

Prozesslenkung: Hierarchische Zustandsautomaten II

Implementierung Hierarchischer Zustandsautomaten

Katja Kirstein, Anne-Lena Kowalka, Marian Triebe, Eugen
Winter

1. November 2014

Themen

- ▶ Implementierungs-Arten
- ▶ Externe State-Variablen
- ▶ Guards
- ▶ Entry- und Exit-Code
- ▶ History
- ▶ Timer

Beispiel Förderband

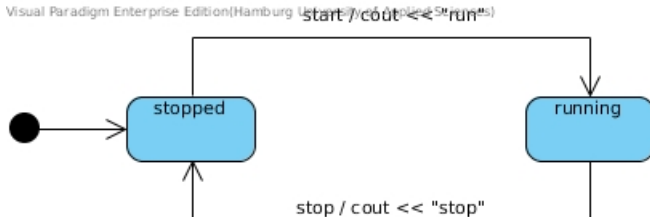
Visual Paradigm Enterprise Edition(Hamburg University of Applied Sciences)

Eingangssignale

Ausgangssignale



Visual Paradigm Enterprise Edition(Hamburg University of Applied Sciences)



Geschachtelte Switch-Case-Anweisung

- ▶ Zustände und Signale werden durch *enums* repräsentiert
- ▶ Der aktuelle Zustand wird durch eine Zustands-Variable repräsentiert
- ▶ 2-Level-Switch-Case erforderlich: Level 1 prüft den aktuellen Zustand, Level 2 prüft das erhaltene Signal
- ▶ Es empfiehlt sich, den Automaten in einer dispatch-Methode zu kapseln

Beispiel der dispatch-Methode

```
enum State{ srunning, sstopped }; // Zustaeende

enum Signal{ Start, Stop }; //Eingangssignale

void dispatch(Signal sig){
    switch(state){ // Zustand pruefen
        case srunning:
            switch(sig){ // Erhaltenes Signal pruefen
                case Start: // Bereits srunning
                    break; // -> Keine Reaktion erforderlich
                case Stop:
                    tran(ssstopped); //Transition
                    stopped(); //cout << "stop"<<endl;
                    break;
            }
            // ...
    }
};
```

Vor- und Nachteile geschachtelter Switch-Case-Automaten

- ▶ Vorteile:
 - ▶ Wenig Speicher benötigt (nur eine State-Variable)
- ▶ Nachteile
 - ▶ Die Verarbeitungszeit hängt von der Anzahl von Zuständen und Signalen ab
 - ▶ Die Umsetzung hierarchischer Automaten ist schwierig
 - ▶ In der Regel unübersichtlich und schwer erweiterbar

Übung: Geschachteltes Switch-Case

Erstellt einen Automaten (geschachteltes Switch-Case) der eine Tür darstellt. (10 Minuten)

- ▶ Eingangssignale
 - ▶ Öffnen
 - ▶ Schließen
- ▶ Zustände
 - ▶ Geöffnet
 - ▶ Geschlossen

Implementierung II: Ungeschachtelte Switch-Case-Anweisung nach Samek

- ▶ Zustände werden durch *typedef* einer Pointer-to-Member-Funktion mit einem Signal als Parameter repräsentiert:
typedef void (Kontext::*State)(unsigned const sig);
- ▶ Die dispatch-Methode übergibt das erhaltene Signal an diese State-Variable (bzw. die Pointer-to-Member-Funktion)
- ▶ Die Zustände werten die Signale dann mittels Switch-Case aus

Beispiel einer Pointer-to-Member-Funktion

```
enum Signal{ Start, Stop }; // Eingangssignale
// ...
void dispatch(unsigned const sig) {
    (this->*myState) (sig);
}
// ...
void Kontext::srunning(Signal sig) {
    switch(sig) { // In srunning das Signal pruefen
        case Start: // Da bereits in srunning...
            break;    // keine Reaktion erforderlich
        case Stop:
            tran(sstopped); // Transition
            stopped();      // cout << "stop" << endl;
            break;
    }
}
```

Vor- und Nachteile ungeschachtelter Switch-Case-Automaten

- ▶ Vorteile:
 - ▶ Leicht verständliche Struktur / einfache Umsetzung
 - ▶ Speicherschonend, da nur der state-Pointer gespeichert wird
 - ▶ Code kann wiederverwendet werden, da der generische Teil (init, dispatch und tran-Methoden) in einer Klasse gekapselt werden kann
 - ▶ Effizient, da ein Switch-Case-Level wegfällt
 - ▶ Änderungen im Automaten sind leicht umsetzbar
- ▶ Nachteile
 - ▶ Die verbliebene Switch-Case-Anweisung ist immernoch von der Anzahl der Signale abgänglich
 - ▶ Nicht für Hierarchische Automaten geeignet

Zustandstabelle

Am Beispiel des Förderbands

δ	start	stop
>stopped	running	\emptyset
running	\emptyset	stopped

Zustandstabelle Implementierung I

```
// Zustände
enum states {
    STOPPED,
    RUNNING,
};

// Eingangssignale
enum input {
    START,
    STOP
};

struct transition {
    states m_state;
    input  m_input;
    states m_next;
    void (*m_fn) (void); // Funktionspointer
};
```

Zustandstabelle Implementierung II

```
// Kontext Klasse
class conveyor {
    static void starten();
    static void stoppen();
    // Transitionstabelle
    static transition trans[];
    // Aktueller Zustand
    states m_state;
public:
    void do_action(input in);
    conveyor() : m_state(STOPPED) { }
};
```

Initialisierung des Zustandsarrays

```
transition conveyor::trans[] = {
    { STOPPED, START, RUNNING, &conveyor::starten },
    { RUNNING, STOP, STOPPED, &conveyor::stoppen }
};
```

Zustandstabelle Implementierung III

Übergangsfunktion in der Kontext Klasse

```
void conveyor::do_action(input in) {  
    for (int i = 0; i < TRANS_COUNT; ++i) {  
        if (trans[i].m_state == this->m_state &&  
            trans[i].m_input == in) {  
            trans[i].m_fn();  
            // Zustandswechsel  
            this->m_state = trans[i].m_next;  
            return;  
        }  
    }  
    // Fehlerausgabe hier moeglich  
    std::cout << "no action" << std::endl;  
}
```

Vor- und Nachteile Zustandstabelle

Überlegt euch kurz die Vor- und Nachteile dieser Implementierung
(5 Minuten)

Vor- und Nachteile Zustandstabelle

- ▶ Vorteile:
 - ▶ Einfach zu Implementieren
 - ▶ Keine NOP Operationen/Funktionen notwendig
 - ▶ Tabelle hat nur so viele Einträge wie Übergänge
- ▶ Nachteil:
 - ▶ Langsamer Zugriff

Zustandsmatrix Implementierung I

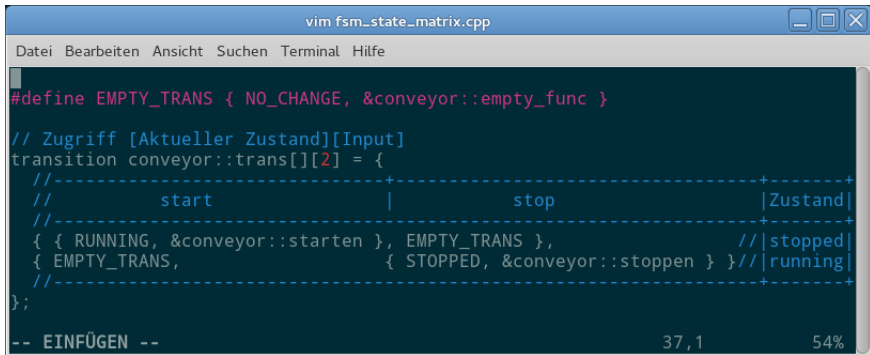
```
// Zustände
enum states {
    NO_CHANGE = -1,
    STOPPED,   // 0
    RUNNING,   // 1
};

// Eingangssignale
enum input {
    START, // 0
    STOP   // 1
};

// Ein Eintrag in der Tabelle
struct transition {
    states m_next;
    void (*m_fn) (void); // Funktionspointer
};
```

Zustandsmatrix Implementierung II

- Deklaration in der Kontext Klasse ist nun ein zweidimensionales Array!



```
vim fsm_state_matrix.cpp
Datei Bearbeiten Ansicht Suchen Terminal Hilfe

#define EMPTY_TRANS { NO_CHANGE, &conveyor::empty_func }

// Zugriff [Aktueller Zustand][Input]
transition conveyor::trans[][2] = {
    //-----+-----+
    //          start          |          stop          |Zustand|
    //-----+-----+
    { { RUNNING, &conveyor::starten }, EMPTY_TRANS },           //|stopped|
    { EMPTY_TRANS,          { STOPPED, &conveyor::stoppen } } //|running|
    //-----+-----+
};

-- EINFÜGEN --                                     37,1          54%
```

Zustandsmatrix Implementierung III

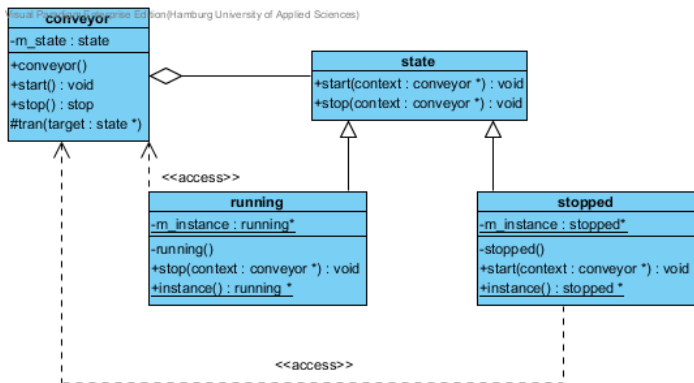
Übergangsfunktion in der Kontext Klasse

```
void conveyor::do_action(input in) {  
    int dim1 = static_cast<int>(this->m_state);  
    int dim2 = static_cast<int>(in);  
    trans[dim1][dim2].m_fn();  
    if (trans[dim1][dim2].m_next != NO_CHANGE) {  
        this->m_state = trans[dim1][dim2].m_next;  
    }  
}
```

Vor- und Nachteile Zustandsmatrix

- ▶ Vorteile:
 - ▶ Gute Performance
 - ▶ Weniger Speicherverbrauch pro Eintrag
- ▶ Nachteil:
 - ▶ Schwerer zu implementieren
 - ▶ Matrix benötigt insgesamt mehr Speicher, da alle Fälle auskodiert werden müssen
 - ▶ NOP Funktion benötigt

GoF State Pattern Struktur



- ▶ Kontext Klasse (conveyor)
- ▶ Zustands Basisklasse (state)
- ▶ Zustände (running, stopped)

GoF State Klasse

- ▶ Alle States erben von einer Oberstate Klasse
- ▶ Der Oberstate implementiert alle Events aller Zustände als leere virtuelle Funktionen
- ▶ Der jeweilige Zustand implementiert nur seine eigenen Events, nicht relevante Events werden an die leeren Methoden der Basisklasse weitergereicht

GoF Kontext Klasse

- ▶ Sicht von Außen auf die FSM
- ▶ Dient als Delegator zu den Zuständen
- ▶ Bedient alle Eingangssignale durch Funktionen

GoF Implementierung I Header

```
class conveyor {  
    friend class stopped;  
    friend class running;  
public:  
    conveyor() : m_state(stopped::instance()) { }  
    void start();  
    void stop();  
private:  
    state* m_state;  
    void tran(state* target);  
};
```


GoF Implementierung II Header

```
class running : public state {  
    static running* m_instance;  
public:  
    void stop(conveyor* context);  
    static running* instance() {  
        if (!m_instance) {  
            m_instance = new running();  
        }  
        return m_instance;  
    }  
private:  
    running() { };  
};
```

GoF Implementierung III

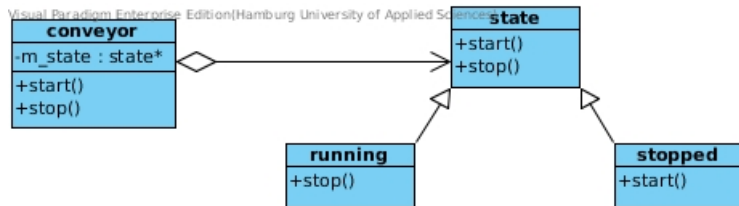
```
// Zustandswechsel Funktion
void conveyor::tran(state* target) {
    m_state = target;
}

// Transition running -> stopped
void running::stop(conveyor* context) {
    cout << "running: stop()" << endl;
    context->tran(stopped::instance());
}
```

Schwächen des klassischen GoF State Pattern

- ▶ Hoher Speicherverbrauch, da jeder Zustand im Speicher gehalten werden muss, auch wenn diese eigentlich nicht verwendet werden
- ▶ Der höhere Speicherverbrauch kann allerdings mit dem *Placement New* Operator umgangen werden
- ▶ Kontext muss eventuell immer mit übergeben werden

GoF nach Pareigis/Manske



GoF States in Code (Header)

```
class state {  
    public:  
        virtual ~state() { }  
        virtual void start() { }  
        virtual void stop() { }  
};  
  
class running : public state {  
    public:  
        void stop();  
};  
  
class stopped : public state {  
    public:  
        void start();  
};
```

GoF States in Code (Implementierung)

```
// Transition running -> stopped
void running::stop() {
    new (this) stopped;
    cout << "stop() / stop" << endl;
}
```

```
// Transition stopped -> running
void stopped::start() {
    new (this) running;
    cout << "start() / run" << endl;
}
```

GoF Kontext Klasse in Code

Header:

```
class conveyor {  
    public:  
        conveyor() : m_state(new stopped) { }  
        ~conveyor() { delete m_state; }  
        void start();  
        void stop();  
    private:  
        state* m_state;  
};
```

Implementierung:

```
void conveyor::start() {  
    m_state->start();  
}  
  
void conveyor::stop() {  
    m_state->stop();  
}
```

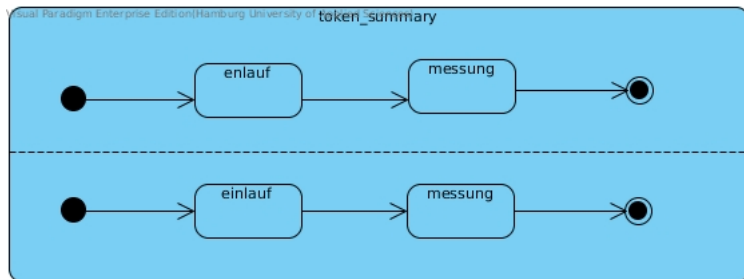
Schritte zum Erstellen einer FSM

1. Erstellen der Systemgrenzen
2. Erstellen einer FSM
3. Umwandlung in Code
 - ▶ Kontext Klasse, Funktionsnamen als Eingangssignale
 - ▶ Basisklasse für Zustände erstellen
 - ▶ Zustandsklassen erben von der Basisklasse und implementieren nur die eigenen Reaktionen neu
4. Prüfen, ob Code und Systemgrenzen/Diagramm zusammenpassen
 - ▶ Sind alle Eingangssignale als Funktionen zu finden?
 - ▶ Werden alle Ausgangssignale verwendet/ausgegeben?
 - ▶ Die Namen für Eingangssignale/Ausgangssignale sollten konsistent sein, da Fehler sonst vorprogrammiert sind!

Übung: Lichtschalter

Erstellt die Systemgrenzen, das Automatendiagramm und den Code eines Lichtschalters mit An- und Aus-Button (15 Minuten)

Orthogonale Automaten



- ▶ Alle Automaten bekommen das Eingangssignal
- ▶ Nur die betreffende Instanz reagiert

Implementierung Orthogonaler Automaten I

```
class token_summary {  
    std::vector<token*> m_tokens;  
public:  
    void entry();  
    void height();  
    void del();  
};
```

Implementierung Orthogonaler Automaten II

```
void token_summary::entry() {  
    m_tokens.push_back(new token());  
    std::cout << "Neuer Token auf Band" << std::endl;  
}
```

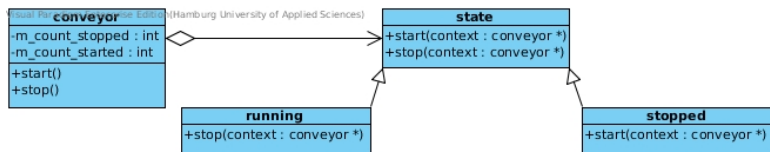
```
void token_summary::height() {  
    for(size_t i = 0; i < m_tokens.size(); ++i) {  
        m_tokens[i]->height();  
    }  
}
```

```
void token_summary::del() {  
    for(size_t i = 0; i < m_tokens.size(); ++i) {  
        m_tokens[i]->del();  
    }  
    token* ptr = m_tokens.front();  
    m_tokens.erase(m_tokens.begin());  
    delete ptr;  
}
```

Orthogonale Automaten in SE2P

- ▶ Jeder Puk/Token ist ein Automat
- ▶ Puks/Tokens müssen verwaltet werden (z.B. token_summary)

Externe State-Variablen



Implementierung von externen State-Variablen I

```
class conveyor {  
    friend class running;  
    friend class stopped;  
public:  
    conveyor()  
        : m_state(new stopped),  
          m_count_stopped(0),  
          m_count_started(0) { }  
    ~conveyor() { delete m_state; }  
    void start();  
    void stop();  
private:  
    state* m_state;  
    int m_count_stopped;  
    int m_count_started;  
};
```

Implementierung von externen State-Variablen II

Header:

```
class state {
public:
    virtual ~state() { }
    virtual void start(conveyor* context) { }
    virtual void stop(conveyor* context) { }
};

class running : public state {
public:
    void stop(conveyor* context);
};

class stopped : public state {
public:
    void start(conveyor* context);
};
```


Implementierung von externen State-Variablen III

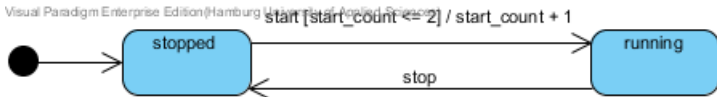
Implementierung:

```
// Transition stopped -> running
void stopped::start(conveyor* context) {
    ++context->m_count_started;
    new (this) running;
}

void conveyor::start() {
    m_state->start(this);
}
```

Guards

Visual Paradigm Enterprise Edition (Hamburg University of Applied Sciences)



Guards Implementierung

Implementierung:

```
// Transition stopped -> running
void stopped::start(conveyor* context) {

    ++context->m_count_started;

    if (context->m_count_started <= 2) {
        new (this) running;
    } else {
        return;
    }
}
```

Übung: Lichtschalter II

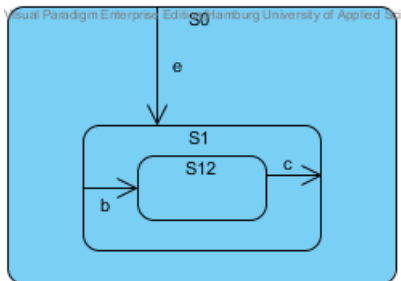
Implementiert einen Lichtschalter, der nach mehrfachem Betätigen der Starttaste in ein anderes Licht wechselt (10 Minuten)

Entry/Exit Code in HSM

- ▶ Zustandswechsel dürfen Hierarchieebenen nicht überspringen
- ▶ `exit()` wird durchgeführt, wenn Signale in der Hierarchie "nach oben" weitergereicht werden
- ▶ `entry()` wird durchgeführt, wenn Signale in der Hierarchie "nach unten" weitergereicht werden

Entry/Exit Code in HSM

- ▶ Beispiel exit: State S12 erhält Signal e



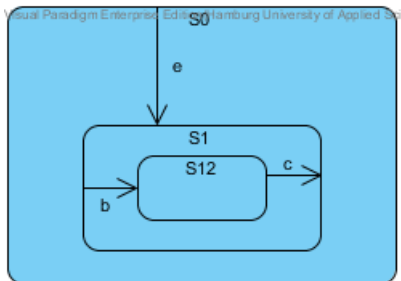
Entry/Exit Code in HSM

```
//State S12 erhaelt Signal e
void StateS12::sigE() {
    exit();
    new (this) StateS1;
    sigE();
}
```

- ▶ Die exit-Funktion ist in jedem Zustand definiert
- ▶ Wenn auch der Top-Zustand nicht auf ein Signal reagiert, sollte dieser z.B. eine failure-Methode haben, um den Fehler anzuzeigen

Entry/Exit Code in HSM

- ▶ Beispiel entry: State S1 erhält Signal b



Entry/Exit Code in HSM

```
// State S1 erhaelt Signal b
void StateS1::init(T* t) {

    // t ist ein Pointer auf die Kontext Klasse
    void* history = t->getStateFromHistory(StateS1ID);

    if(history != 0) {
        memcpy(this, &history, 4);
    } else {
        new (this) StateS12;
        entry();
    }
    init();
}
```

- ▶ Die entry-Funktion ist in jedem Zustand definiert
- ▶ `init()` führt dazu, dass - sofern ein Substate bereits in der History vorliegt - dieser direkt betreten wird.
Ansonsten wird er neu erzeugt
- ▶ Die `init()`-Funktion in der letzten Ebene ist leer

History

- ▶ Implementierung der flachen History
- ▶ Die history-Funktion wird in der exit-Funktion aufgerufen
- ▶ Die Kontext Klasse hält ein History-Array, in dem sich die Subzustände mittels history-Aufruf eintragen (Index-Zuordnung z.B. über *enums*)

History

- ▶ Beispiel: S1 trägt sich in History-Tabelle ein

```
// StateS1 history-Funktion
void StateS1::history(T* t) {
    // t ist ein Pointer auf die Kontext Klasse
    t->setHistory(State::StateID::StateS1_ID,
                 this);
}
```

- ▶ Der Parameter StateS1_ID gibt den Index in der History-Tabelle in der Kontext Klasse an

History

- ▶ setHistory in der Kontext Klasse:

```
void setHistory(int ID, State* ptr) {  
    history_[ID] = *((void**) ptr);  
}
```

- ▶ ptr wird zunächst auf void** gecastet, damit man mit einer weiteren Dereferenzierung * an die virtuelle Funktionstabelle des Zustands gelangt

History

- ▶ getStateFromHistory in der Kontext Klasse:

```
void* getStateFromHistory(int ID) {  
    return history_[ID];  
}
```

- ▶ Über seine ID kann der Zustand sich seine History - sofern eine vorliegt - aus der History-Tabelle in der Kontext Klasse holen

Timer

► Definition der Zeit-Messung eines Timers:

- **Absolut:** Löse aus **am** 24.12.2014
(Zeit vergangen seit 01.01.1970 00:00)
- **Relativ:** Löse aus **in** 10 Sekunden
(Zeit vergangen seit Start des Timers)

Timer

- ▶ Definition des Zeit-Intervalls eines Timers:
 - **Periodisch:** Löse alle 100 Millisekunden aus
 - **One-Shot:** Löse einmalig in 10 Sekunden aus

- ▶ Ereignis beim Auslösen eines Timers:
 - **Pulse Message:** Sende eine Pulse Message
 - **Signal senden:** Sende ein Signal (z.B. SIGTERM)
 - **Thread starten:** Starte einen bestimmten Thread

Timer

- ▶ Erforderliche Schritte für die Nutzung eines Timers:
 1. Timer Objekt erstellen
 2. Entscheiden, wie man benachrichtigt werden möchte (Pulse Message, Signal, Thread) und dementsprechend das **struct sigevent** initialisieren
 3. Entscheiden, welche Art von Timer man wählt (relativ vs. absolut & one-shot vs. periodisch)
 4. Timer starten

Timer Beispiel

Beispiel für die Erstellung eines Timers (1/2):

```
// 1.
timer_t timerid;
struct sigevent event;
struct itimerspec timer;

// 2.
SIGEV_PULSE_INIT(
    &event, // struct sigevent
    coid,  // Connection ID of message receiver
    SIGEV_PULSE_PRIO_INHERIT, // Priority
    MY_CODE_TIMER, // Code for pulse handler
    MY_VALUE_TIMER // Value for pulse handler
);
```

Timer Beispiel

Beispiel für die Erstellung eines Timers (2/2):

```
// 3. Create timer with realtime clock
timer_create(CLOCK_REALTIME, &event, &timerid);

// Setup the timer (2s delay, 1s reload)
// it_value = one-shot value
// it_interval = reload value
timer.it_value.tv_sec = 2;
timer.it_value.tv_nsec = 0;
timer.it_interval.tv_sec = 1;
timer.it_interval.tv_nsec = 0;

// 4. Start the timer
timer_settime(timerid, 0, &timer, NULL);
```

- Hinweis für die Abfrage des Benachrichtigung-Typs:

```
// Don't read the struct sigevent directly
if(event.sigev_notify == SIGEV_PULSE)

// Use this macro instead
if(SIGEV_GET_TYPE(&event) == SIGEV_PULSE)
```

Prüfungsfragen

- ▶ Welche Arten von Implementierungen gibt es?
- ▶ Wie geht man schrittweise vor einen Automaten zu implementieren?
- ▶ Was ist bei State-Variablen zu beachten und welche Nachteile entstehen dadurch?
- ▶ Wie geht man vor einen Timer zu implementieren?
- ▶ Wie würde ein Automat, der Doppelklicks erkennt, aussehen?

Beispielcode / Momentics Projekt

Beispielcode + Momentics Projekt für das Timer Beispiel findet ihr in unserem Git-Repository: <https://github.com/Hamdor/PLHSM2>