

# Embedded Software Engineering

## Realisierung von Finite State Machines

Prof. Reto Bonderer  
HSR Hochschule für Technik Rapperswil  
reto.bonderer@hsr.ch

Oktober 2019

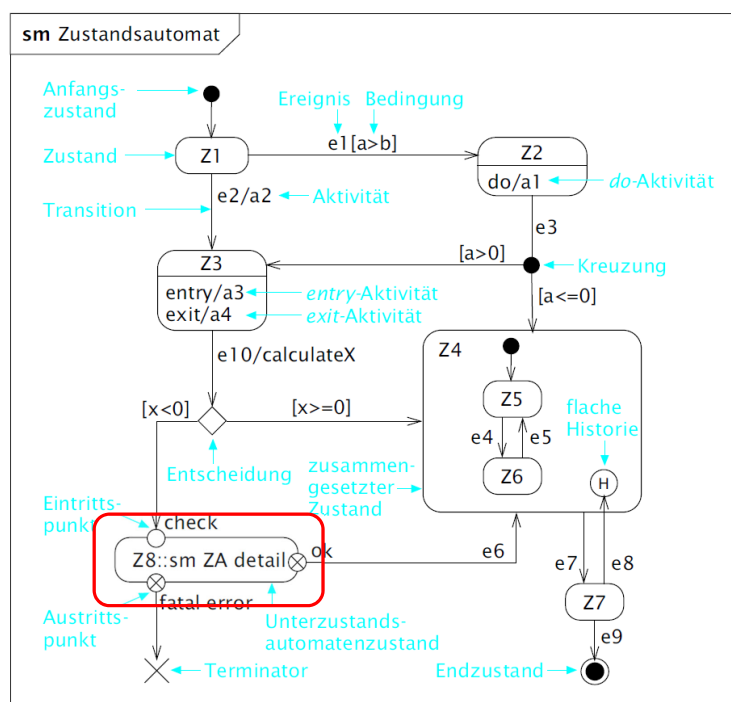
### Themen für heute

- Umsetzung einer flachen (nicht hierarchischen) Finite State Machine (FSM) in C und C++ mit
  - Steuerstruktur (switch-case)
  - Tabelle

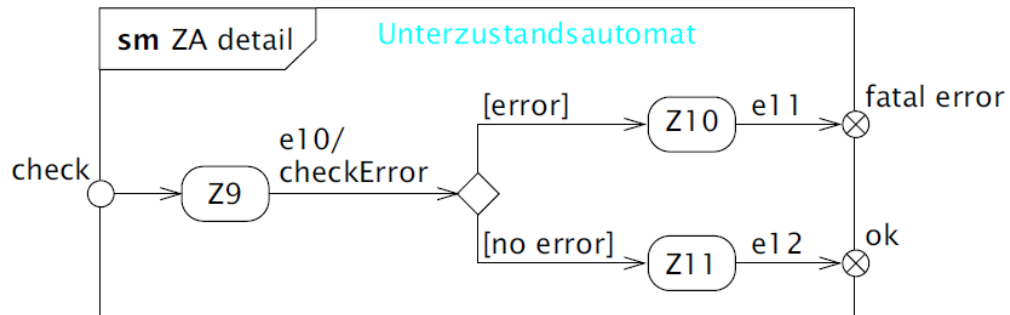


# PRO MEMORIA

## Statechart (aus Balzert)



## Detaillierung des Unterzustandsautomaten



## Realisierung von FSMs

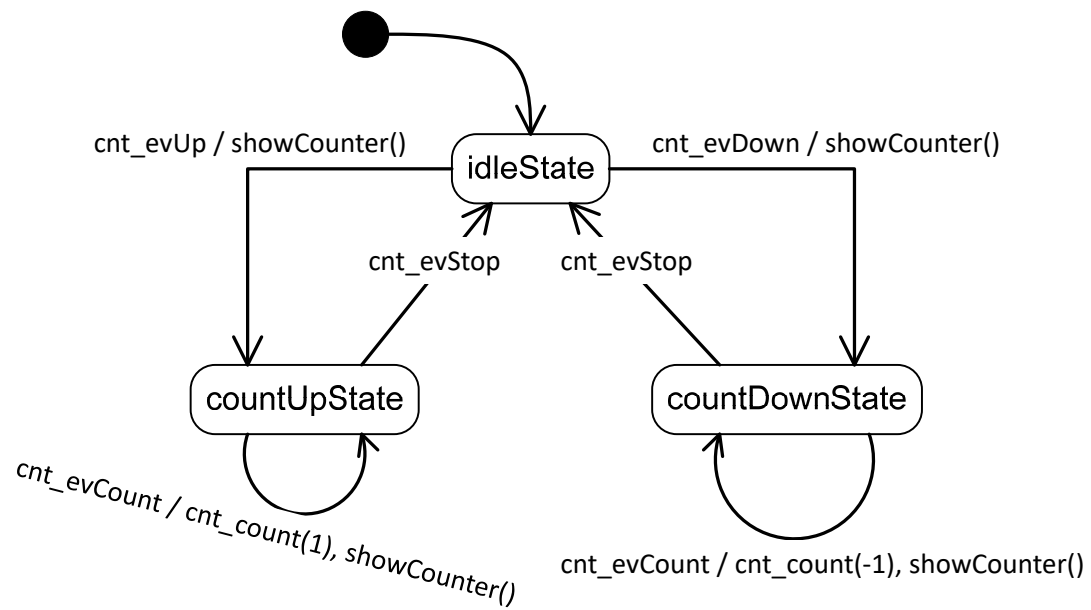
- Die Realisierung von hierarchischen FSMs ist etwas aufwendig (folgt)
- Die Realisierung von flachen (nicht hierarchischen) FSMs ist relativ einfach
- Jede hierarchische FSM kann in eine flache FSM umgewandelt werden
- Das Thema von heute ist die Realisierung von flachen FSMs

# REALISIERUNG VON FLACHEN FSMs

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

## Beispiel für unsere Betrachtungen: Up/Down-Counter



## Demo: Counterapplikation

## Realisierung gemäss Balzert

### Realisierung des Zustandsautomaten

- Kann nicht direkt in eine Programmiersprache umgesetzt werden
- Für einfache Automaten:
  - Jede Klasse erhält im Entwurf ein private-Attribut `classState`, in dem der aktuelle Zustand gespeichert wird
  - Jede Operation muss diesen Zustand abfragen
  - Ist mit der Operation ein Zustandswechsel verbunden, dann muss sie das Zustandsattribut aktualisieren
  - Alternativ kann jede Klasse, die einen Objekt-Lebenszyklus besitzt, eine Operation zur Verfügung stellen, die eintreffende Ereignisse interpretiert und ggf. eine entsprechende Verarbeitung auslöst

#### ■ Kommentar

- doch: die Umsetzung geht direkt und eindeutig
- die Realisierung hängt Balzert an den Operationen (sprich: Events) auf
- dann muss im Diagramm gesucht werden, wo dieser Event überhaupt einen Übergang auslöst
- **diese Variante ist ungeeignet und sollte nicht eingesetzt werden, die FSM wird besser an den Zuständen aufgehängt**

## REALISIERUNG MIT STEUERKONSTRUKT (PROZEDURAL IN C)

## Realisierung mit Steuerkonstrukt (prozedural)

### Zustände (States) werden in einem enum definiert (nicht public!)

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

### Ereignisse (Events) werden in einem enum definiert (public!)

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

## public vs. private

- Bei einer FSM sind die Ereignisse die Schnittstelle nach aussen. Durch die Ereignisse wird der Zustand der FSM geändert.

Die Ereignisse (Events) müssen deshalb in die Schnittstelle (mit Modulkürzel)

- Die Zustände der FSM müssen nach aussen nicht sichtbar sein. Ein Nutzer der FSM muss sich nicht darum kümmern.

Deshalb sind die Zustände (States) private und haben auch keinen Modulkürzel.

## counterCtrl.h

```
//  
// counterCtrl.h  
// implements the Finite State Machine (FSM) of an up/down-Counter  
// (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2019  
//  
  
#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
```

## Realisierung mit Steuerkonstrukt (prozedural)

### Die FSM wird in zwei Funktionen implementiert

```
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
```

Hier werden die Zustände überprüft und  
allfällige Zustandsübergänge veranlasst



## Realisierung mit Steuerkonstrukt (prozedural)

### Der aktuelle Zustand der FSM wird in einer statischen Variablen gehalten

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

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

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

## Bemerkungen zur prozeduralen Implementation

- Da `currentState` `static` ist, kann es nur eine einzige Instanz dieser FSM geben
- Wenn es mehrere Instanzen geben sollte, dann darf `currentState` nicht `static` sein und muss als Parameter mitgegeben werden, bzw. ein Pointer auf die jeweilige Variable
- Das bedingt aber auch, dass der Typ `State` wieder in die Schnittstelle muss oder dass z.B. mit `void*` gearbeitet wird
- Für einfache Anwendungen ist die hier gezeigte Variante dennoch geeignet
- Eine schöne Kapselung ist mit C jedoch nicht möglich

## void cnt\_ctrlProcess(cnt\_Event e)

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

## Anstossen der Finite State Machine

```
#include "counterCtrl.h"
int main(void)
{
    char answer;
    cnt_ctrlInit(0);
    do
    {
        switch (answer = getAnswer())
        {
            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;
}
```

## Gesamte Applikation

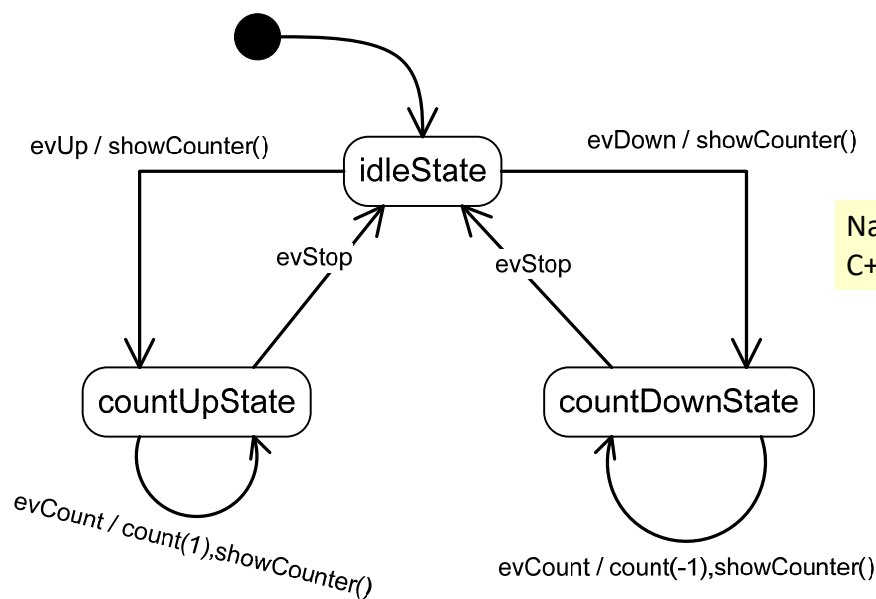
- Demo in Verzeichnis Ctrl\_C
- Eigenschaften der C-Version
  - Von der FSM kann so wie sie implementiert ist nur eine Instanz vorkommen, sonst müsste noch eine Variable für den einzelnen Automaten jedes Mal mitgegeben werden
  - Die Attribute der FSM (currentState) können nicht schön gekapselt werden
  - Die Struktur ist recht einfach und kann gut erweitert werden
  - Die Funktion cnt\_ctrlProcess() kann beliebig aufgerufen werden (z.B. aus einem periodischen Task, laufend, etc.)
  - Die Struktur der FSM ist von aussen nicht sichtbar, der FSM werden nur die eingetretenen Events übergeben
  - Bei exportierten Funktionen und Definitionen muss in C ein Modulkürzel vorangestellt werden (hier: cnt)

## REALISIERUNG MIT STEUERKONSTRUKT (OBJEKTORIENTIERT IN C++)

## Realisierung mit Steuerkonstrukt (Objektorientiert)

- Logisch gesehen funktioniert die objektorientierte Variante völlig identisch wie die prozedurale
- Dank des Klassenkonstrukts kann die FSM sauber gekapselt werden
- Mehrere Instanzen derselben FSM können einfach erstellt werden
- Ein Modulkürzel ist nicht notwendig, da alle Namen im Kontext von Klassen definiert werden
- Der Code wird eleganter, eine Performanceeinbusse ist nicht vorhanden

## Beispiel für unsere Betrachtungen: Up/Down-Counter



Namen können dank C++ vereinfacht werden

## Realisierung mit Steuerkonstrukt (objektorientiert)

States werden im **private-Teil** der Klasse mit einem enum definiert

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

Events werden im **public-Teil** der Klasse mit einem enum definiert  
(public, weil die Events zur Schnittstelle gehören)


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

## Realisierung mit Steuerkonstrukt (objektorientiert)

Die FSM wird in zwei Funktionen implementiert

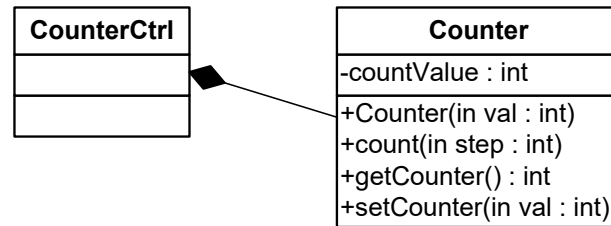
```
CounterCtrl::CounterCtrl(int initValue = 0);
// Ctor initializes counter FSM

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



Hier werden die Zustände überprüft und  
allfällige Zustandsübergänge veranlasst

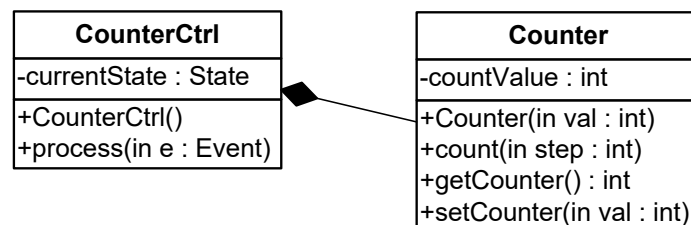
## Zusammenhang mit der Klasse *Counter*



- Die Klasse **Counter** führt die eigentlichen Rechnungsaufgaben durch
- Sie ist bei allen (objektorientierten) Realisierungsarten identisch !!
- Die Klasse **CounterCtrl** ist die FSM, welche den Zugriff auf den **Counter** steuert

## Realisierung mit Steuerkonstrukt (objektorientiert)

Der aktuelle Zustand der FSM (**currentState**) wird in einem Attribut der Klasse **CounterCtrl** gehalten



## CounterCtrl.h

```
#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); // Ctor: initializes FSM
    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;
};
#endif
```

## Ctor CounterCtrl::CounterCtrl(int initValue)

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

Die init-Funktion ist in C++ obsolet, diese Aufgabe übernimmt der Ctor

## void CounterCtrl::process(Event e)

```
void CounterCtrl::process(Event e)
{
    switch (currentState)
    {
        case idleState:
            if (evUp == e)
            { // actions
                cout << "counter = " << myCounter.getCounter() << endl;
                // state transition
                currentState = countUpState;
            }
            else if (evDown == e)
            { // actions
                cout << "counter = " << myCounter.getCounter() << endl;
                // state transition
                currentState = countDownState;
            }
            break;
        case countUpState:
            ...
    }
}
```

## Wo kommen Entry- und Exit-Actions von Zuständen hin?

- **Entry-Actions**  
müssen überall dort hinzugefügt werden, wo in einen neuen Zustand gewechselt wird.  
Üblicherweise muss die Entry-Action für einen bestimmten Zustand bei mehreren Transitionen codiert werden.
- **Exit-Actions**  
müssen überall dort hinzugefügt werden, wo ein Zustand verlassen, d.h. in einen anderen Zustand gewechselt, wird.  
Üblicherweise muss die Exit-Action für einen bestimmten Zustand bei mehreren Transitionen codiert werden



## Entry- und Exit-Actions von Zuständen: wo?

```
void CounterCtrl::process(Event e)
```

```
{
```

```
  switch (currentState)
```

```
  {
```

```
    case idleState:
```

```
      if (evUp == e)
```

```
      { // actions
```

```
        cout << "counter = " << myCounter.getCounter() << endl;
```

```
        // state transition
```

```
        currentState = countUpState;
```

```
      }
```

```
      else if (evDown == e)
```

```
      { // actions
```

```
        cout << "counter = " << myCounter.getCounter() << endl;
```

```
        // state transition
```

```
        currentState = countDownState;
```

```
      }
```

```
      break;
```

```
    case countUpState:
```

```
      ...
```

Exit-Actions von idleState

Entry-Actions von countUpState

Entry-Actions von countDownState

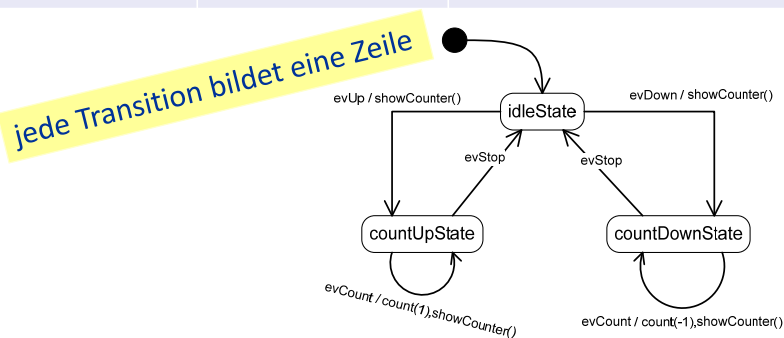
## Gesamte Applikation

- Demo in Verzeichnis Ctrl\_CPP

# REALISIERUNG MIT TABELLE

## Darstellung einer FSM als Tabelle

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



## Prozedurale oder objektorientierte Realisierung der Tabelle

- Die Tabelle kann sowohl prozedural als auch objektorientiert implementiert werden
- Die objektorientierte Variante verwendet einzig die Datenkapselung. Vererbung und Polymorphismus werden nicht benötigt
- Die objektorientierte Variante kann klarer und schöner strukturiert implementiert werden. Im folgenden wird nur diese Variante gezeigt, die C-Version kann jedoch einfach davon abgeleitet werden.

## Grundideen der Tabellenvariante

- Die ganze FSM ist in einer Tabelle gespeichert
- Die Aktionen sind als Funktion implementiert, in der Tabelle steht der entsprechende Funktionspointer
- Die Abarbeitung der FSM erfolgt mit Hilfe einer *Execution Engine*, die in der Tabelle "nachschauf", was zu tun ist
- Die Execution Engine ändert sich nicht, wenn die FSM geändert wird

## Was ändert, was bleibt gleich?

- Das Testprogramm ist völlig unverändert (den Nutzer kümmert es nicht, wie die FSM implementiert ist)
- Die Schnittstelle von CounterCtrl (public-Teil) ist ebenfalls identisch.
- Die Klasse Counter ändert auch nicht.
- Die einzige Änderung liegt im private-Teil der CounterCtrl-Klasse und natürlich in deren Implementation

## Pointer auf Klassenelemente

- Bei der Tabellenversion werden Pointer auf Klassenelemente verwendet
- Ein Beschreibung dazu finden Sie z.B. in [Strasser] Kapitel 11.8.1 *Zeiger auf Klassenelemente*, p 213

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);
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;

    typedef void (CounterCtrl::*Action)(void); // function ptr for action function
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);
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;

    typedef void (CounterCtrl::*Action)(void); // function ptr for action function
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

Events werden im **public-Teil** der Klasse mit einem enum definiert

States werden im **private-Teil** der Klasse mit einem enum definiert

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);

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;

    typedef void (CounterCtrl::*Action)(void); // function ptr for action function
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

Der Ctor initialisiert die FSM, die Methode process() stellt die Execution Engine dar

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);

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;

    typedef void (CounterCtrl::*Action)(void); // function ptr for action function
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

Das Attribut currentState speichert den aktuellen Zustand der FSM, mit dem Attribut myCounter wird die Aggregation zur Klasse Counter implementiert

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);
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;

    typedef void (CounterCtrl::*Action)(void); // ptr for action function
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

Alle (Transitions-)Aktionen werden als Methoden  
deklariert, Action wird als Funktionspointer definiert.  
Diese Methoden müssen alle mit dem Funktionspointer  
übereinstimmen.  
In unserem Fall:  
**void CounterCtrl::foo(void)**

## Tabelle: Klasse CounterCtrl

```
class CounterCtrl
{
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);
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;

    typedef void (CounterCtrl::*Action)(void); // function ptr for act
    // action functions
    void actionIdleUp(void);
    void actionIdleDown(void);
    void actionDoNothing(void);
    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 const Transition fsm[];
};
```

Die Transition wird als klasseninterne Struktur deklariert.  
Sie besteht aus

- Aktueller Zustand
- Event
- Funktionspointer auf Aktionsmethode
- Nächster Zustand

fsm[] wird als statischer offener Array deklariert. Hier  
wird die ganze FSM abgespeichert

## Tabellendefinition in CounterCtrl.cpp

```
const CounterCtrl::Transition CounterCtrl::fsm[] =
// this table defines the fsm
{
//currentState    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}
};
```

Aktueller Zustand	Event	Aktion	Nächster Zustand
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

## Aufbau einer Aktionsmethode

```
void CounterCtrl::actionDownDown(void)
{
  myCounter.count(-1);
  cout << "State: countDownState, counter = " << myCounter.getCounter() << endl;
}
```



## Performancesteigerung mit inline-Funktionen

- Die Action-Funktionen sind oft recht kurz, dennoch werden diese Funktionen immer über einen Funktionspointer aufgerufen.
- Eine naheliegende Lösung ist, alle Action-Funktionen inline zu definieren.
- Leider nützt das nichts, denn eine Funktion wird niemals inlined, wenn ein Pointer auf diese Funktion verwendet wird. Und genau das wird in der Tabelle gemacht.



## Execution Engine

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

Die Execution Engine sucht, ob ein Eintrag mit dem geforderten aktuellen Zustand und Event in der Tabelle vorhanden ist. Falls ja, wird die Aktion über den Funktionspointer aufgerufen und der nächste Zustand gesetzt. Das break verhindert, dass eine weitere Zeile bearbeitet wird, die nun aufgrund der Zustandsänderung gefunden werden könnte.

## Bemerkungen zur Execution Engine

- Vor dem Aufruf der Aktionen über den Funktionspointer müsste streng genommen überprüft werden, ob es sich um einen gültigen Pointer handelt
- Aus Performancegründen wird darauf verzichtet.
- Voraussetzung dafür ist, dass in der Tabelle immer ein gültiger Pointer auf eine Memberfunktion vorhanden ist. Wenn nichts zu tun ist, soll ein Pointer auf eine leere Funktion eingesetzt werden (im Beispiel der Pointer `&CounterCtrl::actionDoNothing`)
- Da sich die Tabelle und die Execution Engine in derselben Datei befinden, ist dieses Vorgehen unproblematisch, bzw. sogar die bevorzugte Variante.

## Gesamte Applikation

- Demo in Verzeichnis Table\_Simple

## Erweiterungsmöglichkeiten der Tabellenversion

- Wenn der Zustandsübergang nicht nur 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

```
typedef bool (CounterCtrl::*Checker)(Event);  
// function ptr for checker function  
// checker functions  
bool checkIdleUp(Event e);  
...  
  
struct Transition  
{  
    State currentState;    // current state  
    Checker pChecker;      // pointer to checker function  
    Action pAction;        // pointer to action function  
    State nextState;      // next state  
};
```

- Ergänzung für die Behandlung von Entry- und Exit-Actions

## Gesamte Applikation

- Demo in Verzeichnis Table

## Ausblick: Themen von nächster Woche

- FSM-Realisierung mit State Pattern
- Vergleich der Realisierungen
- Realisierung von hierarchischen FSMs

