HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Embedded Software Engineering
Realisierung von Finite State Machines (2)

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

Oktober 2019

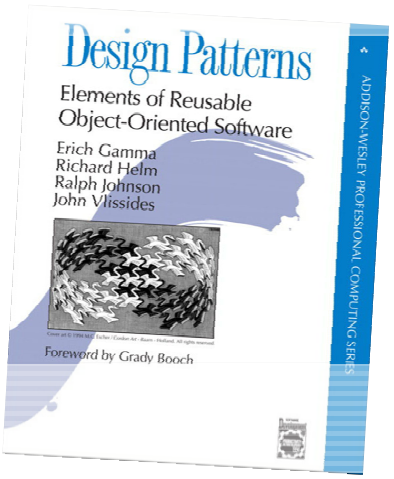
Themen für heute

- Umsetzung einer flachen (nicht hierarchischen) Finite State Machine (FSM) in C++ mit Hilfe des State Patterns
- Realisierung von hierarchischen Finite State Machines

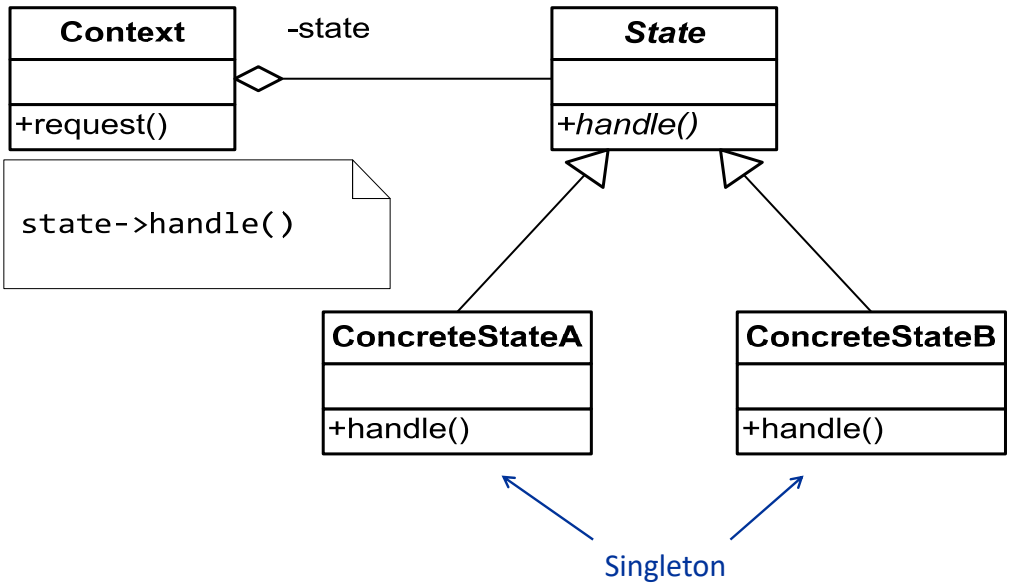
Realisierung mit State Pattern (Objektorientiert in C++)

Realisierung mit State Pattern

- Das *State Pattern* wurde 1994 im berühmten Patterns-Buch der GoF (Gang of Four: Gamma, Helm, Johnson, Vlissides) vorgestellt
- Das Prinzip besteht aus einer abstrakten State-Basisklasse. Pro Zustand muss eine eigene Klasse definiert werden, die von dieser Basisklasse erbt
- Die Realisierung nutzt das Polymorphismus-Konzept, jede konkrete State-Klasse ist ein *Singleton* (siehe ebenfalls [GoF 1994])
- Statt lange switch-case-if-Konstrukte wird beim State Pattern ein bestimmter Zustand vollständig in einer eigenen Klasse realisiert. Die Logik wird somit aufgeteilt.

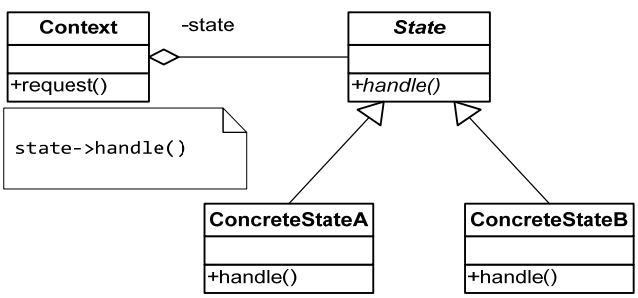


Struktur des State Pattern [GoF 1994]



Teile (participants) des State Pattern [GoF 1994]

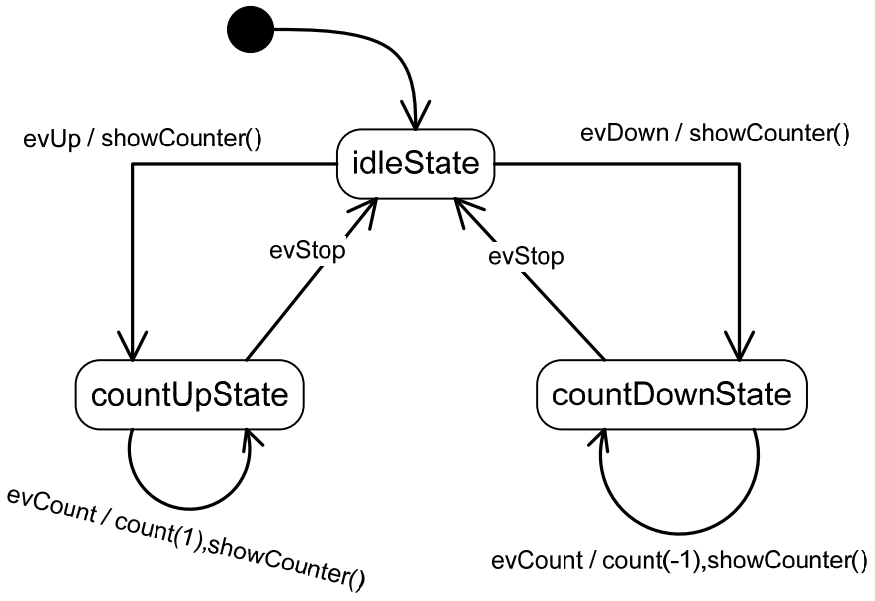
- Context
  - definiert den interessierende Schnittstelle für Clients
  - unterhält eine Instanz einer konkreten Unterklasse von `State`, die den aktuellen Zustand definiert, bzw. repräsentiert
- State
  - definiert eine Schnittstelle zur FSM in Form einer abstrakten Klasse
- ConcreteStateX Unterklassen
  - jede Unterklasse (ist Singleton) implementiert genau einen Zustand



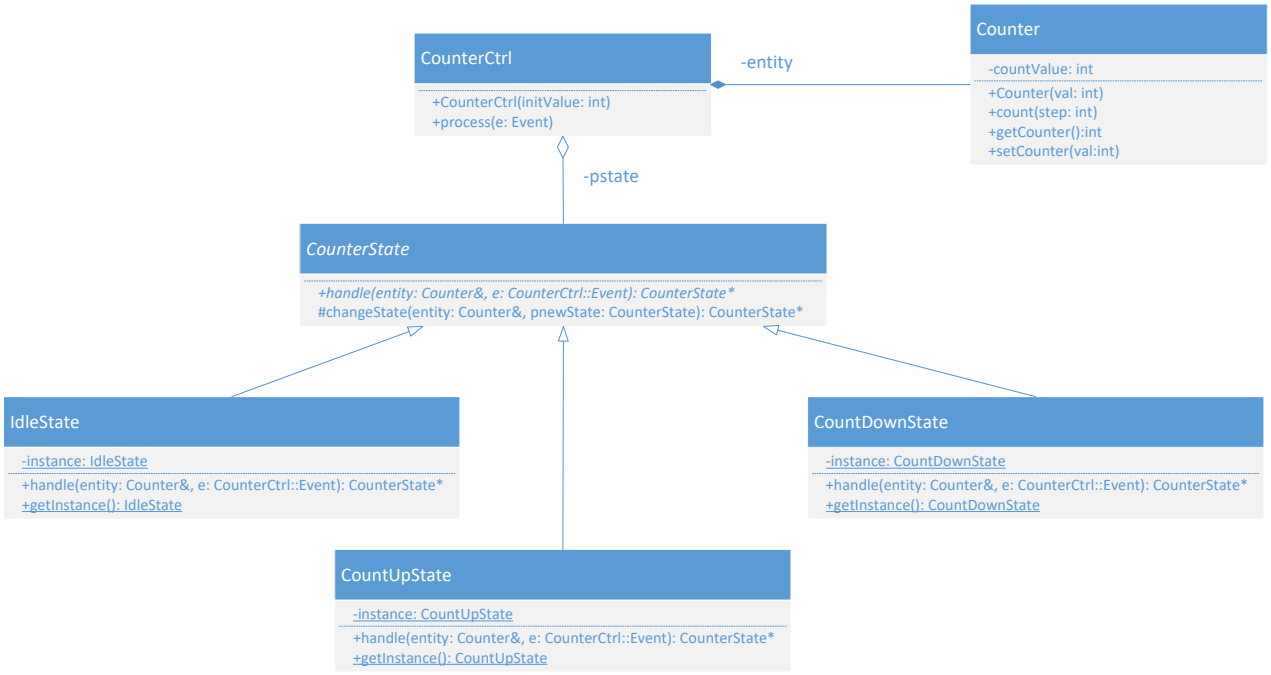
Wer definiert die State Transitions?

- Das State Pattern definiert die Zuständigkeit nicht
- Die Transitionen könnten in der `Context`-Klasse definiert werden. Der Nachteil dieser Variante ist, dass dort zentral sehr viel Intelligenz vorhanden sein müsste. Da diese Klasse auch den Zugriff von der Aussenwelt darstellt, sollte sie möglichst schlank sein.
- Die bessere Variante ist, wenn die `State`-Klassen auch gleich ihre Transitionen realisieren. Diese Variante wir oft mittels `friend`-Deklaration realisiert, was jedoch nicht nötig ist.

Beispiel für unsere Betrachtungen: Up/Down-Counter

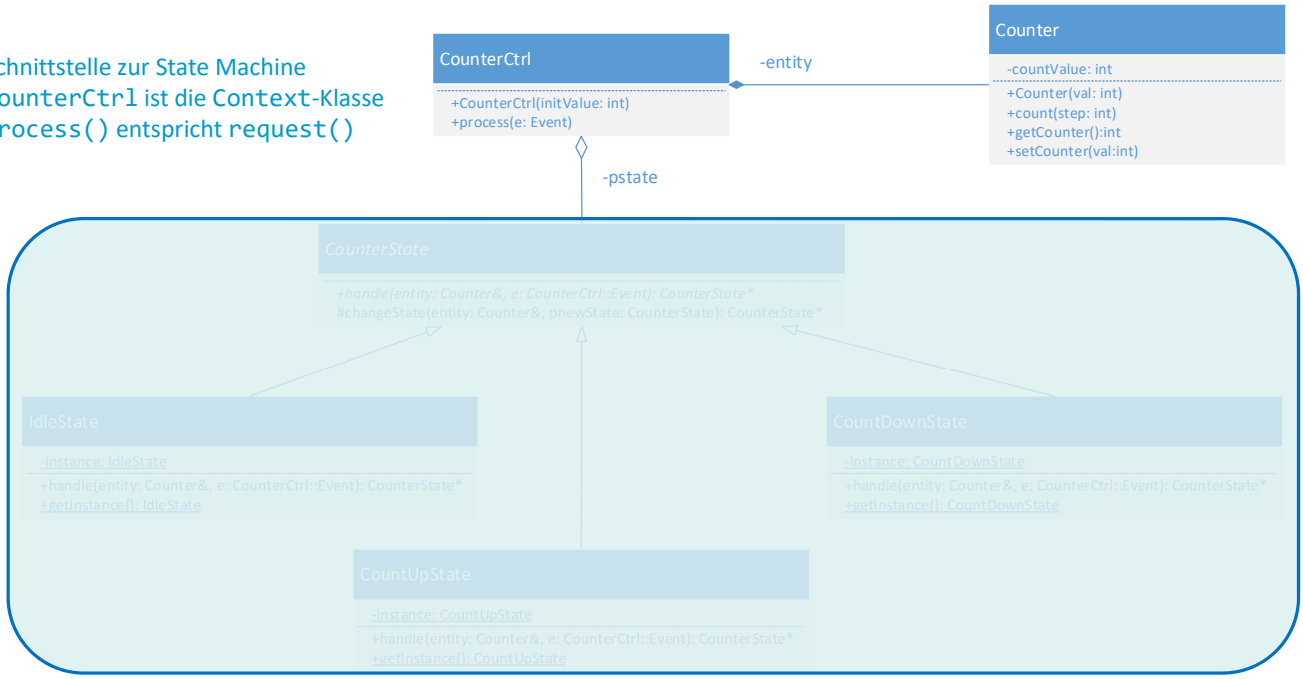


Klassendiagramm des Counters mit dem State Pattern



Klassendiagramm des Counters mit dem State Pattern

Schnittstelle zur State Machine
`CounterCtrl` ist die `Context`-Klasse
`process()` entspricht `request()`



CounterCtrl.h: Schnittstelle zur FSM

```
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);         // the Ctor
    void process(Event e);    // changes the state of the FSM based on the event 'e'
  private:
    Counter entity;
    CounterState* pState;     // holds the current state
};
```


CounterCtrl.cpp

```
#include "CounterCtrl.h"
#include "CounterState.h"

CounterCtrl::CounterCtrl(int initValue) :
  entity(initValue),
  pState(IdleState::getInstance())        // initial state
{
}

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

class CounterState: Interface

```
#include "CounterCtrl.h"  // Events are defined here

class CounterState  // abstract base class
{
  public:
    virtual CounterState* handle(Counter& entity,
                                 CounterCtrl::Event e) = 0;
    // returns new state
  protected:
    CounterState* changeState(Counter& entity,
                              CounterState* pnewState);
};
```


class CounterState: Implementation

```
CounterState* CounterState::changeState(Counter& entity,
                                        CounterState* pnewState)
{
  return pnewState;
}
```

- Diese Funktion könnte für diese Anwendung selbstverständlich auch wegoptimiert werden
- Aus Kompatibilitätsgründen für (noch folgende) Erweiterungen soll sie jedoch beibehalten werden

class CountUpState: Interface

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

- `getInstance()` ist eine statische Funktion mit class scope. Sie ist zusammen mit `instance` charakteristisch für das Singleton
- `handle()` ist spezifisch für diesen State zu implementieren
- Der private Default Ctor verhindert, dass von dieser Klasse Objekte erzeugt werden können

class CountUpState: Implementation

```
CountUpState CountUpState::instance;       // definition of static object
CountUpState* CountUpState::getInstance()
{return &instance;}

CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
{
  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());
  }
  return this;       // Important! Why?
}
```

Gesamte Applikation

- Demo in Verzeichnis `Pattern`

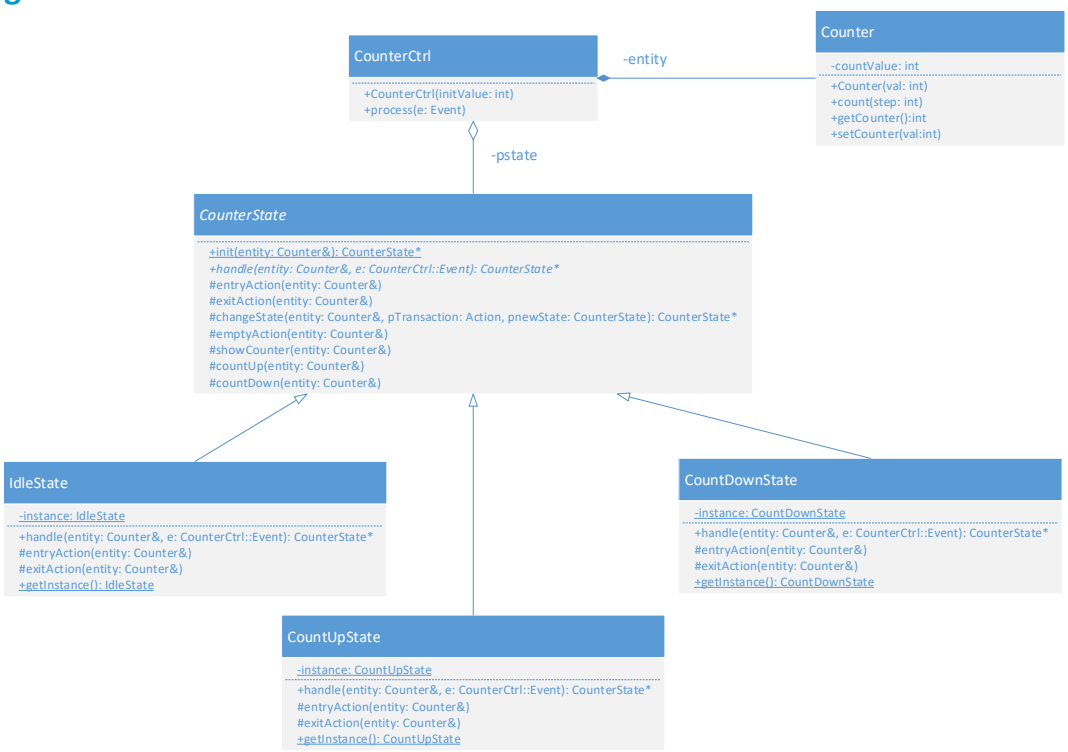
ERWEITERUNG: ENTRY- UND EXIT-ACTIONS

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

Entry-Actions und Exit-Actions

- müssen überall dort aufgerufen werden, wo in einen Zustand gewechselt, bzw. ein Zustand verlassen wird, d.h. die Aufrufe können in der Basisklassen-Methode `changeState()` isoliert vorgenommen werden
- Zwischen der Exit- und der Entry-Action müssen noch allfällige Transitions-Actions ausgeführt werden. Diese wird der Methode als Funktionspointer übergeben.
- In der Basisklasse werden zwei neue virtuelle Methoden `entryAction()` und `exitAction()` deklariert. Diese Methoden können in den abgeleiteten Klassen implementiert werden. Eine Default-Implementation in der Basisklasse ist jedoch sinnvoll.
- Die Schnittstelle zur FSM, d.h. das Interface `CounterCtrl` in `CounterCtrl.h` ändert sich nicht

Klassendiagramm des Counters mit dem State Pattern inkl. Actions



CounterCtrl.cpp

```
#include "Counter.h"
#include "CounterState.h"
#include "CounterCtrl.h"

CounterCtrl::CounterCtrl(int initValue) :
  entity(initValue),
  pState(CounterState::init(entity)) // initial state incl. entryAction
{
}

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


class CounterState: Interface

```
#include "CounterCtrl.h"  // Events are defined here

class CounterState  // abstract base class
{
  public:
    static CounterState* init(Counter& entity);    // should be called first, returns new state
    virtual CounterState* handle(Counter& entity, CounterCtrl::Event e) = 0;
  protected:
    virtual void entryAction(Counter& entity) {};
    virtual void exitAction(Counter& entity) {};
    typedef void (CounterState::*Action)(Counter& entity);
    CounterState* changeState(Counter& entity,
                              Action ptransAction,
                              CounterState* pnewState);
    // transition actions
    void emptyAction(Counter& entity) {};
    void showCounter(Counter& entity);
    // ...
};
```

class CounterState: Implementation

```
CounterState* CounterState::init(Counter& entity)  // is static
{
  CounterState* initState = IdleState::getInstance();
  initState->entryAction(entity);      // executes entry action into init state
  return initState;
}


CounterState* CounterState::changeState(Counter& entity,
                                        Action ptransAction,
                                        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;
}
```


class CountUpState: Interface

```
class CountUpState : public CounterState // it's a singleton
{
  public:
    static CountUpState* getInstance();
    virtual CounterState* handle(Counter& entity,
                                 CounterCtrl::Event e);
  protected:
    virtual void entryAction(Counter& entity);
    virtual void exitAction(Counter& entity);
  private:
    CountUpState() {};
    static CountUpState instance;
};
```

- `entryAction()` und `exitAction()` müssen nur dann deklariert und überschrieben werden, wenn es wirklich etwas Spezifisches zu tun gibt
- Andernfalls wird die Default-Implementation der Basisklasse genommen

class CountUpState: Implementation

```
CountUpState CountUpState::instance;       // Definition des statischen Objekts
CountUpState* CountUpState::getInstance()
{return &instance;}

CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
{
  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;
}
```


class CountUpState: Entry- und Exit-Actions

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

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

Gesamte Applikation

- Demo in Verzeichnis `Pattern_Action`

REALISIERUNG VON HIERARCHISCHEN STATE MACHINES

Toolunterstützung

- IBM Rhapsody
- Mathworks Matlab/Simulink mit RT Workbench
- u.a.

QP-Framework

- Grundidee
  - Framework für Ausführung mit oder ohne Betriebssystem
  - www.state-machine.com
  - Ressourcenbedarf für QP-Framework:
    1 kB RAM
    10 kB ROM
  - Targets:
    z.B. Cortex-M4
  - QP-nano für kleine Controller
    z.B. MSP430, 8051, PIC, AVR
  - Ressourcenbedarf für QP-nano:
    100 Bytes RAM
    2 kB ROM