Embedded Software Engineering 1

HS 2024 – Prof. Reto Bonderer

Autoren: Laurin Heitzer, Simone Stitz

https://github.com/P4ntomime/EmbSW1

V1.0.20241115

Inhaltsverzeichnis

1 Embedded Systems – Allgemein

1.1 Definition

Ein Embedded System...
- ist ein System, das einen Computer beinhaltet, selbst aber kein Computer ist
- besteht üblicherweise aus Hardware (Mechanik, Elektronik) und Software
- ist sehr häufig ein Control System (Steuerung, Regelung)

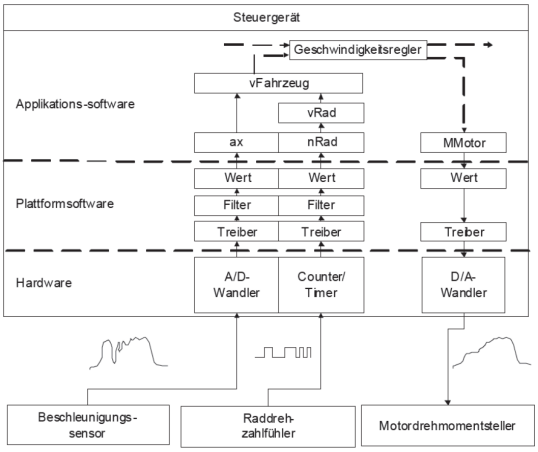
Ein Embedded System beinhaltet typischerweise folgende Komponenten:
- Sensoren
- Aktoren
- Mikrocomputer
- Software (Firmware)
- Hardware (Mechanik, Elektronik)

1.1.1 Charakterisierung von Embedded Systems

Embedded Systems können **(müssen aber nicht)** folgende Eigenschaften haben:
- **reactive systems:** Reaktive Systeme interagieren mit ihrer Umgebung
- **real-time systems:** Echtzeitsysteme haben nebst funktionale Anforderungen auch definierbaren zeilichen Anforderungen zu genügen
- **dependable systems:** Verlässliche Systeme sind Systeme, welche (sehr) hohe Zuverlässigkeitsanderungen erfüllen müssen
- **Weitere (häufige) Anforderungen:**
  - kleiner Energieverbrauch
  - kleine physikalische Abmessungen
  - Lärm, Vibration, etc.

1.1.2 Typischer Aufbau

Ein gutes Design beinhaltet unterschiedliche Abstraktionsschichten ➔ Layer
➔ Siehe Abschnitt 1.8



1.2 Beispiele

Fahrrad-Computer
- GPS-Navigation
- Geschwindigkeits- und Trittfrequenzmessung
- Pulsmesser
- Drahtlosübertragung (ANT+)
- Interface zu elektronischer Gangschaltung
- Barometer, Thermometer
- Trainingsassistent
- Display

Weitere Beispiele
- Smartphone
- Mobile Base Station
- CNC-Bearbeitungszentrum
- Hörgerät

Auto
- Sicherheitsrelevante Aufgaben
  - ABS, ASR
  - Motorenregelung
  - Drive-by-wire
  - Autonom fahrende Autos
- Unterhaltung / Komfort
  - Radio / CD / etc.
  - Navigation
  - Klima
- Mehrere Netzwerke
  - CAN, LIN, Ethernet
- Echtzeitteile und andere
- Von einfachsten μCs bis DSPs und GPUs

➔ Auto ist ein riesiges Embedded System

1.3 Deeply Embedded System

- 'Einfaches' Embedded System, mit **minimaler Benutzerschnittstelle**, üblicherweise mit **keinerlei GUI** und **ohne Betriebssystem**
- Beschränkt auf **eine** Aufgabe (z.B. Regelung eines physikalischen Prozesses)
- Muss oft zeitliche Bedingungen erfüllen ➔ Echtzeitsystem

1.3.1 Beispiele – Deeply Embedded System

- Hörgerät
- Motorenregelung
- ABS-Controller
- 'Sensor' im IoT
- etc...

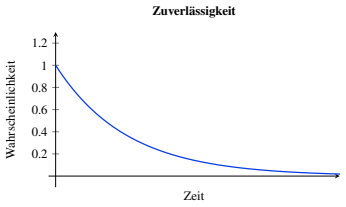
1.4 Betriebssysteme bei Embedded Systems

- Es kommen Betriebssysteme wie (Embedded) Linux oder Android zum Einsatz
  ➔ **Achtung: Linux und Android sind nicht echtzeitfähig!**
- Wenn Echtzeit verlangt wird: real-time operating systems (RTOS)
  - Beispiele: Zephyr, Free RTOS (Amazon), TI-RTOS (Texas Instuments), etc.

1.5 Bare Metal Embedded System

- Es kommt **keinerlei Betriebssystem** zum Einsatz
- Bare Metal Embedded Systems sind recht **häufig**, insbesondere bei **Deeply Embedded Systems**
- Bare Metal Embedded Systems stellen besondere Ansprüche an Programmierung

1.6 Zuverlässigkeit



- Je länger das System läuft, desto weniger zuverlässig ist es
- Die Wahrscheinlichkeit für einen Ausfall steigt stetig

Achtung: Hier ist nur die Alterung der Hardware berücksichtigt

1.7 Verfügbarkeit

Die Verfügbarkeit A (Availability) ist der Anteil der Betriebsdauer innerhalb dessen das System seine Funktion erfüllt.

$$\text{Verfügbarkeit} = \frac{\text{Gesamtzeit} - \text{Ausfallzeit}}{\text{Gesamtzeit}}$$

1.8 Abstraktionsschichten

- Bei µC-Programmierung (Firmware) müssen oft Bitmuster in Register geschrieben werden
- Solche Register-Zugriffe dürfen **nicht** 'willkürlich' überall im Code erfolgen
  ➔ schlecht lesbar, schlecht portiertbar, fehleranfällig
- **Damit Code lesbarer und besser auf andere Platform portierbar wird, beinhaltet jeder professionelle Code einen Hardware Abstraction Layer (HAL)**
- HAL führt **nicht** zum Verlust bei Laufzeit, wenn korrekt implementiert

1.8.1 Hardware-abstraction-layer (HAL)

- Trennt HW-Implementierung von SW-Logik
- Gleiche SW kann auf verschiedene HW verwendet werden ➔ Portabilität
- HW-Komponenten können einfach ausgetauscht werden ➔ Flexibilität

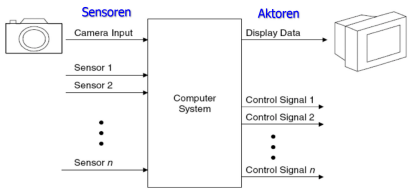
2 Real-Time System (Echtzeitsystem)

2.1 Definitionen

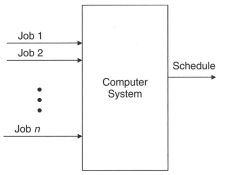
2.1.1 Real-Time System (Echtzeitsystem)

- Ein Echtzeitsystem ist ein System, das Informationen **innerhalb einer definierten Zeit (deadline) bearbeiten** muss.
  ➔ Explizite Anforderungen an **turnaround-time** (Antwortzeit) müssen erfüllt sein
- Wenn diese Zeit nicht eingehalten werden kann, ist mit einer **Fehlfunktion** zu rechnen.

Typisches Echtzeitsystem

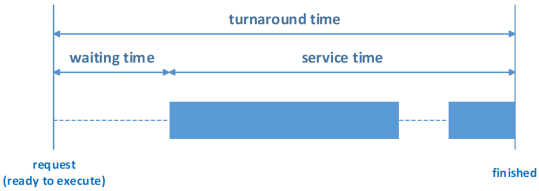


Repräsentation RT-System



Sequenz von Aufgaben (Jobs) müssen zeitlich geplant (scheduled) werden

2.1.2 Zeitdefinitionen (Task)



- **turnaround time:** (response time, Antwortzeit)
  - Startet, wenn der Task bereit zur Ausführung ist und endet, wenn der Task fertig abgearbeitet ist
  - Zeit zwischen dem Vorhandensein von Eingangswerten an das System (Stimulus) bis zum Erscheinen der gewünschten Ausgangswerte.
- **waiting time:** (Wartezeit)
  - Zeit zwischen Anliegen der Eingangswert und Beginn der Abarbeitung des Tasks
- **service time:** (Bearbeitungszeit)
  - Zeit für Abarbeitung des Tasks ➔ Unterbrechungen bzw. (preemptions) möglich

2.2 Fehlverhalten eines Systems (failed system)

- Ein fehlerhaftes System (failed system = missglücktes System) ist ein System, das nicht alle formal definierten Systemspezifikationen erfüllt.
- **Die Korrektheit eines RT Systems bedingt sowohl die Korrektheit der Outputs als auch die Einhaltung der zeitlichen Anforderungen.**

2.3 Echtzeitdefinition – Verschiedene Echtzeitsysteme

- **soft real-time system** (weiches Echtzeitsystem)
  - Durch Verletzung der Antwortzeiten wird das System **nicht** ernsthaft beeinflusst
  - Es kommt zu Komforteinbussen
- **hard real-time system** (hartes Echtzeitsystem)
  - Durch Verletzung der Antwortzeiten wird das **System ernsthaft beeinflusst**
  - Es kann zu einem kompletten Ausfall oder katastrophalem Fehlverhalten kommen
- **firm real-time system** (festes Echtzeitsystem)
  - Kombination aus soft real-time system und hard real-time system
  - Durch Verletzung einiger weniger Antwortzeiten wird das System nicht ernsthaft beeinflusst
  - Bei vielen Verletzungen der Antwortzeiten kann es zu einem kompletten Ausfall oder katastrophalem Fehlverhalten kommen

2.3.1 Beispiele verschiedener Echtzeitsysteme

System	Klassifizierung	Erlärung
Geldautomat	soft	Auch wenn mehrere Deadlines nicht eingehalten werden können, entsteht dadurch keine Katastrophe. Im schlimmsten Fall erhält ein Kunde sein Geld nicht.
GPS-gesteuerter Rasenmäher	firm	Wenn die Positionsbestimmung versagt, könnte das Blumenbeet der Nachbarn platt gemäht werden.
Regelung eines Quadrocopters	hard	Das Versagen der Regelung kann dazu führen, dass der Quadrocopter ausser Kontrolle gerät und abstürzt.

2.4 Determinsismus (determinacy)

Ein System ist deterministisch, wenn für jeden möglichen Zustand und für alle möglichen Eingabewerte **jederzeit der nächste Zustand und die Ausgabewerte definiert** sind.

Insbesondere race conditions können dazu führen, dass der nächste Zustand davon abhängt, 'wer das Rennen gewonnen hat und wie gross die Bestzeit ist', d.h. der nächste Zustand ist nicht klar bestimmt.
➔ Nicht mehr deterministisch und nicht mehr echtzeittauglich

2.5 Auslastung (utilization)

Die (CPU-) Auslastung (utilization) ist der Prozentsatz der Zeit, zu der die XPU **nützliche (non-idle) Aufgaben** ausführt.

2.5.1 Berechungen zur Auslastung (utilzation)

Annahmen:
- System mit $n \geq 1$ periodischen Tasks T_i und Periode p_1
- Jeder Task T_i het bekannte / geschätzte maximale (worst case) execution time e_i

Auslastungsfaktor eines Tasks

$$u_i = \frac{e_i}{p_i}$$

➔ utilization factor

Gesamtauslastung des Systems

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

➔ utilization

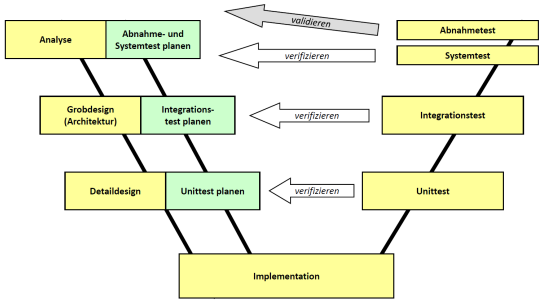
➔ Bei 69 % Auslastung ist das **'theoretical limit'**

2.6 Real-time Scheduling

- Alle kritischen Zeiteinschränkungen (deadlines, response time) sollen eingehalten werden
- Im Notfall muss der Scheduling Algorithmus entscheiden, um die kritischsten Tasks einhalten zu können.
  - Unter Umständen müssen dabei Deadlines von weniger kritischen Tasks verletzt werden.

3 Modellierung eines Embedded Systems

3.1 V-Modell für Software-Entwicklungszyklus



➔ Nur Anforderungen (requirements) definieren, welche man auch testen kann!

3.2 Model Driven Development (MDD)

- Bei **modellbasierter Entwicklung** kommen in **allen Entwicklungsphasen** durchgängig Modelle zum zur Anwendung
- MDD geht davon aus, dass aus formalen Modellen lauffähige Software erzeugt wird ➔ Codegeneratoren
- Modelle werden traditionell als Werkzeug der Dokumentation angesehen
  - Unter Umständen wird zweimal dasslbe beschrieben (Code und Diagramm) ➔ **unbedingt zu vermeiden!**

3.3 Vorgehen bei der Modellierung

1. Systemgrenze definieren
   - Kontextdiagramm: Use-Case-Diagramm
   - Kontextdiagramm: Sequenzdiagramm
2. Systemprozess finden
   - Kontextdiagramm: Use-Case-Diagramm
   - Kontextdiagramm: Sequenzdiagramm
3. Verteilungen festlegen
   - Verteilungsdiagramm (deployment diagram)
4. Systemprozesse detaillieren
   - Umgangssprachlicher Text
   - Sequenzdiagramm
   - Aktivitätsdigramm
   - Statecharts
   - Code (C, C++, ...)

Stukturmodellierung (Statische Aspekte)

Modellierung der dynamischen Aspekte

3.4 Systemgrenze definieren & Systemprozesse finden

3.4.1 Systemgrenze definieren

Die Festlegung der Systemgrenze ist das Wichtigste und Allererste bei sämtlichen Systemen!
Man sollte sich die folgenden Fragen stellen und diese beantworten:

- Was macht das System, d.h. was liegt innerhalb der Systemgrenze?
  - Was macht das System **nicht**?
- Mit welchen Teilen ausserhalb des Systems kommuniziert das System?
- Welches sind die Schnittstellen zu den Nachbarsystemen (Umsystemen, periheral system)?

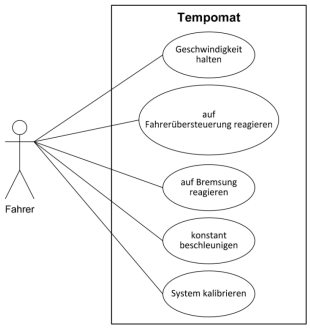
3.4.2 Systemprozesse finden (use-cases)

Da man sich noch immer in der **Analyse** befindet, sollen nur die **Anforderungen** definiert werden. Die Umsetzung ist Teil des Designs!
Um die Use-Cases zu identifizieren, sollte folgendes beachtet werden:

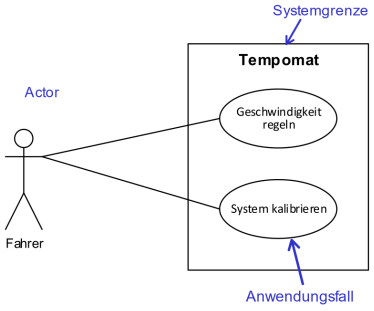
- Aussenbetrachtung des Systems (**oberflächlich!**)
  - Nicht komplizierter als nötig
- System als Blackbox betrachten
  - **Was** soll System können; (nicht: wie soll das System etwas machen)
- RTE-Systeme bestehen häufig aus nur einem einzigen Systemprozess
  - speziell wenn System 'nur' ein Regler ist

3.4.3 Kontextdiagramm: Use-Case Diagramm

Tempomat: zu detailliert



Tempomat: verbesserte Version



3.4.4 Kontextdiagramm: Sequenzdiagramm

- Speziell bei Syetemen, deren Grenzen durch **Nachrichtenflüsse** charakterisiert werden können
- Details zu Sequenzdiagrammen siehe Abschnitt 3.6.1

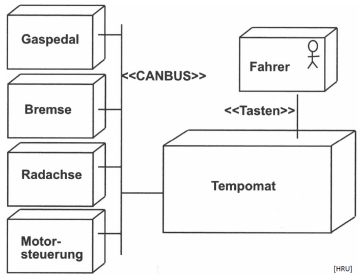
3.5 Verteilungen festlegen

- Bei Embedded Systems werden häufg **mehrere Rechnersysteme** verwendet, um die verschiedenen Aufgaben zu erledigen
- Rechner sind örtlich verteilt und mittels Kommunikationskanal verbunden ➔ **Verteilte Systeme (distributed systems)**

3.5.1 Verteilungsdiagramm

Knoten: Darstellung der örtlichen Verteilung der Systeme
Knoten können auch hierarchisch aufgebaut sein
Linien: Physikalische Verbindungen der Knoten (Netzwerke, Kabel, Wireless, etc.)

Beispiel: Tempomat

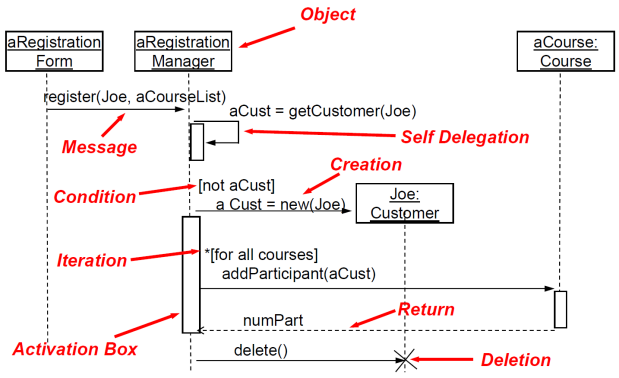


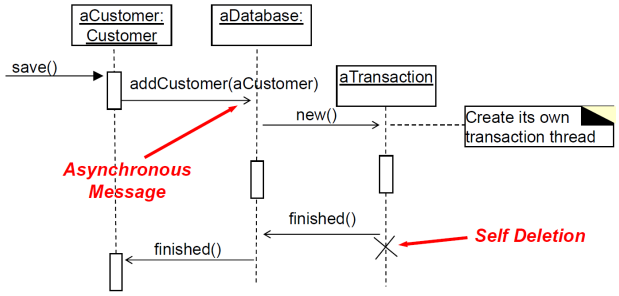
3.6 Systemprozesse detaillieren

- Die gefundenen Systemprozesse (use-cases) müssen genauer spezifiziert werden
  - **Nicht detaillierter spezifizieren als sinnvoll / gefordert!**
  - Jede weitere Spezifizierung soll eien 'added value' liefern
- Verschiedene Detaillierungsstufen für verschiedene Zielgruppen
  - Auftraggeber: Überblick (z.B. in Form von Umgangssprachlichem Text)
  - Systementwickler: 'Normale Sicht' enthält mehr Details

3.6.1 Sequenzdiagramm

- Gute Darstellung für **Austausch von Meldungen** zwischen Objekten innerhalb einer **beschränkten Zeitdauer**
  - Nachrichtenflüsse
  - Kommunikationsprotokolle
- Ideal für...
  - kurze Zeitdauer
  - wenige Objekte
  - wenige Verschachtelungen
  - wenige Verzweigungen



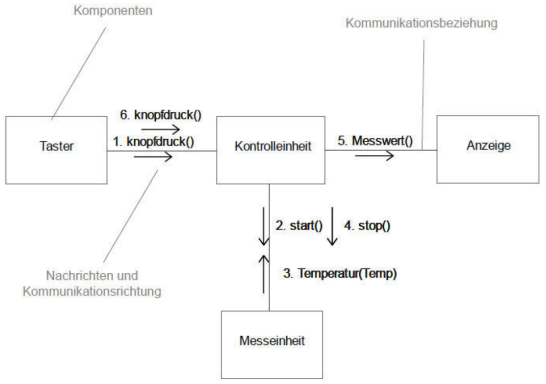


Pfeiltyp	Semantik
→	Synchrone Aufrufe
⇢	Asynchrone Nachrichten
- - - →	Datenfluss

- Beim Zeichnen von Hand unbedingt die **Pfeilkonventionen** beachten!
- Diagramme generell nicht 'überladen'

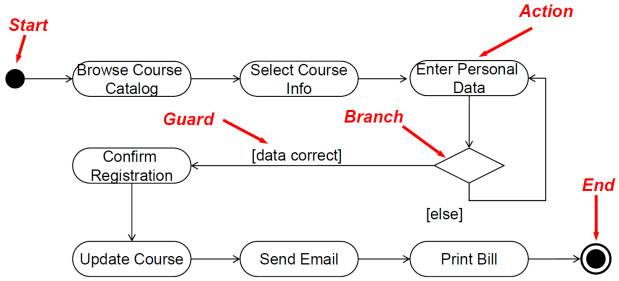
3.6.2 Kommunikationsdiagramm (Kollaborationsdiagramm)

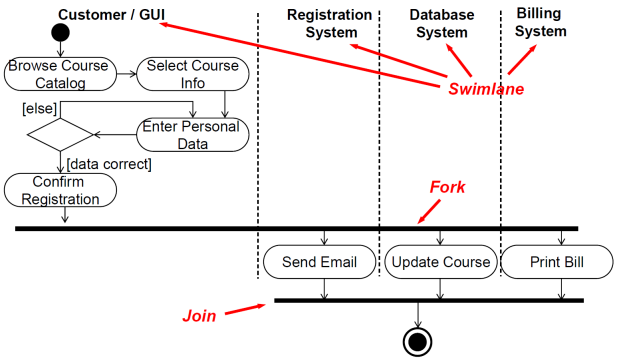
- Kommunikationsdiagramm zeigt **dieselbe Information** wie Sequenzdiagramm
- **Schwerpunkt: Informationsfluss** zwischen den Objekten
  ➡ Beim Sequenzdiagramm liegt der Schwerpunkt auf dem zeitlichen Ablauf



3.6.3 Aktivitätsdigramm

- Gut geeignet für ...
  - **workflow modelling**
  - Sequenzielle Abläufe
  - Prozess- und Steuerfluss
  - Gleichzeitige Prozesse (fork, join)
- Weniger geeignet für ...
  - komplexe logische Bedingungen





4 Hardware-Software-Codesign

4.1 Ziele

- Entwurf (Design) **so lange wie sinnvoll** (nicht so lange wie möglich) **lösungsneutral**
- **Systemdesign fördern**, statt separate Designs für Mechanik, Elektronik, Firmware, Software, etc., die sich unter Umständen auch widersprechen können
- Systemspezifikation erfolgt idealerweise mit Hilfe einer **eindeutigen Spezifikationssprache**, nicht in Prosa
- Die Spezifikation sollte simuliert (ausgeführt) werden können
- Implementationen können einfach geändert werden: HW ↔ SW
- Zielplattformen: diskrete Elektronik, ASIC, µC, DSP, **FPGA**, Software

4.2 Anforderungen für praktische Anwendungen

- Methoden / Tools sollten beim Systemdesign nicht zu fachlastig sein
  - Methoden sollten für Elektronik-, Firmware- und wenn möglich auch Mechanikentwickler anwendbar sein
- Wenn möglich gute Toolunterstützung
- (Automatische Synthese aus dem Modell)

4.3 Spezifikationssprachen

- **Formale Sprachen sind eindeutig** (Prosa immer mehrdeutig)
- Spezifikation kann compiliert und ausgeführt werden ➡ Simulationen
- Die ausführbare Spezifikation dient als **Golden Reference** für die künftigen Entwicklungsschritte

Beispiele für Spezifikationssprachen

- SystemC (eine C++-Template Library)
- SysML
- SpecC
- SystemVerilog
- Esterel
- Matlab/Simulink
- Statecharts

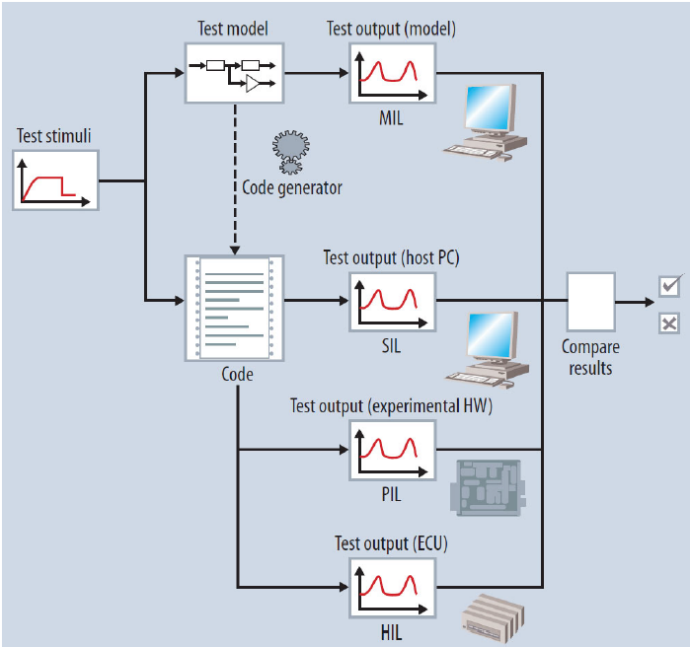
4.4 Virtuelle Prototypen

- Die Simulation des Systems kann unterschiedlich stark detailliert werden
- **Die simulierten Systeme sind Virtuelle Prototypen**
- Während der Entwicklung können einzelne (virtuelle) Teile des Prototyps laufend durch physische Teile ersetzt werden

4.5 X-in-the-loop

- **Model-in-the-Loop (MIL):** vollständig als Modell vorliegender virtuellen Prototyp
- Je mehr der Prototyp durch konkretere Implementationen ersetzt wird, spricht man von
  - Software-in-the loop (SIL)
  - Processor-in-the loop (PIL)
  - Hardwarein-the loop (HIL)

➡ Test outputs werden jeweils mit **Golden Reference** verglichen



4.6 Entwicklungsplattformen

Als Entwicklungsplattformen eignen sich häufig **FPGA basierte Systeme.**

- Hardware mit VHDL
- Software/Firmware in C/C++
  - auf integriertem µC (z.B. Zynq von AMD/Xilinx) (Hard core)
  - auf Soft Core innerhalb FPGA (z.B. Nios II von Intel/Altera)

5 Zustandsbasierte Systeme

5.1 Asynchrone vs. synchrone FSM

- **Asynchron**
  - geänderte Inputsignale führen **direkt** zur Zustandsänderung
  - schneller, aber enorm anfällig auf Glitches
- **Synchron**
  - Inputsignale werden nur zu diskreten Zeitpunkten betrachtet
    ➔ getaktete Systeme
- Softwareimplementationen sind eigentlich immer **synchron**, da Rechner getaktet sind
- Rein softwareseitig besteht die Problematik der Asynchronizität nicht

5.2 Finite State Machines (FSM)

Eine FSM besitzt die folgenden Eigenschaften:

- Eine FSM befindet sich immer in einem **definierten Zustand**
- Die **Inputs** X bezeichnen üblicherweise **Ereignisse (Events)**
- Die **Outputs** Y werden oft auch **Actions** genannt
- Eine FSM benötigt immer **Speicherelemente** zur Speicherung des internen Zustands
  - Eine FSM ist ein sequenzielles und kein kombinatorisches System

Eine FSM kann auf zwei Arten dargestellt werden:

- State-Event-Diagramm
- Zustandstabelle

5.2.1 Mealy-Automat

- Nächster Zustand Z_{n+1} abhängig vom Input X und vom internen Zustand Z_n
  - $Z_{n+1} = f(Z_n, X)$
- Output Y ist abhängig vom internen Zustand Z_n **und vom Input** X
  - $Y = g(Z_n, X)$
- Actions liegen bei den Transitionen

5.2.2 Moore-Automat

- Nächster Zustand Z_{n+1} abhängig vom Input X und vom internen Zustand Z_n
  - $Z_{n+1} = f(Z_n, X)$
- Output Y ist **nur** abhängig vom internen Zustand Z_n
  - $Y = g(Z_n)$
- Actions liegen bei den Zuständen

➔ **Wenn immer möglich sollten Moore-Automaten verwendet werden**

5.2.3 Medvedev-Automat

- Nächster Zustand Z_{n+1} abhängig vom Input X und vom internen Zustand Z_n
  - $Z_{n+1} = f(Z_n, X)$
- Output Y entspricht **entspricht direkt** internem Zustand Z_n
  - $Y = Z_n$
- Actions liegen bei den Zuständen

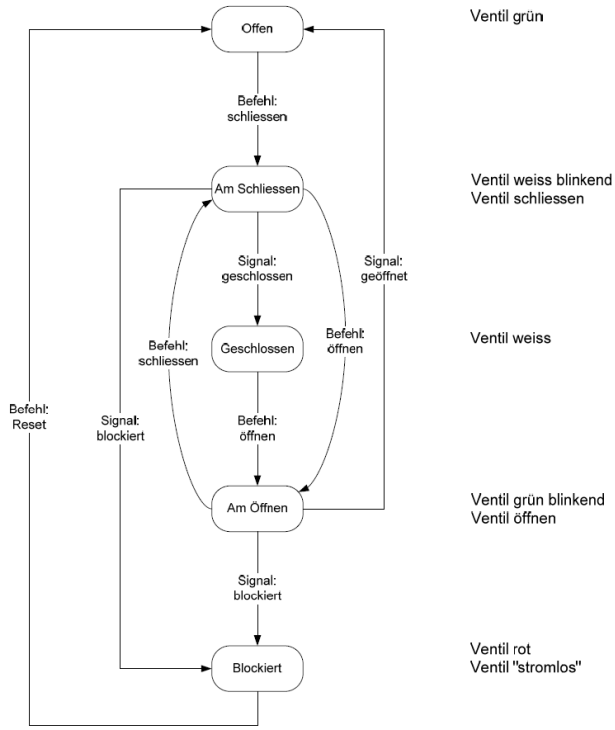
➔ Wird hier nicht weiter behandelt...

5.3 State-Event-Diagramm (Zustandsdiagramm)

Ein State-Event-Diagramm ist eine grafische Möglichkeit, um eine FSM zu beschreiben. In einem State-Event-Diagramm gelten folgende Darstellungsformen

- **Zustände** werden mit einem **Kreis** gezeichnet
- **Ereignisse** werden mit **Pfeilen** zwischen Zuständen dargestellt **(Transitionen)**
- **Aktionen** werden entweder bei Zuständen oder bei Transitionen geschrieben (je nach Automatentyp)
- Ausführung einer Transition ist unendlich schneller
  ➔ Bei Modellierung sind Zwischenzustäde vorgehesen, z.B. 'closing', starting up'

Beispiel: State-Event-Diagramm – Moore Automat



5.4 Zustandstabelle

Nebst der grafischen Darstellung einer FSM mittels State-Event-Diagramm kann die FSM auch tabellarisch mittels Zustandstabelle beschrieben werden.

Beispiel: Zustandstabelle für Elektromotor

Momentaner Zustand	Ereignis	Nächster Zustand	Aktionen
AUS	EIN-Taste	Hochlaufen	Motor ausschalten Kühlung ausschalten Grüne Lampe aus Rote Lampe aus
Hochlaufen	Drehzahl_erreicht Signal	Drehzahl_ok	Motor einschalten Kühlung einschalten
	Aus-Taste	AUS	
	Wasserkühlung Störung	Störung	
Drehzahl_ok	Wasserkühlung Störung	Störung	Grüne Lampe anzeigen
	AUS-Taste	AUS	
Störung	RESET-Taste	AUS	Motor ausschalten Kühlung ausschalten Rote Lampe anzeigen

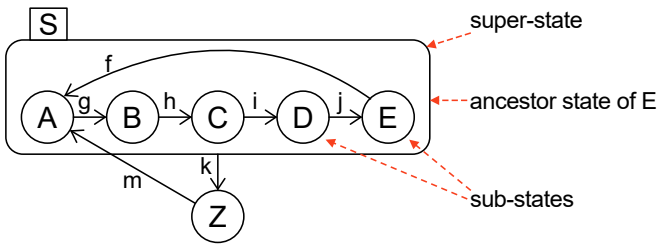
6 Statecharts (nach Marwedel)

6.1 Nachteile von State-Event-Diagrammen

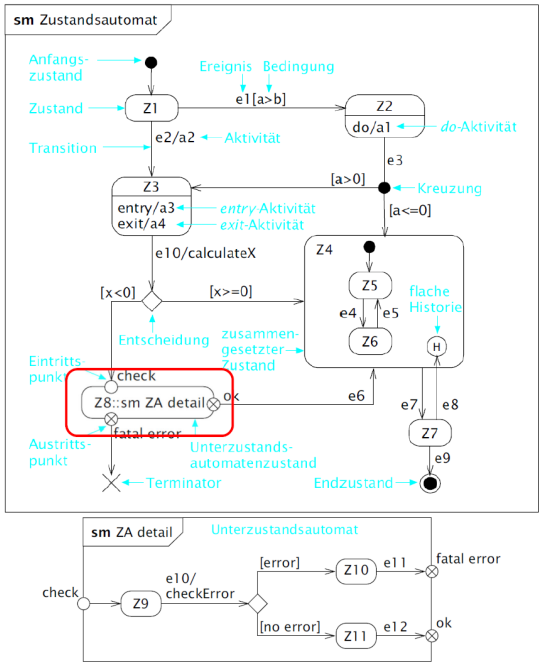
- Zustandsdiagramme sind flach (es gibt keine Hierarchie) ➔ schnell unübersichtlicht
- Es kann keine zeitliche Parallelität modelliert werden

6.2 Definitionen

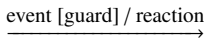
active state: Aktueller Zustand der FSM
basic states: Zustände, die **nicht** aus anderen Zuständen bestehen
super states: Zustände, die andere Zustände enthalten
ancestor states: Für jeden basic state s werden die super states, die s enthalten, als **ancestor states** bezeichnet
OR-super-states: Super-states S werden OR-super-states genannt, wenn **genau einer** der sub-states von S aktiv ist, wenn S aktiv ist ➔ Hierarchie
AND-super-states: Super-states S werden AND-super-states genannt, wenn **mehrere** der sub-states von S **gleichzeitig** aktiv sind, wenn S aktiv ist ➔ Parallelität
➔ Werden auch **Teilautomaten** genannt



6.2.1 Elemente der Statecharts



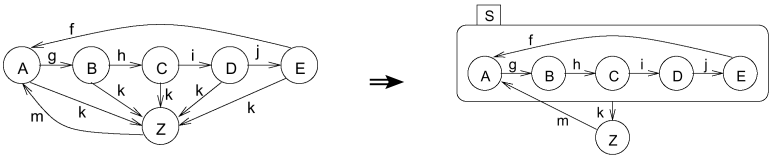
6.2.2 Allgemeine Syntax für Transitions-Pfeile



event auftretendes Event
guard Bedingung, welche zutreffen **muss**, damit Zustand überhaupt gewechselt wird
reaction Zuweisung einer Variablen / Erzeugung eines Events beim Zustandswechsel

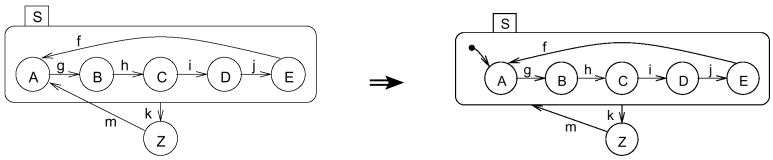
6.3 Hierarchie (OR-super-states)

- Die FSM befindet sich in **genau einem** sub-state von S, wenn S aktiv ist.
  (➔ either in A **OR** in B **OR** ...)



6.4 Default-State

- Ziel: Interne Struktur des states vor der Aussenwelt verstecken ➔ default state
- Ausgefüllter Kreis beschreibt den sub-state, welcher 'betreten' wird, wenn der super-state S 'betreten' wird



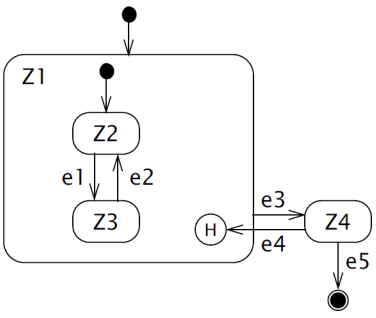
6.5 History

- Wenn Input m auftritt, wird in S derjenige sub-state betreten, in welchem man war, **bevor S verlassen wurde**
  - Wenn S zum ersten Mal betreten wird, ist der **default-Mechanismus** aktiv
- History und Default-Mechanismus können hierarchisch verwendet werden

6.5.1 Shallow History

- Der Histroy-Mechanismus merkt sich den entsprechenden sub-state
- Kennzeichnung: H

Beispiel: Shallow-History

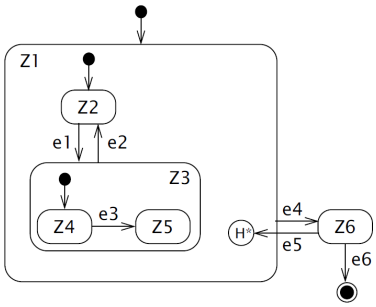


- In Zustand $Z3$ bewirkt der Event $e3$ einen Übergang zu $Z4$
- Der Event $e4$ führt nu zu einem Übergang zum früheren Zustand $Z3$

6.5.2 Deep History

- Der Histroy-Mechanismus merkt sich frühere Zustände **bis in die unterste Hierarchie**
- Kennezeichnung: H^*

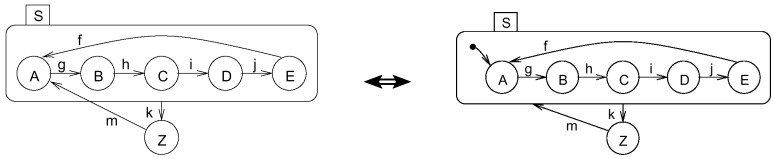
Beispiel: Deep-History



- Im Zustand $Z5$ bewirkt der Event $e4$ einen Übergang zu $Z6$
- Der Event $e5$ führt nun zu einem Übergang zum früheren Zustand $Z5$
  - Eine Shallow History würde bei $e5$ nur in den Zustand $Z3$, und damit in $Z4$, wechseln

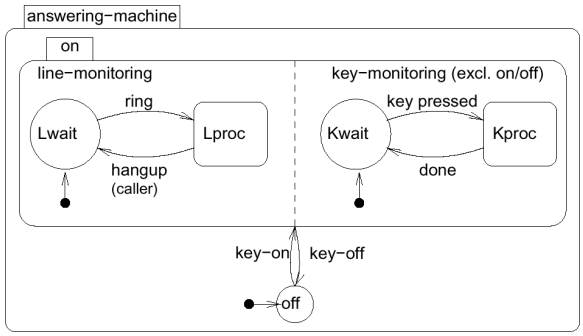
6.6 Kombination: History- und Default-Mechanismus

Folgende statecharts bilden genau das Gleiche ab



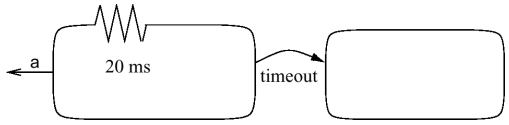
6.7 Parallelität (AND-super-state, Teilautomaten)

- Die FSM befindet sich in **allen** sub-states von einem super-state S, wenn S aktiv ist.
  (➔ in A **AND** in B **AND** ...)

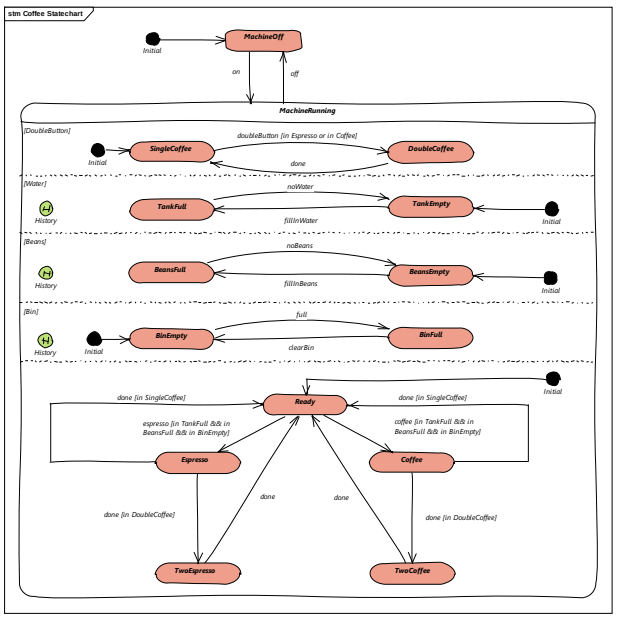


6.8 Timers

- Wenn Evend a nicht eintritt, während das System für 20 ms im linken state ist, wird ein timeout passieren
- Eigentlich sind Timers nicht nötig, da die Wartezeit auch als Übergangsbedingung (Ereignis) zwischen zwei states formuliert werden könnte



6.9 Beispiel – Armbanduhr als Statechart



7 Realisierung flache FSM

7.1 Mögliche Realisierungen von flachen FSMs

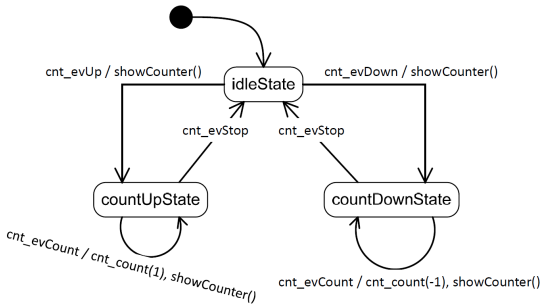
- Steuerkonstrukt (typischerweise mit **switch-case**)
  - prozedural oder objektorientiert
- Definition und Abarbeitung einer **Tabelle**
  - prozedural oder objektorientiert
- **State Pattern** (Gang of Four, GoF)
  - nur objektorientiert
- Generisch mit Templates
  - nur mit einer Sprache, die Templates unterstützt (z.B. C++)

Jede hierarchische FSM kann in eine flache FSM umgewandelt werden.

➡ Alle Varianten haben wie immer sowohl Vor- als auch Nachteile
➡ Bei allen Varianten sind auch Variationen vorhanden

7.2 Realisieurng mit Steuerkonstrukt (prozedural in C)

7.2.1 State-Event-Diagram – Up/Down-Counter



7.2.2 Implementation der Prozeduralen Realisierung in C

- **Ereignisse (events)**
  - Schnittstelle nach aussen ➡ ändern Zustand der FSM
  - In enum definiert (**public**) ➡ header-file
  - Einzelne Events und enum Bezeichnung enthalten **Unitkürzel** (hier: `cnt_`)
- **Zustände (states)**
  - In enum definiert (**nicht** public) ➡ sourcecode-file
- **Aktueller Zustand** wird in einer **statischen Varianlen** gehalten
- Die FSM wird in **zwei Funktionen** implementiert
  - Initialiserungs-Funktion (hier: `void cnt_ctrlInit(int initValue)`)
  - Prozess-Funktion (hier: `void cnt_ctrlProcess(cnt_Event e)`)
    ➡ Zustände prüfen, Zustandsübergänge veranlassen
- Anstossen einer FSM
  - Initialisierung in `main`-Funktion
  - Überprüfung, welches Event aufgetreten ist meist in `do-while`-Schleife

7.2.3 Eigenschaften der Prozeduralen Realisierung in C

- Da aktueller Zustand eine statische Variable ist, kann es nur **eine einzige Instanz** der FSM geben
- Bei mehreren Instanzen in C...
  - darf `currentState` nicht `static` sein und muss als Parameter mitgegenen werden, bzw. ein Pointer auf die jeweilige Variable
  - Zustands-enum muss in die Schnittstelle (header-file) oder es muss z.B. mit `void*` gearbeitet werden
- In C ist **keine schöne Kapselung** der Attribute möglich (`currentState`)
- Funktion `cnt_ctrlProcess()` kann beliebig aufgerufen werden (periodischer Task, laufend, etc.)
- Bei exponierten Funktionen / Definitionen muss in C ein Unitkürzel vorangestellt werden (hier: `cnt_`)

Beispiel: Up/Down-Counter (prozedural in C)

Schnittstelle Counter:

```c
// counter.h
// implements an up/down-Counter

#ifndef COUNTER_H__
#define COUNTER_H__

void cnt_init(int val);
// initializes counter to val

void cnt_count(int step);
// counts the counter up (step>0)
// or down (step<0) by step

int cnt_getCounter();
// returns the counter value

void cnt_setCounter(int val);
// sets the counter to val

#endif
```

Implementation Counter:

```c
// counter.c
// implements an up/down-Counter

#include "counter.h"

static int countValue;

void cnt_init(int val)
{
  countValue = val;
}

void cnt_count(int step)
{
  countValue += step;
}

int cnt_getCounter()
{
  return countValue;
}

void cnt_setCounter(int val)
{
  countValue = val;
}
```

Schnittstelle FSM:

```c
// counterCtrl.h
// implements the Finite State Machine (FSM) of an up/down-Counter

#ifndef COUNTERCTRL_H__
#define COUNTERCTRL_H__

typedef enum {cnt_evUp,       // count upwards
              cnt_evDown,     // count downwards
              cnt_evCount,    // count (up or down)
              cnt_evStop}     // stop counting
             cnt_Event;

void cnt_ctrlInit(int initValue);
// initializes counter FSM

void cnt_ctrlProcess(cnt_Event e);
// changes the state of the FSM based on the event 'e'
// starts the actions

#endif
```

Implementation FSM:

```c
// counterCtrl.c
// implements the Finite State Machine (FSM) of an up/down-Counter

#include <stdio.h>
#include "counterCtrl.h"
#include "counter.h"

typedef enum {idleState,        // idle state
              countUpState,     // counting up at each count event
              countDownState}   // counting down at each count event
             State;

static State currentState = idleState; // holds the current state of the FSM

void cnt_ctrlInit(int initValue)
{
  currentState = idleState;   // set init-state
  cnt_init(initValue);        // set initValue
}

void cnt_ctrlProcess(cnt_Event e)
{
  switch (currentState)
  {
    case idleState:
      printf("State: idleState\n");
      if (cnt_evUp == e)
      {
        // actions (and exit-actions from idleState)
        printf("State: idleState, counter = %d\n", cnt_getCounter());
        // state transition (and entry-actions from countUpState)
        printf("Changing to State: countUpState\n");
        currentState = countUpState;
      }
      else if (cnt_evDown == e)
      {
        // actions (and exit-actions from idleState)
        printf("State: idleState, counter = %d\n", cnt_getCounter());
        // state transition (and entry-actions from countDownState)
        printf("Changing to State: countDownState\n");
        currentState = countDownState;
      }
      break;

    case countUpState:
      printf("State: countUpState\n");
      if (cnt_evCount == e)
      {
        // actions
        cnt_count(1);
        printf("State: countUpState, counter = %d\n", cnt_getCounter());
        // state transition
      }
      else if (cnt_evStop == e)
      {
        // actions
        // state transition
        printf("Changing to State: idleState\n");
        currentState = idleState;
      }
      break;

    case countUpState:
      // ...
      break;

    default:
      break;
  }
}
```

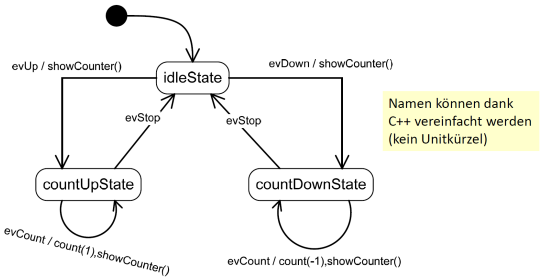
Anstossen der FSM:

```c
// counterTest.c
// Test program for the Finite State Machine (FSM) of an up/down-Counter

#include <stdio.h>
#include "counterCtrl.h"

int main(void)
{
  char answer;
  cnt_ctrlInit(0);

  do
  {
    printf("\n-----------------------------------------\n");
    printf("    u   Count up\n");
    printf("    d   Count down\n");
    printf("    c   Count\n");
    printf("    s   Stop counting\n");
    printf("    q   Quit\n");
```

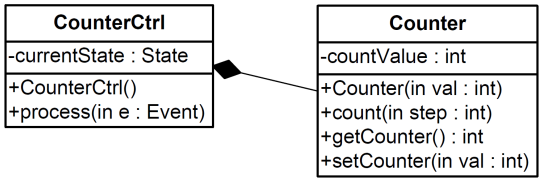
```
    printf("\nPlease press key: ");
    scanf("%c", &answer);
    getchar();  // nach scanf() ist noch ein '\n' im Inputbuffer: auslesen und wegwerfen
    printf("\n");

    switch (answer)
    {
      case 'u':
        cnt_ctrlProcess(cnt_evUp);
        break;
      case 'd':
        cnt_ctrlProcess(cnt_evDown);
        break;
      case 'c':
        cnt_ctrlProcess(cnt_evCount);
        break;
      case 's':
        cnt_ctrlProcess(cnt_evStop);
        break;
      default:
        break;
    }
  } while (answer != 'q');

  return 0;
}
```

7.3 Realisieurng mit Steuerkonstrukt (objektorientiert in C++)

7.3.1 State-Event-Diagram – Up/Down-Counter



7.3.2 Zusammenhang der Klassen Counter und CounterCtrl



- Klasse `Counter` führt eigentliche Rechenaufgaben durch
  - ist bei **allen** (objektorientierten) Realiseurngsarten **identisch**
- Klasse `CounterCtrl` ist FSM, welche Zugriff auf den Counter steuert

➠ **Generell sollten Steuerung und Element, das gesteuert wird, getrennt werden!**

7.3.3 Implementation der Prozeduralen Realisierung in C++

- **Ereignisse (events)**
  - Schnittstelle nach aussen ➠ ändern Zustand der FSM
  - Im **public** Teil der Klasse als enum definiert
  - Keine Unitkürzel nötig
- **Zustände (states)**
  - Im **private** Teil der Klasse als enum definiert ➠ header-file
- **Aktueller Zustand** `currentState` wird in **privatem Attribut** der Schnittstelle gehalten
- Die FSM wird in **zwei Funktionen** implementiert
  - Kontruktor (hier: `CounterCtrl::CounterCtrl(int initValue=0)`)
  - Prozess-Funktion (hier: `void CounterCtrl::process(CounterCtrl::Event e)`)
    ➠ Zustände prüfen, Zustandsübergänge veranlassen
- Anstossen einer FSM
  - Initialisierung in `main`-Funktion
  - Überprüfung, welches Event aufgetreten ist meist in `do-while`-Schleife

Beispiel: Up/Down-Counter (prozedural in C++)

Schnittstelle Counter:

```
// Counter.h
// implements an up/down-Counter

#ifndef COUNTER_H__
#define COUNTER_H__

class Counter
{
  public:
    Counter(int val = 0);

    void count(int step);
    // counts the counter up (step>0)
    // or down (step<0) by step

    int getCounter() const;
    // returns the counter value

    void setCounter(int val);
    // sets the counter to val
  private:
    int countValue;
};
#endif
```

Implementation Counter:

```
// Counter.cpp
// implements an up/down-Counter

#include "Counter.h"

Counter::Counter(int val): countValue(val)
{
}

void Counter::count(int step)
{
  countValue += step;
}

int Counter::getCounter() const
{
  return countValue;
}

void Counter::setCounter(int val)
{
  countValue = val;
}
```

Schnittstelle FSM:

```
// CounterCtrl.h
// implements the Finite State Machine (FSM) of an up/down-Counter

#ifndef COUNTERCTRL_H__
#define COUNTERCTRL_H__
#include "Counter.h"

class CounterCtrl
{
  public:
    enum Event{evUp,       // count upwards
               evDown,     // count downwards
               evCount,    // count (up or down)
               evStop};    // stop counting

    CounterCtrl(int initValue = 0); // C-tor

    void process(Event e);
    // changes the state of the FSM based on the event 'e'
    // starts the actions

  private:
    enum State{idleState,        // idle state
               countUpState,     // counting up at each count event
               countDownState};  // counting down at each count event

    State currentState;          // holds the current state of the FSM
    Counter myCounter;           // holds the counter for calculations
};
#endif
```

Implementation FSM:

```
// CounterCtrl.cpp
// implements the Finite State Machine (FSM) of an up/down-Counter

#include <iostream>
#include "CounterCtrl.h"
#include "Counter.h"
using namespace std;

CounterCtrl::CounterCtrl(int initValue) :
  currentState(idleState),
  myCounter(initValue)
{
}

void CounterCtrl::process(Event e)
{
  switch (currentState)
  {
    case idleState:
      cout << "State: idleState" << endl;
      if (evUp == e)
      {
        // actions (and exit-actions from idleState)
        cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
        // state transition (and entry-actions from countUpState)
        cout << "Changing to State: countUpState" << endl;
        currentState = countUpState;
      }
      else if (evDown == e)
      {
        // actions (and exit-actions from idleState)
        cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
        // state transition (and entry-actions from countDownState)
        cout << "Changing to State: countDownState" << endl;
        currentState = countDownState;
      }
      break;

    case countUpState:
      cout << "State: countUpState" << endl;
      if (evCount == e)
      {
        // actions
        myCounter.count(1);
        cout << "State: countUpState, counter = " << myCounter.getCounter() << endl;
        // state transition
      }
      else if (evStop == e)
      {
        // actions
        // state transition
        cout << "Changing to State: idleState" << endl;
        currentState = idleState;
      }
      break;

    case countDownState:
      // ...
      break;

    default:
      break;
  }
}
```

Anstossen der FSM:

```
// counterTest.cpp
// Test program for the Finite State Machine (FSM) of an up/down-Counter

#include <iostream>
#include "CounterCtrl.h"
using namespace std;

int main(void)
{
  char answer;
  CounterCtrl myFsm(0);  // initValue of counter == 0
```

```
  do
  {
    cout << endl << "-----------------------------------------" << endl;
    cout << "    u   Count up" << endl;
    cout << "    d   Count down" << endl;
    cout << "    c   Count" << endl;
    cout << "    s   Stop counting" << endl;
    cout << "    q   Quit" << endl;

    cout << endl << "Please press key: ";
    cin >> answer;
    cout << endl;

    switch (answer)
    {
      case 'u':
        myFsm.process(CounterCtrl::evUp);
        break;
      case 'd':
        myFsm.process(CounterCtrl::evDown);
        break;
      case 'c':
        myFsm.process(CounterCtrl::evCount);
        break;
      case 's':
        myFsm.process(CounterCtrl::evStop);
        break;
      default:
        break;
    }
  } while (answer != 'q');

  return 0;
}
```

7.4 Realisierung mit Tabelle

7.4.1 State-Event-Diagram – Up/Down-Counter

Siehe Abschnitt 7.3.1

7.4.2 FSM in Tabellenform

Das State-Event-Diagramm wird in eine Tabelle 'übersetzt'. **Jede Zeile der Tabelle entspricht einer Transition (Pfeil) im State-Event-Diagramm**

Current State	Event	Action	Next State
idleState	evUp	showCounter()	countUpState
idleState	evDown	showCounter()	countDownState
countUpState	evCount	count(1); showCounter();	countUpState
countUpState	evStop	-	idleState
countDownState	evCount	count(-1); showCounter();	countDownState
countDownState	evStop	-	idleState

7.4.3 Implementation der Realisierung mittels Tabelle in C++

- Die ganze FSM ist in einer Tabelle gespeichert
- **Aktionen** sind als **Funktion** implementiert, in der **Tabelle** steht der entsprichende **Funktionspointer**
- **Abarbeitung** der FSM erfolgt mittels **Execution Engine**, die in der Tabelle 'nachschaut', was zu tun ist
  - Execution Engine **ändert sich nicht**, wenn FSM geändert wird!
- **Transition** wird als klasseninterner `struct` deklariert
  - enthält aktuellen Zustand, Event, Funktionspointer auf Aktionsmethode und nächsten Zustand
- **FSM** wird als statischer, offener Array deklariert
  - Hier wird ganze FSM gespeichert
  - ein `struct` bildet konkret eine Zeile der Tabelle ab

7.4.4 Eigenschaften der Realisierung mittels Tabelle

- Die Tabelle kann prozedural oder **objektorientiert** implementiert werden
  - Objektorientierte Variante verwendet einzig die Datenkapselung (keine Vererbung, kein Polymorphismus)
  - Objektorientierte Variante ist klarer / schöner strukturiert
- **Aktions-Funktionen** können **nicht inlined** werden, da ein Pointer auf die Funktionen verwendet wird

7.4.5 Tabelle vs. prozedural

Gemeinsamkeiten

- Testprogramm `counterTest.cpp`
- Schnittstelle (public-Teil) von Klasse `CounterCtrl`
- Gesamte Klasse `Counter`

Unterschiede

- private-Teil von Klasse `CounterCtrl` und Implementation davon

Beispiel: Up/Down-Counter (mit Tabelle in C++)

Schnittstelle und Implementation von Counter:
Die Schnittstelle `counter.h` und die Implementation `counter.cpp` ändern sich nicht!
➔ Code-Beispiele siehe 7.3

Anstossen der FSM:
Die Implementation des Testprogramms `counterTest.cpp` ändern sich nicht!
➔ Code-Beispiele siehe 7.3

Schnittstelle FSM:

```
// CounterCtrl.h
// implements the Finite State Machine (FSM) of an up/down-Counter as a simple table

#ifndef COUNTERCTRL_H__
#define COUNTERCTRL_H__
#include "Counter.h"

class CounterCtrl
{
  /* ------------------------------ NO CHANGES ------------------------------*/
  public:
    enum Event{evUp,       // count upwards
               evDown,     // count downwards
               evCount,    // count (up or down)
               evStop};    // stop counting

    CounterCtrl(int initValue = 0);   // C-tor

    void process(Event e);  // execution engine
    // changes the state of the FSM based on the event 'e'
    // starts the actions

  private:
    enum State{idleState,          // idle state
               countUpState,       // counting up at each count event
               countDownState};    // counting down at each count event

    State currentState;            // holds the current state of the FSM
    Counter myCounter;             // holds the counter for calculation

    /* ------------------------------ CHANGES ------------------------------*/

    typedef void (CounterCtrl::*Action)(void);   // function ptr for action function

    // action functions (must match with  function pointer!)
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);   // ensure that there is always a valid fkt-ptr
    void actionUpUp(void);
    void actionDownDown(void);

    struct Transition
    {
      State currentState;   // current state
      Event ev;             // event triggering the transition
      Action pAction;       // pointer to action function
      State nextState;      // next state
    };

    // static open array for transision structs
    static const Transition fsm[];
};
#endif
```

Implementation FSM:

```
// CounterCtrl.cpp
// implements the Finite State Machine (FSM) of an up/down-Counter as a simple table

#include <iostream>
#include "CounterCtrl.h"
#include "Counter.h"
using namespace std;

const CounterCtrl::Transition CounterCtrl::fsm[] =  // this table defines the fsm
{//currentState     triggering event  action function                 next state
  {idleState,       evUp,             &CounterCtrl::actionIdleUp,     countUpState},
  {idleState,       evDown,           &CounterCtrl::actionIdleDown,   countDownState},
  {countUpState,    evCount,          &CounterCtrl::actionUpUp,       countUpState},
  {countUpState,    evStop,           &CounterCtrl::actionDoNothing,  idleState},
  {countDownState,  evCount,          &CounterCtrl::actionDownDown,   countDownState},
  {countDownState,  evStop,           &CounterCtrl::actionDoNothing,  idleState}
};

CounterCtrl::CounterCtrl(int initValue) :   // initializations with initialization list
  currentState(idleState),
  myCounter(initValue)
{
}

void CounterCtrl::process(Event e)    // execution engine, this function never changes!
{
  // determine number of transitions automatically
  for (size_t i = 0; i < sizeof(fsm) / sizeof(Transition); ++i)
  {
    // is there an entry in the table?
    if (fsm[i].currentState == currentState && fsm[i].ev == e)
    {
      (this->*fsm[i].pAction)();
      currentState = fsm[i].nextState
      break;
    }
  }
}

// action functions
void CounterCtrl::actionIdleUp(void)
{
  cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
}

void CounterCtrl::actionIdleDown(void)
{
  cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
}

void CounterCtrl::actionDoNothing(void)
{
}

// ...
```

7.5 Erweiterung der Realisierung mittels Tabellen

- Wenn der Zustandsübergang nicht durch einen Event, sondern eine **komplexere Prüfung (Event und Guard)** ausgelöst wird, dann könnte der **Event-Eintrag** in der Tabelle durch einen weiteren **Funktionspointer** auf eine **Checkfunktion ersetzt** werden.
- Ergänzung für die Behandlung von Entry- und Exit-Actions

Beispiel: Up/Down-Counter (mit Checker-Tabelle in C++)

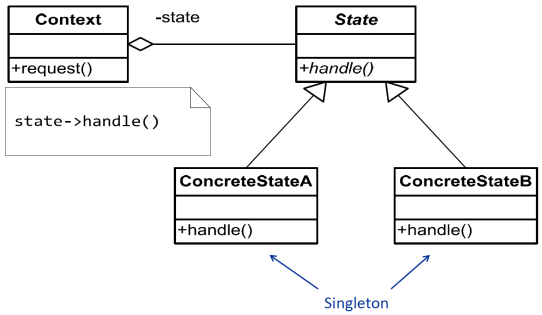
- Änderungen in `CounterCtrl.h` ➾ siehe Beispiel-Code
- Änderungen in `CounterCtrl.cpp`
  - checker-Funktionen müssen implementiert werden
  - In Tabelle steht statt Event die Adresse der checker-Funktion (analog zu action-Funktionen)

```
// ...

typedef bool (CounterCtrl::*Checker)(Event); // function ptr for checker function
typedef void (CounterCtrl::*Action)(void);   // no change here!

// check functions
bool checkIdleUp(Event e);
bool checkIdleDown(Event e);
// ...

struct Transition
{
  State currentState;   // current state
  Checker pChecker;     // pointer to checker function
  Action pAction;       // pointer to action function
  State nextState;      // next state
};

// ...
```

7.6 Realisieurng mit StatePattern

7.6.1 Grundidee von StatePatterns

Das Grundkonzept von StatePatterns ist **Polymorphismus** (Vererbung)



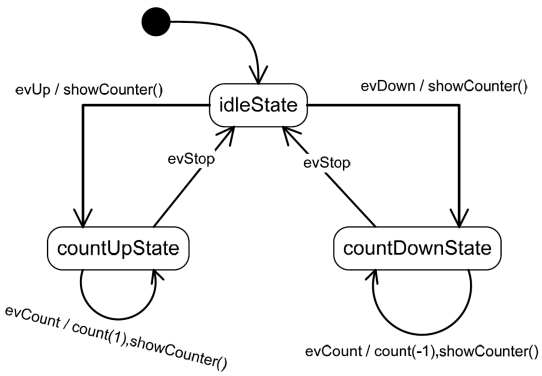
- **Context**-Klasse
  - definiert **Schnittstelle** für Clients
  - unterhält eine **Instanz** einer konkreten Unterklasse von `State`, die den aktuellen Zustand repräsentiert
- **State**-Klasse
  - definiert die Schnittstelle zur FSM in Form einer **abstrakten Klasse**
- **ConreteStateX** Unterklassen
  - Jede Unterklasse (Singleton) implementiert genau **einen Zustand**

7.6.2 Transitions in StatePatterns

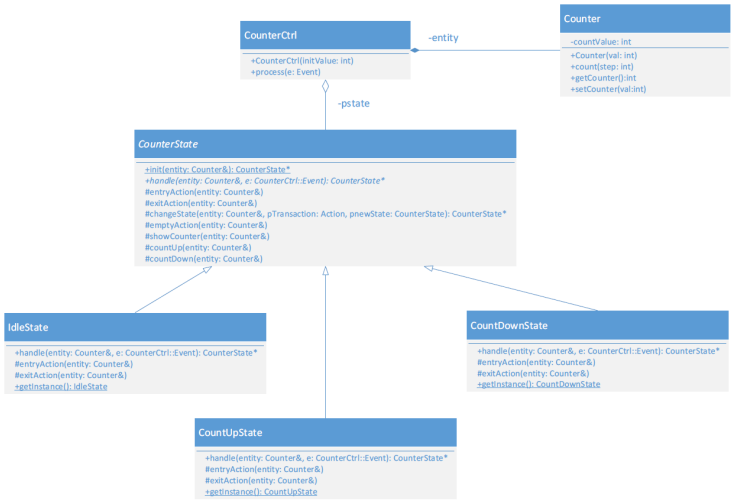
StatePattern definiert nicht, wo die Transitions umgesetzt werden sollen. Es gibt daher die zwei folgenden Varianten.
➾ **Variante 2 ist klar zu bevorzugen!**

1. Transitionen könnten in der `Context`-Klasse definiert werden.
   Nachteil: dort müsste zentral sehr viel Intelligenz vorhanden sein
   Da diese Klasse auch den Zugriff von der Aussenwelt darstellt, sollte sie möglichst schlank sein.
2. `State`-Klassen **realisieren ihre Transitionen selbst.**
   ➾ wir oft mittels `friend`-Deklaration realisiert, was jedoch nicht nötig ist

7.6.3 State-Event-Diagram – Up/Down-Counter



7.6.4 Klassendiagramm – Up/Down-Counter



7.6.5 Implementation der Realisierung mittels StatePattern

- **Ereignisse (events)**
  - Schnittstelle nach aussen ➾ ändern Zustand der FSM
  - Im **public** Teil der abstrakten Basisklasse als enum definiert
- **Zustände (states)**
  - Jeder Zustand als eigene (Sub-)Klasse definert
- **Aktueller Zustand** `pState` wird in **privatem Attribut (Pointer!)** der Schnittstelle gehalten
  - Es braucht daher in der `Context`-Klasse eine **forward declaration** der `State`-Klasse
- Die FSM wird in **zwei Funktionen** implementiert
  - Kontruktor (hier: `CounterCtrl::CounterCtrl(int initValue=0)`)
  - Prozess-Funktion (hier: `void CounterCtrl::process(CounterCtrl::Event e)`)
    ➾ Zustände prüfen, Zustandsübergänge veranlassen
- **Entry- und Exit-Actions**
  - Können in Basisklassen-Methode `changeState()` isoliert vorgenommen werden
  - Zwischen Exit- und Entry-Action müssen allfällige Transition-Actions ausgeführt werden. Diese wird in der Methode `changeState()` als **Funktionspointer** übergeben
  - In Basisklasse werden zwei **virtuelle Methoden** `entryAction()` und `exitAction()` deklariert
    ➾ Default-Implementation sinnvoll!

Beispiel: Up/Down-Counter mit StatePattern

Schnittstelle und Implementation von Counter:
Die Schnittstelle `counter.h` und die Implementation `counter.cpp` ändern sich nicht!
➾ Code-Beispiele siehe 7.3

Schnittstelle zur FSM:

```
// CounterCtrl.h
// implements the Finite State Machine (FSM) of an up/down-Counter

#ifndef COUNTERCTRL_H__
#define COUNTERCTRL_H__
#include "Counter.h"

class CounterState;  // forward declaration

class CounterCtrl    // this is the 'Context' class of the State pattern
{
  public:
    enum Event{evUp,        // count upwards
               evDown,      // count downwards
               evCount,     // count (up or down)
               evStop};     // stop counting
    CounterCtrl(int initValue = 0);
    void process(Event e);
    // changes the state of the FSM based on the event 'e'

  private:
    Counter entity;
    CounterState* pState;  // holds the current state
};
#endif
```

Implementation der FSM:

```
// CounterCtrl.cpp
// implements the Finite State Machine (FSM) of an up/down-Counter
// CounterCtrl is the Context class in the State pattern

#include "Counter.h"  // only needed if there is an entryAction in initState
#include "CounterCtrl.h"
#include "CounterState.h"

CounterCtrl::CounterCtrl(int initValue):
  entity(initValue),                  // use either line 11 or 12
  pState(CounterState::init(entitiy)) // initial state incl. entryAction
  pState(IdleState::getInstance())    // initial state without entryAction
{
}

void CounterCtrl::process(Event e)
{ // delegates all requests to CounterState
  pState =  pState->handle(entity, e);
}
```

Schnittstelle abstrakte State-Basisklasse:

```
// CounterState.h
// implements an up/down-Counter

#ifndef COUNTERSTATE_H__
#define COUNTERSTATE_H__
#include "CounterCtrl.h"  // Events are defined here

class CounterState  // abstract base class
{
  public:
    // should be called first, returns new state (if actions are used)
    static CounterState* init(Counter& entity);

    virtual CounterState* handle(Counter& entity, CounterCtrl::Event e) = 0;
    // returns new state
  protected:    // only inherited classes may use these member functions

    // if actions are used:
    virtual void entryAction(Counter& entity) {};
    virtual void exitAction(Counter& entity) {};
    typedef void (CounterState::*Action)(Counter& entity);   // ptr to action function

    // if actions are used: transition actions
    void emptyAction(Counter& entity) {};
    void showCounter(Counter& entity);
    void countUp(Counter& entity);
    void countDown(Counter& entity);

    // always (see extra comment)
    CounterState* changeState(Counter& entity,
                              Action ptransAction,  // only if actions are used
                              CounterState* pnewState);

};
```

Implementation abstrakte State-Basisklasse:

```
// CounterState.cpp
// implements all states of an up/down-Counter

#include <iostream>
#include "CounterState.h"
using namespace std;

// only if actions are used:
CounterState* CounterState::init(Counter& entity)  // it's static
{
  CounterState* initState = IdleState::getInstance();
  initState->entryAction(entity);      // executes entry action into init state
  return initState;
}

CounterState* CounterState::changeState(Counter& entity,
                                        Action ptransAction, // only with actions
                                        CounterState* pnewState)
{
  exitAction(entity);               // polymorphic call of exit action
  (this->*ptransAction)(entity);    // call of transition action
  pnewState->entryAction(entity);   // polymorphic call of entry action
  return pnewState;
}

// default implementations of entryActions() and exitAction()
// ...
```

Schnittstelle ConcreteStateX-Klassen (CountUpState):

```
// CountUpState.h
// interface of the CountUpState of an up/down-Counter

#ifndef COUNTUPSTATE_H__
#define COUNTUPSTATE_H__
#include "CounterCtrl.h"  // Events are defined here

class CountUpState : public CounterState // it's a singleton
{
  public:
    static CountUpState* getInstance();
    CounterState*  handle(Counter& entity, CounterCtrl::Event e) override;

  /* -------------------- if actions are used ---------------------------------- */
  protected:
    void entryAction(Counter& entity) override;  // only if  default is not enough
    void exitAction(Counter& entity) override;   // only if  default is not enough
  /* --------------------------------------------------------------------------- */
  private:
    CountUpState() {};
};
#endif
```

Implementation ConcreteStateX-Klassen (CountUpState – no actions):

```
// CountUpState.cpp
// implements the CountUpState of an up/down-Counter without actions

#include <iostream>
#include "CountUpState.h"
#include "CounterCtrl.h"  // Events are defined here
using namespace std;

CountUpState* CountUpState::getInstance()
{
  static CountUpState instance;
  return &instance;
}

CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
{
  cout << "State: countUpState" << endl;
  if (CounterCtrl::evCount == e)
  {
    // transition actions
    entity.count(1);
    cout << "counter = " << entity.getCounter() << endl;
    // state transition
    return changeState(entity, CountUpState::getInstance());
  }
  else if (CounterCtrl::evStop == e)
  {
    // transition actions
    // state transition
    return changeState(entity, IdleState::getInstance());
```

Implementation ConcreteStateX-Klassen (CountUpState – with actions):

```
// CountUpState.cpp
// implements the CountUpState of an up/down-Counter with actions

#include <iostream>
#include "CountUpState.h"
#include "CounterCtrl.h"  // Events are defined here
using namespace std;

// class CountUpState
CountUpState* CountUpState::getInstance()
{
  static CountUpState instance;
  return &instance;
}

CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
{
  cout << "State: countUpState" << endl;
  if (CounterCtrl::evCount == e)
  {
    // state transition
    return changeState(entity, &CountUpState::countUp, CountUpState::getInstance());
  }
  else if (CounterCtrl::evStop == e)
  {
    // state transition
    return changeState(entity, &CountUpState::emptyAction, IdleState::getInstance());
  }
  return this;
}

void CountUpState::entryAction(Counter& entity)
{
  cout << "Entering countUpState" << endl;
}

void CountUpState::exitAction(Counter& entity)
{
  cout << "Exiting from countUpState" << endl;
}
```

Anstossen der FSM:
Die Implementation des Testprogramms `counterTest.cpp` ändern sich nicht!
➠ Code-Beispiele siehe 7.3

8 Modularisierung

Ziel der Modularisierung ist eine **Reduktion der Komplexität.**

$$\sum_i \text{complexity(problem)}_i < \text{complexity}\Big(\sum_i \text{problem}_i\Big)$$

8.1 Grundprinzip Modularisierung

- **Problem in (einfachere) Unterprobleme aufteilen** und diese Unterprobleme jeweils **einzeln angehen**
- Abstraktion

8.1.1 Motivation für Modularisierung

- **Grosse Projekte – 'richtige' Softwaresysteme**
  - Systematischer Designansatz und strukturierter Aufbau ermöglichen effiziente **Arbeit im Team**
  - Schnittstellen müssen klar definiert werden
- **Informatin Hiding**
  - Für die Nutzung eines Moduls (Unit) muss es gnügen, **nur die Schnittstellen** zu kennen

8.1.2 Phasenunterteilung beim Entwurf

- **Grobentwurf, Architektur (architectural design)**
  - (Software-) System im Grossen
  - Schnittstellen zu anderen (Nicht-Software-) Systemen
  - Datenstruktur im Grossen
  - **Aufteilung in Subsysteme**
  - **Schnittstellen zwischen Subsystemen**
- **Feinentwurf**
  - Innenleben und Datenstruktur im Kleinen

8.2 Bewertung einer Zerlegung

- **Kopplung (coupling)**
  - Mass für Komplexität der Schnittstelle
- **Kohäsion (cohesion)**
  - Aussage, wie stark eine funktionale Einheit wirklich zusammengehört
  - Mass die die Stärke des inneren Zusammenhangs

➠ Ziel ist eine schawache Kopplung mit starker Köhäsion!

8.3 Kopplung

schwach (gut)

- **Keine direkte Kopplung**
- **Datenkopplung**
  - Kommunikation ausschliesslich über Parameter
- **Datenbereichskopplung**
  - Ein Modul hat Zugriff auf eine Datenstruktur eines anderen Moduls. Es werden allerdings nur einzelne Komponenten wirklich benötigt.
- **Steuerflusskopplung** (control flow)
  - Ein Modul beeinflusst Steuerfluss eines anderen Moduls
- **Globale Kopplung**
  - Kommunikation über globale Variablen, jedes Modul hat Zugriff
- **Inhaltskopplung (Todsünde!)**
  - Aus einem Modul heraus werden lokale Daten eines anderen Moduls modifiziert, obwohl dieses Modul gar nicht vom anderen Modul aufgerufen wird.

stark (schlecht)

8.4 Kohäsion

schwach (gut)

- **funktional**
  - Die Teile einer Einheit bilden zusammen eine Funktion, bzw. eine Funktionsgruppe
- **sequentiell**
  - Teilfunktionen einer Einheit werden nacheinander ausgeführt, wobei das Resultat einer Funktion als Eingabe für die nächste verwendet wird
- **kommunikativ**
  - Die Teilfunktionen einer Einheit werden auf den gleichen Daten ausgeführt, Reihenfolge spielt keine Rolle
- **prozedural**
  - Teilfunktionen werden nacheinander ausgeführt, verknüpft über Steuerfluss
- **zeitlich**
  - Die Teile einer Einheit sind alle zu einer bestimmten Zeit auszuführen
  - Typischer Fall: alle Initialisierungsfunktionen werden zusammengefasst
- **logisch**
  - (nicht zusammengehörende) Teilfunktionen einer Einheit gehören zu einer Einheit
- **zufällig**
  - Die Teilfunktionen einer Einheit haben keinen sinnvollen Zusammenhang

stark (schlecht)

8.4.1 Ziele bezüglich Kohäsion

- Kohäsion soll maximiert werden
  ⇒ **starke Kohäsion führt automatisch zu schwacher Kopplung!**
- Den genauen Wert der Kohäsion zu ermitteln ist **kein** Ziel

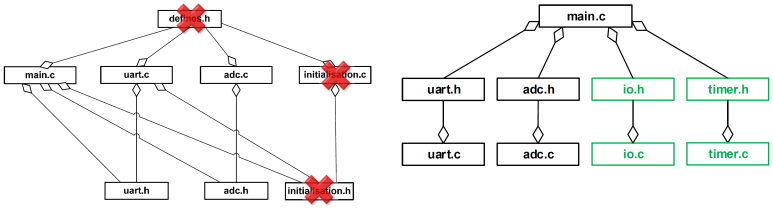
⇒ **Zusammengehörendes zusammennehmen!**

8.5 Guidelines – gute Modularisierung

- **Zusammengehörendes zusammennehmen**
  - Defines für spezigisches Modul in Header-File des Moduls
- Passende / aussagekräftige Namen für Variablen
- 'Interne' (private) Funktionen in `.c`-file deklarieren und definieren
- Schnittstellenbeschreibung in Header-Dateien
  - Falls möglich: Doxygen verwenden
- Lokale Funktionen (z.B. in `main.c`) bei Funktionsdeklarationen kommentieren
- Allenfalls 'globalen' Header für Typdefinitionen
  - besser: Typen aus `stdint.h` verwenden
- `uint8_t` etc. verwenden, wenn gezielt ein 8 Bit register angesprochen wird (und nur dann!)
- Keine `initialization.h` Dateien ⇒ zeitliche Kohäsion!
  - generell keine Dateien wie: `global.h`, `defines.h`, `util.h`, `project.h`

Hinweis: Für die Zurechtfindung in einem bestehenden Projekt müssen generell immer zuerst die **Header-Files studiert werden!**

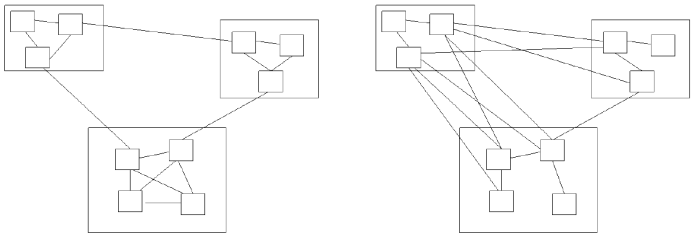
Beispiel: Schlechte vs. gute Modularisierung



8.6 Package-Diagramm

- Ein Package besteht aus mindestens einer, üblich aus mehreren Klassen, die zusammengehören (Stichwort: **Kohäsion**)
- Im Package-Diagramm kann dargestellt werden, welche Packages mit welchen anderen Packages Verbindungen haben (dürfen)
  - **Abhängigkeiten zwischen Packages** können sichtbar gemacht werden
- Packagekonzept in C++: Namespaces umgesetzt
  - ein Namespace entspricht einem Package

Beispiel: Schlechtes vs. gutes Packaging



links: hohe Kohäsion, tiefe Kopplung ⇒ gut
rechts: tiefe Kohäsion, hohe Kopplung ⇒ schlecht

9 Patterns (Lösungsmuster)

Ein Software Pattern ist eine bekannte Lösung für eine Klasse von Problemen.

+ Rad muss nicht immer neu erfunden werden	- Wichtige Patterns müssen bekannt sein
+ Getestete / funktionierende Lösungen	- Problemstellungen müssen als solche erkannt werden

9.1 Arten von Patterns

- **Architekturmuster (Architectural Pattern)**
  - Legt die grundlegende Organisation einer Anwendung und die Interaktion zwischen den Komponenten fest
- **Entwurfsmuster (Design Pattern)**
  - Die ursprüngliche Form des Pattern-Ansatzes
- **Implementationsmuster (Implementation Pattern)**
  - Behandelt grundsätzliche Implementationen immer wiederkehrender Codefragmente

9.2 Wichtige Patterns für Embedded Systems

9.2.1 Bereits bekannte Patterns

- **FSM Implementationen**
  - State Pattern
  - Singleton Pattern
  - (Steuerkonstrukt mit switch-case)
  - (Tabellenvariante)
- **'Mini-Patterns'**
  - Setzen / Löschen einzelner Bits
  - Behandlung asynchroner Ereignisse
    - Interrupts
    - Polling

9.2.2 Creational Patterns

Creational Patterns behandeln die **Erzeugung (und Vernichtung) von Objekten.**

- **Factory (Dependency injection)**
  - Definition einer Schnittstelle zur Erzeugung eines Objekts, statt der direkten Erzeugung auf der Client-Seite
- **Singleton**
  - stellt sicher, dass eine Klasse nur **ein einziges Objekt** besitzt
- **RAII (Resource Acquisition Is Initialization)**
  - Die Belegung und Freigabe einer Ressource wird an die Lebensdauer eines Objektes gebunden. Dadurch wird eine Ressource z.B. 'automatisch' freigegeben.

9.2.3 Structural Patterns

Structural Patterns **vereinfachen Beziehungen** zu anderen Teilen.

- **Adapter (Wrapper, Translator, glue code)**
  - Wandelt (adaptiert) eine Schnittstelle in eine für einen Client passendere Schnittstelle um
- **Facade**
  - Bietet eine **einfache Schnittstelle** für die Nutzung einer meist viel **grösseren Library**
- **Proxy**
  - 'A proxy, in its most general form, is a class functioning as an interface to something else.'
  - Oft ist es eine SW-Repräsentation eines HW-Teils, z.B. die Repräsentation einer Netzwerkverbindung

9.2.4 Behavioral Patterns

Behavioral Patterns identifizieren **gemeinsame Kommunikationspatterns** zwischen Objekten und implementieren diese.

- **Mediator**
  - definiert ein Objekt, welches das Zusammenspiel einer Menge von Objekten regelt
  - ein Embedded System, das aus **mehreren Teilen** wie Sensoren und Aktoren besteht, wird **im Mediator softwaremässig zusammengebaut**
- **Observer (MVC)**
  - Nicht nur bei Embedded Systems wichtig
  - Wird als objektorientierte Variante präsentiert
  - MVC-Prinzip kann auch prozedural mit Callbackfunktionen implementiert werden

Beispiel Mediator: Bei einem Drucker mit mehreren Druckaufträgen von mehreren Personen teilt der Mediator die Aufträge jeweils korrekt

9.2.5 Concurrency Patterns

Concurrency Patterns kümmern sich um die **Ausführung in multi-threaded Umgebungen.**

- **Active Object**
  - entkoppelt den Methodenaufruf von der Methodenausführung
    Methode soll sich nicht kümmern, in welchem Kontext sie aufgerufen wird
- **Lock**
  - Synchronisationsprimitive, welche den unteilbaren Zugriff read-modify-write implementiert
- **Monitor**
  - Monitor versteckt Synchronisationsanforderungen vor Client

10 Event-based Systems

10.1 Ereignisse (Events)

Reaktive Systeme reagieren auf (oft externe) Ereignisse (z.B. Digitale Inputs, Timer, Buttonclicks, etc.). Solche **Ereignisse sind per Definition asynchron und treten somit zu einem beliebigen Zeitpunkt auf.** Die Ereignisse können jedoch **synchron oder asynchron** umgesetzt werden.

10.2 Synchrone Umsetzung von Ereignissen

Ein '**normales' Programm** ist immer **synchron**. (Programm gibt vor, was wann ausgeführt wird.)

10.2.1 Polling

- Programm fragt periodisch oder dauernd ab, ob irgendein Ereignis eingetreten ist
- Maximale Reaktionszeit wird durch Abfrageperiode und Anzahl Abfragen definiert (Looptime bei SPS)

\+ Sehr einfach zu implementieren

\- Leerabfragen (Abfragen, bei welchen nichts eingetreten ist) können durch periodisches Abfragen (mittels Timer) reduziert, aber nicht vermieden werden

10.3 Asynchrone Umsetzung von Ereignissen

Ziel der asynchronen Verarbeitung von Events ist es, dass die Prozessorzeit **genau dann und nur dann** beansprucht wird, wenn ein Ereignis eingetreten ist. ➠ Interrupts

10.4 Interrupt-Verarbeitung

1. I/O-Element generiert einen Interrupt Request
2. Die CPU unterbricht das laufende Programm
3. **Die Interrupts werden disabled (ausgeschaltet)**
4. Das I/O-Element wird informiert, dieses deaktiviert den Interrupt Request
5. Die Interrupt Service Routine (ISR) wird ausgeführt
6. **Die Interrupts werden wieder enabled (eingeschaltet)**
7. Die CPU führt das Programm an der unterbrochenen Stelle weiter

Sprungadresse nach Interrupt-Auslösung (ISR):

- **Non-vectored Interrupt (zentral)**
  - Alle Interrupts verzweigen zu einer **gemeinsamen Adresse**. Dort wird die Ursache bestimmt und zu einer spezifischen Behandlungsroutine verzweigt.

  \+ Nur eine zentrale Routine für die Behandlung notwendig

  \- Information über die Ursache ist beim Eintreten bereits bekannt. Dann verzweigt man in die zentrale Routine, d.h. diese Information ist dann verloren. In der Routine muss diese Information wieder ermittelt werden.
- **Vectored Interrupt (spezifisch)**
  - In einer Tabelle (**Interruptvektortabelle**, IVT) wird gespeichert, wohin bei welchem Interruptvektor verzweigt werden muss.
    ➠ zu bevorzugende Methode!

10.5 Interruptvektortabelle (IVT)

Für jeden Vektor muss eingetragen werden, welches die **Anfangsadresse** der Interrupt Service Routine (ISR) ist, d.h. die **IVT ist nichts anderes als eine Tabelle (Array) von Funktionspointern.**

➠ Dieses Konzept kommt bei **allen asynchronen Mechanismen** zur Anwendung

10.6 Model View Controller (MVC) aka Observer Pattern

Ausgangslage: Daten (model) und verschiedene Darstellungsformen (views) der Daten (z.B. Balkendiagramm, Kuchendiagramm, Tabelle, etc.)
➠ Die views (clients) sollen unbedingt vom model (server) getrennt werden!

Wie kann nun erreicht werden, dass bei **jeder Änderung** der Daten (model) alle Darstellungen aktualisiert werden? ➠ Callback-Funktionen!

10.7 Callback-Funktionen

\+ Views werden **asynchron** genau informiert, wenn sich etwas im **model geändert** hat

\+ An und für sich sind alle registrierten Funktionen nichts anderes als **Eventhandler eines bestimmten Events** ➠ Darstellung (Definition der registrierten Funktionen) sauber von den Daten (model) **entkoppelt**

10.8 Umsetzung der Callback-Funktionen in C (clientseitig)

Event registieren (attach):

- Jeder client meldet beim server an, welche Ereignisse ihn interessieren
  - Anmeldung erfolgt über eine Funktion, welche der server anbietet

```
int foo_registerCb(foo_Event e, foo_cbFunction f);
// registers function 'f' on event 'e' -> 'id' is returned
// sometimes called attach()
```

- Der server trägt diesen **Funktionspointer** `f` in eine Tabelle ein und ruft **beim Eintreten des Ereignisses alle registrierten Funktionen** der Reihe nach je über den eingetragenen Funktionspointer auf

Event austragen (detach):

- Ein client kann sein Interesse an einem Ereignis beim Server auch wieder austragen
  - Abmeldung erfolgt über eine Funktion, welche ebenfalls der server anbietet
  - Der Server löscht dann den entsprechenden Eintrag (**Funktionspointer** `f`) wieder aus der Tabelle

```
int foo_unregisterCb(foo_Event e, int id);
// unregisters functionId 'id' on event 'e' -> 'id' is returned
// sometimes called detach()
```

10.9 Umsetzung der Callback-Funktionen in C (serverseitig)

- Funktionspointer `foo_cbFunction` zu Callback-Funktionen definieren

```
typedef void (*foo_cbFunction)(int);
// Schnittstelle: void f(int)
```

- Tabelle von Funktionspointern für jeden Event definieren und mit **Nullpointern initialisieren**

```
foo_cbFunction evClient[evNum] = {0};
// Note: NULL instead of 0 if stdio.h is included
```

- Aufruf der registrierten Clientfunktionen beim Eintreten des Events

```
void notify(foo_cbFunction client[], int evNum, int arg)
{
    for(size_t i = 0; i < evNum; ++i)
    {
        if(evClient[i] != 0)    // entry found
        {
            evClient[i](arg);   // call client through function ptf
        }
    }
}
```

Mitteilung eines Events:

Sobald (**asynchron**) ein Event eingetreten ist, kann dieser dem Server mit der Funktion `void foo_signalEvent(foo_Event e);` mitgeteilt werden.

Beispiel: Callback-Funktionen in C

Test-Applikation – Client:

```
// testApp.c
// this is the client

#include <stdio.h>
#include "fooServer.h"
#include "fooSigGen.h"

// functions to be registered (prototypes)
static void f1(int a);
// ...

int main(void)
{
  enum {maxId = 8};
  int fId[maxId] = {0};
  // register functions on events
  fId[0] = foo_registerCb(foo_ev1, f1);
  fId[1] = foo_registerCb(foo_ev1, f2);
  fId[2] = foo_registerCb(foo_ev1, f3);
  fId[3] = foo_registerCb(foo_ev2, f4);
  fId[4] = foo_registerCb(foo_ev2, f2); // same function registered on two events
  fId[5] = foo_registerCb(foo_ev3, f5);

  for (size_t i = 0; i < maxId; ++i)
  {
    if (foo_failed == fId[i])
      printf("fId[%zu] failed to register\n", i);
  }
  foo_generateSignals();

  // unregister some functions
  if (foo_unregisterCb(foo_ev1, fId[0]) == fId[0])
    printf("f1 successfully unregistered from foo_ev1\n");
  else
    printf("failed to unregister f1 from foo_ev1\n");

  if (foo_unregisterCb(foo_ev1, 27) == 27) // should fail: unknown id
    printf("xy successfully unregistered from qr\n");
  else
    printf("failed to unregister (unknown id)\n");

  // register functions on events
  printf("try to register f4 on foo_ev2 at fId[6]\n");
  fId[6] = foo_registerCb(foo_ev2, f4); // should fail: too many registered functions
  for (size_t i = 0; i < maxId; ++i)
  {
    if (foo_failed == fId[i])
    {
      printf("fId[%zu] failed to register\n", i);
    }
  }

  foo_generateSignals();
  return 0;
}

// local functions
void f1(int a)
{
  printf("f1() called. Event# = %d.\n", a);
}

// ...
```

Server – Header-File:

```
// fooServer.h -> Callback server

#ifndef FOO_SERVER_H__
#define FOO_SERVER_H__

// typeless enum with integer for a failed result
enum {foo_failed = -1};

//function pointer to callback functions
typedef void (*foo_cbFunction)(int);

// enum with possible events
typedef enum {foo_ev1 = 1,        // foo example event 1
              foo_ev2,            // foo example event 2
              foo_ev3             // foo example event 3
              }foo_Event;

// registers function 'f' on event 'e'
// returns id: success or foo_failed: no success
int foo_registerCb(foo_Event e, foo_cbFunction f);

// unregisters functionId 'id' on event 'e'
// returns id: success or foo_failed: no success
int foo_unregisterCb(foo_Event e, int id);

// signals that event 'e' occurred
void foo_signalEvent(foo_Event e);
#endif
```

Server – Implementation:

```
// fooServer.c --> Callback server

#include "fooServer.h"
#include <stddef.h>

enum {ev1Num = 3,   // max number of registered functions for event ev1
      ev2Num = 2,   // dito ev2
      ev3Num = 2};  // dito ev3

// definition of function tables
static foo_cbFunction ev1Client[ev1Num] = {0};  // clients for event ev1
static foo_cbFunction ev2Client[ev2Num] = {0};  // clients for event ev2
static foo_cbFunction ev3Client[ev3Num] = {0};  // clients for event ev3

// local function declarations
static int insertCb(foo_cbFunction f, foo_cbFunction client[], int evNum);
// inserts callback function 'f' in list 'client[]'

static int deleteCb(int id, foo_cbFunction client[], int evNum);
// deletes callback functionId 'id' in list 'client[]'

static void notify(foo_cbFunction client[], int evNum, int arg);
// notifies all registered clients

// interface functions' definitions
int foo_registerCb(foo_Event e, foo_cbFunction f)
{
  switch (e)
  {
    case foo_ev1:
      return insertCb(f, ev1Client, ev1Num);
    case foo_ev2:
      return insertCb(f, ev2Client, ev2Num);
    case foo_ev3:
      return insertCb(f, ev3Client, ev3Num);
    default:
      break;
  }
  return foo_failed; // no success if I get here
}

int foo_unregisterCb(foo_Event e, int id)
{
  switch (e)
  {
    case foo_ev1:
      return deleteCb(id, ev1Client, ev1Num);
    case foo_ev2:
      return deleteCb(id, ev2Client, ev2Num);
    case foo_ev3:
      return deleteCb(id, ev3Client, ev3Num);
    default:
      break;
  }
  return foo_failed; // no success if I get here
}

void foo_signalEvent(foo_Event e)
{
  switch (e)
  {
    case foo_ev1:
      // in this example, only the event # is passed as argument
      notify(ev1Client, ev1Num, e);
      break;
    case foo_ev2:
      notify(ev2Client, ev2Num, e);
      break;
    case foo_ev3:
      notify(ev3Client, ev3Num, e);
      break;
    default:
      break;
  }
}

// local functions
int insertCb(foo_cbFunction f, foo_cbFunction client[], int evNum)
{
  for (size_t i = 0; i < evNum; ++i)
  {
    if (0 == client[i]) // free entry found
    {
      client[i] = f;
      return i;  // success
    }
  }
  return foo_failed;   // number of registered functions exceeded
}

int deleteCb(int id, foo_cbFunction client[], int evNum)
{
  if (id < evNum && id >= 0)
  {
    client[id] = 0;
    return id;  // success
  }
  else
  {
    return foo_failed;   // illegal id
  }
}

void notify(foo_cbFunction client[], int evNum, int arg)
{
  for (size_t i = 0; i < evNum; ++i)
  {
    if (client[i] != 0) // entry found
    {
      client[i](arg);   // call the registered client through function pointer
    }
  }
}
```

Signal Generator – Header-File:

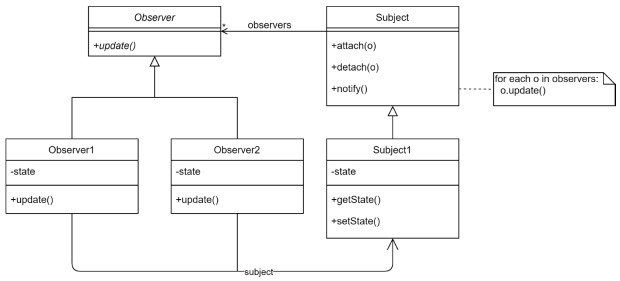
```
// fooSigGen.h -> signal generator
// generates (emulates) external signals

#ifndef FOO_SIGGEN_H__
#define FOO_SIGGEN_H__

void foo_generateSignals(void);

#endif
```

Signal Generator – Implementation:

```
// fooSigGen.c
// Signal generator implementation

#include "fooServer.h"
#include <stdio.h>

// interface functions' definitions
void foo_generateSignals(void)
{
  int answer;

  do
  {
    printf("------------------------------\n");
    printf("\nChoose event to be signalled\n");
    printf("    (1)     Event 1\n");
    printf("    (2)     Event 2\n");
    printf("    (3)     Event 3\n");
    printf("\n    (0)     Exit\n");
    printf("\n\nYour choice: ");
    scanf("%d", &answer);
    printf("\n------------------------------\n");
    switch (answer)
    {
      case 1:
        foo_signalEvent(foo_ev1);
        break;
      case 2:
        foo_signalEvent(foo_ev2);
        break;
      case 3:
        foo_signalEvent(foo_ev3);
        break;
      default:
        break;
    }
  } while (answer != 0);
}
```

10.10 Observer Pattern

10.10.1 Klassendiagramm (abstrakte Observer Basisklasse)



10.10.2 Implementation in C++

- **Observer-Klasse (abstrakte Basisklasse)**
  - Die Klasse muss **nicht** geändert werden
- **Observer-Subklassen** (View)
  - Enthalten jeweils eine private Referenz auf ein konkretes Subject
  - state entspricht z.B. einem counter-Wert, welcher jeweils updated wird
- **Subjekt Klasse** (server, Model)
  - liefert Administration für alle Subjects
  - Die Klasse muss **nicht** geändert werden
  - Enthält privates array mit Pointern auf Observer `const Observer* observers[size]`
  - `attach(o)` und `detach(o)` benutzen `const` Referenzen auf Observer als Parameter
- **Subjekt1 Subklasse**
  - Konkretes Sujekt (Server, Model)

Beispiel: Observer Pattern in C++

Test-Applikation:

```cpp
// testApp.cpp
// client using observer pattern

#include <iostream>
#include "Subject1.h"
#include "Observer1.h"
using std::cout;
using std::endl;

int main(void)
{
  Subject1 myS;
  Observer1 myO(myS);
  myS.setState(23);
  myS.setState(87);

  return 0;
}
```

Observer – Abstrakte Basisklasse:

```cpp
// Observer.h

#ifndef OBSERVER_H__
#define OBSERVER_H__

class Observer
{
  public:
    // method to update something
    // (pure virtual)
    virtual void update() const = 0;

    // Dtor
    virtual ~Observer() {}
};
#endif
```

Observer1 – Konkreter Observer (View) – Header-file:

```cpp
// Observer1.h -> implements an observer

#ifndef OBSERVER1_H__
#define OBSERVER1_H__

#include "Observer.h"

class Subject1;  // forward declaration to subject

class Observer1 : public Observer
{
  public:
    Observer1(Subject1& s);       // Ctor with reference to subject
    void update() const override; // method to update something
    virtual ~Observer1();         // Dtor

  private:
    Subject1& sub;
};
#endif
```

Observer1 – Konkreter Observer (View) – Implementation:

```cpp
// Observer1.cpp -> implements an observer

#include "Observer1.h"
#include <iostream>
#include "Subject1.h"
using namespace std;

Observer1::Observer1(Subject1& s) : sub(s)
{
    sub.attach(*this);
}

void Observer1::update() const
{
  cout << "Observer1 view: " << sub.getState() << endl;
}

Observer1::~Observer1()
{
    sub.detach(*this);
}
```

Subject – Basisklasse (Server, Model) – Header-file:

```cpp
// Subject.h -> Server, Model, Subject

#ifndef SUBJECT_H__
#define SUBJECT_H__

class Observer; // forward declaration

class Subject
{
  public:
    enum {ok = 0,       // return value for good
          failed = -1   // return value for error/failure
         };

    int attach(const Observer& ob);   // attaches observer 'ob'
    // return: ok: success or failed: no success

    int detach(const Observer& ob);   // detaches observer 'ob'
    // return: ok: success or failed: no success

    void notify() const;  // notifies all attached observers (read-only)

  private:
    enum {size = 4};                          // may use template parameter
    const Observer* observers[size] = {nullptr};  // may use vector<> instead
};
#endif
```

Subject – Basisklasse (Server, Model) – Implementation:

```cpp
// Subject.cpp -> Server, Model, Subject

#include <iostream>
#include "Subject.h"
#include "Observer.h"
using namespace std;

int Subject::attach(const Observer& ob)
{
  for (size_t i = 0; i < size; ++i)
  {
    if (nullptr == observers[i]) // free entry found
    {
      observers[i] = &ob;
      return ok;   // success
    }
  }
  return failed;   // number of observers exceeded
}

int Subject::detach(const Observer& ob)
{
  for (size_t i = 0; i < size; ++i)
  {
    if (&ob == observers[i])
    {
      observers[i] = nullptr;
      return ok;  // success
    }
  }
  return failed;   // illegal observer
}

void Subject::notify() const
{
  for (size_t i = 0; i < size; ++i)
  {
    if (observers[i] != nullptr) // entry found
      observers[i]->update();
  }
}
```

Subject1 – Header-file:

```cpp
// Subject1.h
// observed entity, Model, Subject

#ifndef SUBJECT1_H__
#define SUBJECT1_H__

#include "Subject.h"

class Subject1 : public Subject
{
  public:
    // sets the state of the subject
    void setState(unsigned int newState);

    // returns ok: success
    // or failed: no success
    unsigned int getState() const;
  private:
    unsigned int state = 0;
};
#endif
```

Subject1 – Implementation:

```cpp
// Subject1.cpp
// The observed entity, Model, Subject

#include "Subject1.h"
#include <iostream>
using namespace std;

void Subject1::setState(unsigned int
      newState)
{
  state = newState;
  notify();  // inform observers
}

unsigned int Subject1::getState() const
{
    return state;
}
```

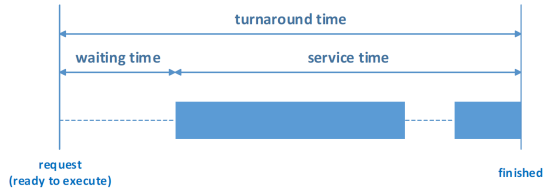
11 Scheduling

11.1 Multitasking

Mehrere (gleiche oder unterschiedliche) Tasks müssen erledigt werden. Dazu werden Ressourcen benötigt (z.B. **CPU**, Speicher, ...). Wenn mehrere Tasks **dieselben Ressourcen** benötigen, nimmt der **Scheduler** die Zuteilung der Ressourcen an die einzelnen Tasks vor.

Bei der Zuteilung der Ressourcen wird darauf geachtet, dass alle **kritischen deadlines eingehalten** werden.
➠ Der Scheduler **priorisiert** also die **kritischen Tasks.**
Unter Umständen werden somit Deadlines von weniger kritischen Tasks verletzt.

11.1.1 Zeitdefinitionen (Task)



- **turnaround time:** (response time, Antwortzeit)
  - Startet, wenn der Task bereit zur Ausführung ist und endet, wenn der Task fertig abgearbeitet ist
  - Zeit zwischen dem Vorhandensein von Eingangswerten an das System (Stimulus) bis zum Erscheinen der gewünschten Ausgangswerte.
- **waiting time:** (Wartezeit)
  - Zeit zwischen Anliegen der Eingangswert und Beginn der Abarbeitung des Tasks
- **service time:** (Bearbeitungszeit)
  - Zeit für Abarbeitung des Tasks ➠ Unterbrechungen bzw. (preemptions) möglich

11.1.2 Leistungsmerkmale

- Durchsatz (throughput)
  - Anzahl erledigte Tasks pro Zeiteinheit
- Mittlere Wartezeit (average waiting time)
- Auslastung (utilization)
  - Prozentuale Auslastung einer Ressource
- Weitere

11.2 Scheduability

Eine Menge von Tasks ist dann **scheduable**, wenn **alle Tasks zu allen Zeiten ihre deadlines einhalten** können. ➠ Das ist immer das Ziel!

11.2.1 Deadline – Definition

- Spätestmöglicher Abschlusszeitpunkt (eines Tasks)
  - Bei periodischen Tasks ist dias meist gleichzeitig mit Beginn der nächsten Periode

11.3 Scheduling-Strategien

Folgende Algorithmen können dür die Zuteilung der Ressourcen (Scheduling) verwendet werden:

- **FCFS (First Come First Served)**
  - Einfachste Variante
- **Round Robin**
  - Rund herum in fixer Reihenfolge
- **Random**
- **SJF (Shortest Job First)**
  - \+ Mittlere Wartezeit minimal
  - \- längere Tasks können 'verhungern'
- **Priority Scheduling**
  - unterbrechbar (preemptive) oder nicht unterbrechbar (non-preemptive)
  - \- tief priorisierte Taks können 'verhungern'

Hinweis: 'verhungern' heisst, dass ein Task gar keine Ressourcen erhält

11.4 Cooperative Multitasking

Kooperative Task-Zuteilung ist bei **fairen** Tasks möglich.

- Aktiver Task entscheidet selbst, wann er CPU wieder für andere Tasks freigibt
  - Unfaire und abgestürzte Tasks blockiert andere Tasks
- Nächster Task kann mit beliebigem Algorithmus ermittelt werden
  ➠ siehe Abschnitt 11.3
- Sehr einfach zu implementieren

11.5 Preemptive Multitasking / Scheduling

Preemptive Multitasking wird meistens in RTOS verwendet.
Der Task mit höchster Priorität wird immer ausgeführt. Unter Umständen muss dabei ein Task mit niedrigerer Priorität **unterbrochen** werden.

Es gibt zwei Arten von Preemptive Multitasking Algorithmen:

- **dynamic-priority Algorithmen**
  - Prioritäten werden zur Laufzeit laufen angepasst (z.B. aufgrund von vorhandenen deadlines)
- **static-priority Algorithmen**
  - Prioritäten werden zur Entwicklungszeit festgelegt und nicht geändert.
  - Einfacher als dynamic-priority Algorithmen!

11.6 Rate Monotonic Scheduling (RMS)

RMS beschreibt Regel, bei deren Einhaltung eine Konfiguration **immer scheduable** ist.

11.7 Rate Monotonic Scheduling Theorem

11.7.1 Zwingende Voraussetungen

- Perioische Tasks
- static priority preemptive scheduling ➠ siehe Abschnitt 11.5

11.7.2 Regeln für optimales Scheduling

Für jeden Task T_i wird die Periode p_i und die (worst case) execution time e_i ermittelt, bzw. geschätzt.
Die Prioritäten mussen den Tasks zwingend folgendermassen zugewiesen werden:

Tasks mit kürzerer Periode (d.h. mit hoher Rate) erhalten höhere Priorität (rate-monotonic)

11.7.3 Berechnung der Auslastung einer Ressource

Jeder Task T_i trägt mit der Teilauslastung $u_i = \frac{e_i}{p_i}$ zur Gesamtauslastung U bei.

$$U = \sum_i \frac{e_i}{p_i}$$

11.8 Vorgehen – Rate Monotonic Scheduling

1. Tasks priorisieren (Task mit kleinster Periode hat höchste Priorität!)
2. Task mit höchster Priorität aufzeichnen
3. Task mit zweithöchster Priorität 'regulär' zeichnen mit folgenden Sonderregelungen
   - Bei Bedarf warten (W), bis höher priorisierter Task abgeschlossen ist
   - Höher priorisierte Tasks (bereits gezeichnet) unterbrechen (P) aktuellen Task
4. Punkt 3 wiederholen, bis alle Tasks aufgezeichnet sind und sich das Muster wiederholt

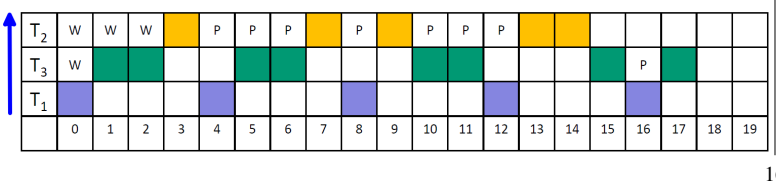
Beispiel: Rate Monotonic Scheduling

Gemäss gegebener Tabelle sind die Tasks folgendermassen priorisiert:

$$T_1 > T_3 > T_2$$

In dieser Reihenfolge werden die Tasks aufgezeichnet!

Task	e_i	p_i	$u_i = e_i / p_i$
T_1	1	4	0.25
T_2	5	20	0.25
T_3	2	5	0.40



11.9 RMA Bound (RMA = Rate Monotonic Approach)

Jede Konfiguration mit n periodischen Tasks ist **immer** RM scheduable, wenn die Gesamtauslastung U **unterhalb oder gleich** der RMA Bound $U(n)$ liegt

RMA-Bound $U(n) = n \cdot (2^{\frac{1}{n}} - 1)$

n	2	3	4	5	10	∞
***U*(*n*) in %**	82.4	78.0	75.7	74.4	71.7	**ln(2) ≈ 69.3**

➠ Liegt die Gesamtauslastung unter 69.3 %, ist die Konfiguration **immer** RM schedulable

11.10 Anleitung für Zuweisung der Prioritäten bei RMS

- Prioritäten immer gemäss RMS zuweisen. (manuelle Zuweisung gibt keine bessere Lösung)
- Falls Auslastung nicht grösser als RMA Grenze, so ist Konfiguration RM schedulable
- Falls Auslastung **grösser** ist, muss **manuell analysiert** werden, on Konfiguration schedulable ist
- 100 % Auslastung könnte erreicht werden, wenn alle Perioden harmonisch sind, d.h. jede längere Periode ist ein exaktes Vielfaches aller Perioden kürzerer Dauer, z.B. (10, 20, 40, 80)
- Harmonische Perioden verringern die Unterbrechung (preemptions) von niedriger priorisierten Tasks
  ➠ (10, 20, 40) ist gegenüber (10, 20, 50) zu bevorzugen, falls möglich

12 Concurrency (Gleichzeitigkeit)

Programme von praktischem Nutzen führen meist mehrere Arbeiten 'gleichzeitig' durch. Beispielsweise soll bei einem Embedded System ein Roboterarm bewegt werden, während 'gleichzeitig' mit einem übergeordneten System kommuniziert wird.

12.1 Parallel Computing vs. Concurrent Computing

- **Parallel Computing**
  - Ausführung verschiedener Tasks **tatsächlich gleichzeitig**
  - Nicht möglich auf single-core System
- **Concurrent Computing**
  - Ausführung verschiedener Tasks **wirkt nur gleichzeitig**
  - Verschiedene Tasks erhalten verschiedene 'time slices' ➠ Ein Task pro time slice
  - Auf single- und multi-core Systemen möglich

12.2 Warum man Concurrency nicht verwenden sollte

- **Concurrency (mit Prozessen, Tasks, Threads) kostet immer**
  - Stack
  - Braucht context switch (Umschalten von einen zum anderen Prozess, Task, Thread) ➠ Alter context (Registerwerte, Steck, etc.) muss gespeichert, neuer geladen werden
  - Zugriff auf gemeinsame Ressourcen muss synchronisiert werden ➠ fehleranfällig (wird vergessen / falsch gemacht)
- **Komplexität steigt**
  - Sequenzielle Programme sind einfacher zu verstehen als parallele Programme

➠ **Concurrency nur dann einsetzen, wenn wirklich ein Nutzen vorhanden ist!**

12.3 Synchronisation

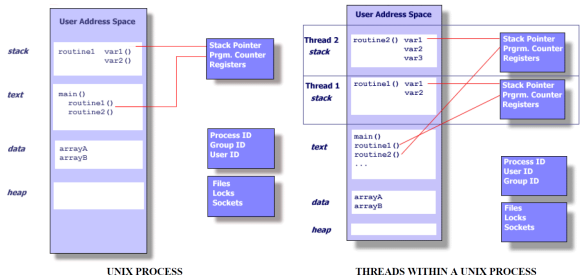
Wenn parallele Einheiten **gemeinsame Ressourcen** benützen, muss der Zugriff auf die Ressourcen geregelt (synchronisiert) werden. Wenn dies nicht gemacht wird kann es sein, dass zwei Tasks dieselbe Ressource 'falsch' verwenden. ➠ 'Deadlock'

Achtung: 'Ein Bisschen warten' ist **keine** Synchronisation!

13 POSIX Threads Programming

Für UNIX Systeme steht ein stardardisiertes threads programming interface in C zur Verfügung (POSIX threads / pthreads).

13.1 UNIX Process vs. UNIX Thread



13.1.1 UNIX Process

- **heavyweight process** (generiert von Betriebssystem)
- Prozess erfordert **erheblichen overhead**, da Informationen über Programmressourcen und den Ausführungsstatus des Programms, beispielsweise:
  - Prozess-ID, Prozessgruppen-ID, Benutzer-ID und Gruppen-ID
  - Environment, Programmanweisungen
  - Register, Stack, Heap
  - Datei-Deskriptoren, Signal-Aktionen
  - Gemeinsame Bibliotheken
  - Werkzeuge für die prozessübergreifende Kommunikation

13.1.2 UNIX Thread

- lightweight 'process' (weniger overhead)
- Unabhängiger 'stream of instructions', welcher simultan mit anderen 'streams of instructions' ablaufen kann
- Prozedur, welche unabhängig von ihrem (aufrufenden) main-Programm abläuft
- **Threadsexistieren in einem Prozess und nutzen dessen Ressourcen**
  - Sobald ein Prozess ended, enden auch die darin existierenden Threads!
- **Ein Thread benutzt den gleichen Adressraum wie andere Threads im gleichen Prozess**
  - Daten können einfach mit anderen Threads im gleichen Prozess geteilt werden
- Threads werden vom Betriebssystem 'gescheduled'
- Ein Thread dupliziert nur die essenziellen Ressourcen die er braucht, um unabhängig 'schedulable' zu sein:
  - Stack pointer, Register
  - Scheduling properties (policy / priority)
  - Set of pendding and blocked signals
  - Thread-spezifische Daten

➠ **Gleichzeitigkeit wird in der Programmierung mit Threads umgesetzt!**

13.2 pthreads API

13.2.1 Includes / Compile & Link

- `#include <pthread.h>` wird benötigt
- Methoden der pthreads API starten mit `pthread_`
- Source files, welche pthreads verwenden, sollen mit `-pthread` kompiliert werden
- Für das file-linking muss der command `-lpthread` verwenet werden

Beispiel: Compiling / Linking file printer.c

Compiling: `clang -c -Wall -pthread printer.c`
Linking: `clang -o printer printer.o -Wall -lpthread`

13.2.2 Thread starten / beenden

- Jede Funktion mit der folgenden interface kann eine Thread-Methode werden
  - Als Parameter / Return-Wert sind alle Pointer-Datentypen möglich

```
void* threadRoutine(void* arg);
```

- Ein Thread wird mit der folgenden Funktion gestartet:

```
int pthread_create(pthread_t* thread,              // ptr to pthread_t instance
                   const pthread_attr_t* attr,     // ptr to pthread_attr_t
                                                   // structure, often 0
                                                   // (default attributes)
                   void* (*startRoutine) (void*),  // function ptr to thread routine
                   void* arg);                     // single argument that may be
                                                   // passed to startRoutine
// returns 0 if thread is started successfully
```

- Ein Thread kann mit einer der folgenden drei Arten beendet werden
  - Thread ruft Funktion `pthread_exit()` auf
  - Thread springt aus Thread Routine `startRoutine` zurück
  - Thread wird mit Funktion `pthread_cancel()` abgebrochen

13.2.3 Warten, bis ein Thread beendet ist

- Nach dem Starten des Threads bzw. am Ende des main-Programms kann eine Endlos-Schleife eingefügt werden
  - **Dies sollte nie gemacht werden**, da der Prozess so die gesamten CPU-Ressourcen braucht
- Entsprechende Funktion aus pthreads API verwenden

```
int pthread_join(pthread_t thread,  // pthread_t instance
                 void** status)     // ptr to status argum. passed at end of thread
// returns 0 if thread terminated successfully
```

13.3 Beispiel: thread API

```
#include <pthread.h>  // for threads API
#include <stdio.h>
#include <unistd.h>   // for usleep()

// function prototype
void* printDashes(void* arg);

int main(void)
{
  int ret;
  pthread_t dasher; // pthread_t instance

  printf("start");

  // starts thread -> immediately returns
  // (thread maybe not fully started yet)
  ret = pthread_create(&dasher, 0,
                       printDashes, 0);
  if (ret)
  {
    printf("ERROR CODE: %d\n", ret);
    return -1;
  }

  // main thread shall wait until
  // dasher is finished
  ret = pthread_join(dasher, 0);
  if (ret)
  {
    printf("ERROR CODE: %d\n", ret);
    return -1;
  }

  printf("end\n");
  return 0;
}

void* printDashes(void* arg)
{
  for (size_t i = 0; i<20; ++i)
  {
    usleep(40000);
    putchar('-');
    fflush(stdout); // write character-
                    // wise and
                    // don't buffer
  }
  return 0;
}
```

13.4 Thread-safeness

Thread-safeness bezieht sich auf die Fähigkeit einer Anwendung, mehrere Threads gleichzeitig auszuführen, **ohne 'clubbering' und 'race conditions'** zu verursachen. Damit Thread-safeness gewährleistet werden kann, ist **Synchronisation** erforderlich.

clubbering: Speicher durcheinander bringen, wenn mehrere Threads den gleichen Speicher benötigen und 'falsch' darauf zugreifen

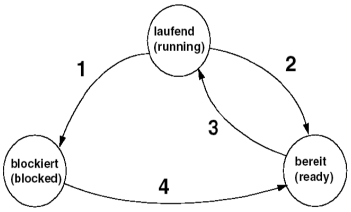
race conditions: Programmablauf und Endergebnis hängen davon ab, in welcher Reihenfolge 'gleichzeitig' ablaufende Threads auf z.B. eine globale Variable im Speicher zufreifen und das Verhalten somit unvorhersehbar wird

13.4.1 Empfehlung: Thread-Safeness

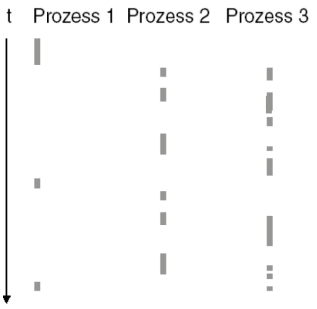
Wenn Thread-safeness nicht explizit garantiert ist (z.B. von einer Library, welche verwendet wird), muss angenommen werden, dass sie **nicht thread-safe** ist!
Um in einem solchen Fall Thread-safeness zu gewährleisten, können die Aufrufe einer 'unsicheren' Funktion **serialisiert** werden.

13.5 Quasi-Parallelität / 'Prozess'-Zustände

13.5.1 Prozess-Zustände



1. I/O Operation, Warten auf Bedingung
2. Scheduler entzieht CPU
3. Scheduler weist CPU zu
4. I/O beendet, Bedigung erfüllt



- Prozesse / Threads warten die 'meiste Zeit' ➠ blocked (z.B. `join` blockiert andere Threads)
- Scheduler ordnet CPU denjenigen Prozess / Thread zu, die im Zustand 'ready' sind und 'etwas zu tun haben'
- Die Zuordnung hängt vom verwendeten Scheduling-Algorithmus ab:
  - First come First serve Scheduling: Eine Queue mit allen Prozessen, wobei nächster Prozess jeweils hinten angehängt wird und erster Eintrag der Queue aktuell ausgeführt wird
  - Priority Scheduling: Pro Priorität gibt es eine Queue. Abarbeitung je nach Algorithmus anders

13.6 Synchronisation

Synchronisation wird benötigt, um den **Zugriff auf gemeinsame Ressourcen** in Critical Sections (CS) zu 'kontrollieren'.

13.6.1 Defintition: Critical Section (CS)

- Codebereich, in dem nebenläufige oder parallele Prozesse auf gemeinsame Ressourcen zugreifen
  - Zu jeder Zeit darf sich **höchstenns ein Prozess** im kritischen Abschnitt befinden
- Der Exklusive Zugriff durch höchstens einen Prozess wird mittels **gegenseitigem Ausschluss (Mutex)** sichergestellt ➠ Siehe Abschnitt 13.7

13.6.2 Forderungen an die Synchronisation

1. Maximal ein Prozess in einem kritischen Abschnitt (CS)
2. Über Abarbeitungsgeschwindigkeit, bzw. Anzahl Prozesse dürfen keine Annahmen getroffen werden
3. Kein Prozess darf **ausserhalb** eines kritischen Abschnitts einen anderen blockieren
4. Jeder Prozess, der am Eingang eines kritischen Abshcnitts wartet, muss irgendwann den Abschnitt betreten dürfen (**fairness condition**) ➠ Verhinderung von 'starvation'

13.7 Mutex (mutual exclusion)

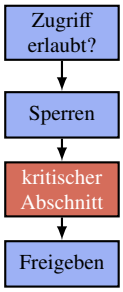
Die Lösungsstruktur 'Mutex' (gegenseitiger Ausschluss) stellt sicher, dass höchstens ein Prozess auf eine Critical Section (CS) zugreift.

13.7.1 Mutex – Ablauf

Zugriffsprüfung: Warten bis der Zugang frei wird

Sperren: Signal wird für andere auf Rot gesetzt, damit nur ein Prozess im kritischen Abschnitt sein kann

Freigeben: Rotes Signal wird wieder gelöscht



`waitFor(signal)`

`send(signal)`

13.7.2 Verwendung von Signalen und Semaphoren

- Jeder Prozess wartet vor dem Betreten der CS auf ein gemeinsames Signal
  - Wenn das Signal gesetzt ist, ist CS frei
  - Mehrere Prozesse können gleichzeitig warten ➠ Schedulingalgorithmus bestimmt 'nächsten' Thread
- `waitFor(signal)` blockiert **aufrufenden** Prozess, falls Signal nicht gesetzt
- Jeder Prozess, der fertig ist, setzt das Signal mit `send(signal)`

Semaphoren:
- 'Semaphor' ist ein spezieller Name für ein Signal für den **Zutritt zu einer CS**
- Es gibt zwei atomare (nicht unterbrachbare) Operationen auf einer Semaphoren `s`
  - Passieren P(s): Beim Eintritt in CS ➠ `waitFor(s)`

  - Verlassen V(s): Beim Austritt aus CS ➠ `send(s)`

Bei der Verwendung von Semaphoren treten folgende Probleme auf

- Ressourcen können besetzt bleiben, wenn V(s) vergessen wird
  - **Für jedes P(s) braucht es auch ein V(s)**
- Grössere Programme: Es können subtile Probleme entstehen, falls z.B. das V(s) in einer **if-Bedingung** gemacht wird
- Beim Auftreten von Exceptions kann das Freigeben schwierig werden

➠ Lösung für das Freigabe-Problem: RAII (siehe Abschnitt)

13.7.3 Busy Waiting

- Prozesse warten **aktiv** in einer Schleife (**spin lock**)
  - Wartende Prozesse **belasten** unnötigerweise den Prozessor

Die Lösung für Busy Waiting ist, die wartenden Prozesse in eine **Warteschlange** einzutragen (**sleep and wakeup**)

13.8 Thread Synchronisierung in C mit pthreads API

Code Synchronisation wird mittels Mutex (**lock pattern**) sichergestellt. Das Konzept von Mutex ist, dass eine Mutex Variable **nur einem Thread gleichzeitig gehören kann.**

13.8.1 Ablaub einer Mutex-Sequenz in C

1. Mutex Variable erstellen / instanzieren
   - 'Schloss', welches Zugang zu CS schützt
2. Mehrere Threads versuchen, die Mutex Variable zu blockieren
   ➠ **Nur ein Thread** ist erfolgreich ➠ diesem Thread ('owner') gehört die Mutex Variable
3. Dieser 'owner thread' führt Aktionen in der Critial Section (CS) aus
   - Häufig Update einer globalen (shared) Variable
4. 'owner' entblockt (unlock) die Mutex Variable
5. Dem nächsten Thread gehört die Mutex Variable ➠ zurück zu Schritt 2
6. Wenn alle Threads abgearbeitet sind, wird die Mutex Variable zerstört

➠ Dies ist ein sicherer Weg, um sicherzustellen, dass, wenn **mehrere Threads** dieselbe Variable aktualisieren, der **Endwert derselbe** ist, wie wenn nur **ein Thread** die **Aktualisierung durchführen** würde.

Beispiel: Mutex in C

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* for usleep */

static volatile int val = 0;    // shared resource
static pthread_mutex_t valMtx;  // create mutex_t variable

void* threadRoutine(void* arg); // prototype

int main(void)
{
  pthread_t t1; // create pthread_t variable
  pthread_t t2; // create pthread_t variable

  pthread_mutex_init(&valMtx, 0); // init mutex

  pthread_create(&t1, 0, threadRoutine, 0);
  pthread_create(&t2, 0, threadRoutine, 0);

  pthread_join(t1, 0);  // wait for thread to finish
  pthread_join(t2, 0);  // wait for thread to finish

  pthread_mutex_destroy(&valMtx); // destory mutex

  return 0;
}

// main routine of each counter thread
void* threadRoutine(void* arg)
{
  unsigned int rState = 17;

  while(1)
  {
    /* non critical section; simulate with usleep() */
    usleep(rand_r(&rState) % 200000);

    /* start of critical section */
    pthread_mutex_lock(&valMtx);  // lock mutex

    if (val < 20)
    {
      /* wait random time between 0s up to 0.3s */
      usleep(rand_r(&rState) % 300000);
      val = val + 1;  // change shared resource
      printf("val = %2d\n", val);
    }
    else
    {
      /* end of critical section */
      pthread_mutex_unlock(&valMtx);  // unlock mutex
      break;                          // exit while(1)
    }
    /* end of critical section */
    pthread_mutex_unlock(&valMtx);    // unlock mutex
  }
  pthread_exit(0); // optional, good programming style!
}
```

13.9 Monitorprinzip (Monitor Pattern)

Das Monitorprinzpt beschreibt eine Art Abstraktion des Mutex / Lock Patterns.
Dabei muss sich der **Aufrufer nicht mehr um die Synchronisation der Threads kümmern**. Das Problem wird einmal im Monitor gelöst.

- Es wird ein Abstrakter Datentyp (ADT) definiert, der genau die Funktionen in der Schnittstelle anbietet, die notwendig sind
- Der Aufrufer ruft diese Funktion auf, muss sich aber **nicht um Synchronisation kümmern**
  - Synchronisation (z.B. mit Semaphoren) ist Implementation des Monitors lokal gelöst

13.10 'Stolperfallen' bei Synchronisation

13.10.1 Starvation (Verhungern)

- Zustand, bei dem ein Prozess nie dran kommt ➠ er verhungert
- Kann auftreten bei:
  - prioritätsgetriebenen Systemen bei Prozessen mit niederer Priorität passieren
  - SJF (shortest job first) Systeme ➠ kurze Jobs bremsen längere Jobs aus
- Fairness condition besagt, dass Starvation verhindert werden muss

13.10.2 Deadlock

- Situation, bei der sich **zwei Prozesse gegenseitig blockieren**
  - Zwei Prozesse benötigen gemeinsame Ressourcen A und B. Wenn Prozess 1 die Ressource A bereits besitzt und Prozess 2 die Ressource B, dann warten beide unendlich lange auf die jeweils andere Ressource

Deadlock kann vermieden werden, indem alle Prozesse die gemeinsamen Ressourcen immer in derselben Reihenfolge anfordern (z.B. zuerst A, dann B)

13.11 Informationen zwischen Threads austauschen

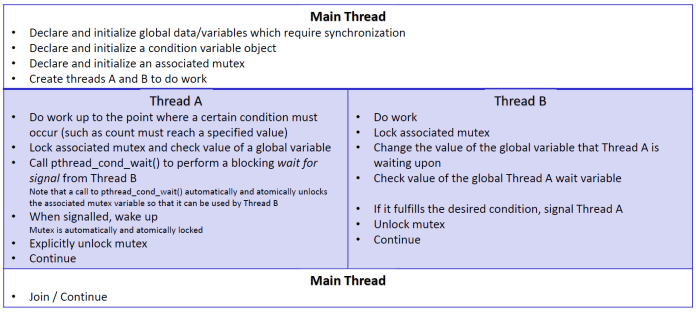
Der Austauschen von Daten zwischen verschiedenen Threads (z.B. anderen Thread benachrichtigen, wenn in eigenem Thread etwas passiert ist / warten, bis in anderem Thread etwas passiert ist) ist mittels **shared resources** möglich.
Der **Nachteil** davon ist aber, dass diese shared resource mit **polling** abgefragt werden muss ➠ nicht effizient!

Der korrekte Weg für den Informations-Austausch zwischen Threads sind **condition variables.**

13.11.1 Condition Variables

- Mit Hilfe von condition variables können Threads auf der Grundlage des aktuellen Datenwerts synchronisiert werden
  - **Kein polling nötig!**
- Condition Variables werden immer **zusammen mit einem 'Mutex lock'** verwendet

Beispiel: Anwendungsbeispiel für Condition Variables



13.12 Condition Variables mit pthreads

13.12.1 Erstellen / inizialisieren von Conditon Variables

```c
int pthread_cond_init(pthread_cond_t* condVar,          // ptr to condition variable
                      const pthread_condattr_t *attr)   // ptr to pthread_condattr_t
                                                        // structure, often 0
                                                        // (default attributes)
// returns 0 if condition variable is initiated successfully
```

13.12.2 Zerstören von Condition Variables

```c
int pthread_cond_destroy(pthread_cond_t* condVar)       // ptr to condition variable
// returns 0 if condition variable is destroyed successfully
```

13.12.3 Auf Conditon Variables warten

```c
int pthread_cond_wait(pthread_cond_t* condVar,      // ptr to condition variable
                      pthread_mutex_t* mutex)       // ptr to pthread_mutex_t instance
// returns 0 if waiting (blocking) is successful
```

13.12.4 Signalisierung mit Conditon Variables

```c
int pthread_mutex_signal(pthread_cond_t* condVar)   // ptr to condition variable
// returns 0 if signaling (unblocking) is successful
```