3
Mittel	10	7	4	5	4
Hoch	15	10	6	7	6

Komplexität pro Anzahl RETs & DETs

RETs	DETs		
	1-19	20-50	mehr als 50
1	gering	gering	mittel
2-5	gering	mittel	hoch
mehr als 6	mittel	hoch	hoch

Function Points Ratings für EI

FTRs	DETs		
	1-4	5-15	mehr als 15
0-1	gering	gering	mittel
2	gering	mittel	hoch
mehr als 3	mittel	hoch	hoch

Function Points Ratings für EO und EQ

FTRs	DETs		
	1-5	6-19	mehr als 19
0-1	gering	gering	mittel
2-3	gering	mittel	hoch
mehr als 3	mittel	hoch	hoch

7.4 Aron (Selbststudium), Broy Kap. 6.3.3.1

**7.5 COCOMO (Selbststudium), Broy Kap. 6.3.3.2, Hummel Kap.
COCOMO S. 70 ff**

**7.6 COCOMO II (Selbststudium), Broy Kap. 6.3.3.2, Hummel Kap.
COCOMO S. 70 ff**

8 Projektplanung

Planung ersetzt den Zufall durch den Irrtum. Albert Einstein

Plans are nothing; planning is everything. Dwight D. Eisenhower

”Plane – und du wirst irren. Plane nicht – und du wirst nicht wissen, wo du geirrt hast.”

8.1 Projekt- und Arbeitsplanung

8.1.1 Kernbestandteile der Projektplanung

- Projektstrukturplan - Aufteilung in Themenblöcke:
 - Phasenorientiert: Anforderungerhebung, Architektur & Design, Implementation
 - Objektorientiert: Benutzerschnittstelle, Datenhaltung, Anwendungsfunktionalität, Kommunikation, Systemdienste
 - nach Verantwortung: QS, Fortschrittskontrolle, Anwendungs- und/oder Nutzungsfälle (Use Cases)
- Arbeitspakete
beschreiben eine klar definierte Tätigkeit, definierte Dauer, zugeordnete Ressourcen sowie (logische oder zeitliche) Abhängigkeiten von anderen Arbeitspaketen
- Zeit-/Terminplan (ggf. nur die nächsten Arbeitspakete ausdetailieren”)
- Meilensteinplan
 - Feststellen der Zielerreichung
 - Kann erfüllt oder nicht erfüllt sein
 - Blockierende Meilensteine (z.B. Projekt nicht genehmigt - Projekt steht still)
 - Meilensteine sollten nicht zu nahe zusammenliegen)
- Schätzungen (siehe VL 7 Aufwandsschätzung)

Der Inhalt von Arbeitspaketen umfasst einen eindeutigen Identifikator, einen Namen zur Bezeichnung, eine Beschreibung des Arbeitspakets, Verantwortliche und mitwirkende Rollen/Personen, die geschätzte Dauer des Arbeitspakets, Ergebnisdefinition (geforderte Artefakte), Qualitätsanforderungen, Abnahmekriterien, technische Abhängigkeiten sowie zu erfassende Kennzahlen

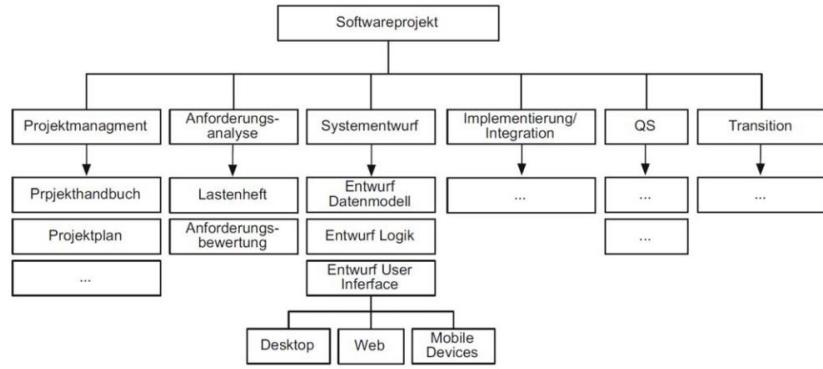


Abbildung 8.1: Projektstrukturplan

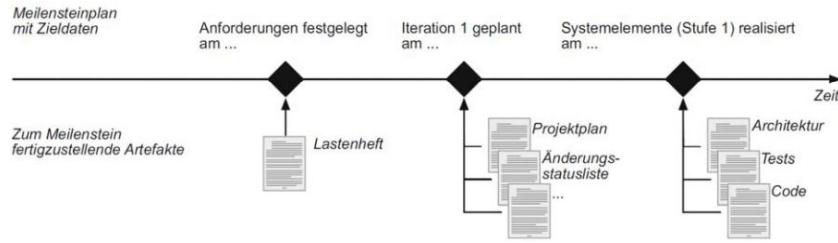


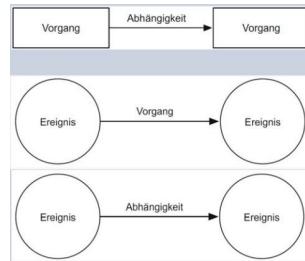
Abbildung 8.2: Meilensteinplan

8.1.2 Netzplantechnik

Grafische Darstellung der Zeitpunkte für die Durchführung von Tätigkeiten
 Logische Beziehungen und zeitliche Anordnung zwischen den Arbeitspaketen wird dargestellt.
 Wie lange wird das Projekt (mindestens) dauern? Welche kritischen Arbeitspakete können das gesamte Projekt verzögern?

Methoden

- Critical Path Method
- Program Evaluation Review Technique (PERT)
- Meta Potential Method (MPM)



Modellierung

Kritischer Pfad berechnen: Wie in Digitaltechnik - Zeiten berechnen für ein Signal im Netzwerk.
 Verkettung von derjenigen Vorgänge, bei deren zeitlicher Änderung sich der Endtermin des

Netzplanes verschiebt. Aktivität a liegt auf einem kritischen Pfad bei $p(a) = 0$. Falls es keinen Projekt-Endtermin gibt hat man mindestens einen kritischen Pfad.

Vorwärtsrechnung:

Frühster Endtermin $fet(a)$: Frühster Anfangstermin $fat(a)$ + Vorgangsdauer $d(a)$

Rückwärtsrechnung:

Spätester Anfangstermin $sat(a)$ = spätester Endtermin $set(a)$ - Vorgangsdauer $d(a)$

Puffer:

Zeitdauer in der man die Aktivität herum schieben kann, ohne, dass sich das Projekt verzögert.

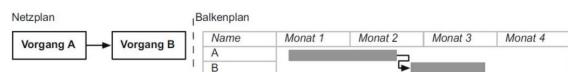
Puffer $p(a)$ = spätester Endtermin $set(a)$ - frühster Endtermin $fet(a)$

= spätester Anfangstermin $sat(a)$ - frühster Anfangstermin $fat(a)$

8.1.3 Balkenplantechnik - Gantt-Diagramm

Normalfolge: End-Anfang-Beziehung

B startet wenn A fertig ist



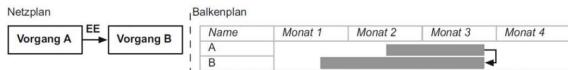
Anfangsfolge: Anfang-Anfang-Beziehung

A und B müssen zeitgleich starten



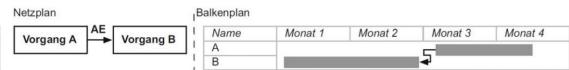
Endfolge: End-End-Beziehung

A und B müssen zeitgleich beendet werden



Sprungfolge: Anfang-End-Beziehung

B kann erst beendet werden, wenn A anfängt



Projektplanung in der Praxis

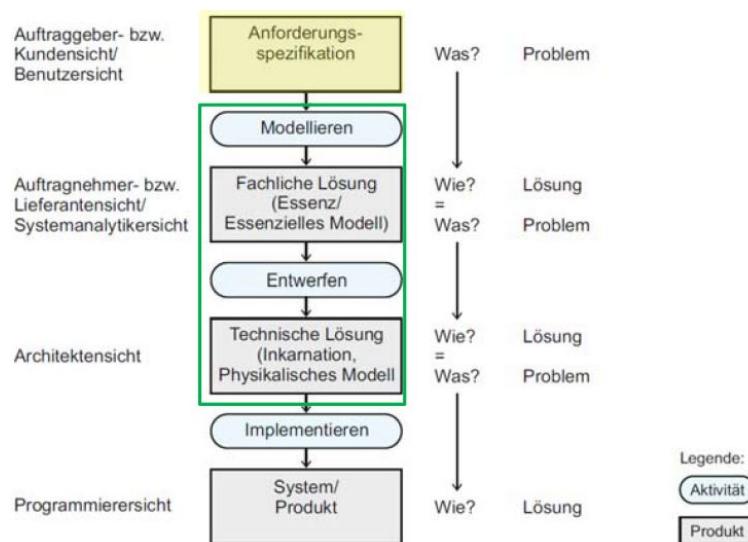
- Netzpläne: für Grobplanung und prozessorientierte Darstellung
- Arbeitspakete werden oft zu stark unterteilt, Arbeitspakete mit Dauer > 1 Woche sollten ggf. überdacht werden
- Keine Abhängigkeiten zwischen Arbeitspaketen, wo keine notwendig sind:
 - > Zuerst Pflichtenheft erstellen, dann Design / Implementation (Abhängigkeit)
 - > Implementation und Test können parallel gemacht werden (keine Abhängigkeit)
 - > Lieber eine Abhängigkeit weniger machen

Es gibt verschiedene Philosophien zur Projektplanung. Darunter sind Meilenstein-orientierte Planung, Fast Tracking, Time Boxing, Critical Chain und Kanban.

9 Requirements Engineering

Requirements Engineering heisst es Anforderungen an ein neues Softwareprodukt ermitteln, spezifizieren, analysieren und validieren. Daraus soll eine fachliche Lösung abgeleitet werden.

9.1 Vom Problem zur Lösung



Elemente davon sind:

- Anforderungen: Pflichtenheft
- Fachliche Lösung: OOA-Modell
- Technische Lösung: OOD-Modell

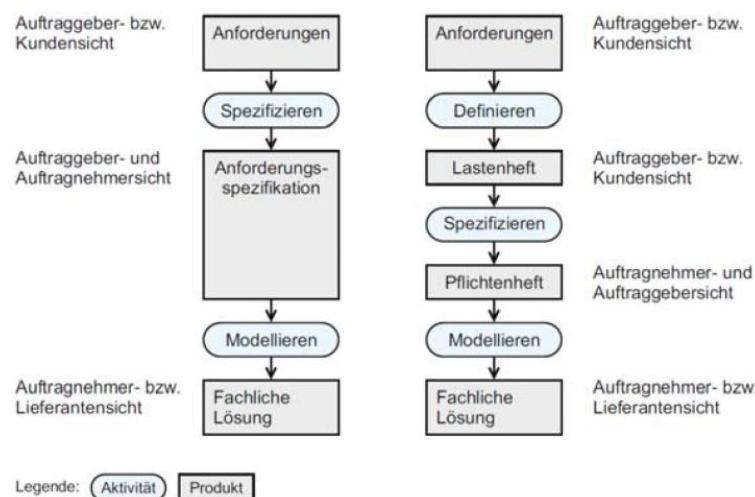
Das OOA- und OOD-Modell sind Thema im Modul OOAD. Je weitere im Prozess, desto stärker wird der Lösungsraum eingeschränkt, wobei der Detaillierungsgrad der Lösung zeitgleich steigt.

9.1.1 Hauptgründe für einen Projektabbruch

- Änderungen der Anforderungen - 33 %
- Mangelnde Einbindung des höheren Managements - 33 % (Top Mgmt Commitment)
- Engpass im Budget - 28 %
- Fehlende Projektmanagement-Fähigkeiten - 28%
- Siehe Bild zu den "Relativen Bugfixing Kosten"

9.2 Anforderungen und Anforderungsarten

Die Anforderungen legen fest, was man (Stakeholder / Auftraggeber) von einem Softwaresystem als Eigenschaft (Visionen, Ziele, Rahmenbedingungen) erwartet. Eigenschaften lassen sich in funktionale und nichtfunktionale Anforderungen unterteilen. Die Anforderungsspezifikationen werden bei der Projektausschreibung in Lasten- und Pflichtenheft unterteilt.



Lastenheft

- Zusammenstellung der Anforderungen
- WAS und WOFÜR aus Sicht des Auftraggebers

Pflichtenheft

- Beschreibung der Realisierung der Anforderungen vom Lastenheft

Vision

- Beschreibt eine realitätsnahe Vorstellung der gewünschten Zukunft
- Sie beschreibt, was erreicht werden soll, sagt aber nicht wie

Regeln zur Definition von Zielen:

Ausgehend von einer Vision dienen Ziele dazu, die Vision zu verfeinern und zu operationalisieren.

1. Kurz und Prägnant – ; Füllwörter vermeiden
2. Aktivformulierungen verwenden – ; Akteur klar benennen
3. Überprüfbare Ziele formulieren – ; Spezifisch
4. Ziele, die nicht überprüft werden können, nicht verfeinern
5. Den Mehrwert eines Ziels hervorheben
6. Das Ziel sollte begründet werden – ; Die Zielbegründung führt zur Identifikation weiterer Ziele
7. Keine Lösungsansätze geben, da sie den Lösungsraum zu früh einschränken würden

Rahmenbedingungen

Eine Rahmenbedingung – auch Restriktion genannt – legt organisatorische und/oder technische Restriktionen für das Softwaresystem und/oder den Entwicklungsprozess fest.

Organisatorische gen

- Anwendungsbereiche
- Zielgruppen
- Betriebsbedingungen
- Technische Rahmenbedingungen

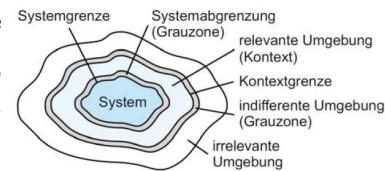
Rahmenbedingun-

Technische Produktumgebungen

- Welche SW / HW Komponenten sind auf der Zielmaschine
- Anforderungen an die Entwicklungsumgebung
- SW / HW

Kontext und Überblick

Festlegung der Einbettung in materielle und immaterielle Umgebung wie Sensoren, Personen, Übertragungsmedien, APIs, Internet zur Ermittlung der Systemgrenze (Use Case Diagramm).



Funktionale & Nichtfunktionale Anforderungen

Funktionale Anforderungen

- das System beschreibende Anforderungen
- Statik
- Dynamik
- Logik

Nichtfunktionale Anforderungen

- Qualitätsanforderungen
- Genauigkeit, Verfügbarkeit, Zuverlässigkeit
- Sicherheit vs. Benutzbarkeit
- Speichereffizienz vs. Laufzeiteffizienz

Nichtfunktionale Anforderungen betreffen mehrere oder alle funktionalen Anforderungen und können sich gegenseitig beeinflussen

Natürlichsprachliche Anforderungen

Die Verwendung ist einfach

+ flexibel

- Synonyme (z.B. City – Innenstadt) und Homonyme (z.B. Bank) führen zu einer lexikalischen Mehrdeutigkeit

Syntaktische Mehrdeutigkeit

Die letzten 10 Buchungen und die Stornierungen des Kunden werden im Fenster angezeigt"

Semantische Mehrdeutigkeit

"Jeder Sensor ist mit einem Service verbunden"

Referentielle Mehrdeutigkeit

Beim Login muss zuerst das Benutzerkennzeichen und dann das Passwort eingegeben werden.
Ist dies nicht korrekt, schlägt die Anmeldung fehl."

Vage Begriffe

Der Sensor muss neben der Tür angebracht werden"

Sprachliche Anforderungsschablone können verwendet werden.

9.3 Anforderungsschablonen

1 Einleitung (Introduction)

Gibt einen Überblick über die Anforderungsdefinition.

- 1.1 Zielsetzung (Purpose)
- 1.2 Produktziele (Scope)
- 1.3 Definitionen, Akronyme und Abkürzungen (Definitions, Acronyms and Abbreviations)
- 1.4 Referenzen (References)
- 1.5 Überblick (Overview)

2 Übersichtsbeschreibung (Overall Description)

Gibt einen Überblick über das Produkt und die allgemeinen Faktoren, die seine Konzeption beeinflussen.

- 2.1 Produkt-Umgebung (Product Perspective)
- 2.2 Produkt-Funktionen (Product Functions)
- 2.3 Benutzer-Eigenschaften (User Characteristics)
- 2.4 Restriktionen (Constraints)
- 2.5 Annahmen und Abhängigkeiten (Assumptions and Dependencies)

3 Spezifische Anforderungen (Specific Requirements)

Beschreibung aller Details, die für die Erstellung des System-Entwurfs benötigt werden. Das am besten geeignete Gliederungsschema dieses Kapitels hängt von der Anwendung und der zu spezifizierenden Software ab. Die IEEE-Richtlinie enthält dazu acht Vorschläge.

Abbildung 9.1: IEEE Schablone

3 Spezifische Anforderungen (Specific Requirements)

Beschreibung aller Details, die für die Erstellung des System-Entwurfs benötigt werden. Das am besten geeignete Gliederungsschema dieses Kapitels hängt von der Anwendung und der zu spezifizierenden Software ab. Die IEEE-Richtlinie enthält dazu acht Vorschläge.

- Externe Schnittstellen-Anforderungen (External Interface Requirements)
- Funktionale Anforderungen (Functional Requirements)
- Leistungsanforderungen (Performance Requirements)
- Entwurfsrestriktionen (Design Constraints)

- Eigenschaften des Softwaresystems (Software Systems Attributes)
 - Andere Anforderungen (Other Requirements)

9.4 Anforderungen ermitteln und spezifizieren

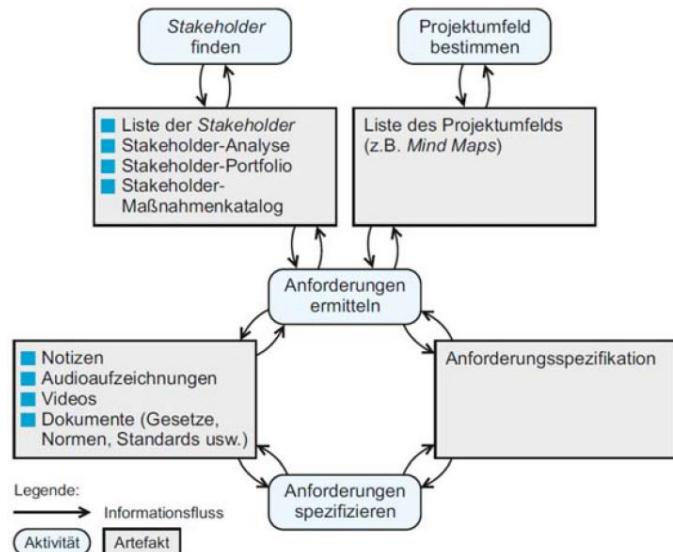


Abbildung 9.2: Prozess zur Anforderungsdefinition

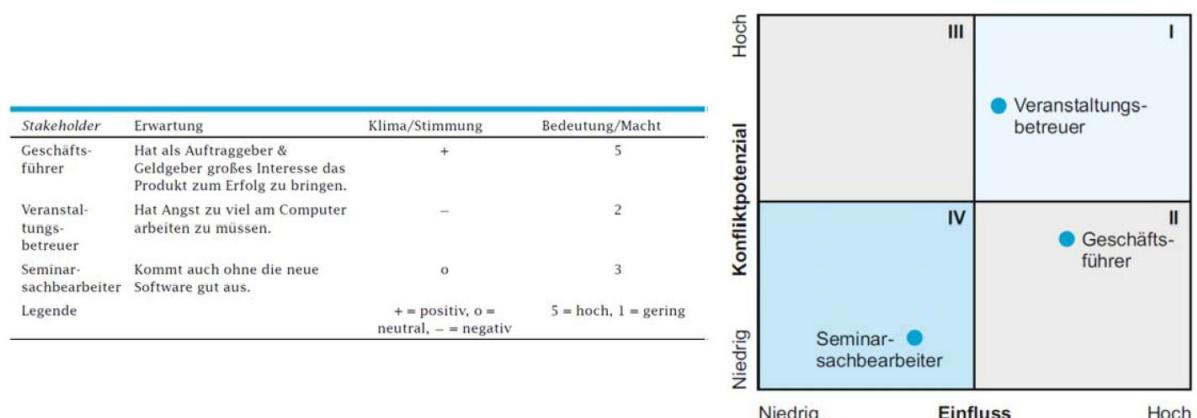


Abbildung 9.3: Stakeholderanalyse & -portfolio

9.4.1 Projektumfeld ermitteln

Es geht darum das Umfeld des Projekts, welche das Projekt wesentlich beeinflussen können zu ermitteln. Das sind meist Vorgaben, Gesetze, Standards, Normen sowie ökologische, ökonomische,

gesellschaftliche und kulturelle Einflüsse.

9.4.2 Anforderungen ermitteln

Die Anforderungen können über Stakeholderbefragungen ermittelt werden.

- Erwarten Sie keine präzise Anforderung
- Erwarten Sie viel mehr schwammige, unvollständige, widersprüchliche Anforderungen
- Überlassen sie die Federführung für die Ermittlung von Anforderungen nicht einem Stakeholder

Befragungstechniken

- Strukturiertes Interview anhand Anforderungsschablone
- Selbstaufschreibung
- Stakeholder schreiben Arbeitsabläufe und Wünsche auf
- Analyse des Altsystems, falls vorhanden

Die Ergebnisse der Anforderungsermittlung können nun systematisch ausgewertet und schrittweise in die Anforderungsschablone eingetragen werden

9.5 Anforderungen priorisieren

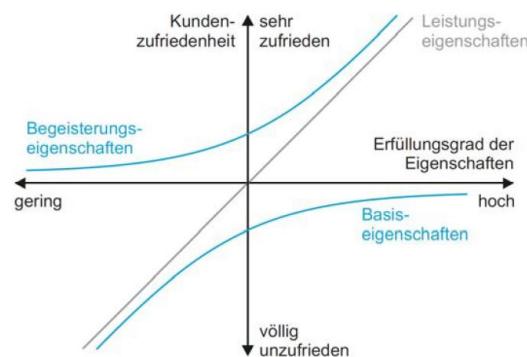
Klassifizierung und Priorisierungsmethoden der Anforderungen

- Nach Kriterium: Häufig wird Notwendigkeit als Kriterium verwendet
- Mögliche Unterteilung nach IEEE830, S.13 in essenziell (essential), Bedingt notwendig (conditional), Optional (optional)

Einige Priorisierungsmethoden

- Ad-hoc Anordnung
- Reihenfolge der Anforderungen von Stakeholder oder Gruppe
- Top-Ten-Methode
- n-Anforderungen werden anhand Kriterium in Rangfolge gebracht

9.5.1 Kano-Modell



Kano-Klassifikation

- Basiseigenschaft: vom Kunden implizit erwartet
- Leistungseigenschaft: vom Kunden explizit gefordert
- Begeisterungseigenschaft: vom Kunden nicht erwartet

Das Kano-Modell kann auch für Anforderungen verwendet werden.

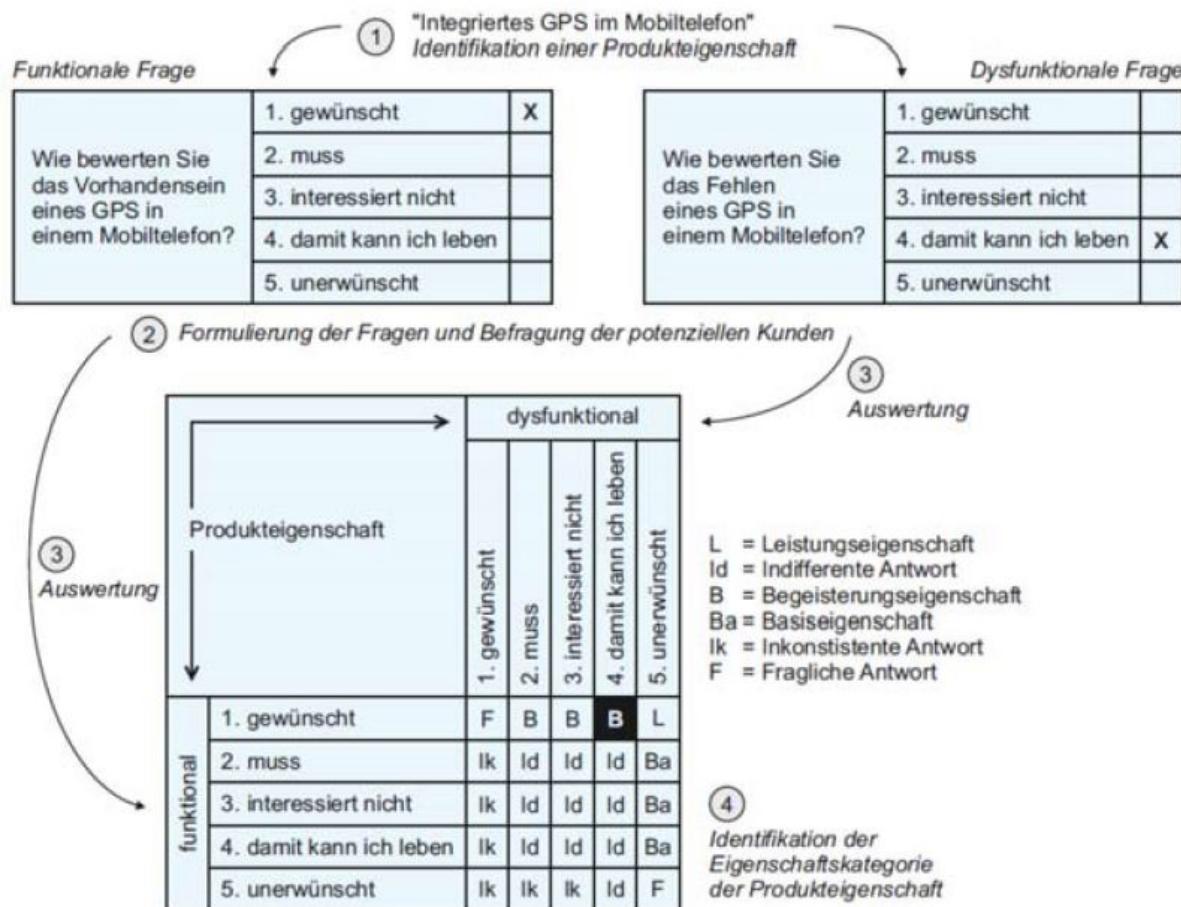


Abbildung 9.4: Prozess zur Anforderungsdefinition

10 Versionskontrolle Einführung - Git

10.1 Git

- Git = (engl.) Blödmann
- freie Software zur verteilten Versionsverwaltung von Dateien
- entwickelt für Quellcode-Verwaltung des Linus-Kernels
- Vokale Systeme: "Local Version Control Systems" (LVCS) - SVN - Subversion
- Verteilte Systeme: "Distributed Version Control Systems"
Mehrere verteilte Repositories für die zu archivierenden Dateien. Bsp. Git

Ziele von Git

- Geschwindigkeit
- Einfaches Design
- Vollständig verteilt (distributed Repository)
- Gute Unterstützung von nicht-linearer Entwicklung
- ..

10.1.1 Sicherungsart im Vergleich

Git

- stream of snapshots
- Speichert Snapshot von aktuellen Filesystem
- Link falls Datei nicht geändert (Pointer)

- Erst dann werden Deltas verwendet

SVN

- Speichert die Datei beim ersten Mal und danach nur noch alle Änderungen
- Wird ein File aufgerufen, dann muss es erst das File erstellen und alle Änderungen durchführen

Bilder einfügen!

10.1.2 Status von Dateien

committed: Daten sicher gespeichert im lokalen Repo

modified: Datei ist geändert zu (letztem Stand) aber nicht committed

staged: Modifizierte Files ist markiert mit aktueller Version um im nächsten Commit-Snapshot zu gehen

-

10.1.3 Bereiche

- Working Directory:
- Staging Area/ Index:
- .git directory (lokal):

10.1.4 Git Workflow

Erstellen eines Ordners **-mkdir gitTest**

Erstellen eines Git Repositories (.git Ordner): **git init**

Erstellung eines lokalen Repositories erfolgt.

Tracked vs. Untracked: Dateien die Git bekannt sind. Die unter Versionsverwaltung stehen.
Add the file Edit the file Stage the file Commit the file Remove the file

Übersicht über die aktuelle Lage im Repository: **git status**

Funktionalität:

Registerzustand aller Files, Aktueller lokaler Branch, Abweichungen vom entfernten Branch.

Hinzufügen eines Files, welches bereits in den entsprechenden Ordner gespeichert wurde: **git add *file name***

Überprüfung der Aktion: **git status**

File dem Repository hinzufügen: **git commit -m "Comment"**

Fragt den Namen und die Email nach: **git config - global user**

Differenz zwischen Staging Area und Working Directory feststellen: **git diff *file name***

Wiederum das File ätagen": **git add *file name***

File dem Working Directory hinzufügen: **git commit -m "Comment"** File aus der Versionsverwaltung raus nehmen: **git rm *file name***

Histories anzeigen: **git log**

GUI:

- **git gui**
- **gitk**: "native gui" von git
(Alle Branches werden angezeigt mit –all)

Commit Baum Darstellung: Jeder Commit hat einen eigenen Hash-Wert und es gibt dazu jeweils einen Snapshot. Branches sind Zeiger, die auf bestimmte Commits zeigen. Der Masterbranch zeigt auf den aktuellen Commit.

Frühere Versionen zurückholen Aus dem lokalen Repo wird etwas wieder in die Staging Area zurückgeholt: **git reset *path***

Datei(en) ünstagen", indem die Dateiversion vom letzten Commit verwendet wird.

Working Directory mit alter Version überschreiben: **git checkout HEAD *path***

10.1.5 **git clone *URL***

`git clone https://git.hsr.ch/git/reponame` Beinhaltet folgende Schritte:

1. Erstellt neues Verzeichnis
2. Wechselt in das neu erstellte Verzeichnis
3. Führt `git init` aus um lokales Repo zu erstellen
4. Führt `git remote add origin URL` aus
5. Führt `git fetch` aus
6. Führt `git checkout -b branch origin/branch` (HEAD vom remote Repo)

Von einem entfernten Repo wird ein lokales Repo erstellt.

10.1.6 git remote add *|kurzname|* *|URL|*

Verbindung zu einem entfernten Repo erstellen. Bsp.: git remote add origin https://git.hsr.ch/git/gitTest
per default wird origin als Kurznamen für ein remote Repo verwendet
Normalerweise verwendet man git clone und nicht git remote.

10.1.7 git fetch *|remotename|*

Aktualisiert lokales Repo vom Remote Repo. Bsp.: git fetch origin.

Die Commits, die wir schon hatten bleiben, werden jedoch als "Branch" der aktualisierten Version des Remote Repo, welche wir ins lokale Repo geladen haben, angezeigt. Nun hat man jedoch 2 Zweige, welche wieder vereint werden sollen.

10.1.8 git pull

Aktualisiert lokales Repo vom entfernten Repo und merged Änderungen zum lokalen Repo. Ist wie git fetch, jedoch werden die 2 Zweige ineinander verwoben "merge". // git pull origin master entspricht git fetch origin & git merge origin/branchname Merge-Konflikte: Identische Codezeilen wurden geändert - Manuelle Überprüfung der 2 Versionen und Entscheid für eine Version. Es ist wichtig, dass man modified files zuerst committed bevor man einen pull macht.

10.1.9 git push

Veröffentlicht lokales Repo und der master-Branch wird aktualisiert. git push *|remotename|* *|branchname|*

10.2 Tagging

10.2.1 git tag

Tags sind da, um spezifische Punkte in der History zu taggen.

- Lightweight Tag
git tag -a v2.3
- Annotated Tags (Kommentierter Tag)
git tag -a v1.3 -m "myComment"

10.3 Branching

Ein Branch ist ein verschiebbarer Zeiger. Es gibt HEAD-Zeiger und Tags. Der Master Branch wird per Default erstellt.

10.3.1 git branch `|branchname|`

Erstellen eines neuen Branch. Jedoch wird nicht auf diesen Branch gewechselt (Head bleibt stehen), git checkout `|branchname|`; Head-Zeiger wird auf neuen Branch verschoben. Nun erfolgt ein commit auf einem Branch.

git checkout master Der Head-Zeiger ist wieder auf dem master-branch. git commit - nun wird wieder etwas auf dem Master-branch committed. Es resultiert eine Verzweigung.

10.3.2 git merge

Zusammenführen von zwei Branches. Es gibt Fast-Forward merge: git merge `|branchname|`. Der master-branch-Zeiger wird entlang der Kette verschoben. Es resultiert kein Merge-Problem. Three-way Merge: Merging-Konflikte sind möglich. Git entscheidet selbst ob ein FF-merge oder ein three-way merge erfolgen soll.

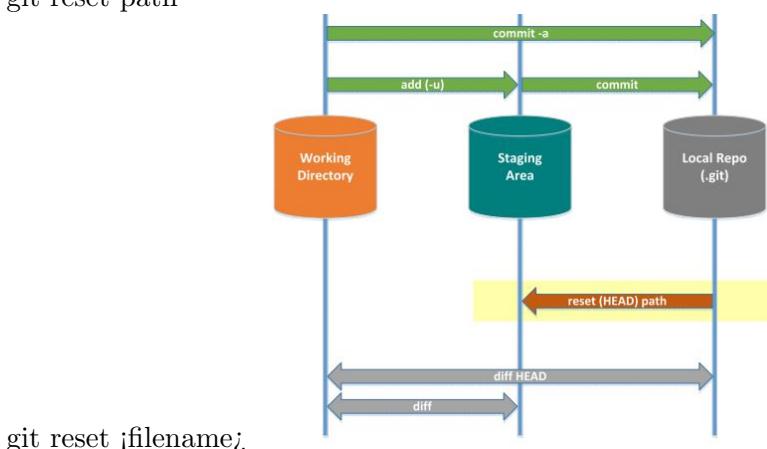
Mergekonflikte:

Mit GUI: KDiff3 mittels **mergetool**

Eine von beiden Dateien komplett verwenden: **git checkout [-theirs/-ours]**

10.3.3 git reset

Datei(en) «unstaged», indem die Dateiversion vom letzten commit verwendet wird
git reset path



10.3.4 git checkout

Working Directory mit alter Version überschreiben: git checkout HEAD path

File(s) in Staging Area wird von local Repo überschrieben: git checkout *{filename}*

File(s) in Staging Area und Working Directory wird von local Repo überschrieben: git checkout HEAD README

Achtung: Kann Daten zerstören

10.3.5 gitignore

Hinzufügen von Files mit bestimmten Endungen zu gitignore:

```
echo *.aux >> .gitignore
```

Files verwerfen, die bereits in der Staged Area sind:

```
git rm --cached {filename}
```

11 Code Dokumentation mit Doxygen

11.1 Dokumentation & Code-Dokumentation

11.1.1 Zweck der Dokumentation

- Wissenssicherung:
Wissen über ein System macht einen beträchtlichen Wert des Systems aus – Menschen vergessen schnell
- Förderung der Kommunikation & Genauigkeit
- Sichtbarmachen des Projektfortschrittes:
Fertigstellung und Freigabe von Dokumenten markiert nachprüfbar(!) den Projektfortschritt

Bei der Programm-Dokumentation werden bestehende Programmteile (v.a. Schnittstellenbeschreibungen (APIs)) verwendet. Der Entwickler / Programmierer dokumentiert, wenn das bestehende Programm verändert wird.

Die Dokumentation im Code hat den Vorteil, dass Dokumentationen bei Änderungen häufig nicht nachgeführt wurden, da Code und Dokumentation getrennt waren. Dies führte zu Inkonsistenzen sowie fehlerhaften und unvollständigen Dokumentationen.

Code-Dokumentationen integrieren Beschreibungen in den Source-Code mit Hilfe von speziellen Kommentaren. Die Dokumentation wird erzeugt mit Hilfe von Dokumentationswerkzeugen, die den Sourcecode analysieren und die Beschreibung daraus extrahieren.

11.1.2 Vorteile von Code-Dokumentation

- Erzeugung der Dokumentation kann leicht automatisiert werden. –*i* Dokumentation bleibt aktuell
- Gewisse Informationen, wie z.B. Funktionsköpfe, werden direkt aus dem Programmcode entnommen.

- Dadurch, dass Beschreibungen und Source-Code unmittelbar nebeneinander im Source-Code-Dokument vorkommen, ist die Wahrscheinlichkeit gross, dass Programmcode und Beschreibung übereinstimmen. → Dokumentation bleibt konsistent
- Vor allem geeignet zur API-Dokumentation von Klassen bei der Objektorientierten Programmierung.
- Hinweis auf gewisse nicht dokumentierte Codestellen (z.B. Parameter) leichter. → Dokumentation ist vollständig

11.2 Doxygen

- The KDE API Reference (Linux Desktop): <http://api.kde.org/>
- D-Bus documentation (SW Message Bus System): <http://freedesktop.org/software/dbus/doc/api/html/index.html>
- The Xerces-C++ Documentation (XML-Parser): <http://xerces.apache.org/xerces-c/apiDocs-3/classes.html>

Weit verbreitetes Dokumentationswerkzeug:

- Open Source, für Windows, Unix / Linux, Mac, ...
- unterstützt: C, C++, Java, C#, etc.

Erzeugt Dokumente in folgenden Formaten:

- HTML (für Browser) – am häufigsten
 - LaTeX (dvi, pdf mit Links, PostScript ...)
 - Unix man pages
 - RTF (MS-Word)
 - XML (für maschinelle Weiterverarbeitung)
- Erstellt primär API-Beschreibungen, aber (mit Zusätzen) auch Klassendiagramme und andere Diagramme.

Weitere Infos, Website: <http://www.stack.nl/~dimitri/doxygen/>

Doxygen ist ein "Command-Line-Tool".

GUI-Frontend "doxywizard" sowie Plugins z.B. für Eclipse («Eclox»)

Hauptbefehl (command-line): \$ doxygen

Doxygen benötigt eine Konfigurationsdatei: "Doxyfile"

Aktuelle Anzahl möglicher Konfigurationen: 267

Aufruf-Beispiele:

\$ doxygen -g // erzeugt Konfigurationsdatei "Doxyfile"

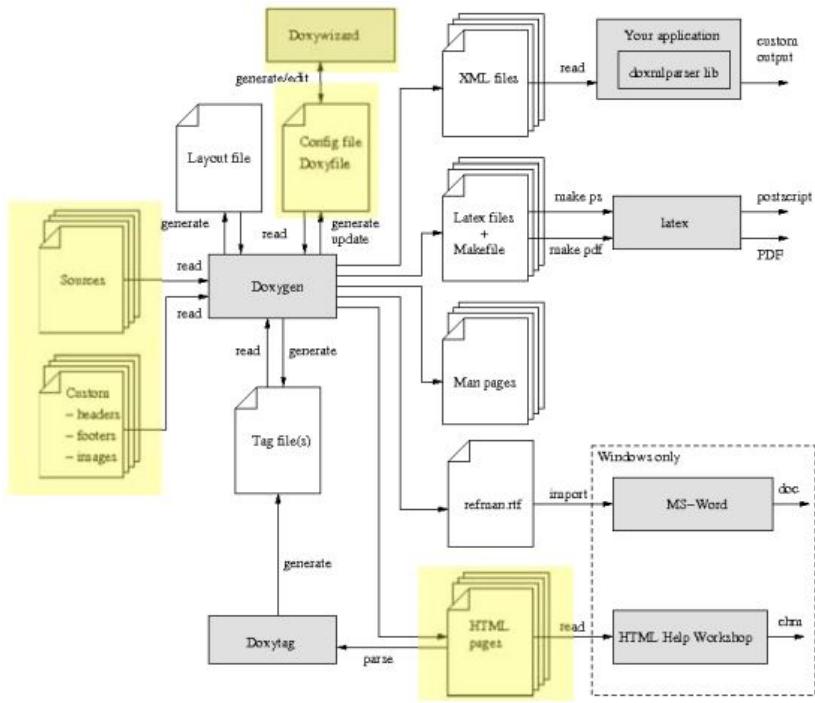


Abbildung 11.1: Doxygen Architektur

\$ doxygen Doxyfile // erzeugt Dokumentation, falls Konfig'datei Doxyfile heisst, reicht:\$ doxygen

11.2.1 Wichtigste Einstellungen

- Projektname: PROJECT_NAME = "Voltage Divider"
- Input: INPUT = . ./src ./tests README.md
- Datei-Filter: FILE_PATTERNS = *.cpp, *.h
- Output: OUTPUT_DIRECTORY = doc
- README.md als mainpage: USE_MDFILE_AS_MAINPAGE = README.md

11.2.2 Wichtigste Befehle für Doxygen

Allgemein

- Dateikopf
- file
- date

- author

Funktionsbeschreibung

- brief
- param
- pre
- post

Beispiel:

```
/// \brief Calculates the optimal values for R1 and R2. The base formula
/// behind this calculation is \math{u1 = u2 \frac{r2}{r2+r1} nf}.
/// \pre \p u1 < \p u2
/// \pre \p lowerRTh & \p upperRTh
/// \param u1 Value for U1 in V
/// \param u2 Value for U2 in V
/// \param lowerRTh set the lower bound for resistor value output
/// \param upperRTh set the higher bound for resistor value output
/// \param resDecade gives the resistor e-decade on which the resistor are calculated virtual
void calc(double u1, double u2, double lowerRTh, double upperRTh, const EDecade& resDecade);
```

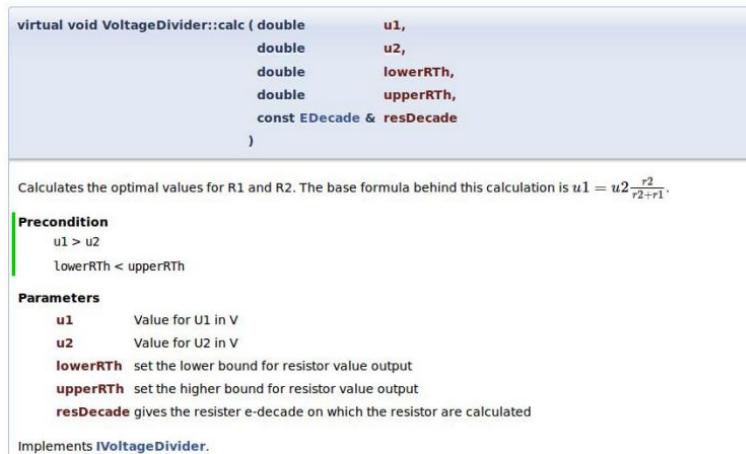


Abbildung 11.2: Ausschnitt aus dem HTML-Output

Die Doxygen Hauptseite wird als erstes in der HTML Dokumentation angezeigt.

Zweck: Sie soll einen Überblick über das gesamte Projekt vermitteln

Formatierungsanweisung: @mainpage oder markdown file (z.B. README.md, empfohlen)

11.2.3 Weitere Informationen zu Doxygen

Ein Markdown-File (typischerweise README.md genannt) enthält eine Plain Text Formatierungssyntax. Das Ziel ist dabei es so leserlich wie möglich zu machen. Es soll als plain Text publiziert werden wie es ist, ohne visueller Überladung durch zusätzliche Syntax.

Beispiel: # This is an H1 Header # oder ## This is an H2 Header ##

Es kann mit folgendem Befehl verwendet werden: USE_MDFILE_AS_MAINPAGE = README.md

INPUT = ./tests ./src README.md

Die erste Zeile wird im Titelbalken dargestellt. Mit einem Punkt wird der Projektname verwendet. Der Nachteil bei dieser Lösung ist eine Warnung beim Übersetzen

Weitere Syntax: <http://daringfireball.net/projects/markdown/syntax>

Code Documentation Style

Java / C	C	C++	C++
<pre>/** Brief description which ends at this dot. Details follow * here. */</pre>	<pre>/*! ... text ... */</pre>	<pre>/// ... text ... ///</pre>	<pre>//! ... text ... //!</pre>

\\" sind hier wichtig. Sie schliessen einen normalen Kommentarblock und öffnen einen C-Style Kommentarblock.

Weitere Befehle für Doxygen

Jeder Befehl beginnt mit @ oder \ wird häufig für C/C++ und @cmd oft für Java genutzt. LaTeX Formeln sind kompatibel für Outputs in LaTeX und HTML.

Dabei muss USE_MATHJAX eingeschaltet sein.

Beispiel: \f\$formel\f\$

Bilder sind kompatibel für Outputs in LaTeX, HTML, Docbook und Rtf.

Dabei muss IMAGE_PATH auf den Pfad mit den Bildern zeigen.

Beispiel: \image format file "caption"

\image html Kontextdiagramm.gif "Kontext-Diagramm"

\image latex Kontextdiagramm.eps "Kontext-Diagramm" width=10cm

12 Testing & Unit Testing

Definition: Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.(G. J. Myers, The Art of Software Testing, 1979)

”To test a program is to make it fail.” – Bertrand Meyer, 2008

Jeder durch Testen gefundene Fehler ist ein Erfolg. Nach sorgfältigem Testen eines Programms steigt die Wahrscheinlichkeit, dass das Programm sich auch in nicht getesteten Fällen wunschgemäß verhält. Durch das Testen kann die Korrektheit eines Programms aber nicht bewiesen werden.

Je mehr Fehler gefunden werden, umso höher ist die Wahrscheinlichkeit für weitere Fehler.



12.0.1 Testen versus Debugging

Ziel von Testen:

Aufzeigen, beweisen, dass Fehler (Bugs) existieren –; Destruktives Testen
”Jeder gefundene Bug ist ein Gold-Nugget!”

Ziel von Debugging:

Die durch Testen gefundenen Bugs beseitigen

12.0.2 Laufversuch versus systematischer Test

Laufversuch

- Entwickler testet Code während Implementation fortlaufend

- Kann mittels Debugging festgestellt werden
- Fehler werden sofort korrigiert

Ist wichtig um zu überprüfen, ob das Programm das tut was man will

Systematischer Test

- Systematische Fehlersuche
- Test werden geplant
- SOLL wird mit IST verglichen
- Resultate werden festgehalten
- Wenn möglich nicht durch Entwickler der Software

Ist wichtig um die Richtigkeit einer Software jederzeit reproduzieren und nachvollziehen zu können

12.0.3 Testarten

Abnahmetest (Anforderungen; Kunde): Validierung, Besondere Test-Form, Soll nicht Fehler aufzeigen, sondern zeigen, dass das System nach Anforderungen (gemäss Pflichtenheft) fehlerfrei funktioniert

Systemtest (Architektur; Tester): Test des gesamten Systems (z.B. inkl. Mechanik)

Integrationstest (Entwurf; Entwickler/Tester): Integration von Programmeinheiten

Modultest (Detailentwurf/Implementation; Entwickler): Unit-Test

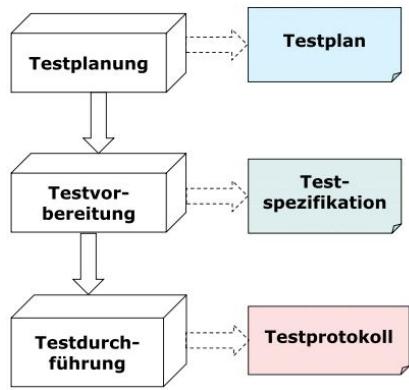
Die Tests sollten entsprechend den verschiedenen Phasen gemacht werden (Phase und Aktor in Klammern geschrieben).

12.0.4 Verifikation und Validierung

Validierung (Validation) „Are we building the right product?“ Überprüft das Software-Produkt, ob es die Anforderungen des Auftraggebers erfüllt.

Verifikation (Verification): „Are we building the product right?“ Überprüft die Artifacts während der Entwicklung, ob sie ihre Vorgaben erfüllen (Artifacts = künstliche, d.h. von Menschenhand geschaffene Dinge wie Dokumente etc.)

12.0.5 Prinzipieller Ablauf eines Tests



Vorzüge von einem Testablauf:

- Reproduzierbar
- Wissen, was getestet wurde
- Unabhängig von testender Person
- Wenn möglich automatisiert
- Testspezifikation fortlaufend erweitern

12.0.6 Testspezifikation / Testprotokoll

Redundanz vermeiden; Möglichst in Code integrieren oder/und automatisieren

Elemente:

Test-Spezifikation (Test-Code)

Test-Protokoll: Wird typischerweise von Test-Frameworks erstellt

Test-Dokumentation (z.B. mit Hilfe von Doxygen)

12.0.7 Auswahl der Testfälle

Mit den Testfällen die Anforderungen überprüfen Ziel: Mit möglichst wenig Testfällen möglichst viele Fehler finden

Testmethoden:

- Black-Box Testing
- White-Box / Glass-Box Testing

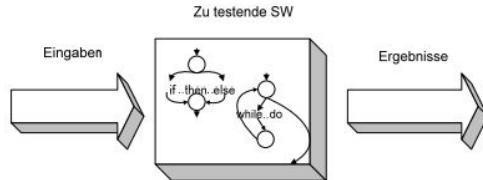
Black-Box Tests

- Testfälle mit Kenntnis der Funktion
- Programmstruktur kann unbekannt sein



White-Box-Tests

- Testfälle mit Kenntnis der inneren Struktur



- Einige Methoden:

- Äquivalenzklassen
- Grenzwertanalyse

- Einige Methoden:

- Programmablauf-Test
 - Anweisungsüberdeckung
 - Zweigüberdeckung
 - Pfadüberdeckung

12.1 BlackBox Testing

12.1.1 Äquivalenzklassenanalyse

Äquivalenzklasse = Wertebereich einer Eingabegröße, für welche der Prüfling voraussichtlich das gleiche Verhalten zeigt. Ist Korrektheit der Eingabegrößen nicht gesichert auch Äquivalenzklassen für ungültige Wertebereiche.

ein Testfall pro Äquivalenzklasse

Wert für Testfall kann zufällig sein

12.1.2 Grenzwertanalyse

Erfahrung: Fehler sehr oft an Grenzen zulässiger Eingabewertebereiche Grenzwertanalyse wählt Testfälle an den Grenzen Werte auf Grenze, knapp darunter und darüber. Kann Äquivalenzklassenmethode erweitern

12.1.3 Vergleich

- für eine Einkommen von 80'000.- bis 110'000.- sei der Steuersatz 25%, darunter 20%, darüber 30%.

- Testfälle Äquivalenzklassen

Einkommen	Steuersatz
< 80'000.-	20 %
= 80'000.- und < 110'000.-	25 %
> 110'000.-	30 %

- Testfälle Grenzwertanalyse

Einkommen	Steuersatz
79'999.-	20 %
80'000.-	25 %
80'001.-	25 %
109'999.-	25 %
110'000.-	30 %
110'001.-	30 %

- Weitere mögliche Klassen

- 0 <
- Grenzen des Datentyps

12.2 White Box / Glass Box Testing

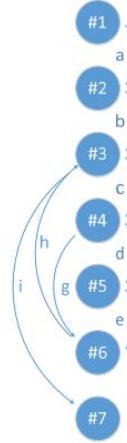
Kenntnis der Kontrollstrukturen als Basis für Testfälle (Kontrollflussgraph). Jedes Statement als Knoten bzw. sequentielle zu einem Knoten zusammenfassen. Kanten verbinden Knoten entsprechend Verzweigungen und Schleifen.

Code

```
int main()
{
    int vec[] = { 10, 5, -12, 4};
    unsigned int vecLen = sizeof(vec)/sizeof(int);
    unsigned int peakIdx = findPeak(vec, vecLen);
}

unsigned int findPeak(int vec[], unsigned int len)
{
    int peakIdx = 0;                      // #1
    unsigned int i=1;                     // #2
    while (i<len)                       // #3
    {
        if (vec[i] > vec[peakIdx])       // #4
            peakIdx = i;                // #5
        ++i;                           // #6
    }
    return peakIdx;                      // #7
}
```

Graph



12.2.1 Anweisungsüberdeckung (Statement Coverage)

Prozentualer Anteil der Anweisungen, die in Test ausgeführt werden 100% Anweisungsüberdeckung ist Minimum Alle Knoten werden genau einmal durchlaufen.

Testfälle: 1. int vec[] = 1, 10;

12.2.2 Zweigüberdeckung

Zweigüberdeckung (branch coverage) Prozentualer Anteil der Kanten (Zweige), die in Test durchlaufen werden Eine 100% Zweigüberdeckung enthält auch eine 100% Anweisungsüberdeckung Zusätzlich werden auch alle "leeren Zweige" durchlaufen

Testfälle: 1. int vec[] = 1, 10, 2; Ablauf: abcde - zurück auf mit i 3-4g - zurück auf 3 i

12.2.3 Pfadüberdeckung (Path Coverage)

Prozentualer Anteil der Pfade, die in Test durchlaufen werden Ein Pfad ist ein möglicher Weg durch den Kontrollgraph 100% Pfadüberdeckung kaum zu erreichen Die findPeak Funktion kann mit unendlichem Speicher unendlich viele Pfade besitzen

Testfälle: 1. int vec[] = 1, 10;

a, b, c, d, e, h, i

2. int vec[] = 1, 10, 2;

a, b, c, d, e, h, c, g, h, i

3. int vec[] = 1, 10, 2, ... ;

12.2.4 Güte

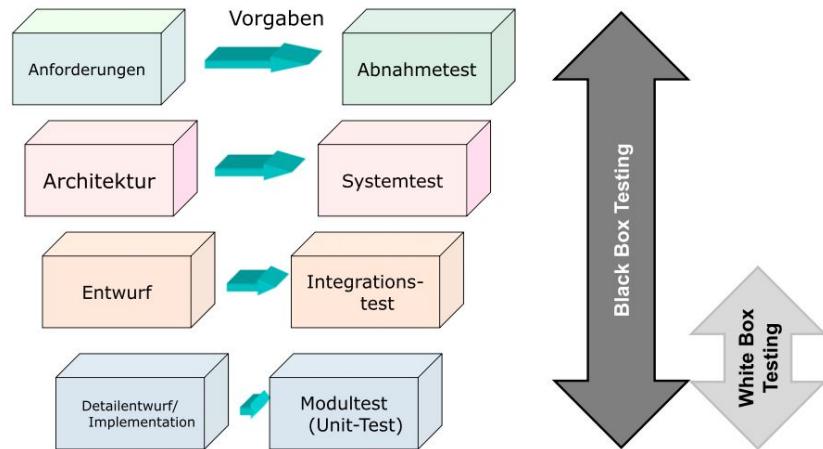
Die Testgüte hängt von gewählter Überdeckung und erreichtem Überdeckungsgrad ab Überdeckungsgrad – Prozentuales Verhältnis der Anzahl überdeckter Elemente zur Anzahl vorhandener Elemente
Beispiel Anwendungüberdeckung int vec[] = 1 ;

12.2.5 Greybox-Testing: Pragmatisches Testen

1. Zuerst Black-Box Test
2. Überprüfung der Anweisungsüberdeckung / Coverage
3. Ergänzen mittels White-Box Test
4. Falls gewünschte Coverage nicht erreicht, zum zweiten Schritt springen

Die Kombination von Black- und Whitebox-Testing wird auch Greybox-Testing genannt und wird in der Praxis häufig angewendet.

12.2.6 Wann wird welche Testmethode verwendet



12.3 Teststrukturen

12.3.1 Testgeschirr

Zum Testen unvollständiger Software wird ein Testgeschirr (testharness) benötigt - wird für das isolierte Testen von Units und Integrationstests benutzt

- unabhängig von der Vollständigkeit der Software

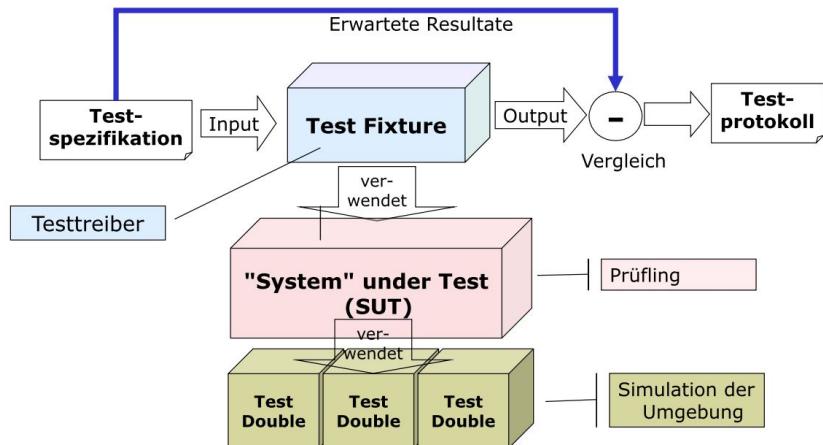
Ein Testgeschirr besteht aus:

- Test-Driver: Ruft den Prüfling auf und Versorgt den Prüfling mit Daten und Nimmt Resultate entgegen und protokolliert sie
- Test-Double: berechnet oder simuliert die Ergebnisse einer vom Prüfling aufgerufenen Operation. Kann Mock, Stub, Dummy oder Spy unterteilt werden, haben jedoch keine eindeutige Definitionen. Wir verwenden im allgemeinen den Begriff Mock

Testarten

Unit-/Komponenten-Tests bezeichnen das isolierte Testen von Programmeinheiten (Unit = Modul = Methode, Klasse, Komponente) oder mehreren Klassen. Sie sollten gut wiederholbar sein. Die gesamte Umgebung einer Komponente muss durch Treiber und Stümpfe simuliert werden.

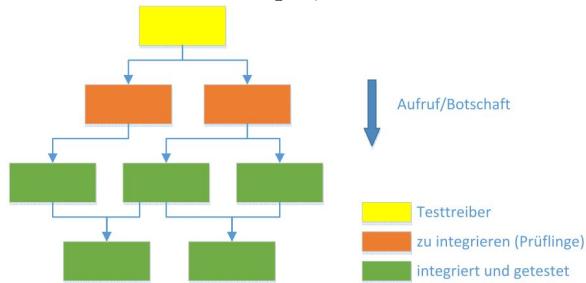
Integrationstest wird angewendet, wenn ein System schrittweise zusammengebaut wird. Dabei wird die Funktionalität der Baugruppen durch Tests überprüft. Dabei werden die noch



nicht integrierten Teile durch Treiber und Stümpfe simuliert.

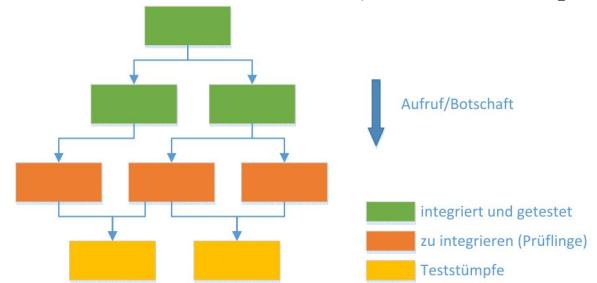
Aufwärtsintegration (bottom-up)

- beginnt mit elementaren Komponenten
- braucht keine Stümpfe, dafür Treiber



Abwärtsintegration (top-down):

- beginnt mit einem "hohlen" Gesamtsystem
- braucht keine Treiber, dafür Stümpfe



Mischformen sind ebenfalls möglich.

```
class A % Buch-Klasse
{
public:
A()
{ b;
}
private:
B b; }
```

```
class B % Author-Klasse
{public:
private:
}
```

```
main
{ A a;
}
```

12.3.2 Beispiel a

```
void authorTestCtors()
{
    Author* author = new Author("Samuel", "Beckett");
    assert(author->getFirstName() == "Samuel");
    assert(author->getLastName() == "Beckett");
    std::cout << "AuthorTest Ctors successful" << std::endl;
    delete author;
}

//Tests nicht abschliessend

int main( int argc, char* argv[] )
{
    authorTestCtors();
    authorTestEmptyAuthorException();
    std::cout << "ALL TESTS SUCCESSFUL" << std::endl;
    return 0;
}
```

Unit- oder
Integrationstest?
Wo ist der Testmock?

```
class A % Buch-Klasse
public:
A(B &b): b(b)
% Jedes Buch A hat einen Autor B
private:
B &b;
```

```
class B % Author-Klasse
{public:
}
```

```
main
{ B b;
A a(b); }
```

```
class Author
{
private:
    std::string firstName;
    std::string lastName;
public:
    Author(const std::string& firstname,
           const std::string& lastname);
    std::string getFirstName() const
    {
        return firstName;
    }
    std::string getLastname() const
    {
        return lastName;
    }
    bool operator==(const Author& rhs) const
    {
        return firstName == rhs.firstName && lastName == rhs.lastName;
    };
}

Author::Author(const std::string& firstname,
              const std::string& lastname)
: firstName(firstname), lastName(lastname)
{
    if (firstname.empty() || lastName.empty())
        throw std::runtime_error("Invalid Author");
}
```

- Lösung:

Unitest (man testet nur diese Klasse);

Testmock: Soweit nicht ersichtlich

12.3.3 Beispiel b

```

void bookTestCtors()
{
    Author myAuthor("Samuel", "Beckett");
    Book mybook("Endspiel", myAuthor);
    assert(mybook.getTitle() == "Endspiel");
    assert(&mybook.getAuthor() == &myAuthor);
    std::cout << "BookTest Ctors successful" << std::endl;
}

... //Tests nicht abschliessend

int main( int argc, char* argv[] )
{
    bookTestCtors();
    ... //Weitere Test-Funktionsaufrufe

    std::cout << "ALL TESTS SUCCESSFUL" << std::endl;
    return 0;
}

```

■ Unit- oder Integrationstest?

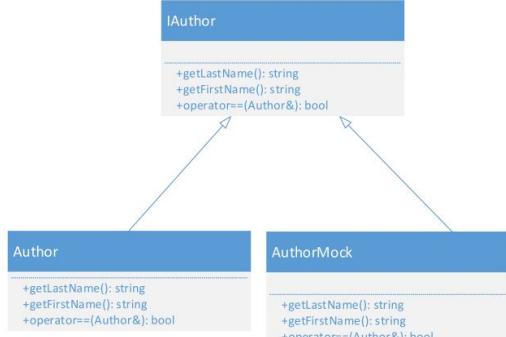


- \downarrow Lösung:

Integrationstest: Buch und Autor werden miteinander getestet und es wird die Beziehung zwischen Buch und Autor geprüft.

12.3.4 Beispiel c: Dependency Injection

Pure Virtual Klassen müssen von den Unterklassen überschrieben werden



```

class Book
{
private:
    Author& author;
    std::string title;
public:
    Book(std::string const &title, const Author& author);
    std::string getTitle() const
    {
        return title;
    }
    const Author& getAuthor() const
    {
        return author;
    }
    bool operator==(const Book& rhs) const
    {
        return author == rhs.author && title == rhs.title;
    }
};
Book::Book(std::string const &title, const Author& author)
: author(author), title(title)
{
}

```

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

```

void bookTestCtors()
{
    AuthorMock myAuthorMock;
    Book mybook("Endspiel", myAuthorMock);
    assert(mybook.getTitle() == "Endspiel");
    assert(&mybook.getAuthor() == &myAuthorMock);
    std::cout << "BookTest Ctors successful" << std::endl;
}

// Tests nicht abschliessend

int main( int argc, char* argv[] )
{
    bookTestCtors();
    // weitere Test-Funktionsaufrufe

    std::cout << "ALL TESTS SUCCESSFUL" << std::endl;
    return 0;
}

```

■ Unit- oder Integrationstest?

■ Wo ist der Testmock?

```

class Book
{
public:
    Book(std::string const &title, const IAuthor& author);
    std::string getTitle() const
    {
        return title;
    }
    const IAuthor& getAuthor() const
    {
        return author;
    }
    bool operator==(const Book& rhs) const
    {
        return author == rhs.author && title == rhs.title;
    }
private:
    IAuthor& author;
    std::string title;
};

Book::Book(std::string const &title, const IAuthor& author)
: author(author), title(title)
{
}

```

Nun kann

die richtige Klasse oder die Testklasse überprüft werden.

Will man Buch testen, kann der Testmock verwendet werden – i Unittest von Buch

Die Abhängigkeit zwischen Buch und Autor wurde gelöst. Falls nun Autor verändert wird, so kann mit dem "fake Autor" Testmock Author gearbeitet werden.

Testmock: AuthorMock (Testklasse für Author)

12.3.5 Wann ist genug getestet?

Wenn mit den in der Testvorschrift festgelegten Testdatensätzen keine Fehler mehr gefunden werden - Sinnvolles Kriterium, wenn der Umfang des Prüflings eine systematische Auswahl von Testfällen mit ausreichender Überdeckung ermöglicht - Übliches Kriterium bei der Abnahme - Wenn die Prüfkosten pro entdecktem Fehler eine im Voraus festgelegte Grenze überschreiten - Sinnvolles Kriterium für das Beenden des Systemtests - Setzt die Erfassung der Prüfkosten und der Anzahl gefundener Fehler voraus

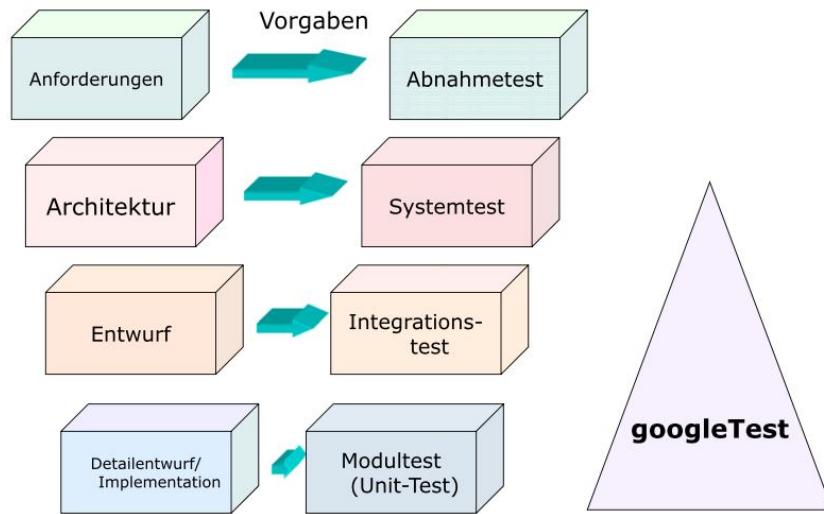
12.4 Automatisierte Tests

Vorteile:

- Wiederholbarkeit
- Absicherung bei Änderung, Portierung, Erweiterung
- geringe/keine Kosten bei Wiederholung
- Eindeutige Spezifikation
- Testcode ist Programmcode und damit eindeutig

Nachteile:

- Mehr Code zu schreiben und zu pflegen
- Testcode ist Programmcode
- Werden wirklich die richtigen Anforderungen getestet?
- Was muss überhaupt getestet werden?



12.4.1 Unit Test Frameworks

Um die automatisierten Test zu vereinfachen, gibt es verschiedene Unit Test Frameworks.
Bekannte C++ Vertreter sind: googleTest, Boost Test, Qt Test und CppUnit

Vorteile von Unit-Testframeworks

- Testfunktionen werden automatisch aufgerufen
- Übersichtliche Organisation der Testfälle
- Aufteilung der Tests auf mehrere Dateien in der Regel
- Zum Beispiel eine Testklasse pro zu testende Klasse.
- Innerhalb der Datei: Organisation der Testfälle in Form von Testfunktionen und Testsuiten
- Unterstützung bei Test Doubles
- Testprotokolle: Ausgabe der Testergebnisse in verschiedenen Formen möglich
- Test result formatter (Outputter")

googleTest Beispiel

■ main.cpp

```
#include <gtest/gtest.h>

int main(int argc, char *argv[])
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

■ MyDateTest.cpp

```
#include <gtest/gtest.h>
#include "../src/MyDate.h"
...
TEST(MyDateTest, testDefaultCtor)
{
    MyDate d1;
    ASSERT_EQ(d1.getDay(), 0);
    ASSERT_EQ(d1.getMonth(), 0);
    ASSERT_EQ(d1.getYear(), 0);
}
```

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(val1,val2);	EXPECT_EQ(val1,val2);	val1 == val2
ASSERT_NE(val1,val2);	EXPECT_NE(val1,val2);	val1 != val2
ASSERT_LT(val1,val2);	EXPECT_LT(val1,val2);	val1 < val2
ASSERT_LE(val1,val2);	EXPECT_LE(val1,val2);	val1 <= val2
ASSERT_GT(val1,val2);	EXPECT_GT(val1,val2);	val1 > val2
ASSERT_GE(val1,val2);	EXPECT_GE(val1,val2);	val1 >= val2

googleTest Assertions

- googleTest nutzt selbst definierte Assertions
- Beispiele in der Tabelle rechts

Weitere Test-Makros können hier gefunden werden: <https://github.com/google/googletest/blob/master/googletest/docs/Primer.md>

Konventionen

- Empfohlene Verzeichnis-Unterteilung - src Ordner: MyDate.h, MyDate.cpp, main.cpp
- tests Ordner: UnitTest main.cpp, MyDateTest.cpp
- Für jede zu testende Klasse wird eine entsprechende Testklasse erstellt. **Namensgebung** - Der Name einer Testklasse beginnt mit dem Namen der zu testenden Klasse und endet mit "Test". Beispiel: Applikations-Klasse SStack-; Testklasse SStackTest".
- Der Name einer Testfunktion beginnt mit "test": Beispiel: "testAddition()".

12.4.2 GCOV Beispiel

```
$ g++ -o FindPeak FindPeak.cpp -fprofile-arcs -ftest-coverage
$ ./FindPeak
$ lcov --capture --directory . --output-file cov.info
$ genhtml --demangle-cpp cov.info -o html
$ firefox html/index.html &
Wichtig: Flags angeben!
```

Was bedeutet ein Überdeckungsgrad unter 100%?

- Mit den vorhandenen Testfällen werden nicht alle Anweisungen/Zweige/Pfade ausgeführt
- Ist u.a. auch ein Gütemass für die Testfälle

Massnahmen:

- Zusätzliche Testfälle definieren
- Nicht überdeckter Code analysieren, z.B. mittels Code Inspection

Beim nicht ausgeführten Code kann es sich auch um toten Code handeln

- wenn logisch falsch: korrigieren bzw. entfernen
- wenn aufgrund defensivem Programmierstil (z.B. default: break): lassen

12.4.3 Arten von Tests

Funktionale Tests

gemäss den funktionalen Anforderungen

- Testen anhand der vorgestellten Theorie

Nichtfunktionale Tests

gemäss nicht funktionalen Anforderungen wie Leistungsanforderungen, Leistungstest, Stress-test, Ressourcenverbrauch, Qualität (wenig ist testbar), Zuverlässigkeit, Benutzbarkeit, Sicherheit

13 Design by Contract & Refactoring

13.1 Liste der anbei NICHT dokumentierten Aufgaben:

- Prakti 1: A5, A6, A7
- Prakti 2: A1, A2, A3, A4
- Prakti 3: A1, A2, A3, A4
- Prakti 4: A1, A2, A3, A4, A5
- Prakti 7: A3
- Prakti 10: A8

Thema, Ziele: Libraries erstellen und verwenden, C++ Auffrischung

Für die folgenden Aufgaben lohnt sich eine saubere Ordner-Struktur. Das jeweilige Library Projekt sollte sich in einem anderen Ordner befinden als das Executable Projekt.

Aufgabe 1: Static Library erstellen

Verpacken Sie die Klasse Stack vom Ordner Vorlage/Stack in eine static Library mit dem Namen libStack.a.

Aufgabe 2: Static Library nutzen

Verwenden Sie die von Aufgabe 1 erstellte Library *libStack.a*. Erstellen Sie ein *main.cpp* File in welchem Sie auf die *Stack* Library zugreifen und zugleich testen.

Welche Dateien brauchen Sie dazu?

Achten Sie beim Erstellen des *main.cpp* Files darauf, dass dieses in einem anderen Ordner als die *Stack* Library liegt.

Aufgabe 3: Shared Library erstellen

Verpacken Sie die Klasse Stack analog der Aufgabe 1 in eine shared Library mit dem Namen libStack.so.

Aufgabe 4: Shared Library nutzen

Verwenden Sie die in Aufgabe 3 erstellte shared Library analog Aufgabe 2, diesmal jedoch mit der in Aufgabe 3 erstelle shared Library *libStack.so*.

Mit dem Linux Befehl `ldd` können die verwendeten shared Libraries in einem ausführbaren Programm angezeigt werden.

Welche shared Libraries werden im soeben erstellten Programm verwendet?

Aufgabe 5: Static Library mit Eclipse erstellen

Erstellen Sie aus der Stack Klasse im Vorlageordner eine statische Library *libStack.a*. Dazu soll ein neues C++Projekt in Eclipse erstellt werden. In Abbildung 1 sind die benötigten Einstellung beim Erstellen eines neuen C++ Projekts ersichtlich. Die Ordnerstruktur wird wie in Abbildung 4 dargestellt empfohlen.

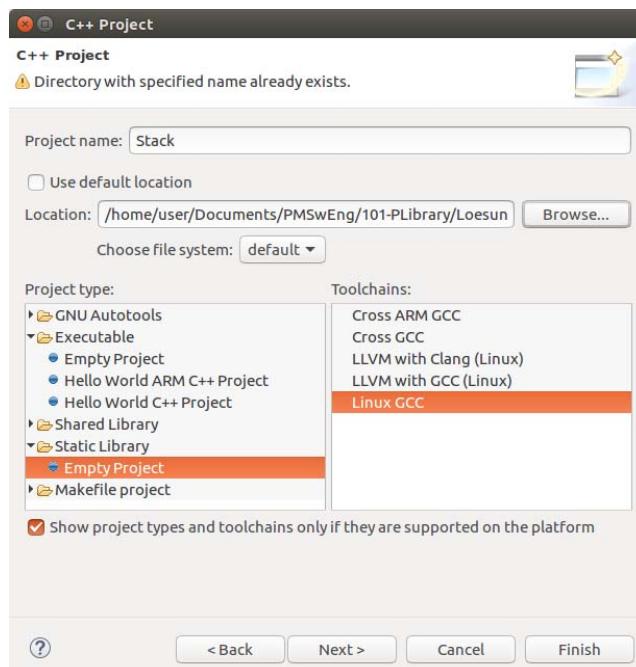


Abbildung 1 C++ Einstellungen für ein neues static Library Projekt

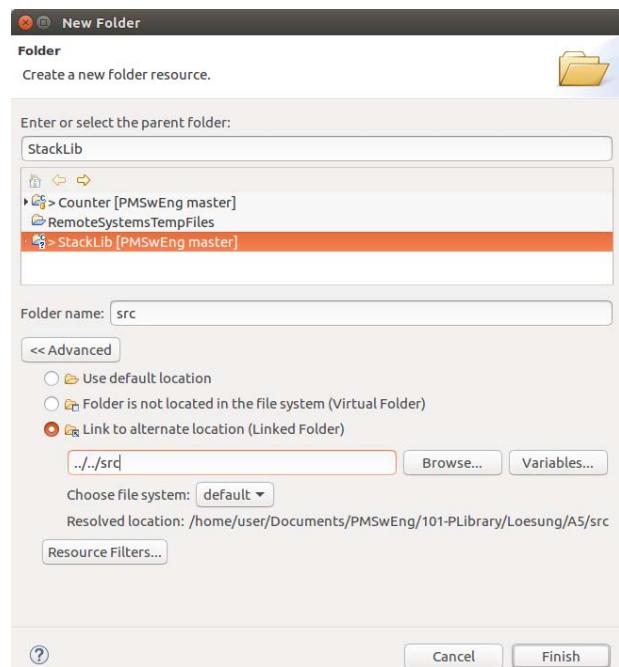


Abbildung 2 Verklinken von src Ordner

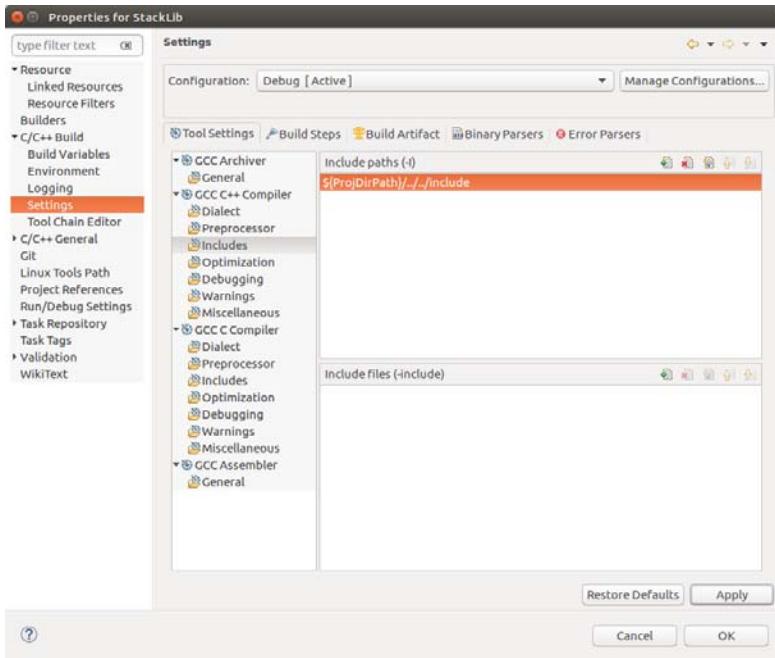


Abbildung 3 Projekteinstellung für *Include* Pfad

Das neu zu erstellende Projekt sollte also im *prj/Eclipse* Pfad abgespeichert werden.

Da Eclipse nur die Dateien im gleichen Ordner oder darunter im Projekt darstellt, müssen die *src* und *include* Ordner verlinkt werden. Wählen Sie dazu *File* → *New* → *Folder* aus. Nun kann der *src* Ordner wie in Abbildung 2 verlinkt werden. Der gleiche Schritt kann für den *include* Ordner wiederholt werden.

Wie in den vorhergehenden Aufgaben muss auch hier der *include* Ordner angegeben werden, damit der Compiler die Header Datei findet. Der Pfad kann in den Projekteinstellungen angegeben werden (siehe Abbildung 3).

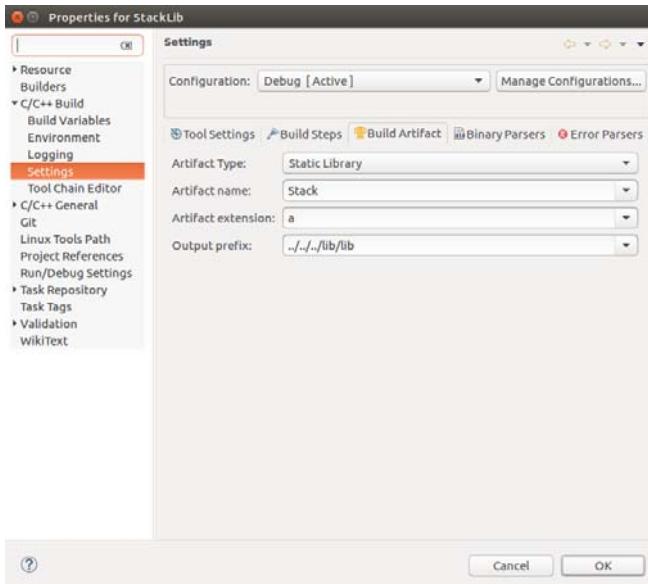


Abbildung 5 Definition des Output Ordners

Eclipse erstellt in einem *managed make* Projekt (typisches Projekt) die generierten *object* und *executable* Files unter dem Eclipse Projektdateien Ordner ab. Falls das generierte Library-File in den */lib* Ordner abgelegt werden soll wie bei den Aufgaben 1 und 3, kann dies in den Projekteinstellungen gemäss Abbildung 5 einstellen.

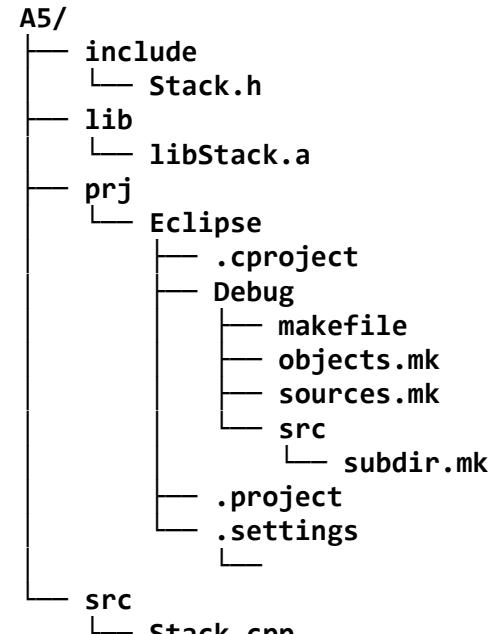


Abbildung 4 Ordner Struktur

Aufgabe 6: Static Library mit Eclipse verwenden

Erstellen Sie wie vom ProgCPP Modul gewohnt ein neues Eclipse Projekt. Beachten Sie die Ordner-Struktur ähnlich der Struktur von Aufgabe 5 (siehe Abbildung 7).

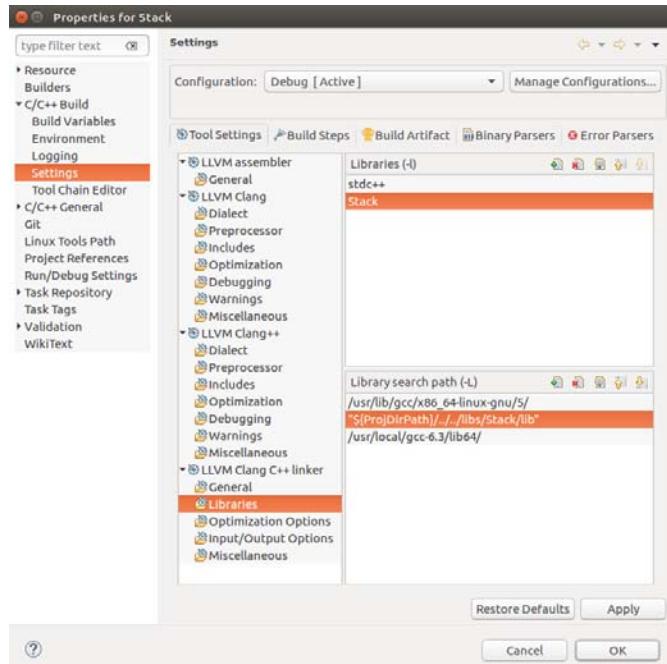


Abbildung 6 Angabe von Library Pfad und Library Name für Linker

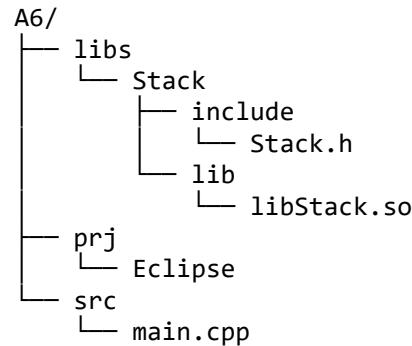


Abbildung 7 Ordnerstruktur Eclipse Executable

Vorgehen:

1. Speichern Sie die Projektdateien im Verzeichnis *prj/Eclipse* ab.
2. Verlinken Sie den *src* Ordner wie in Abbildung 2 dargestellt.
3. Fügen Sie den Includepfad *lib/include* hinzu (ähnlich Abbildung 3, aber anderer Pfad)
4. Fügen Sie den Library-Pfad hinzu sowie die Stack Library (siehe Abbildung 6)

Aufgabe 7: Klasse Counter erstellen

Erstellen Sie eine Klasse die eine Counter Funktionalität abbildet. Folgende API soll zur Verfügung stehen:

```
Counter();  
int getValue() const;  
void incValue();  
void decValue();  
void setValue(int value);  
void clearValue();
```

Im Vorlage Ordner Vorlage/Counter befindet sich das Testprogramm *main.cpp* für Ihre *Counter* Klasse. Die Klasse muss nicht in eine Library verpackt werden.

Thema, Ziele: qmake mit .pro Datei, QObject

Aufgabe 1: Counter mit qmake

Erstellen Sie ein qmake Projekt (*Counter.pro*) mit der Counter Klasse vom letzten Praktikum und dessen Test (siehe Vorlage/Counter Ordner). Verwenden Sie dazu noch nicht den Qt Creator.

Aufgabe 2: Stack App mit Stack Library und qmake

Nutzen Sie die Stack Library vom letzten Praktikum (siehe Vorlagen Ordner) mit Hilfe von qmake. Erstellen Sie wiederum eine Testapplikation *Stack*.

Hinweis: Das Einbinden kann bei qmake respektive dem .pro File wie folgt aussehen:

GCC-Flags	.pro qmake Flags
-Ipath	INCLUDEPATH += path
-Lpath -llibName	LIBS += -Lpath -llibName

Aufgabe 3: QObject auf Heap

Untersuchen Sie das Verhalten von QObjects auf dem Heap.

Vorgehen:

1. Instanziieren Sie zwei QObject Objekte *o1* und *o2* auf dem Heap. Achten Sie dabei darauf, dass *o2* ein Child von *o1* wird.
2. Löschen Sie *o1* vom Heap

Was passiert mit *o2*? Überprüfen Sie das Verhalten mittels ableiten von QObject mit der Klasse *Test*. Schreiben Sie eine entsprechende Konsolenausgabe im D'tor der *Test* Klasse. Nun sollen anstatt den QObjects *o1* und *o2* die Objekte von *Test t1* und *t2* verwendet werden.

Hinweis:

Die Ausgabe kann via *qDebug* (#include <QDebug>) wie folgt aussehen:

```
qDebug() << "Test D'tor called" << this->objectName() << " object called";
```

Dabei wird das Attribut *objectName* gentzt, dass nach dem Erstellen des Objekts gesetzt werden kann:

```
Test* t1 = new Test;
```

```
t1->setObjectName("t1");
```

Aufgabe 4: QObject auf Stack

Wiederholen Sie Aufgabe 4 mit dem Unterschied, dass die Objekte nun auf dem Stack und nicht auf dem Heap zu liegen kommen sollen.

Wieso ist die *setParent*-Methode der *QObject* Klasse bei Objekten auf dem Stack gefährlich?

Thema, Ziele: Qt IDE Qt-Creator, Layouts erstellen

Aufgabe 1: "Hello Qt-World" mit QLabel und HTML

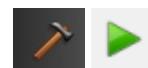
Im Unterverzeichnis *Vorlage\HtmLabel* finden Sie ein "Hello Qt-World" GUI-Programm.

Öffnen Sie das entsprechende Qt Projekt und erweitern Sie die Vorlage gemäss den folgenden Schritten:

- a) Überprüfen Sie mit Hilfe des File-Explorers welche Dateien das Verzeichnis "HtmlLabel" nun enthält. Führen Sie nun im Qt-Qreator den "Run qmake" aus dem "Build"-Menü aus. Überprüfen Sie nun welche Dateien entstanden sind. Welche Informationen enthalten diese?

Lösung:

- b) Kompilieren Sie dieses Projekt im Qt-Creator mit Hilfe des "Hammers" und führen Sie es anschliessend mit dem grünen Startknopf aus.
- c) Untersuchen Sie, welche neuen Dateien und Verzeichnisse durch den "Build"-Vorgang entstanden sind.



Lösung:

- d) Studieren Sie das Source-Programm. Es läuft zwar, aber, ist es auch ein korrekt programmiertes C++ Programm?

Lösung:

- e) Modifizieren Sie das Programm so, dass anschliessend im Titelbalken Ihr Name steht.
- f) Finden Sie heraus, von welchen Klassen *QLabel* abgeleitet (*derived*) ist. Beachten Sie auch Ober-Ober-Klassen etc. Dazu kann die in Qt integrierte Hilfe beigezogen werden. Klicken Sie irgendwo im Code wo *QLabel* steht darauf. Anschliessend drücken Sie F1. Nun sollte die Qt Dokumentation zu *QLabel* angezeigt werden.

Aufgabe 2: Manuelles Layout mit mehreren Widgets (ohne Layout Manager)

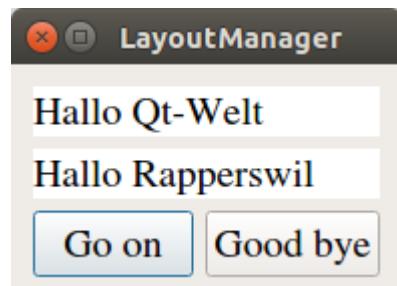
Im Unterverzeichnis *Vorlage\ManualLayout* finden Sie ein anderes "Hello Qt-World" GUI-Programm. Dieses besteht aus dem Hauptfenster 'wTop' und beinhaltet ein *QLabel* zur Anzeige des Textes "Hallo Qt-Welt".

- a) Erweitern Sie dieses Programm so, dass unterhalb des bestehenden *QLabels* ein neues Label mit dem Text "Hallo Rapperswil" zu stehen kommt.
- b) Erweitern Sie Ihr Programm zusätzlich durch zwei "QPushButton"-Objekte, die zuunterst im Hauptfenster nebeneinanderliegen sollen. Der Button links soll mit "Go on", der rechts mit "Good bye" beschriftet sein.

Aufgabe 3: Layout Manager mit mehreren Widgets

Im Unterverzeichnis *Vorlage\LayoutManager* finden Sie ein weiteres "Hello Qt-World" GUI-Programm.

Erstellen Sie ein Layout mit den gleichen Widgets und derselben Anordnung wie bei Aufgabe 2 mit Hilfe von Layout Managern. Die Abbildung rechts zeigt wie das Layout schlussendlich aussehen sollte.



Aufgabe 4: Layout mit Qt GUI Creator erstellen

Erstellen Sie das gleiche Layout wie bei Aufgabe 3 mit Hilfe des QtCreadors. Erstellen Sie dazu ein neues "Qt Widgets Application"-Projekt mit dem Namen QtCreator (Abbildung 2). Das Top-Widget soll ein QWidget sein und die dazugehörige Klasse soll Widget heissen (Abbildung 4) . Das fertige Layout könnte wie Abbildung 3 aussehen.

Nun sollte das Projektverzeichnis wie Abbildung 5 aussehen. Das Aussehen der grafischen Oberfläche kann nun im GUI Creator spezifiziert werden. Dazu kann ein Doppelklick auf Widget.ui durchgeführt werden, worauf sich der GUI Creator öffnet.

Das Hauptlayout des Top-Widgets kann mittels rechtsklick auf Widget (rot eingekreist) → Layout ausgewählt werden. Das funktioniert aber erst, wenn mindestens ein Child Widget bereits hinzugefügt wurde.

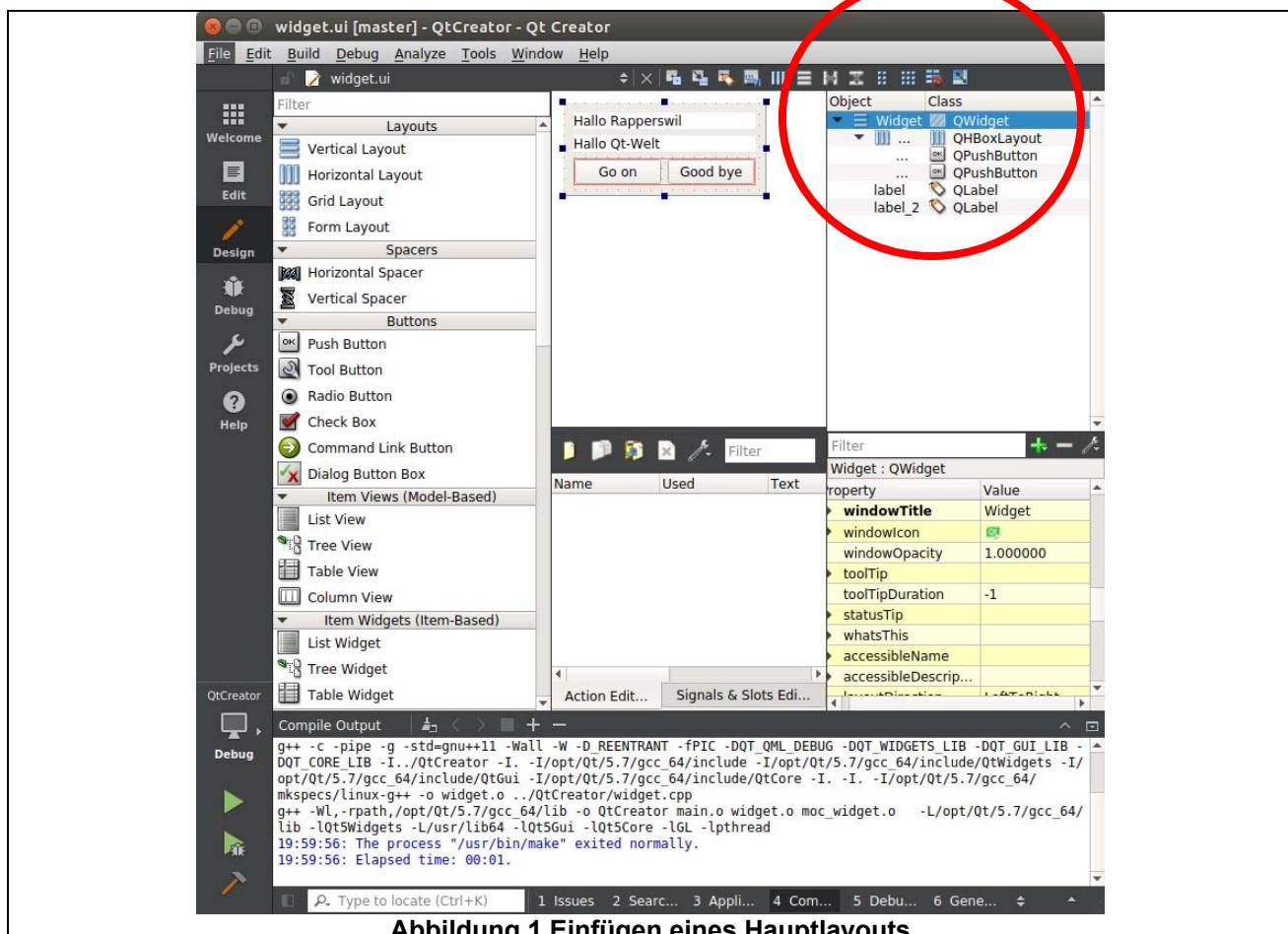


Abbildung 1 Einfügen eines Hauptlayouts

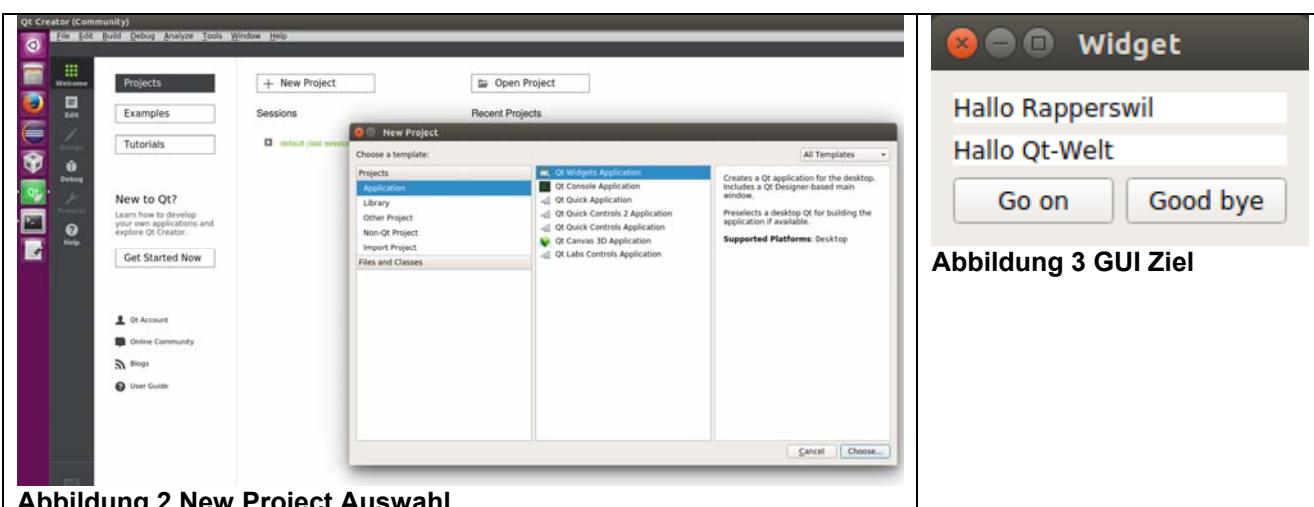
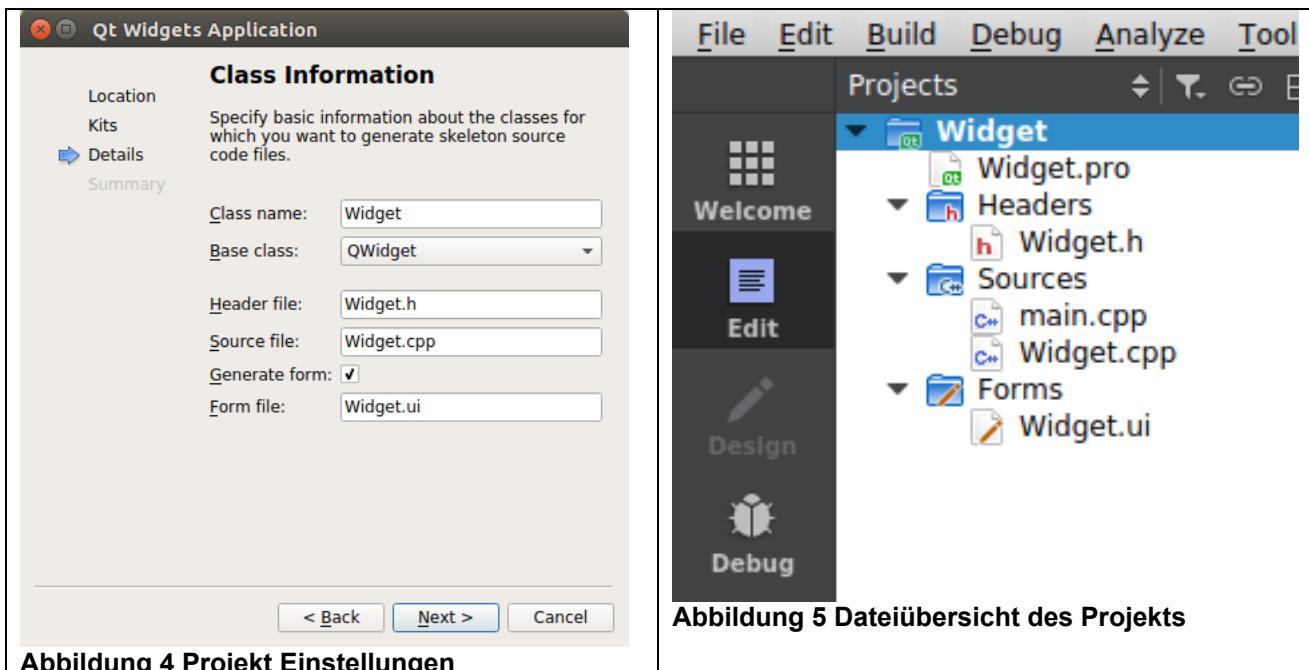


Abbildung 2 New Project Auswahl



Zusätzliche Hinweise zur Qt-Programmierumgebung (Qt Creator)

Verwendete Projektorganisation (Shadow-Building)

Mit Hilfe der Qt-Library ist es bekanntlich möglich, ausgehend von den gleichen Source-Files, ausführbare Programme (sogenannte "Builds") für unterschiedliche Konfigurationen (Plattformen, Compiler etc.) zu erstellen.

Dazu legt der Qt-Creator für jede Konfiguration (Plattform, Compiler etc.) ein eigenes Build-Verzeichnis an, das die konfigurationsspezifischen Daten wie makefiles, object-files etc. enthält. Diese Aufteilung hat den Vorteil, dass die plattformspezifischen Dateien völlig getrennt sind von den (gemeinsamen) Source-Dateien. Außerdem werden dadurch die verschiedenen "Builds" (Plattformen, Compiler etc.) sauber voneinander getrennt. (Dieser Mechanismus wird als "Shadow-Building" bezeichnet.)

Der Name eines Build-Verzeichnisses wird dabei aus dem Namen der Projektdatei (Bsp. "Counter1.pro") und der verwendeten Konfiguration (Plattform, Compiler etc.) gebildet. Das führt dann zum Beispiel zu einem Namen wie "build-Counter1-Desktop-Qt_5_4_1_MinGW_32bit-Debug" für das entsprechende Build-Verzeichnis.

Thema, Ziele: Signal & Slots, eigene Widgets

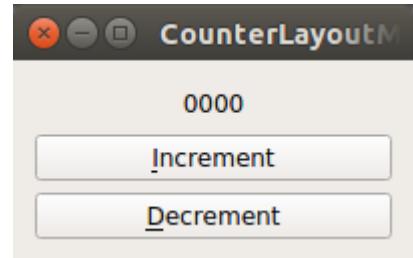
Aufgabe 1: Qt Counter: Layout Manager

Auf dem Skriptserver, im Unterverzeichnis "Vorlage" finden Sie die Klasse Counter (counter.cpp und counter.h). Erstellen Sie ein Qt-Programm mit dem ein Zähler realisiert wird. Sie können dazu die Counter Klasse verwenden.

Verbinden Sie die Signale `clicked` der beiden Buttons mit den Counter Slots (Memberfunktionen) `incValue` resp. `decValue`.

Die beiden Buttons sollen nun so definiert werden, dass sie auch mit den Tasten `<Alt>+I` und `<Alt>+D` (sogenannte "Shortcut's") bedient werden können. – Anleitung: Schreiben Sie ein `&`-Zeichen beim Button-Text vor den Shortcut-Buchstaben. Also "`&Increment`" und "`&Decrement`". Probieren Sie andere Buchstaben.

Hinweis: Ausserhalb eines QObjects muss die `connect`-Methode über den QObject Klassenscope aufgerufen werden → `QObject::connect(...)`

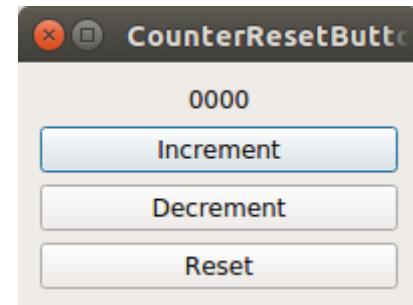


Aufgabe 2: Qt Counter: Reset-Button

Entwerfen Sie eine Counter Applikation mit GUI mittels QtCreator. Zeichnen Sie das GUI mit dem eingebauten GUI-Designer. Verwenden Sie wiederum die Counter Klasse aus dem Vorlageordner auf dem Skripteserver.

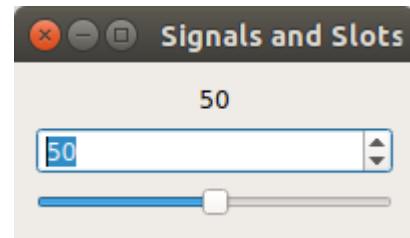
Verbinden Sie auch wieder die entsprechenden Signale mit den dazugehörigen Slots (inklusive Reset-Button).

Das Programm soll in etwa so wie rechts dargestellt aussehen.



Aufgabe 3: Signals and Slots

Erstellen Sie mit dem Qt-Creator und GUI-Designer das rechts dargestellte GUI. Falls der Wert beim QSpinBox-Objekt mit Hilfe der kleinen "Up-Down-Buttons" geändert wird oder den Schieberegler betätigt wird, sollen alle angezeigten Werte (inklusive Schieberegler) dieser Änderung folgen. Verbinden Sie die benötigten Signale mit den entsprechenden Slots (Memberfunktionen).



Hinweise:

Wird beim QSlider (Schieberegler) mit der Maus der Regler verschoben, sendet das QSlider-Objekt das Signal "void valueChanged(int x)" aus. Um andererseits die Schiebereglerposition "von aussen" zu setzen, gibt die Memberfunktion (Slot) "void setValue(int x)". Der Wertebereich des Sliders ist 0..99.

Beim Anklicken der (kleinen) "Up-Down-Buttons" beim QSpinBox-Objekt in der Mitte, wird von diesem Objekt das Signal "void valueChanged(int x)" ausgesendet. Zum Setzen des angezeigten Wertes gibt es bei dieser Klasse den Slot (Memberfunktion) " void setValue(int x)". Diese Signals und Slots heissen also gleich wie beim QSlider. Der Wertebereich der QSpinBox ist 0..99.

Erweiterung "QMessageBox"

Das QLabel- QSpinBox- und QSlider-Objekt zeigen bekanntlich immer den gleichen Wert im Bereich 0..99 an. Beim Start des Programms soll dieser Wert nun 40 sein.

Wenn beim Verstellen dieser Wert nun ≥ 80 oder ≤ 20 wird, soll ein Dialogfenster mit der Fehlermeldung "Wert ausserhalb Bereich!" erscheinen. Nach dem "Wegklicken" dieser Fehlermeldung soll dann der Wert auf 50 gesetzt werden.

Erweitern Sie ihr Programm entsprechend.

Hinweis: Verwenden Sie QMessageBox::warning(this, " ", "Wert ausserhalb Bereich!");

Aufgabe 4: Callback in C

Die in der Vorlesung gezeigte Callback Version für C ist nur für das Registrieren eines Clients gedacht. Der entsprechende Code finden Sie im Vorlagen Ordner unter CCallback. Als Erweiterung sollen sich mehrere Clients registrieren und die Registrierung löschen können (*unregister*). Erweitern Sie den Code entsprechend, damit sich neben dem Modul a auch das Modul a2 registrieren können und die Registrierung wieder löschen kann.

Der folgende Code soll zum Test verwendet werden:

```
int main (void)
{
    aInit();
    a2Init();

    bFooX(); /* fire event */

    printf("aFoo should be called now\n");
    printf("a2Foo should be called now\n");

    aClear();

    bFooX(); /* fire event */
    printf("a2Foo should be called now\n");

    a2Clear();
    aInit();

    bFooX(); /* fire event */
    printf("aFoo should be called now\n");

    return 0;
}
```

Aufgabe 4: Observer mit C++

Ähnlich wie in der vorherigen Aufgabe soll die Observer C++ Variante 3 der Vorlesung für mehrere Observer erweitert werden. Im Vorlage Ordner finden Sie den Code unter *CppObserver*. Erweitern Sie den Code, damit sich mehrere Objekte der Klasse A bei der Klasse B registrieren können.

```
int main()
{
    B b;
    {
        A a1(b);
        A a2(b);

        // fire event
        b.fooX();

        std::cout << "A::foo should be called twice" << std::endl;
    }
    {
        A a2(b);

        // fire event
        b.fooX();

        std::cout << "A::foo should be called once" << std::endl;
    }

    return 0;
}
```

Thema, Ziele: Qt Vertiefung und PM in SwEng

Aufgabe 1: Qt Counter: zweiter Zähler

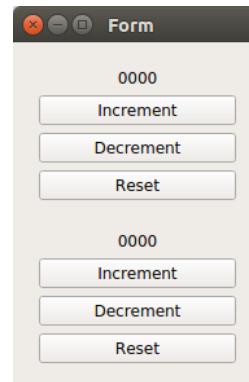
Erweitern Sie Ihr CounterWidget Programm von letzter Woche so, dass zusätzlich noch ein zweiter Zählerstand angezeigt wird und verändert werden kann – siehe nebenstehende Abbildung.

Die beiden Zähler sollen unabhängig voneinander funktionieren.

Erstellen Sie dazu ihr eigenes Widget mit dem Namen *CounterWidget*, welches die benötigten Widgets für einen Counter beinhaltet. Instanzieren Sie entsprechend zwei Objekte der *CounterWidget* Klasse um diese Aufgabe zu lösen.

Hinweis:

Ihr eigene *CounterWidget* Klasse soll von QWidget ableiten.



Aufgabe 2: Projektmanagement und Software Engineering

Zitat (Autor unbekannt): "Aus Fehlern wird man klug."

Ergänzung: "... sofern die nötige Grundklugheit vorhanden ist."

Obwohl in der Raumfahrt bei der Software-Entwicklung konsequent Software-Engineering-Methoden eingesetzt werden, kann es auch hier zu spektakulären Unfällen kommen:

2.1 "The Explosion of the Ariane 5" – 1996

Einen solchen Unfall beschreibt der Artikel "The Explosion of the Ariane 5" auf der nächsten Seite.
Quelle: "<http://www.ima.umn.edu/~arnold/disasters/ariane.html>".

Lesen Sie diesen Artikel und beantworten Sie anschliessend die untenstehenden Fragen.

Beachten Sie, dass aus diesem Artikel nicht hervorgeht, dass das bei der Ariane 5 für das INS ("Inertiales Navigationssystem") verwendete Softwaremodul weitgehend von der Ariane 4 übernommen wurde. Dort lief diese Software lange Zeit erfolgreich und wurde deshalb auch als sicher eingestuft.
(Details siehe "http://de.wikipedia.org/wiki/Ariane_V88")

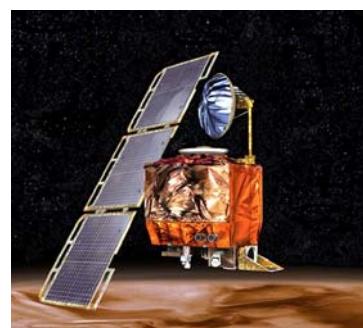
Fragen:

1. Was war der Fehler der zum Unfall mit der Ariane 5 führte?
2. Was war Ihrer Meinung nach die Ursache für diesen Fehler?
3. Wie hätte sich dieser Fehler Ihrer Meinung nach vermeiden lassen können?
4. Welche Schlüsse ziehen Sie aus diesem Fehler in Bezug auf die Entwicklung von Software im Allgemeinen und das Software Engineering im speziellen (Entwicklung, Testen, Fehler, Wartung, etc.)?
5. Was hat dieser Fehler gekostet? (nur direkte Kosten, ohne Entwicklungs- und Folgekosten)

2.2 Verlust der NASA-Sonde "Mars Climate Orbiter" (MCO) – 1999

Lesen Sie den folgenden Wikipedia-Beitrag "http://de.wikipedia.org/wiki/Mars_Climate_Orbiter" und beantworten Sie anschliessend folgende Fragen.

1. Wie lange dauerte der Flug dieser NASA-Sonde bis der Fehler sich bemerkbar machte?
2. Was war der Fehler der zum Verlust führte?
3. Angenommen Sie würden die Fehlerursache nicht kennen. Hätten Sie sich dann vorstellen können, dass in der Raumfahrt solch "dumme" Fehler vorkommen können?
4. Was hat dieser Fehler gekostet?



Aufgabe 3: Management von SwPrj Verständnisfragen

Erarbeiten Sie die Aufgaben in einer 2er oder 3er Gruppe. Die Lösungen können fortlaufend mit dem Praktikumsleiter besprochen werden.

- a) Zählen Sie drei völlig verschiedene Vorhaben auf, die eindeutig Projekte sind.
- b) Zählen Sie drei völlig verschiedene Vorhaben auf, die sicher keine Projekte sind. Geben Sie dabei an, warum sie keine Projekte sind.
- c) In einer Firma muss jedes Jahr im Dezember ein Neujahrsgeschenk für die Kunden "erfunden" werden. Da Sie als kreativer Kopf gelten, landet diese Aufgabe meistens bei Ihnen.
Frage: Handelt es sich bei diesem Vorhaben um ein Projekt oder nicht?
Führen Sie die Argumente auf, die für ein Projekt sprechen und diejenigen Argumente die dagegen sprechen.
- d) "Ein Mann braucht zum Bau einer 2 m langen Brücke 1.5 Tage. Wie lange brauchen 100 Leute für den Bau einer 2 km langen Brücke?"
Berechnen Sie die Dauer mit Hilfe eines Dreisatzes.
Erklären Sie warum diese "Milchmädchenrechnung" hier falsch ist.
- e) Zeichnen Sie eine zweidimensionale Grafik, die zeigt, wie während dem Projektverlauf sich bei einem typischen Projekt, das Wissen (in Prozent) über das zu realisierende Projekt darstellt. (x-Koordinate = Zeit, y-Koordinate = Wissen in Prozent).
- f) Stellen Sie in einer zweidimensionalen Grafik dar, wie sich bei einem typischen Projekt, während dem Projektverlauf der Handlungsspielraum des Projektleiters ändert (x-Koordinate = Zeit, y-Koordinate = Handlungsspielraum).
- g) Eine berühmte Aussage bezüglich Software-Projekten ist:
"adding manpower to a late software project makes it later" (Fred Brooks, 1975)
Erklären Sie diese Aussage, Brooks's law genannt, und begründen Sie, warum sie auch heute, 35 Jahre später für die Softwareentwicklung immer noch richtig ist.
Erklären Sie allgemein bei welcher Art von Projekten diese Regel nicht gilt und geben Sie ein konkretes Beispiel für ein solches Projekt an.
- h) Gegenfrage zu Aufgabe g): Sollten demnach alle Mitarbeiter bereits von Projekt-Beginn an dabei sein?

Aufgabe 4: Projektphasen Verständnisfragen

- a) Was ist das Ziel in der Projektentstehungsphase? Formulieren Sie Ihre Antwort so, dass diese aus maximal 3 Wörtern besteht. – Hinweis: Worin besteht die Hauptarbeit in dieser Phase?
- b) Was sind die Hauptaufgaben der jeweiligen Projektphasen? Geben Sie pro Phase mindestens eine Aufgabe an.

Thema, Ziele: Vorgehensmodelle

Aufgabe 1: Verständnisfragen

- a) Erklären Sie, warum das Original-Wasserfallmodell in der Praxis nicht funktioniert.
(In welchen speziellen Fällen kann es doch funktionieren?)
- b) Erklären Sie den/die Unterschied(e) zwischen dem modifizierten Wasserfallmodell (mit Rückweg zur vorhergehenden Phase) und den anderen iterativen und inkrementellen Vorgehensmodellen z.B. Spiralmodell.
Zeichnen Sie drei mögliche und naheliegende Wege auf für die Tätigkeiten a, b, c, d.
- c) Bei den Vorgehensmodellen spricht man auch von "schwergewichtigen" und "leichtgewichtigen" Prozessen. Was ist da gemeint? Nennen Sie dazu jeweils mindestens ein Vorgehensmodell.
- d) Ist eine Kombination von agilen und klassischen Vorgehensmodellen möglich? Wenn ja, auf welche Art und Weise? Wenn nein, warum nicht?
- e) Angenommen die Kombination von agilen und klassischen Vorgehensmodellen aus Aufgabe d) ist möglich. Zu welchem Zeitpunkt sollen welche Anforderungen definiert sein und wieso?
- f) Nicht alle Vorgehensmodelle eignen sich gleich gut für verschiedene Vertragsformen. Welche Vor- und Nachteile haben bestimmte Vorgehensmodelle mit einem Festpreis-Vertrag?
- g) Erklären Sie, was man unter dem Begriff "Big Bang" bei Software Projekten versteht.
Bei welcher Art von Vorgehen kann es diesen "Big Bang" Effekt am ehesten geben?
- h) Bei welchen Vorgehensmodellen kann die objektorientierte Software-Entwicklung eingesetzt werden?
- i) In der nächsten Aufgabe wird von einem Projektauftrag gesprochen. In welcher Projektphase befindet sich demnach das Projekt?

Aufgabe 2: Projekteinstieg

Bilden Sie ein Team, welches bis zum Ende des Semesters an folgendem Projekt beschäftigt sein wird. Die Aufgabenstellung sowie weitere Informationen zum Projekt finden Sie auf dem Skripte Server unter dem Ordner **910-Projekt/Projektauftrag.pdf**. Lesen Sie dieses Dokument durch und überlegen Sie Ihr Vorgehen. Bei Fragen können Sie sich an den Praktikumsleiter wenden.

Anschliessend können die gebildeten Teams dem Praktikumsleiter gemeldet werden.

Setzten Sie Ihre Entwicklungsumgebung auf, damit Sie in der folgenden Woche ohne Verzögerung mit dem Projekt beginnen können.

Die folgende Liste von Werkzeugen / Tools kann als Checkliste dienen:

- Projektplanungstool, bevorzugt MS Project oder MS Excel (mindestens einer aus der Gruppe)
- Qt Umgebung*
 - inklusive Qt Test*
 - inklusive googletest*
- Git*
- Git Repo erstellt und geklont
 - Workflow-Strategie festgelegt
- Doxygen*
- MS Word oder LateX

* ist bereits auf dem Linux-Image installiert

Aufgabe 3: grobe Anforderungen ermitteln

Wenn wir uns an den Projektphasen orientieren wäre die nächste Phase die Projektdefinition. Diese Phasen sind jedoch eher als grobes Raster und nicht exakte Wissenschaft zu verstehen. D.h. die Phasen können sich überlappen.

Zu diesem Zeitpunkt des Projekts sollen die groben Anforderungen ermittelt werden. Einige Anforderungen sollten aus dem Projektauftrag ersichtlich sein, andere müssen selber ergänzt werden.

Die ermittelten Anforderungen geben dem Entwickler ein besseres Bild des Produktziels und werden auch für die Aufwandschätzung genutzt. Diese Aufwandschätzungen wiederum können zugleich für ein Projektangebot genutzt werden, falls z.B. ein Festpreis-Projekt ausgeschrieben ist.

Das Einarbeiten in fachliche Themen kann zu diesem Zeitpunkt durchaus Sinn machen. Gerade bei Themen in denen die Erfahrung fehlt oder bereits kritische, komplexe Teile des Produkts bekannt sind ist eine Einarbeitung zu empfehlen. Ein gängiges Verfahren dazu ist z.B. *Rapid Prototyping* (siehe Broy Kap. 4.2.3.2).

Ermitteln Sie aufgrund des Projektauftrags die groben Anforderungen. Erstellen Sie dazu ein Kontextdiagramm mittels UML Use-Case Diagramm. Die Theorie zu Use-Case Diagrammen ist im PDF *UseCaseDiagramm.pdf* zu finden.

Thema, Ziele: Algorithmische Schätzungen

Aufgabe 1: Rapid Prototyping

Für welche Themen ist Rapid Prototyping geeignet? Was sind die Vor- und Nachteile von Rapid Prototyping?

Wie lässt sich Rapid Prototyping in den Projektlebenszyklus einordnen und wie sieht die Projekthierarchie aus? Geben Sie auch ein geeignetes Vorgehensmodell an falls nötig.

Aufgabe 2: Function Point Analysis

Aufgrund den im letzten Praktikum erstellten groben Anforderungen mit Hilfe eines Use-Case Diagramms sollen nun ein Functional Point Analyse durchgeführt werden.

Wie viele Functions Points erhalten Sie aus für die Funktionalität der in unserem Projekt zu erstellenden Software?

Als Ausgangslage empfehle ich meinen Use Case Lösungsvorschlag vom letzten Praktikum zu verwenden, damit die Lösungen dieser Aufgabe nicht zu gross abweichen.

Zur Vereinheitlichung der Lösung sind hier die zu schätzenden Elemente aufgelistet:

- Datenelemente
 - ILF VoltageDividerWidget inkl. ESeries
 - 2 RETs
 - VoltageDividerWidget
 - ESeries
 - ? DETs:
 - U1,
 - U2,
 - ?,
 - aktuell ausgewählte Serie
- Transaktionselemente
 - Parameter konfigurieren (worst case Betrachtung mit E-Reihe setzen)
 - 2 FTRs
 - VoltageDividerWidget
 - ESeries
 - ? DETs:
 - ?
 - Widerstand berechnen
 - 2 FTR
 - VoltageDividerWidget
 - ESeries
 - ? DETs:
 - ?

Vorgehen:

1. Bestimmen Sie die fehlenden, mit einem Fragezeichen gekennzeichneten Elemente.
2. Bestimmen Sie die Komplexität der jeweiligen Daten- und Transaktionselemente.
3. Ermitteln Sie die Anzahl Function Points für die zu entwickelnde Anwendung.

Aufgabe 3: Stack GUI implementieren

Implementieren Sie mittels Qt ein Stack GUI. Die Problem-Domain ist bereits implementiert und ist im Vorlage Ordner „Vorlage/A1“ zu finden.

Zu implementierende GUI Darstellungen/Funktionalitäten sind

- *push, pop und peek* Buttons
- Eingabe einer Zahl als *integer* Datentyp
- Es kann davon ausgegangen werden, dass der User die Eingabe immer korrekt macht → keine Eingabeüberprüfung
- Information über vollen und leeren Stack (Zustände müssen ständig sichtbar sein)

Die *Stack* Klassenvorlage ist als Template implementiert. Der *moc* unterstützt derzeit keine Klassen-Templates in Zusammenhang mit *QObject* und *Signal/Slots*. Als Workaround wurden die *Signals/Slots* in die Basisklasse *StackSignals* der *Stack* Klasse verschoben werden.

Thema, Ziele: Projektplanung**Aufgabe 1: COCOMO II Größenfaktoren und Kostentreiber**

Interpretieren Sie folgende Grafik:

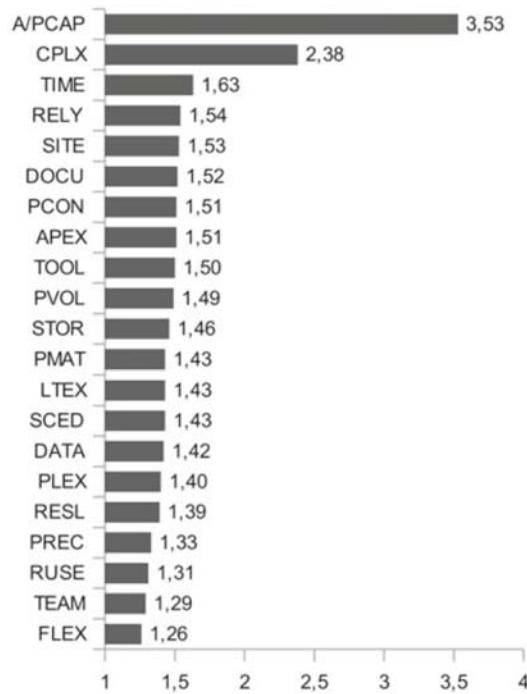


Abbildung 1 Einflussfaktoren von COCOMO II Kostentreibern und Größenfaktoren

Die Abbildung 1 stellt den Einflussfaktor P eines jeden Kostentreibers EM und Größenfaktors SF dar und erlaubt so ein Vergleich der Gewichtung untereinander.

Zur Berechnung der Werte werden die folgenden zwei Formeln verwendet:

$$P_{EM_i} = \frac{\max(EM_i)}{\min(EM_i)}$$

$$P_{SF_i} = \frac{100^{0.91 + (0.01 \max(SF_i))}}{100^{0.91}}$$

Fragen:

1. Was fällt bei dieser Grafik auf?
2. Wie kommt der A/PCAP Wert zustande?

Aufgabe 2: COCOMO II Aufwandschätzung

Berechnen Sie den Aufwand unseres Projekts anhand der vom letzten Praktikum berechneten Funktion Points. Verwenden Sie dazu das COCOMO II Early-Design Modell.

Für die Berechnung kann die Excel Vorlage COCOMOII.xlsx verwendet werden. Die Tabellen zur Eruierung der einzelnen Kostentreiber und Größenfaktoren können dem Word Dokument COCOMOII/Tabellen.docx entnommen werden. Eine weitere Beschreibung zu den Kostentreibern und Größenfaktoren als Ergänzung finden sie im PDF *modelman.pdf*.

Aufgabe 3: Projekt Planung

Arbeiten Sie am Projekt weiter. Erstellen Sie dazu ein Projektplan. Im Ordner *910-Projekt* finden Sie die Anleitung *Projektplan mit MS Project.pdf* um mit Microsoft Project einen einfachen Projektplan zu erstellen. Aufgrund des geschätzten Aufwands von Aufgabe 2 werden Sie bemerken, dass das Projekt nicht in der vorgegebenen Zeit realisierbar ist.

Aus diesem Grund wird die Berechnungsklasse für die Widerstände vorgegeben. So muss nur noch ein GUI dazu erstellt werden. Dazu können Sie bei der Abschätzung auf Ihre eigene Erfahrung zählen, denn Sie haben schon die ähnliche Aufgabe *StackWidget* in Qt bewältigt.

Wenn Sie den Semesterplan anschauen werden Sie bemerken, dass der Projektplan sehr sequentiell aussehen muss. Der Grund liegt in der sequentiell vermittelten Theorie. Für nächste Projekte werden sich Gedanken zu inkrementell/iterativen Vorgehen lohnen. In der Praxis werden so z.B. Implementation, Test und Code-Dokumentation oft quasi parallel ausgeführt. Das SW Design wird oft bereits am Anfang auf einen guten Stand gebracht, damit die groben Strukturen vorgegeben sind anschliessend wird das SW Design im Verlauf des Projekts vervollständigt.

Hinweis:

Das Ziel dieser Aufgabe ist es MS Project kennen zu lernen. Dieses Tool ist für die Grösse unseres Projekts sicher überdimensioniert. Gut möglich, dass das Erstellen eines Plans mit Excel ähnlich schnell oder schneller möglich ist.

A2: COCOMO II Tabellen

Thema, Ziele: Pflichtenheft erstellen

Aufgabe 1: Pflichtenheft erstellen

Erstellen Sie für das Spannungsteiler Projekt ein Pflichtenheft. Im Ordner *Pflichtenheftvorlage* finden Sie die Word und LateX Vorlagen für das Pflichtenheft.

Bei offenen Punkten bezüglich Anforderungen kann der Praktikumsbetreuer befragt werden. Er repräsentiert den Kunden.

Thema, Ziele: Git Basics, Git lokal verwenden, VoltageDivider Implementation

Aufgabe 1: Repo einrichten

In jedem beliebigen Verzeichnis (Pfad) kann ein neues Git-Repo erstellen. Wählen Sie einen geeigneten Pfad um ein Git Repo zu erstellen (z.B. ~/Documents/gitTest) und folgen Sie den untenstehenden Anweisungen:

1. Erstellen Sie ein Git Repo mit `git init`
2. Um später Commits auszuführen, muss der Name und die E-Mail Adresse hinterlegt werden. Diese Informationen können wie folgt konfiguriert werden:
 \$ `git config --global user.name "Max Muster"`
 \$ `git config --global user.email max.muster@bsp.com`

Aufgabe 2: Dateien comitten

Für diese Aufgabe wird Aufgabe 1 vorausgesetzt.

Bei dieser Aufgabe sollen sie die Befehle `add` und `commit` in verschiedenen Variation verwendet werden um Dateien in das lokale Repo zu committen.

Führen Sie die untenstehenden Anweisungen aus. Sie können dabei das in Aufgabe 1 erstellte Repo verwenden.

1. Erstellen Sie eine Textdatei mit dem Namen `Hallo.txt`
2. Bringen Sie diese Datei in die Staged Area
3. Committen Sie diese Datei anschliessend in das lokale Repo. Achten Sie darauf, dass Sie zu jedem Commit immer einen Kommentar schreiben.
4. Überprüfen Sie mittels `gitk` Befehl, ob sich der Commit mit dem entsprechenden Kommentar im lokalen Repo befindet.
5. Ändern Sie den Text in der Datei `Hallo.txt` etwas ab und committen Sie die Datei erneut.
6. Überprüfen Sie erneut, ob der Commit im Repo zu finden ist.

Nun sollten Sie folgende Befehle verwendet haben:

- `git add`
- `git add -u`
- `git commit -m "Kommentar"`
- `git commit -a -m "Kommentar"`
- `gitk`

Es werden nicht unbedingt alle aufgelisteten Befehle benötigt um die obenstehenden Anweisungen auszuführen. Wiederholen Sie die obigen Anweisungen, bis Sie alle Befehle einmal verwendet haben.

Aufgabe 3: getrackte Dateien

Für diese Aufgabe wird Aufgabe 2 vorausgesetzt.

In der Praxis werden nicht immer alle Dateien in einem Verzeichnis getrackt. Mit dem Befehl `git ls-tree branchname` können Sie die von Git getrackten Dateien anzeigen lassen:

```
git ls-tree master
```

Überprüfen Sie, ob `Hallo.txt` getrackt ist. Erstellen Sie eine weitere Datei mit dem Namen `Welt.txt` und überprüfen Sie erneut, welche Dateien getrackt sind.

Fügen Sie anschliessend die `Welt.txt` Datei ebenfalls zum Repo hinzu.

Aufgabe 4: git diff

Für diese Aufgabe wird Aufgabe 3 vorausgesetzt. Bei dieser Aufgabe sollen die Änderungen an Dateien verglichen werden. Dabei befinden sich die Änderungen sowohl im Working Directory sowie in der Staging Area. Mit den folgenden Anweisungen sollen die Zusammenhänge der drei Bereiche (Working Directory, Staging Area und local Repo) gut erkennbar sein:

1. Ändern Sie die Datei Hallo.txt ab. Achten Sie dabei darauf, dass Sie Zeichen oder Zeilen weglassen, einige Zeichen stehen lassen und einige Zeichen oder Zeilen hinzufügen.
2. Überprüfen Sie die Änderungen mit der Konsole und mit dem GUI
3. Stagen Sie diese Änderungen und überprüfen Sie anschliessend die Änderungen mit der Konsole sowie auch mit dem GUI. Sehen Sie sich die Differenzen von der Working Directory zur Staging Area und zum lokalen Repo an.
4. Ändern Sie die Datei Hallo.txt erneut. Vergleichen Sie nun die beiden Versionen der Hallo.txt Datei zwischen Working Directory und Staging Area.
5. Committen Sie die Änderungen in der Staged Area
6. Committen Sie die Änderungen im Working Directory

Aufgabe 5: git reset mit Pfad/File

Für diese Aufgabe wird Aufgabe 4 vorausgesetzt.

Falls Sie eine Änderung von einer oder mehreren Dateien im Staged Bereich verwerfen wollen, können Sie den `reset`-Befehl einsetzen.

Die folgenden Anweisungen zeigen einen möglichen Ablauf:

1. Ändern Sie den Inhalt der Dateien Hallo.txt und Welt.txt etwas ab und stagern beide Dateien. (Aufgrund der vorherigen Aufgabe sollte nur noch Hallo.txt gestaged werden müssen)
2. Bevor Sie committen stellen Sie fest, dass die Änderungen an der Datei Hallo.txt nicht in den Commit sollen
3. Unstagen Sie die Datei Hallo.txt
4. Committen Sie nun (nur) die Dateien aus der Staged Area

Aufgabe 6: git reset ohne Pfad/File

Für diese Aufgabe wird Aufgabe 5 vorausgesetzt.

Falls Sie auf eine frühere Version (einen früheren Commit) wechseln wollen, können Sie den Befehl `git reset` verwenden. Der Unterschied zu Aufgabe 5 besteht darin, dass hier kein Pfad/File hinter `git reset` geschrieben wird. Durch diesen Unterschied verhält sich dieser Befehl entsprechend anders.

Mit den folgenden Anweisungen kann ein typischer Ablauf in der Praxis nachvollzogen werden:

1. Mit dem letzten Commit aus Aufgabe 5 haben sie einen nicht brauchbaren Stand commited. Sie möchten nun der vorletzte Commit auschecken
2. Setzten Sie die Staged Area sowie auch das Working Directory zurück

Aufgabe 7: .gitignore

Generierte Dateien, temporäre Dateien oder von Editoren erstellte Sicherungskopien möchten in der Regel nicht unter Versionsverwaltung gestellt werden. Durch Einträge in die Datei `.gitignore` im Rootverzeichnis des Working Directoys können diese für Git „unsichtbar“ gemacht werden.

Erstellen Sie ein Qt Projekt und ein `.gitignore` File. Welche Qt Projekt abhängigen Dateien sollen in das `.gitignore` File geschrieben werden? Schreiben Sie ein entsprechendes `.gitignore` File.

Aufgabe 8: VoltageDivider Anwendung implementieren

Implementieren Sie die `VoltageDivider` Anwendung anhand der im Pflichtenheft (siehe 910-Projekt/Pflichtenheft.pdf) spezifizierten Anforderungen. Wie bereits erwähnt ist eine komplette Implementierung der Anwendung in der zur Verfügung stehenden Zeit unrealistisch. Aus diesem Grund wird die Lösung für die Problem-Domain vorgegeben. Sie finden diese im Vorlagen Ordner dieses Praktikums. Binden Sie die vorgegebenen Klassen in Ihr GUI inkl. Interaktionen ein.

Thema, Ziele: Git branch, merge und remote Repos, Code Dokumentieren

Aufgabe 1: clone Repo

Erstellen Sie auf dem HSR Git-Portal ein entferntes Repo. Öffnen Sie einen Internetbrowser und geben Sie die URL <https://git.hsr.ch> ein. Loggen Sie sich mit dem HSR Login ein. Wählen im linken Menü den Eintrag „Repositories“ aus. Nun können Sie mittels des „Hinzufügen“-Buttons ein neues Repo auf dem HSR Git Server erstellen. Folgende Informationen können/müssen angegeben werden:

Feld	Beschreibung
Name	Name des Git Repos. Mit <code>git clone</code> wird automatisch ein Ordner mit diesem Namen im aktuellen Verzeichnis erstellt.
Typ	Hier soll Git ausgewählt werden.
Kontakt	Geben Sie hier Ihre HSR E-Mail Adresse an.
Beschreibung	Hier wäre eine kurze und prägnante Beschreibung ihres Projekts / Repos sinnvoll.
Öffentlich	Ist „Öffentlich“ markiert, kann dieses Repo von jeder Person mit Zugriff auf git.hsr.ch gelesen werden.

Nachdem ein Repository auf dem HSR Git-Server erstellt wurde, kann dieses mittels `git clone` mit einem lokalen Repo verknüpft werden. Dieses lokale Repo wird ebenfalls durch den `git clone` Befehl erstellt.

Hinweis:

Gemäss den Vorgaben der IT-Servicedesks müssen die Repos nach Beendigung des Projekts gelöscht werden. In diesem Falle wäre es sinnvoll, wenn Sie die in diesem Praktikum erstellten Repos nach Beendigung des Praktikums löschen würden.

Aufgabe 2: push

Für diese Aufgabe wird Aufgabe 1 vorausgesetzt.

Erstellen Sie einige Textdateien und comitten Sie diese. Anschliessend soll dieser Commit auf des remote Repo hochgeladen werden.

Aufgabe 3: neuen Branch erstellen

Für diese Aufgabe wird Aufgabe 2 vorausgesetzt.

Auf welchem Branch haben Sie die letzten Änderungen in Aufgabe 2 committet?

Erstellen Sie einen neuen Branch mit dem Namen `featureA` und wechseln Sie auf diesen Branch. Ändern Sie einige getrakte Dateien. Comitten und pushen Sie diese anschliessend.

Aufgabe 4: zwei verschiedene Branches zusammenführen

Für diese Aufgabe wird Aufgabe 3 vorausgesetzt.

Schritt 1:

Fügen Sie die Branches `master` und `featureA` wieder zusammen. Dabei soll der `featureA` Branch in den `master` Branch eingefügt werden.

Schritt 2:

Wechseln Sie wieder auf den `featureA` Branch und machen Sie ein Commit. Achten Sie darauf, dass Sie vor diesem Commit nur eine Datei verändern. Wechseln Sie auf den `master` Branch. Führen Sie nun auch auf dem `master` Branch ein Commit aus. Achten Sie darauf, dass die Änderungen nicht im gleichen Bereich gemacht werden wie auf dem `featureA` Branch (einige Zeilen Abstand). Mergen Sie anschliessend den `featureA` Brach wieder in den `master` Branch. Das Mergen der Braches sollte ohne Probleme funktionieren.

Falls nicht, sind die Änderungen der Datei bei den beiden Commits (*master* und *featureA* Branch) zu ähnlich. Brechen Sie entweder den Merge mit `git merge --abort` ab oder gehen Sie direkt zu Schritt 3 ohne den zweiten Schritt nochmals zu wiederholen.

Schritt 3:

Wiederholen Sie Schritt 2. Achten nun darauf, dass bei den jeweiligen Commits auf dem *master* und *featureA* Branch sehr ähnliche Änderungen gemacht werden, um ein Mergekonflikt zu provozieren. Beim Mergen des *featureA* Branches in den *master* Branch wird es entsprechend ein Konflikt geben. Lösen Sie den Konflikt mittels dem Git Mergetool `git mergetool`.

Aufgabe 5: Merge Konflikt von binären Dateien

Erstellen Sie eine binäre Datei und fügen Sie diese zur Versionsverwaltung hinzu. Unter Linux können Sie z.B. mit folgendem Befehl drei Bytes mit den Werten 0x3, 0x2 und 0x1 in eine Datei schreiben:

```
echo -n $'\x03\x02\x01' > binary.dat
```

Weitere Schritte:

1. Committen Sie diese Datei.
2. Gehen Sie um den gerade erstellten zurück und erstellen Sie einen neuen Branch.
3. Erstellen Sie ebenfalls eine binäre Datei mit demselben Namen, jedoch mit anderem Inhalt als die bereits erstellte.
4. Committen Sie diese Datei.
5. Wechseln Sie auf den ursprünglichen Branch zurück und mergen Sie mit dem eben neu erstellten Branch.
6. Der Merge wird fehlschlagen. Lösen Sie diesen Konflikt, in dem Sie eine von beiden Versionen übernehmen (*ours* oder *theirs*).

Aufgabe 6: Stash

In dieser Aufgabe sollen Änderungen zwischengespeichert werden und zu einem späteren Zeitpunkt (z.B. nach einem commit) wieder geladen werden. Die Theorie dazu wurde nicht in der Vorlesung behandelt. Informationen zum Stashen finden Sie im "Pro Git" Buch auf Seite 266.

Die folgenden Schritte geben ein mögliches Vorgehen vor:

1. Erstellen Sie zwei Dateien mit dem Namen Hallo.txt und Welt.txt.
2. Schreiben Sie irgendetwas in die Datei und committen Sie die beiden Files
3. Ändern Sie den Inhalt der Dateien Hallo.txt und Welt.txt etwas ab.
4. Stagen Sie die Datei Welt.txt
5. Zwischenspeichern Sie nun die Änderungen mittels dem `stash` Befehl
6. Erstellen Sie eine neue Datei mit dem Namen PMSwEng.txt und schreiben Sie irgendein Text in die Datei.
7. Ändern Sie die Datei Hallo.txt erneut
8. Holen Sie die *gestasheten* Änderungen aus dem Zwischenspeicher zurück
→ das wird nicht funktionieren
9. Sie müssen zuerst die aktuellen Änderungen committen oder verwerfen bevor die *gestasheten* Änderungen zurückgeholt werden können. In diesem Beispiel sollen die Änderungen verworfen werden. Dies betrifft nur die Datei Hallo.txt, da die Änderungen im Working Directory überschrieben würden. PMSwEng.txt ist nicht von Git getrackt.
10. Anschliessend können Sie nun die Änderungen wieder aus dem Stash-Speicher holen

Aufgabe 7: commit - pull - push

Klonen Sie ein Repo eines anderen Praktikum-Teilnehmers. Dazu muss dieser die Rechte für Ihren Zugriff auf das entsprechende Repo hinzufügen. Dies kann ebenfalls über das HSR Git-Portal erfolgen.

Arbeiten Sie nun gemeinsam im gleichen Branch (typischerweise *master* Branch). Dieser Workflow wird in der Praxis oft angewendet und ähnelt dem Workflow mit einem zentralisierten Repository. Beachten Sie dabei die folgenden Punkte:

1. Wenn Sie Änderungen im Working Directory haben, die Sie auf das remote Repo legen möchten, ist es wichtig, dass Sie die Änderungen zuerst committen.
2. Danach müssen allfällige Commits des remote/master Branches zuerst gefetched und gemerged werden. Dies geht am schnellsten mit dem `git pull` Befehl.
3. Anschliessend kann der lokale Commit mit dem `git push` Befehl auf das remote Repo übertragen werden.

Sprechen Sie sich mit Ihrem Arbeitspartner ab und spielen Sie die oben genannten Schritte ein paar Mal durch. Was passiert, wenn ein `pull` vor einem `commit` gemacht wird, wenn bereits Änderungen im Working Directory vorhanden sind?

Aufgabe 8: VoltageDivider Anwendung um Dokumentation erweitern

Dokumentieren Sie Ihr Programm mit Doxygen inklusive Mainpage. Verwenden Sie dazu die VoltageDivider Anwendungs-Lösung vom letzten Praktikum.

Thema, Ziele: Testen allgemein

Aufgabe 1: Rabatt Rechnung Äquivalenzklassen

Ein Supermarkt ist an den Wochentagen (Mo.-Sa.) von 08:00 bis 20:00 Uhr geöffnet. Ein Kassenprogramm in dem Supermarkt hat folgende Anforderungen:

1. Ab einem Rechnungsbetrag von insgesamt CHF 150 wird ein Rabatt von CHF 10 gewährt.
2. Da es morgens immer ziemlich leer im Supermarkt ist, soll ein weiterer Rabatt für grösseren Umsatz sorgen: Auf alle Waren, die morgens bis 10:00 Uhr eingekauft werden, wird ein Rabatt von 5% gewährt (Frühkaufrabatt).
3. Der Frühkaufrabatt kann nicht mit dem Rechnungsbetrag Rabatt kombiniert werden .Es soll der für den Kunden günstigere der beiden Rabatte gelten.

Der für den Rabatt entscheidende Baustein des Kassenprogramms sei die Funktion:

```
size_t rabatt(size_t rechnungsbetrag, size_t stunde, size_t minute)
```

Der zurückgegebene Rabatt und der Rechnungsbetrag werden als Rappen-Beträge angesehen, weil es keine kleinere Einheit gibt. Der Datentyp `size_t` wird gewählt, weil mit ihm die grösstmöglichen unsigned-Zahlen darstellbar sind. Eine mögliche Ursache für Fehler, die durch eine Typumwandlung entstehen, entfällt dadurch. Aus Plausibilitätsgründen wird eine Obergrenze von CHF 10'000 festgelegt. Ein höherer Wert gilt als Fehler. Pfandgeld wird nicht berücksichtigt, deshalb muss der Rechnungsbetrag > CHF 0 sein.

Definieren Sie alle Äquivalenzklassen (gültig und ungültig).

Aufgabe 2: Rabatt Rechnung Testfälle mit Grenzwertanalyse

Definieren Sie konkrete Testfälle (Input und Output) für die in Aufgabe 1 gebildeten Äquivalenzklassen mittels Grenzwertanalyse.

Falls ungültigen Äquivalenzklassen wird eine Exception von der Funktion `rabatt` geworfen.

Hinweis: Das Makro `SIZE_MAX` aus dem Header `stdint.h` repräsentiert der maximale Wert welcher mit `size_t` dargestellt werden kann.

Aufgabe 3: Testfälle für `VoltageDivider` Anwendung

Definieren Sie alle Testfälle für die Funktionen `VoltageDivider::calc` (siehe Vorlage `VoltageDivider.h`). Verwenden Sie dazu Äquivalenzklassen und die Grenzwertanalyse. Die resultierenden Testfälle dienen als Vorlage für das nächste Praktikum, damit automatisiert getestet werden kann.

Hinweise:

- Das Makro `DBL_MAX` aus dem Header `cfloat` repräsentiert der maximale Wert welcher mit `double` dargestellt werden kann.
- Die Funktion `nextafter` gibt der nächste Repräsentant eines `double` Wertes zurück. Diese Funktion ist im `cmath` Header.

Thema, Ziele: automatisierte Unit Tests

Die Vorlage für alle folgenden Aufgaben ist im Ordner *Vorlage* dieses Praktikums zu finden.

Aufgabe 0: Qt Projekt inkl. googletest für Till Unit Test einrichten

Für den automatisierten Test müssen Sie zuerst ein neues Qt Projekt mit dem Namen *RabattTest* erstellen. Anders als gewohnt reicht hier das Projekt Template *Qt Console Application*. Anschliessend soll die *googleTest* Library zum Projekt hinzugefügt werden.

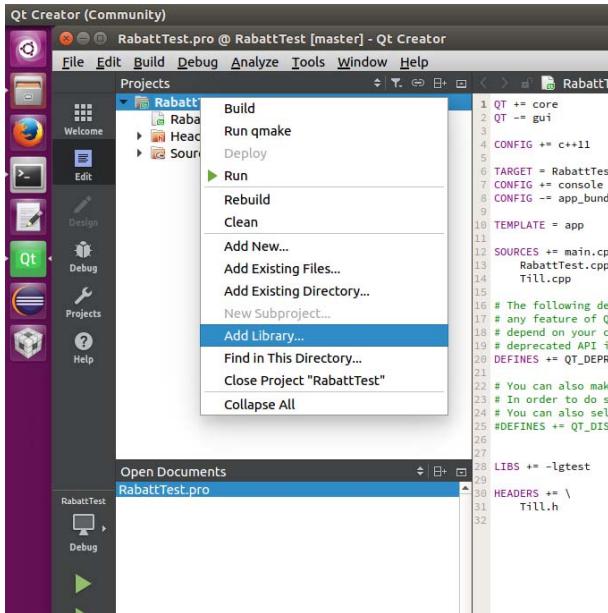


Abbildung 1 Add Library Kontextmenü in QtCreator

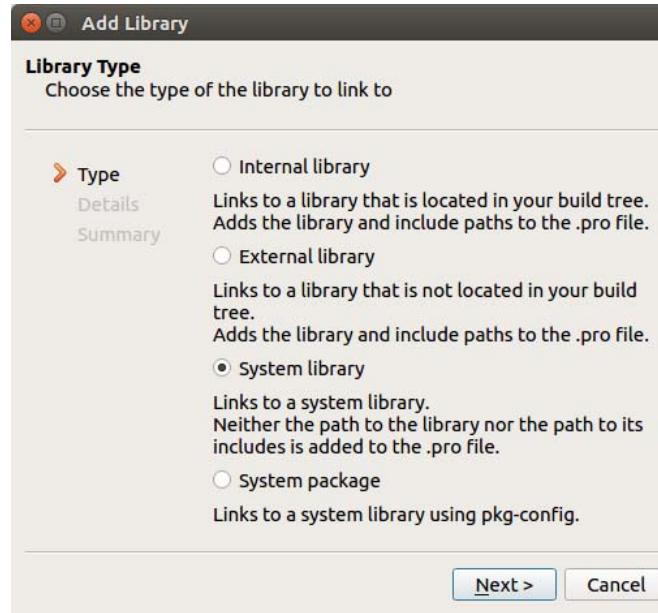


Abbildung 2 System Library auswählen

Die folgende Anleitung zeigt, wie die GoogleTest Library zum Projekt hinzugefügt wird:

Machen Sie zuerst einen Rechtsklick auf Ihrem Projekt (siehe Abbildung 1). Nach Abbildung 2 drücken Sie auf *Next >*. Wählen Sie die *gtest* Library aus. Sie finden Sie im GoogleTest Ordner im Pfad */usr/local/lib/libgtest.a*.

Der Pfad zu den Headerfiles ist für System Libraries bereits hinterlegt. Hier müssen keinen zusätzlichen Pfad angeben.

Dem Projekt kann die Klasse *Till* hinzugefügt werden. Dies kann über *Add Existing Files...* über das Kontextmenü in Abbildung 1 gemacht werden.

Das Test main.cpp File könnte wie folgt aussehen:

```
///\file main.cpp
///\date 30.11.2016
///\author Michael Trummer

#include "VoltageDividerTest.h"
#include "EDecadeTest.h"
#include "VoltageDividerWidgetTest.h"

#include <gtest/gtest.h>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); //wird bei Verwendung von QWidgets benötigt

    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Aufgabe 1: Till Unit Test schreiben

Erstellen Sie ein weiteres Cpp File mit dem Namen *TillTest.cpp* und fügen Sie es zum Projekt hinzu.
Der Inhalt der Datei kann wie folgt beginnen:

```
#include <gtest/gtest.h>
#include "Till.h"
TEST(TillTest, rabatt_iec11_1)
{
    EXPECT_DEATH(Till::rabatt(10001, 0, 0), "");
}

// ... weitere TESTs...
```

Schreiben Sie den Unit Test für die *Till* Klasse. Da die *Till* Klasse nur die Rabatt Funktion enthält, beschränkt sich dieser Unit Test auf das Testen der *rabatt* Funktion.

Implementieren Sie alle Testfälle, die im letzten Praktikum für die *rabatt* Funktion mittels Äquivalenzklassen und Grenzwertanalyse erstellt haben.

Aufgabe 2: Till Unit Test Code Coverage

Nachdem das Test-Programm von Aufgabe 1 ausgeführt werden kann, kann die Coverage mittels folgenden Schritten angezeigt werden:

- Mit Qt kann das Coverage Tool gcov in Zusammenhang mit lcov genutzt werden. Um dieses Tools einfach zu nutzen ist ein Script mit dem Namen *generateCoverageView.sh* im Übungsverzeichnis abgelegt. Dieses kann verwendet werden, um direkt eine HTML-Seite mit den Coverage-Informationen zu erhalten.
- Im .pro-File müssen die entsprechenden Flags gesetzt werden und die gcov Library muss dazu gelinkt werden. Das sieht wie folgt aus:

```
#GCOV Settings
QMAKE_CXXFLAGS += -fprofile-arcs -ftest-coverage
QMAKE_LDFLAGS += -fprofile-arcs -ftest-coverage
LIBS += -lgcov
```

- Abschliessend kann das Script wie folgt aufgerufen werden
./generateCoverageView.sh RabattTest

Im Firefox wird Ihnen die Coverage von beteiligten File angezeigt. Überprüfen Sie die Coverage von der Klasse *Till*. Die Coverage wird nicht 100 % betragen. Ergänzen Sie die Tests für die *Till* Klasse, damit Sie 100 % Coverage erreichen.

Aufgabe 3: Author Mock für Unit Test in Library Projekt

Bei der nächsten Aufgabe soll die Klasse *Book* (siehe Vorlageordner) mittels Unit Test getestet werden. Damit *Book* ohne Abhängigkeiten zu der Klasse *Author* getestet werden kann, muss *Author* mittels Interface *IAuthor* zu *Book* getrennt werden.

Erstellen Sie das Interface und passen Sie *Author* so an, dass die Klasse *Author* von *IAuthor* ableitet. Übergeben Sie dem Ctor von *Book* neu eine Referenz auf ein *IAuthor* Object anstatt einem *Author* Object.

Bringen Sie das Projekt entsprechend wieder zum Laufen, damit der gleiche Konsolenoutput wie vorher ausgegeben wird.

Aufgabe 4: Book Unit Test in Library Projekt

Erstellen Sie einen Author Mock mit dem Namen *AuthorMock* im Test Projektpfad. Die Klasse *AuthorMock* soll von *IAuthor* ableiten.

Erstellen Sie anschliessend Anweisungsüberdeckungs-Testfälle für die Klasse *Book*. Verwenden Sie ein Objekt von *AuthorMock* anstatt von *Author*.

Weitere Informationen: Verwenden von gcov in Eclipse

In Eclipse ist gcov bereits mittels Plugin eingebunden. Es kann über das in der untenstehenden Abbildung markierte Icon konfiguriert werden (siehe Abbildung 3).



Abbildung 3 Icon um Profile zu konfigurieren

Danach *Profile Configurations* auswählen.

In der linken Spalte kann das gewünschte Projekt ausgewählt werden, und im Tab *Profile* kann gcov ausgewählt werden (siehe Abbildung 4).

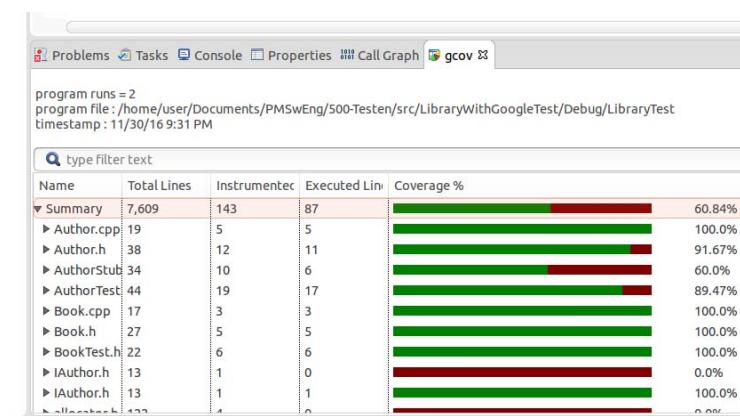
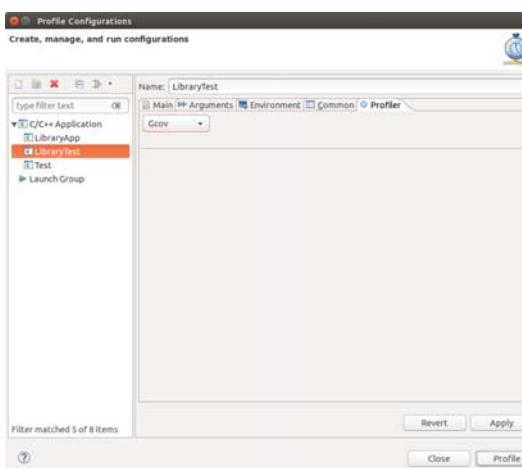
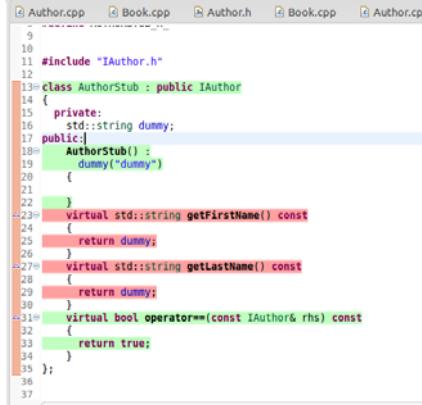


Abbildung 5 Gcov coverage output

Abbildung 4 Gcov als Profile auswählen

Nach dem Bestätigen wird das Programm inklusive gcov ausgeführt. Es erscheint ein neuer Tab im unter Bereich, welcher das Resultat von gcov präsentiert (siehe Abbildung 5).

Werden die Files über den *Project Explorer* im linken Bereich von Eclipse ausgewählt, werden die Codezeilen entsprechend der Anweisungsüberdeckung eingefärbt (siehe Abbildung 6)



```
#include "IAuthor.h"
class AuthorStub : public IAuthor
{
private:
    std::string dummy;
public:
    AuthorStub() :
        dummy("dummy")
    {
    }
    virtual std::string getFirstName() const
    {
        return dummy;
    }
    virtual std::string getLastName() const
    {
        return dummy;
    }
    virtual bool operator==(const IAuthor& rhs) const
    {
        return true;
    }
};
```

Abbildung 6 Eingefärbte Codezelle durch gcov