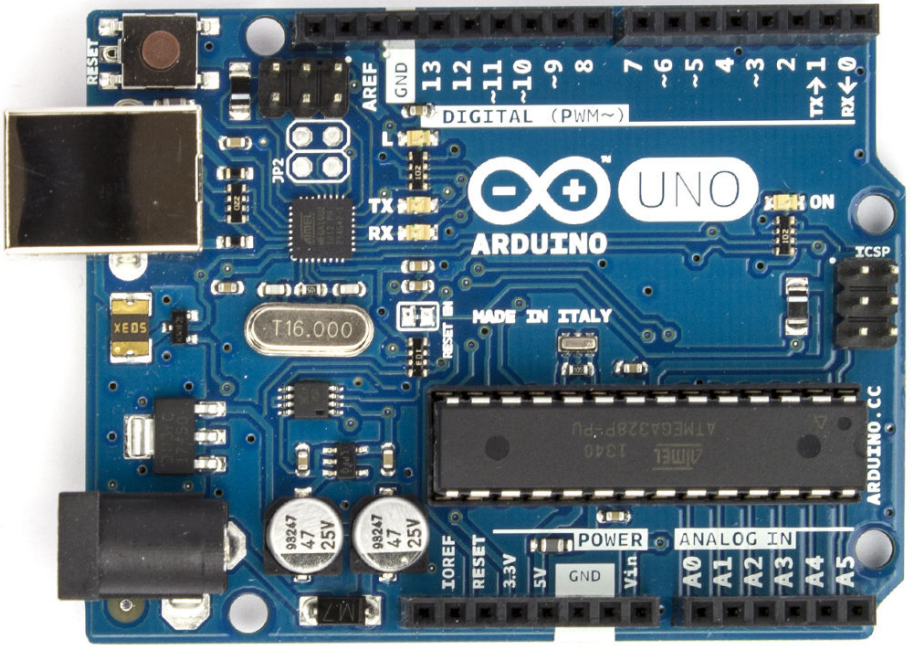
Ereignisorientierte Mikrocontroller-Programmierung mit Rizzly

Urs Fässler

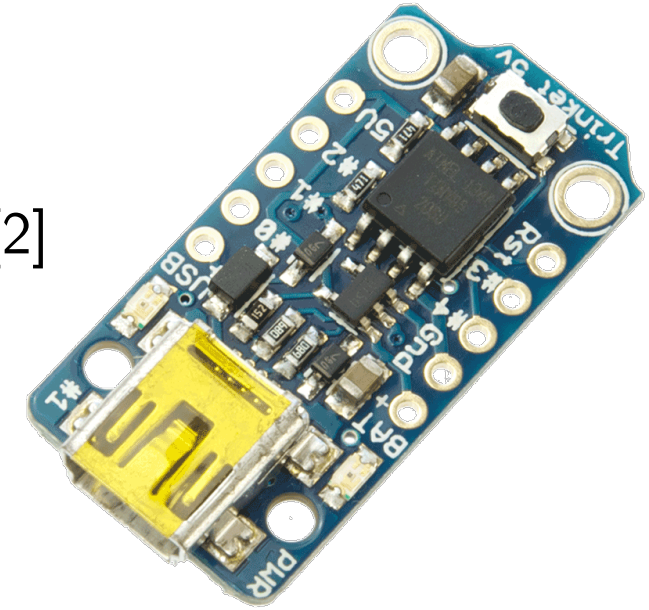
16. LinuxDay
Dornbirn

22.11.2014

Mikrocontroller



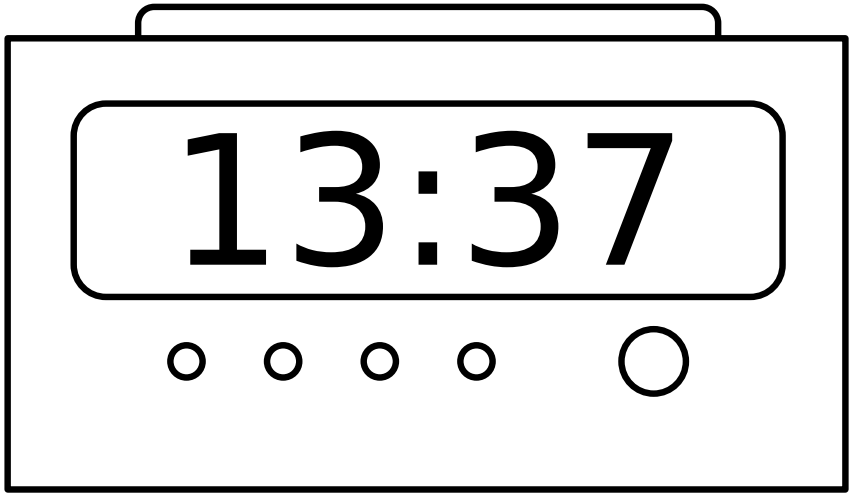
[2]

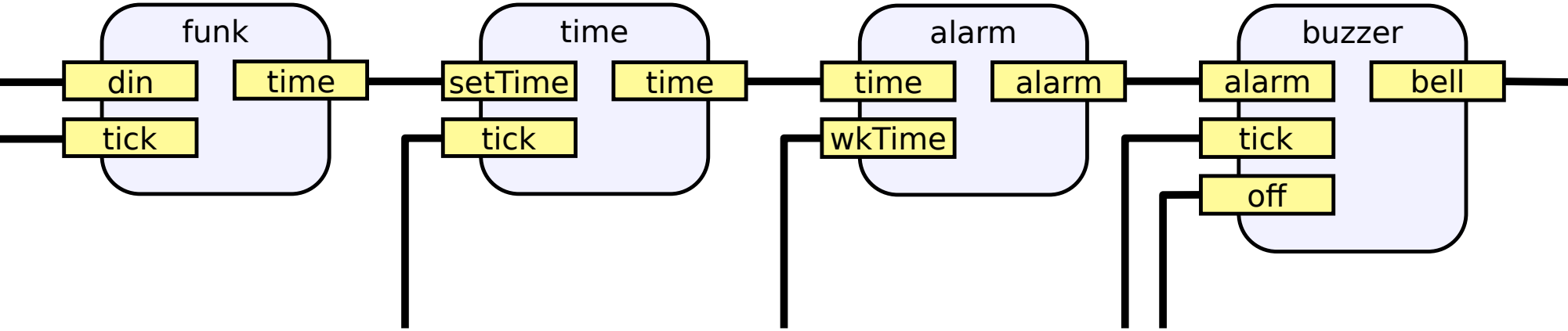


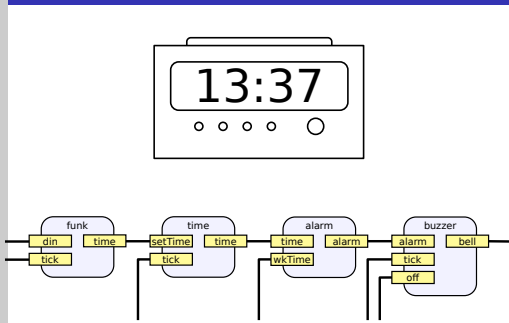
[1]

- 8 Bit
- ab 4 MHz
- ab 128 Byte RAM
- ab 2 KiB ROM
- kein OS, keine Speicherverwaltung
- ganze Umgebung zur Kompilierzeit bekannt
- Interrupts

Beschäftigung, Laufzeit und Kontrolle



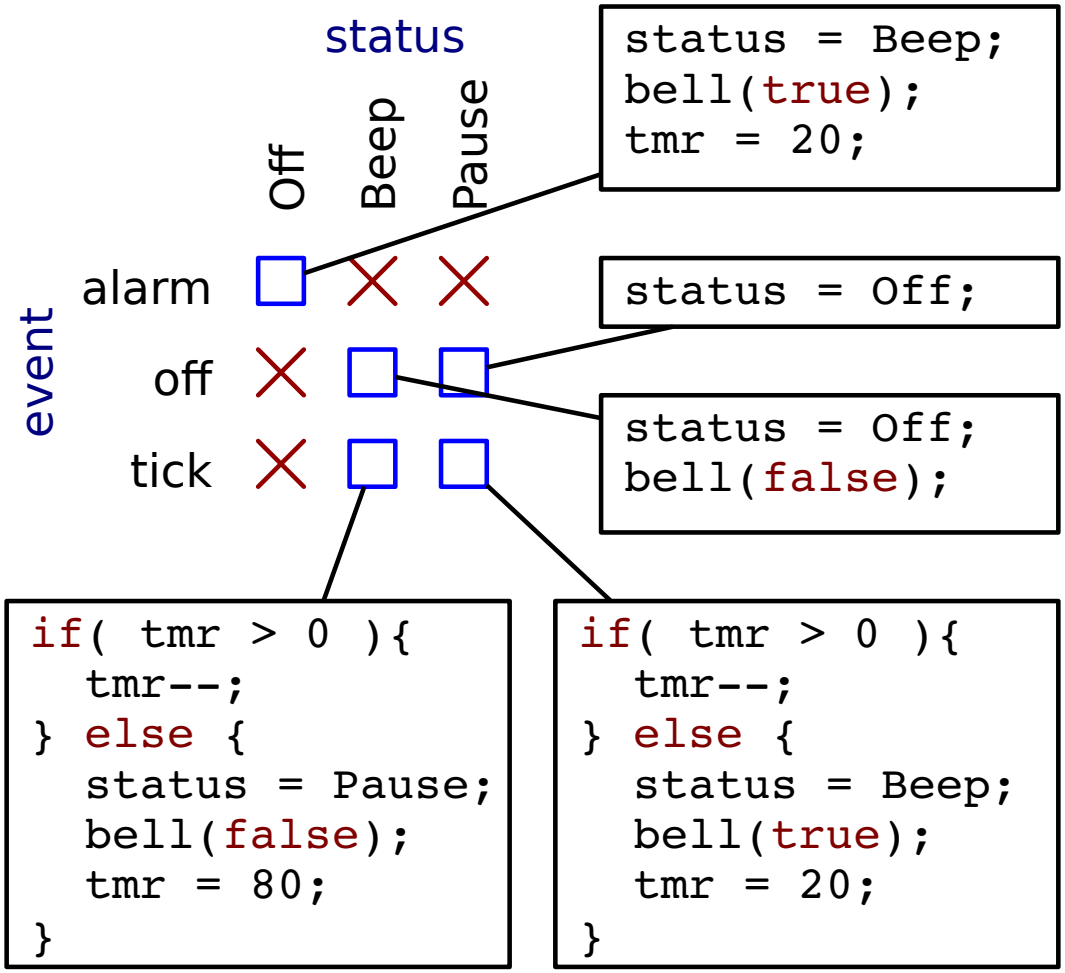




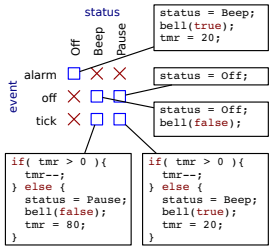
- Wir schauen nicht ganze Zeit auf Wecker
- Wecker ist ereignisorientiertes System
- Programm läuft sehr lange
- Prozessor ist meist am Nichtstun

- Hollywood principle[10]
- Inversion of Control[8]
- muss nicht selbst Daten holen
- man wird informiert
- callback
- Interrupt

Programmfluss



Programmfluss



- hoechts nicht-linearer Programmfluss
  - viele Ereignisse möglich
  - Reaktion auch Zustandsabhängig
- State Machine gute Lösung
- Beispiel Beeper
  - alarm: starte
  - off: stoppe, je nach Zustand buzzer ausschalten
  - tick: wenn aktiv timer dekrementieren; wenn 0 Zustand wechseln

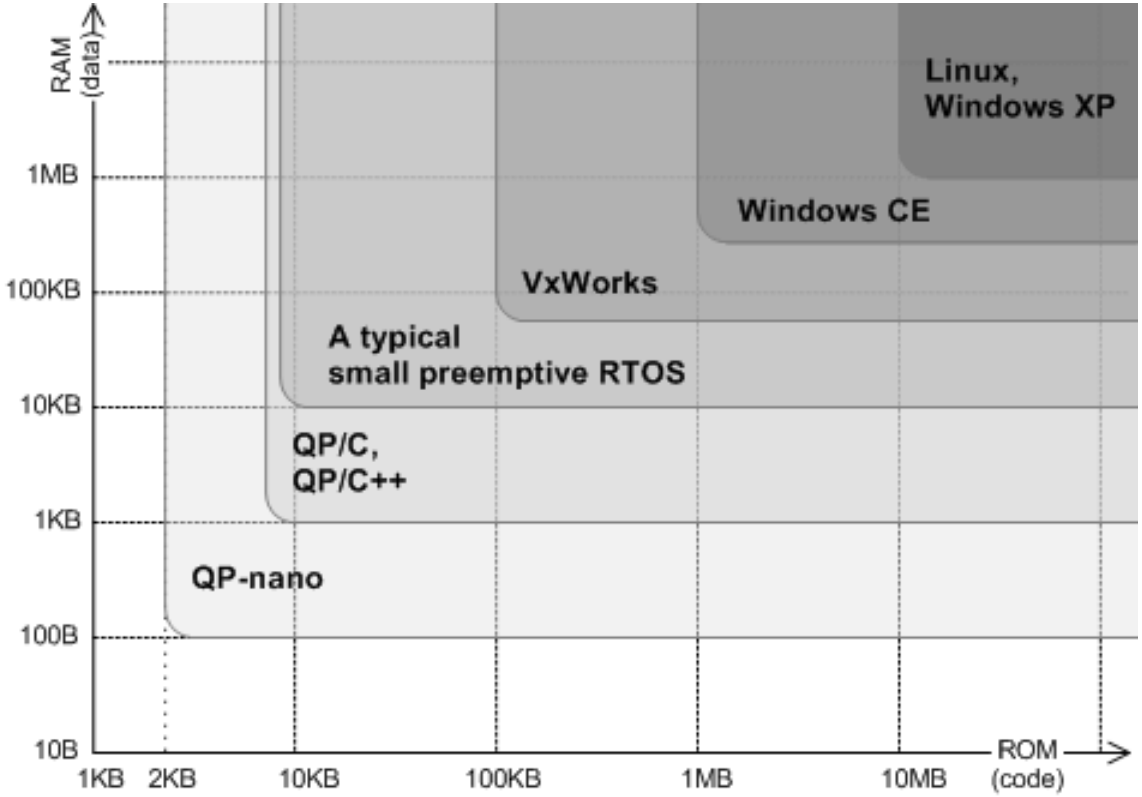
Beispiel: Bare Metal Microcontroller

```
#define MAX  3

typedef void (*Callback)(void *pData);

extern const Callback CB_FUNC[MAX];
extern void* const    CB_DATA[MAX];

void timer_isr()
{
  clearInterrupt();
  for( int i = 0; i < MAX; i++ ){
    CB_FUNC[i]( CB_DATA[i] );
  }
}
```

- Observer Pattern

Beispiel: qp Framework

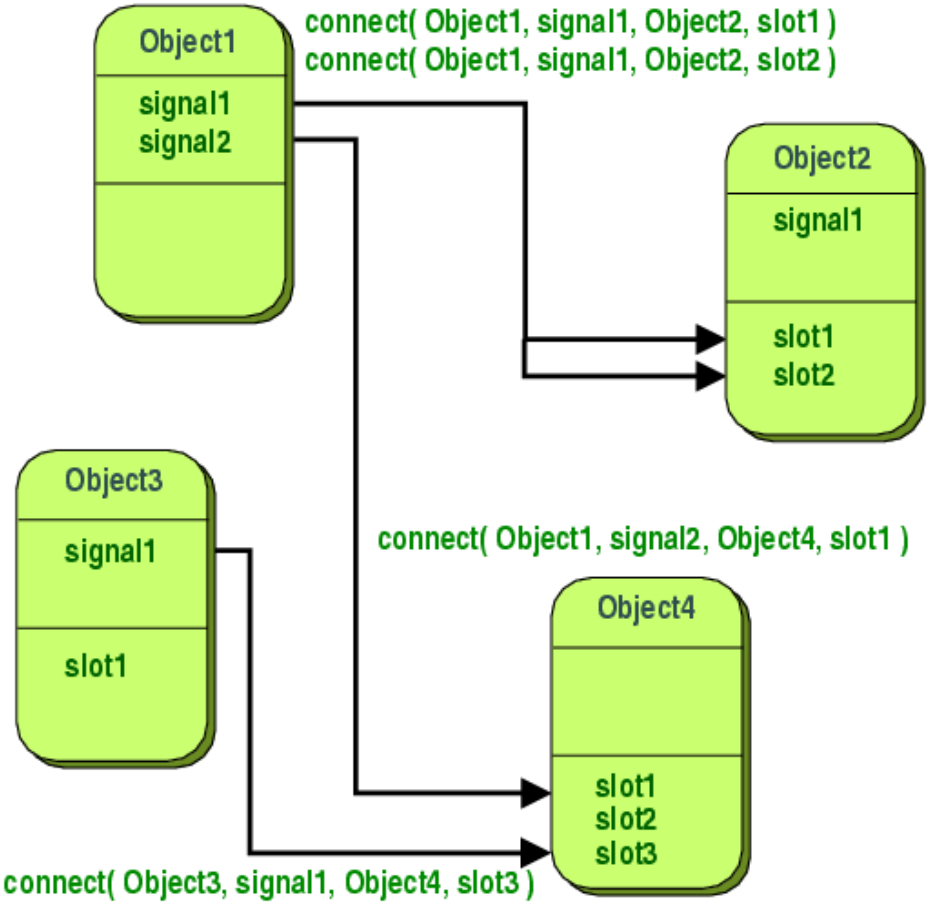
- Mikrocontroller
- State-Machine
- Ereignisorientiert



[4]

Beispiel: Qt

- Signal und Slots
- Metaobject Compiler
- benötigt OS



[5]

Probleme mit existierenden Lösungsansätzen

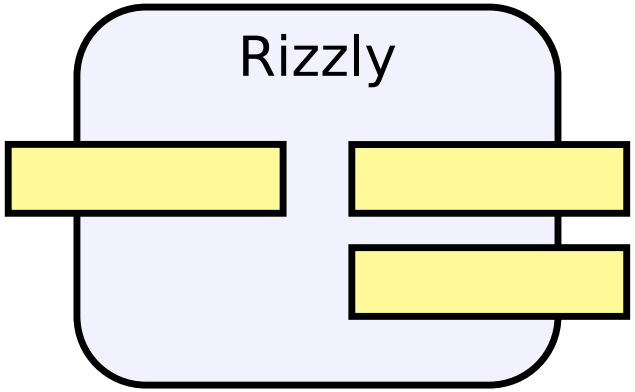
- umständlich wenn kein Framework
- Overhead durch Framework[1]
- Ressourcen-hungrig
- C basiert: Hacks
- Compiler kann schlecht optimieren[2]

→ falsches Programmierparadigma

[1] Workarounds für Programmiersprachen

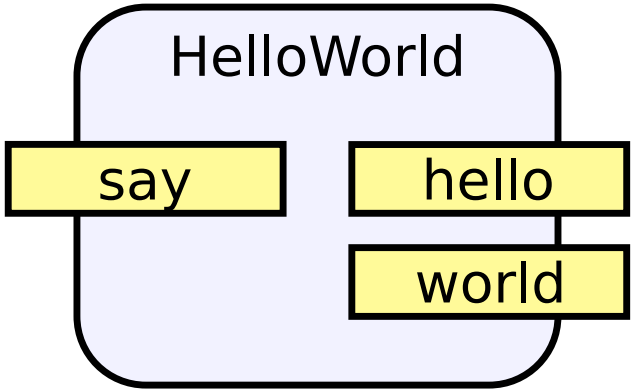
[2] weil der Compiler das Modell nicht kennt, Hacks

Rizzly



- Ereignisorientierte Programmiersprache
- kleinste Mikrocontroller
- hohe Maschinenabstraktion
- hohe Code-Wiederverwendbarkeit
  - Module
  - Minimierung der Seiteneffekte
  - Templates
  - Compile Time Function Evaluation

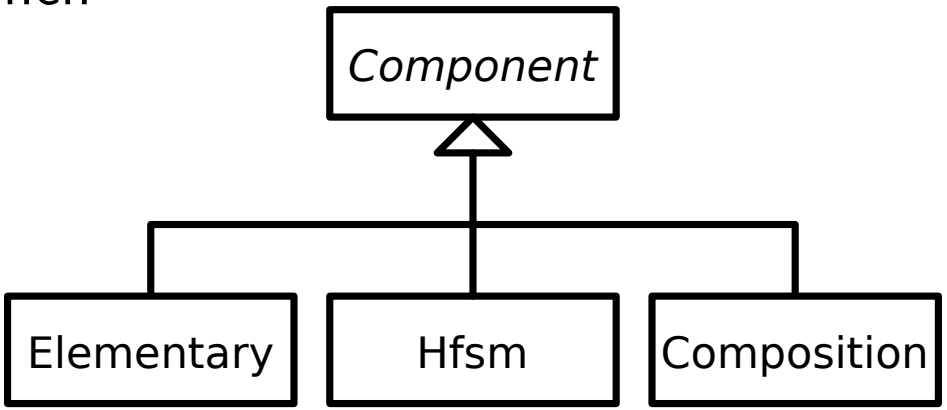
Komponente



- Slots bilden Input Interface
- Signale bilden Output Interface

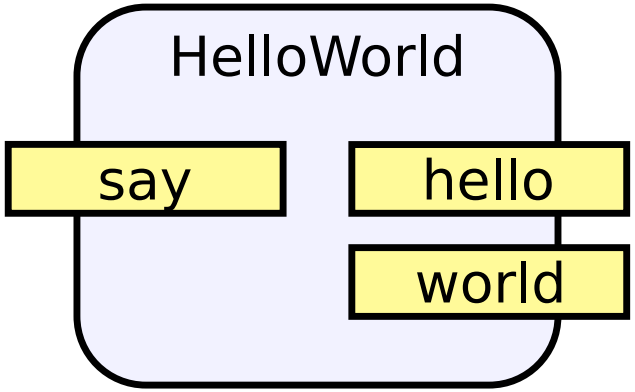
→ einzige Kommunikationsmöglichkeit

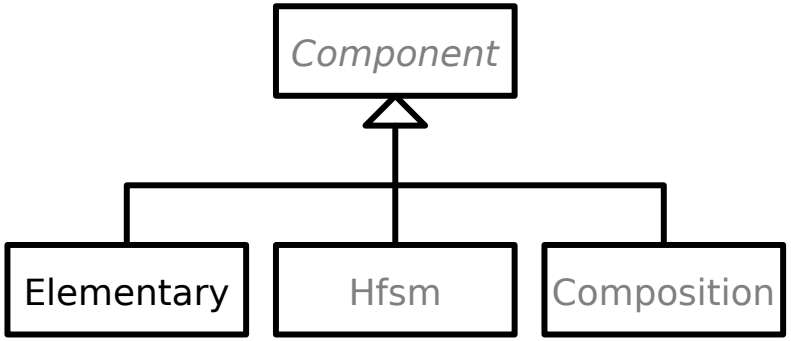
- besitzt einen Status
- verschiedene Implemetationen



Elementary

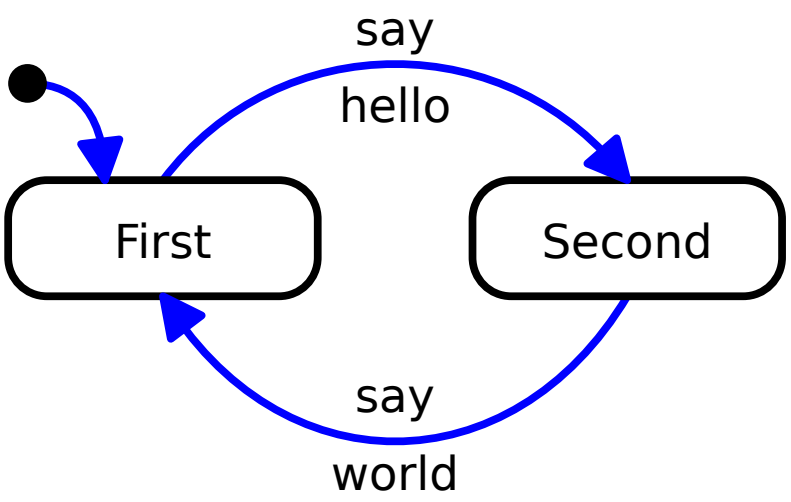
```
HelloWorld = Elementary
  first : Boolean = True;

  hello : signal();
  world : signal();

  say : slot()
    if first then
      hello();
    else
      world();
    end
    first := not first;
  end
end
```

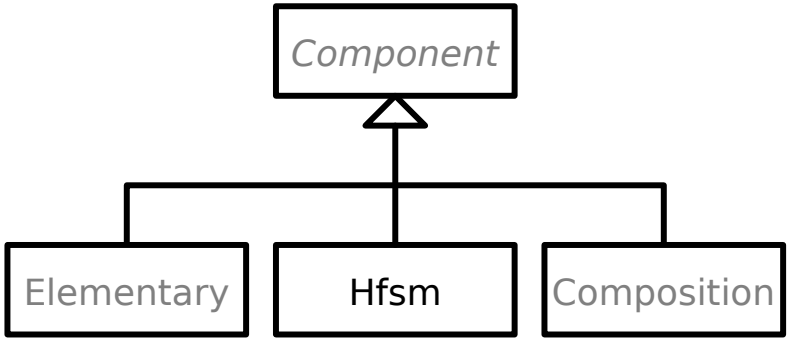




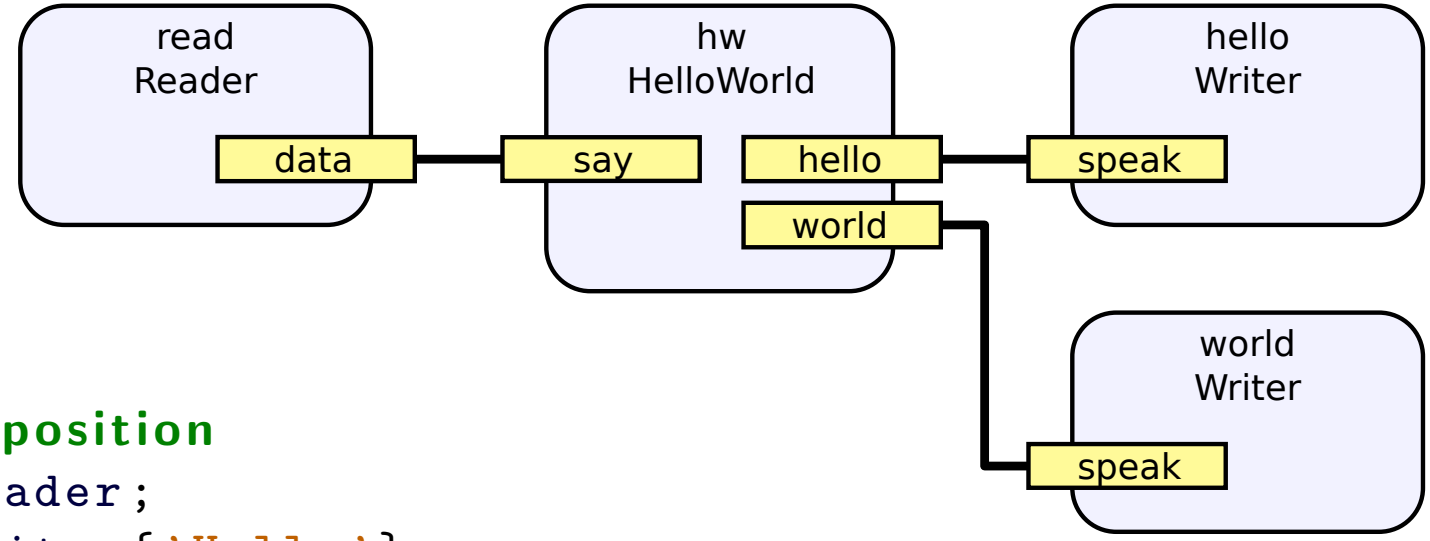
Hirarchical Finite State Machine

```
HelloWorld = Hfsm
  hello : signal();
  world : signal();
  say   : slot();

  state(First)
    First  : state;
    Second : state;

    First to Second by say() do
      hello();
    end
    Second to First by say() do
      world();
    end
  end
end
```

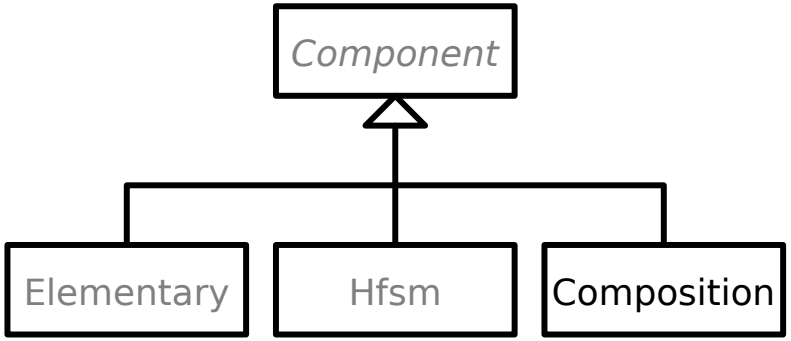




Composition

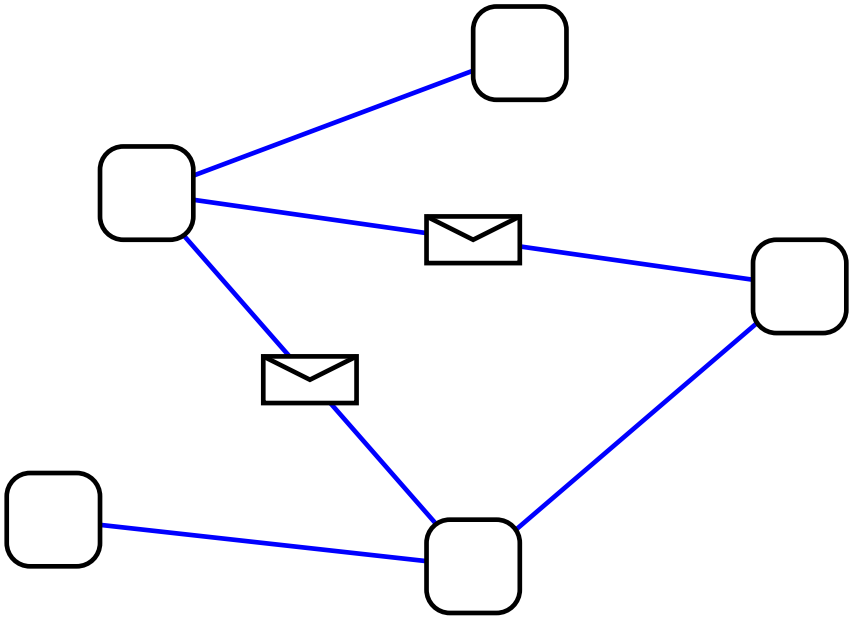



```
World = Composition
  read  : Reader;
  hello : Writer{'Hello'};
  world : Writer{'World'};
  hw    : HelloWorld;

  read.data -> hw.say;
  hw.hello -> hello.speak;
  hw.world -> world.speak;
end
```



Atom-Modell

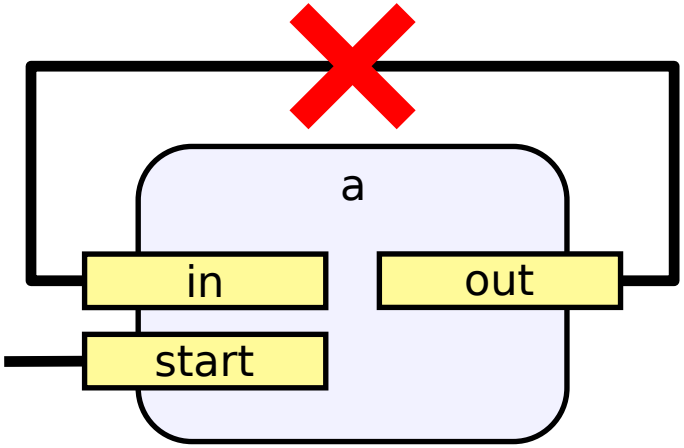
- Kommunikation nur über Ereignisse
- Übertragung braucht Zeit
- Ausführung braucht keine Zeit
- Komponente kann nur durch Ereignis aktiviert werden
- Programm ist verteiltes System



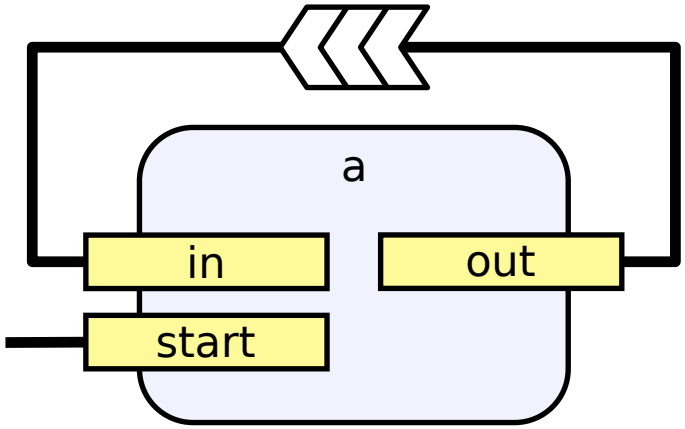
Umsetzung der Ereignisse

- Ereignisse sind Funktionsaufrufe
  - Loops verletzen Modell
- asynchrone Verbindungen
  - Ereignis wird in Queue gespeichert[3]
  - später Queue abarbeiten

```
start -> a.start;
a.out -> a.in;
```




```
start -> a.start;
a.out >> a.in;
```



[3] Kontrolle kehrt sofort zurück

Bereichstyp

- R{von, bis}[4]
- mathematisches Modell

```
a : R{-2, 5};
b : R{ 2, 3};
c : const[5] = 42;     // R{42,42}

d : R{ 0, 8} = a + b;
e : R{-6,15} = a * b;

f : R{0,3} = d mod 4;

if d <= 2 then
  g : R{0,2} = R{0,2}( d )[6];
```

[4] ganzzahliger Bereichstyp
[5] Typ wird automatisch ermittelt[11]
[6] Explizite Typumwandlung, wird zur Laufzeit überprüft

Für kleinste Mikrocontroller

- strikte Sprache
- Compiler versteht Modell
- Compiler sieht das gesamte Programm

→ bessere Optimierungen[7]

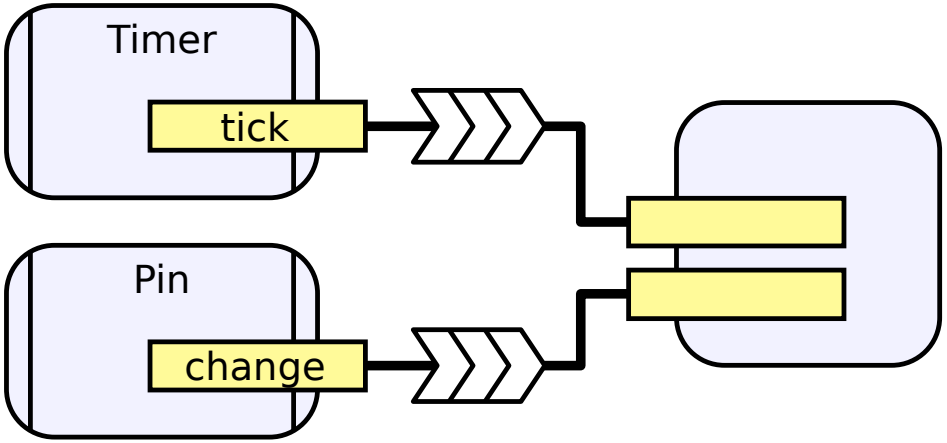
- statischer Code[8]
- Compiler kann Code ausführen (Siehe Compile-time function evaluation 6)

[7] z.B. eliminieren unerreichbarer States in Hfsm

[8] keine Funktions-Pointer, keine Dynamik

TODO: Hardware ansprechen / Interrupts

```
Timer = Elementary
  tick = signal();

  reload       : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```



Rizzly18[width=8cm]
└─Rizzly
  └─TODO
    └─TODO: Hardware ansprechen / Interrupts

TODO: Hardware ansprechen / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

reload ist ein Register der CPU was durch den Register-Modifier angegeben wird. Das Template Argument spezifiziert die Adresse des Registers. Dem Compiler muss noch ein Mapping von dem Rizzly Typ auf die Bit-Repräsentation angegeben werden. In diesem Fall dürfte dies ein 1:1 Mapping sein.

TODO: Hardware ansprechen / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

interrupt ist eine spezielle Prozedur. Das Template Argument kann die Interrupt-Nummer sein oder die Adresse an welcher die Interrupt Funktion liegen muss.

Rizzly18[width=8cm]
└─Rizzly
  └─TODO
    └─TODO: Hardware ansprechen / Interrupts

TODO: Hardware ansprechen / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

Die Komponente ist nicht mehr "rein". Sie kann von sich aus Ereignisse verschicken. Um dies in das Modell zu bringen können solche Komponenten nur über Queues zusammengeführt werden. Die Verwendung solcher Komponenten sollte spärlich und nur auf der obersten Ebene geschehen.

Rizzly18[width=8cm]
└─Rizzly
  └─TODO
    └─TODO: Hardware ansprechen / Interrupts

TODO: Hardware ansprechen / Interrupts

```
Timer = Elementary
  tick = signal();

  reload      : Register{R{0,255}, 2048, 0, 8, MAP11};

  interrupt{INTERRUPT_TIMER}
    // clear interrupt flag
    reload := ...;
    tick();
  end
end
```

Im jetzigen Stand muss Glue Code geschrieben werden, siehe Verwendung 5.

TODO

- foreach Schleife[9][9]
- Tupel[10]
- llvm[11]/gcc als Middle- und Backend
- Eclipse Integration[12]

[9] Konzept dass es allgemeingültig ist

[10] u.a. für mehrere Rückgabewerte von Funktionen, zieht Syntax/Semantik-Änderung von Funktionen nach sich

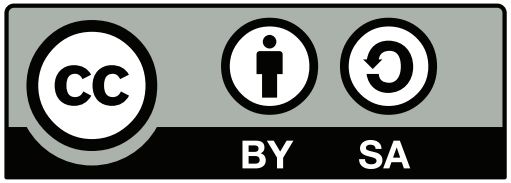
[11] kein avr Backend fertig/verfügbar

[12] Fleissarbeit

Zusammenfassung

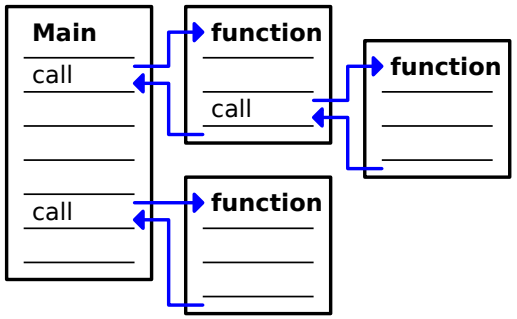
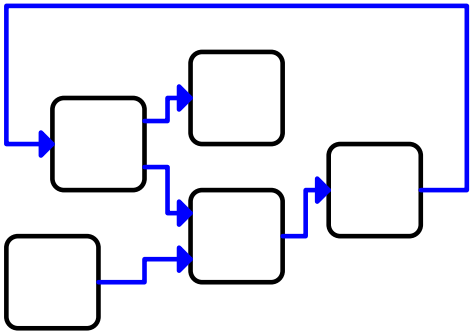
- Es existieren Ereignis-basierte Probleme
- imperative Sprachen sind das falsche Werkzeug

→ Rizzly
  - Ereignisorientiert
  - Mikrocontroller
- Es gibt viel zu tun
- `bitzgi.ch`
- `gitorious.org/rizzly`



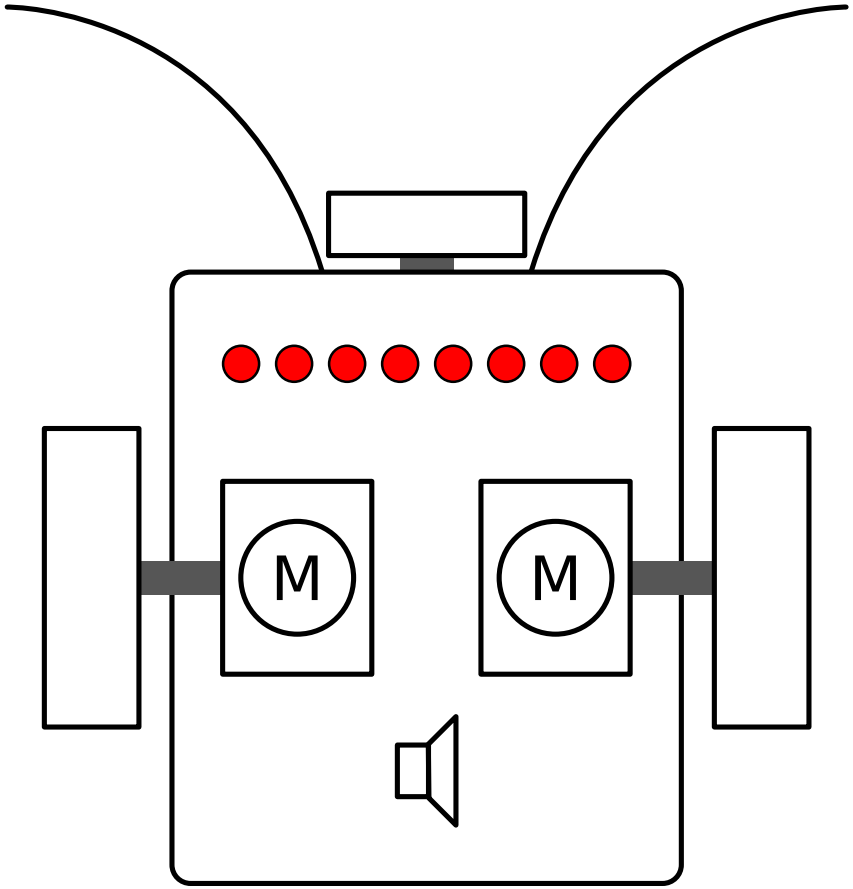
Exklusive fremde Inhalte

Imperativ ↔ Ereignisorientiert

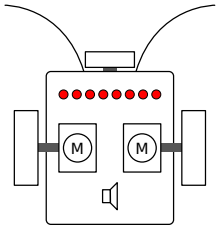
	Imperativ	Ereignisorientiert
Kontrolle	Programm	Umgebung
Programmfluss	linear	nichtlinear
Beschäftigung	rechnen	warten
Laufzeit	kurz	lang
Parallel	nein	ja
Kopplung	stark[13]	schwach
		

[13] Sub-Funktionen, verwendete Objekte/Listener-Interfaces

Parallelität

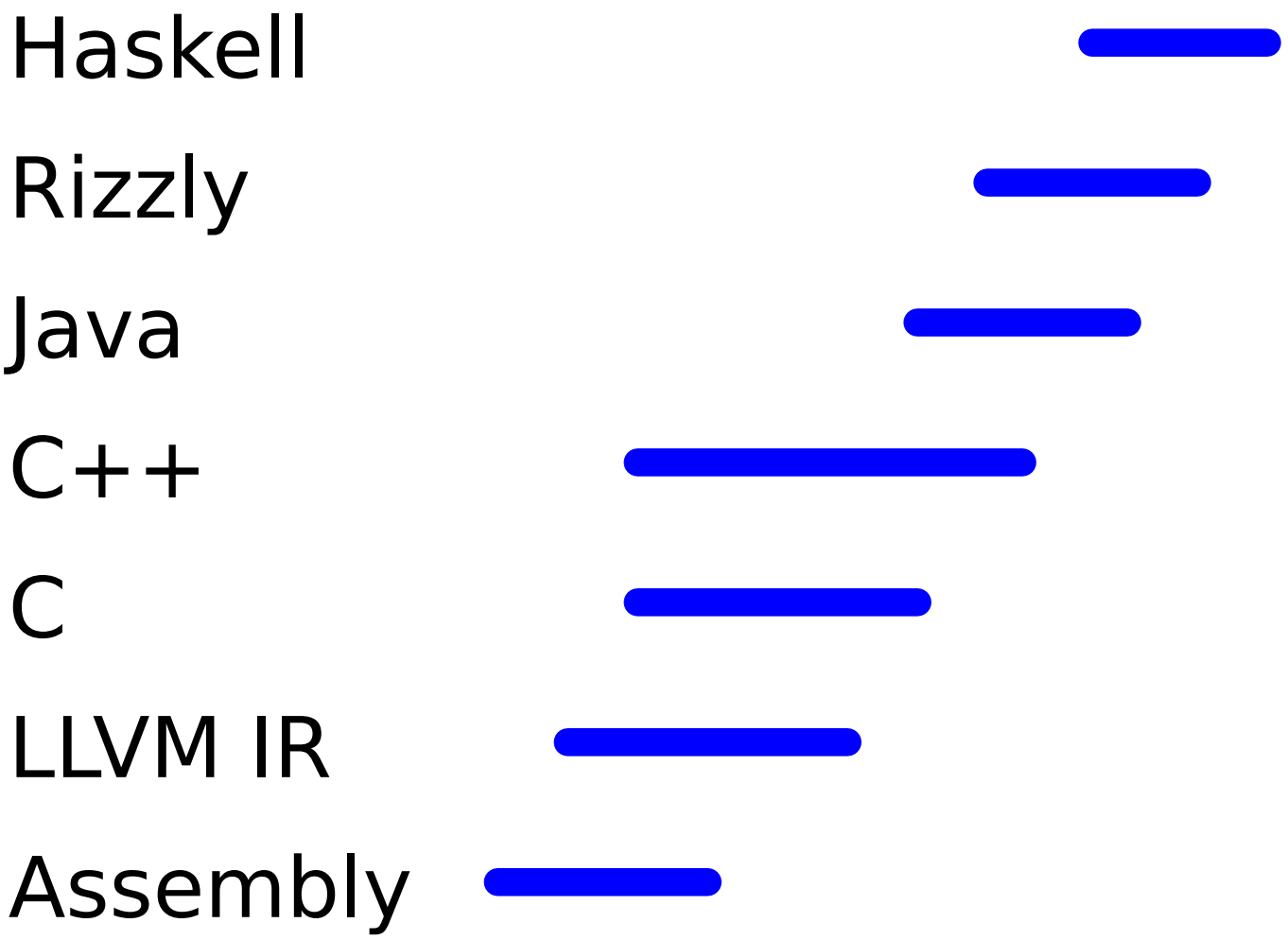


Parallelität

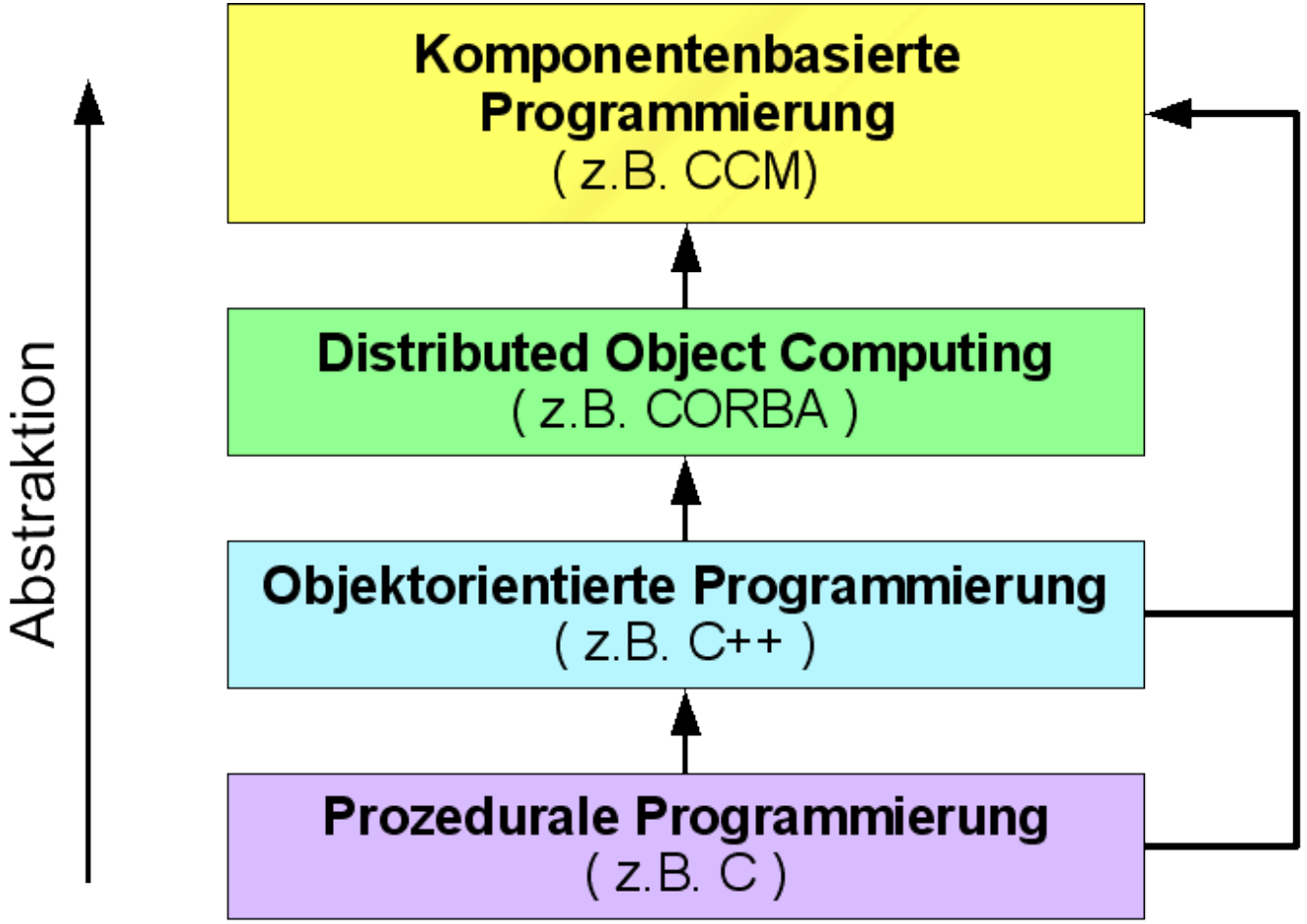


- generell stark parallel
  - auf Sensoren reagieren
  - Motoren ansteuern
  - Lauflicht Programm steuern
  - Lautsprecher ansteuern

Maschinen-Abstraktion verschiedener Sprachen

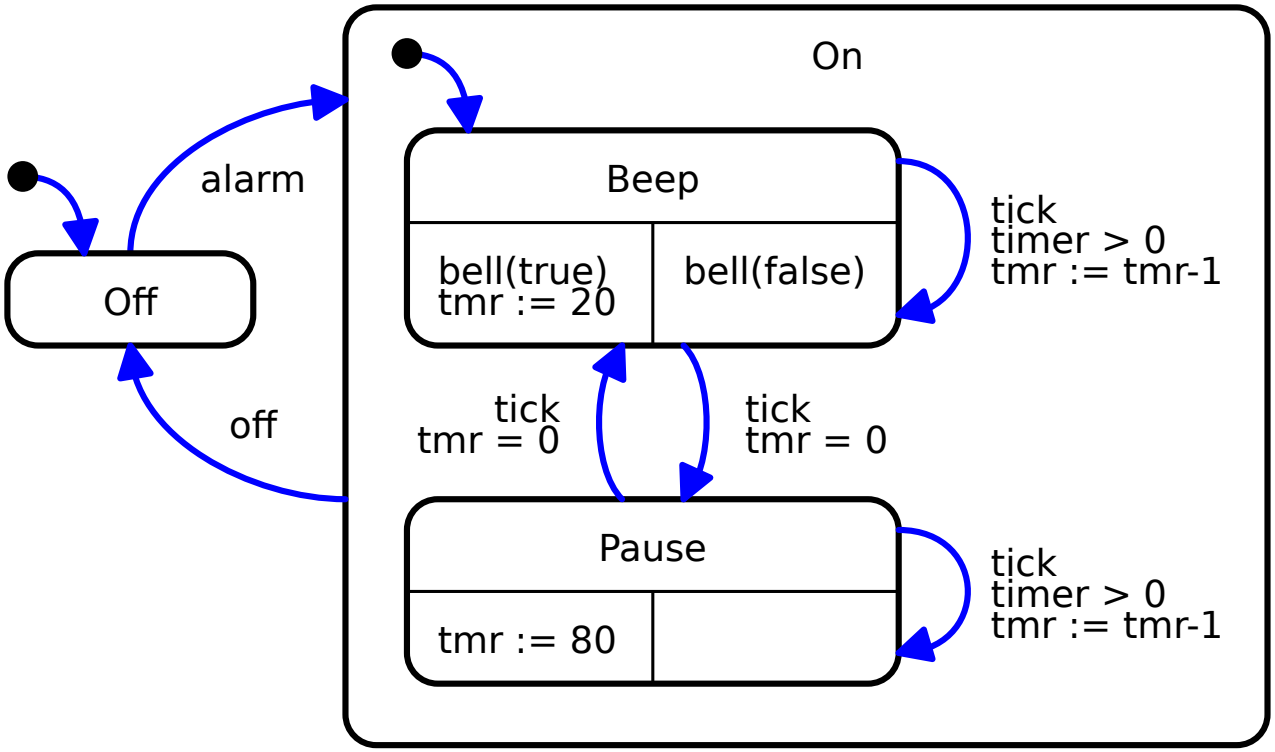


Komponentenbasierte Entwicklung



[7]

Beeper State Machine



Beispiel: GUI

Objektinspektor

Components (filter)

- Shape2: TShape
- Shape3: TShape
- Shape4: TShape
- Shape5: TShape
- Shape6: TShape
- Button1: TButton

Eigenschaften | Ereignisse

Action	
BorderSpacin	(TControlBorderSpacir
Constraints	(TSizeConstraints)
OnChangeBo	
OnClick	Button1Click
OnContextPo	
OnDragDrop	
OnDragOver	
OnEndDrag	
OnEnter	
OnExit	
OnKeyDown	
OnKeyPress	
OnKeyUp	
OnMouseDov	

Form1

Button1 Button2 Button3

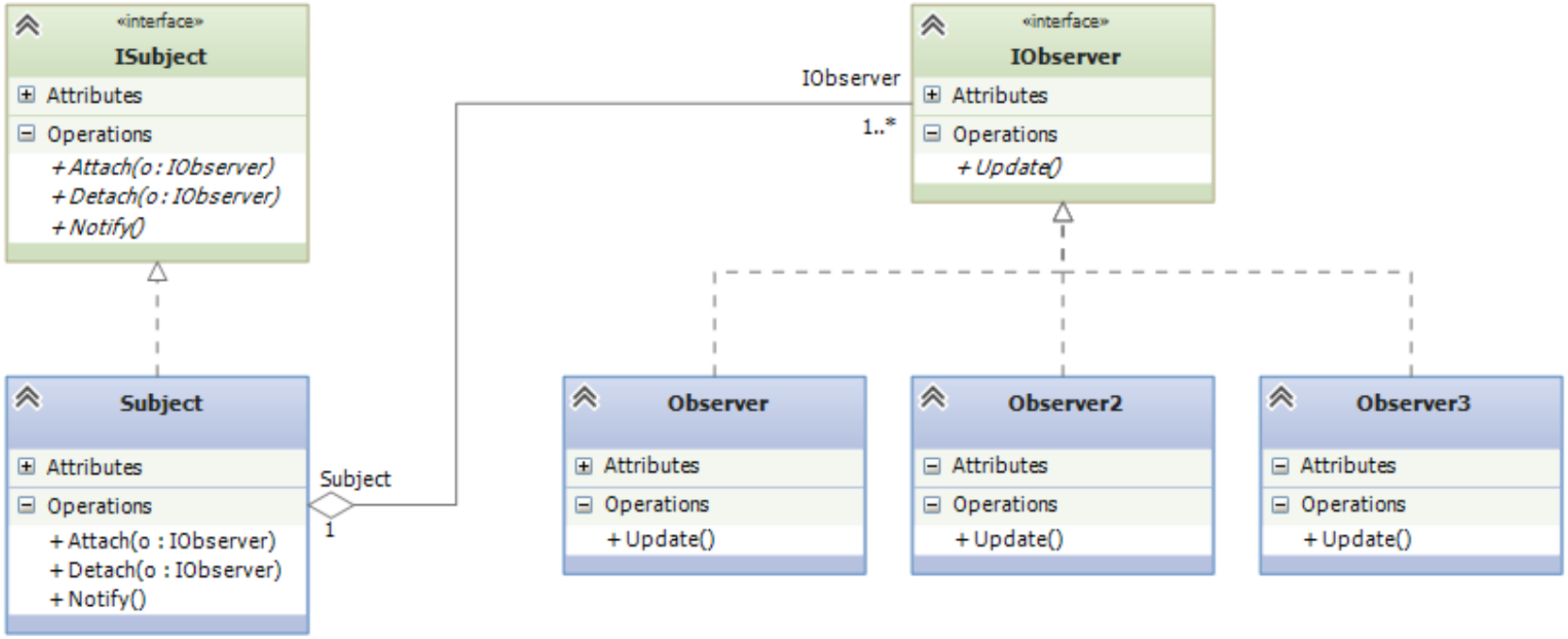
Button4 Button5 Button6

Quelltexteditor

main

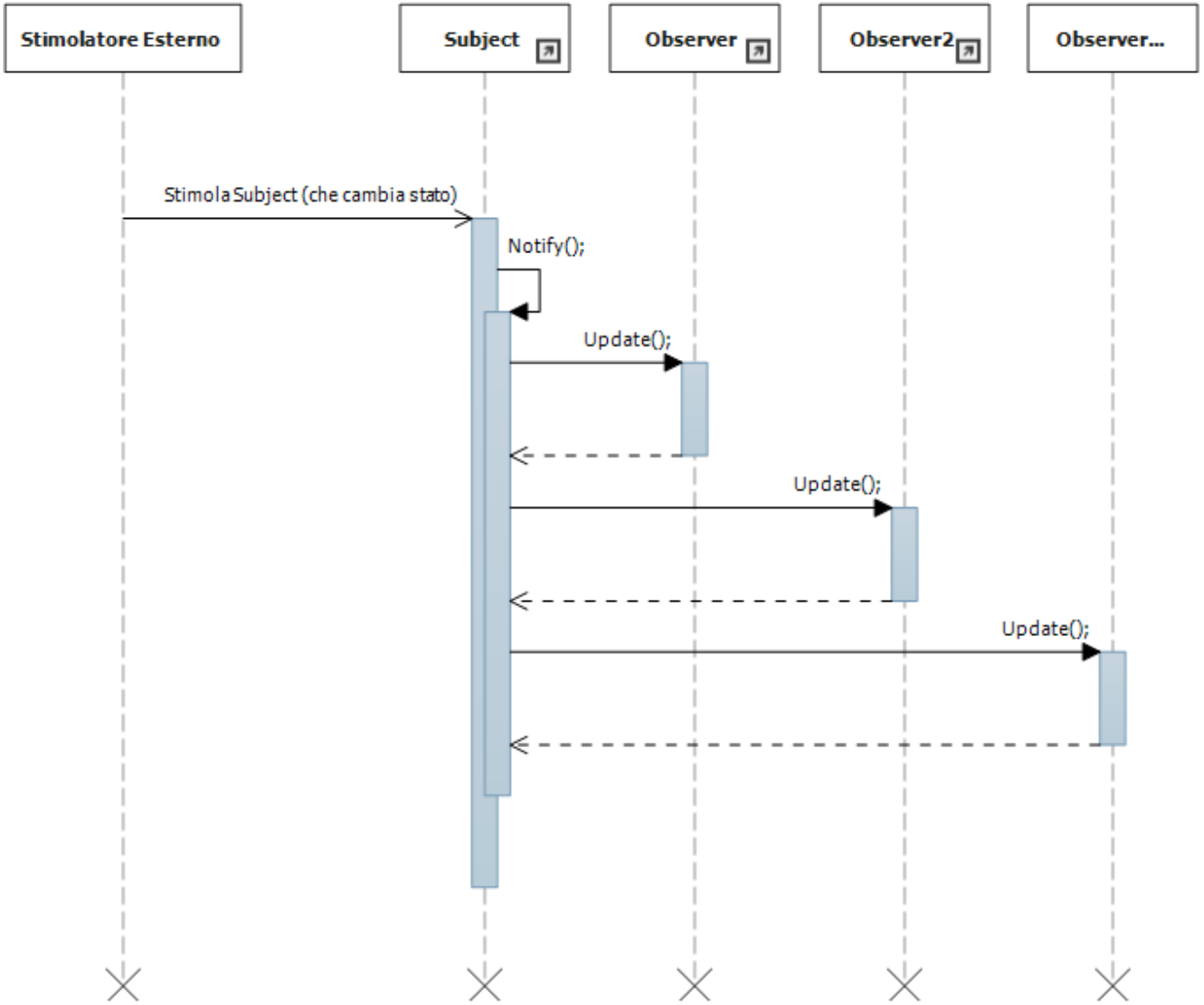
```
    Form1: TForm1;

  implementation

  {$R *.lfm}

  { TForm1 }

  procedure TForm1.Button1Click(Sender: TObject);
  begin
    if nr = 5
      then up := False;
    if nr = 0
      then up := True;

    if up
      then inc( nr )
      else dec( nr );

    setLed(nr);
  end;
```

Beispiel: OOP



[6]

Observer Pattern: Sequenzdiagram



[6]

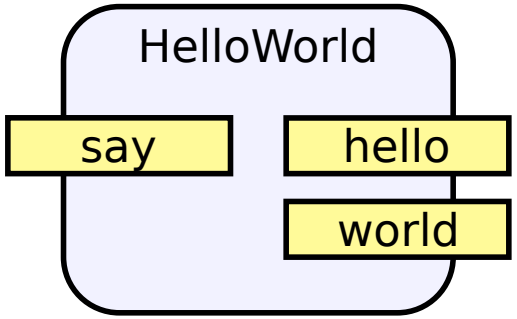
Templates

```
Point{T: Type{Integer}} = Record
  x : T;
  y : T;
end

max{N: R{0,100}} = function(x: R{0,100}):R{0,N}
  if x > N then
    return N;
  else
    return x;
  end
end

a : Point{R{-10,10}};
y := max{80}( 42 );
```

Verwendung

Main (Glue Code)

```
ISR(INT0_vect){
  inst_say();
}
void inst_hello(){
  LED_HELLO = 1;
}
void inst_world(){
  LED_WORLD = 1;
}
void main(){
  ...
  inst__construct();
  ...
```



Generierter Header

```
extern void inst__construct();
extern void inst__destruct();
extern void inst_say();
// void inst_hello();
// void inst_world();
```

Compile-time function evaluation[15]

```
lookuptable : const = calcTable(57);

calcTable = function(n: R{0,100}):Array{10,R{0,100}}
  res : Array{10,R{0,100}};
  i   : R{0,11} = 0;
  while[14] i < 11 do
    idx : R{0,10} = R{0,10}(i);
    res[idx] := idx * n / 10;
    i := idx + 1;
  end
  return res;
end
```

[14] for Schleife ist geplant

[15] Compiler kann Code ausführen

Beispiel: Linux Treiber

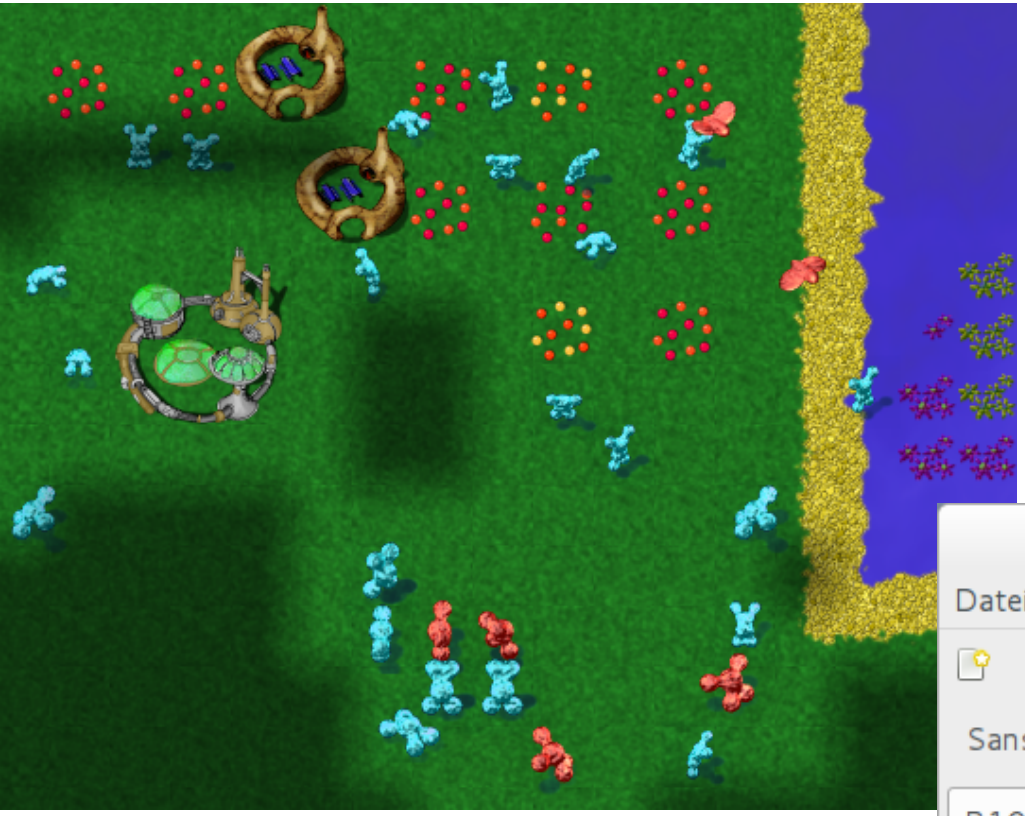
```
struct scull_dev {
  int quantum;
  ...

ssize_t scull_read(...);
ssize_t scull_write(...)
{
  struct scull_dev *dev = filp->private_data;
  printk(KERN_NOTICE "write");
  ...

struct file_operations scull_fops = {
  .read = scull_read,
  .write = scull_write,
  ...
```

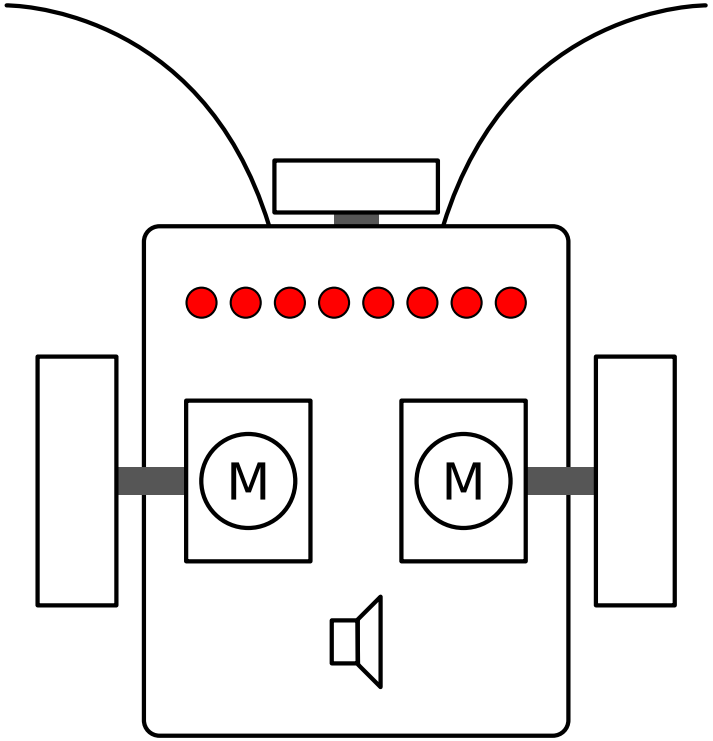
[3]

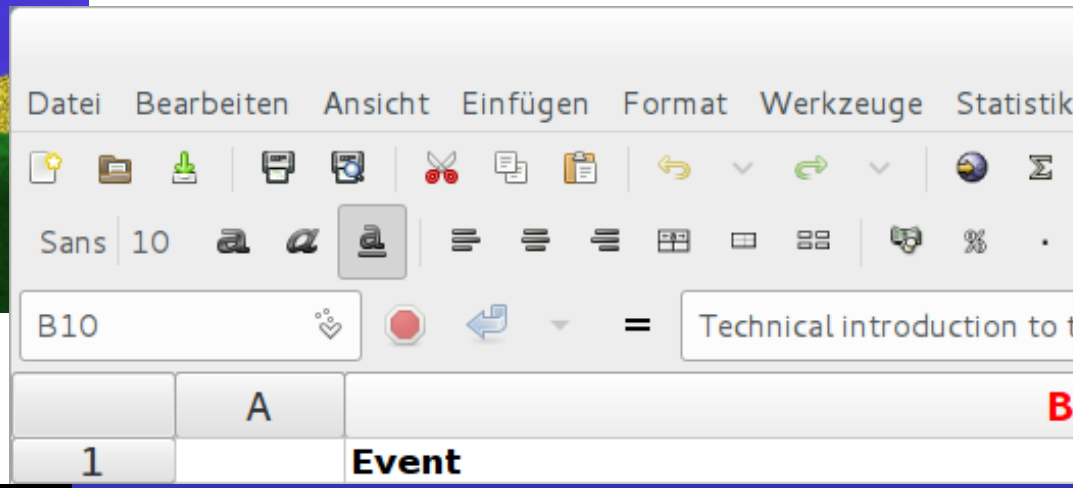
Interrupt ist Ereignisquelle

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Capture Event
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B
9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	EE_RDY	EEPROM Ready
16	0x000F	USI_STR	USI START
17	0x0010	USI_OVF	USI Overflow

Einführung



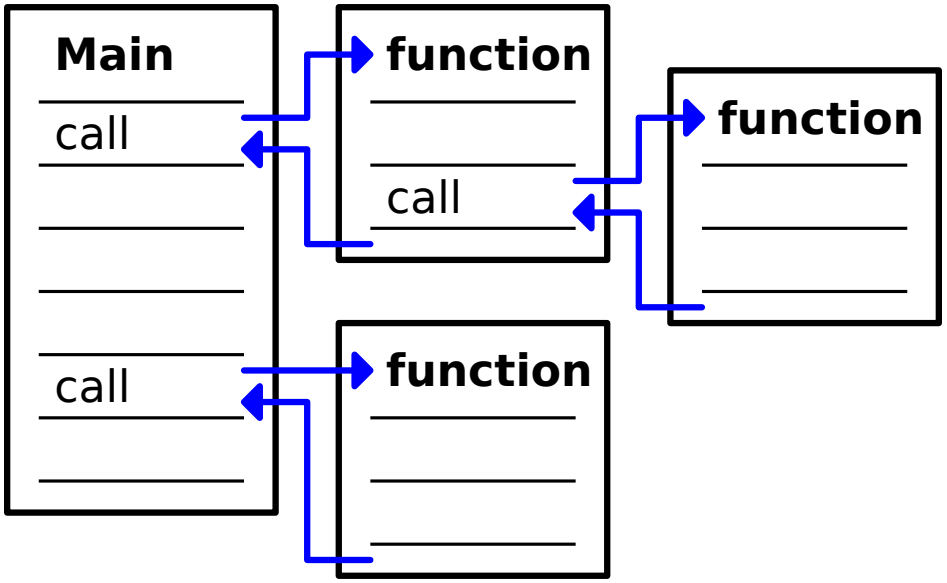

M
M


Datei Bearbeiten Ansicht Einfügen Format Werkzeuge Statistik
Sans 10
B10
=
Technical introduction to t
A
B
1
Event

Wie programmieren wir?

- LaTeX erzeugen
- Daten durchsuchen
- Berechnung
- Compiler

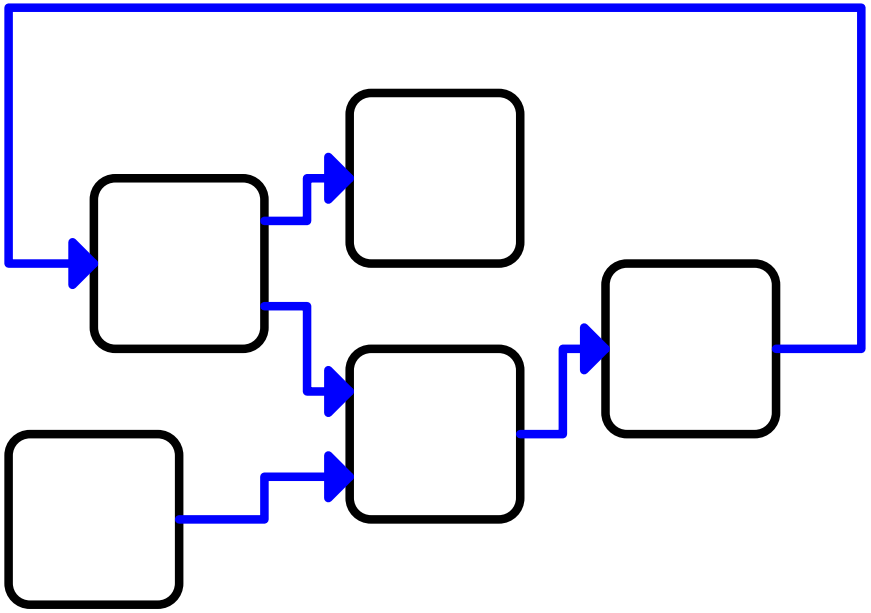
→ Imperativ



Wie programmieren wir?

- GUI Programme
- Linux Treiber
- Spiele
- Roboter

→ Ereignisorientiert



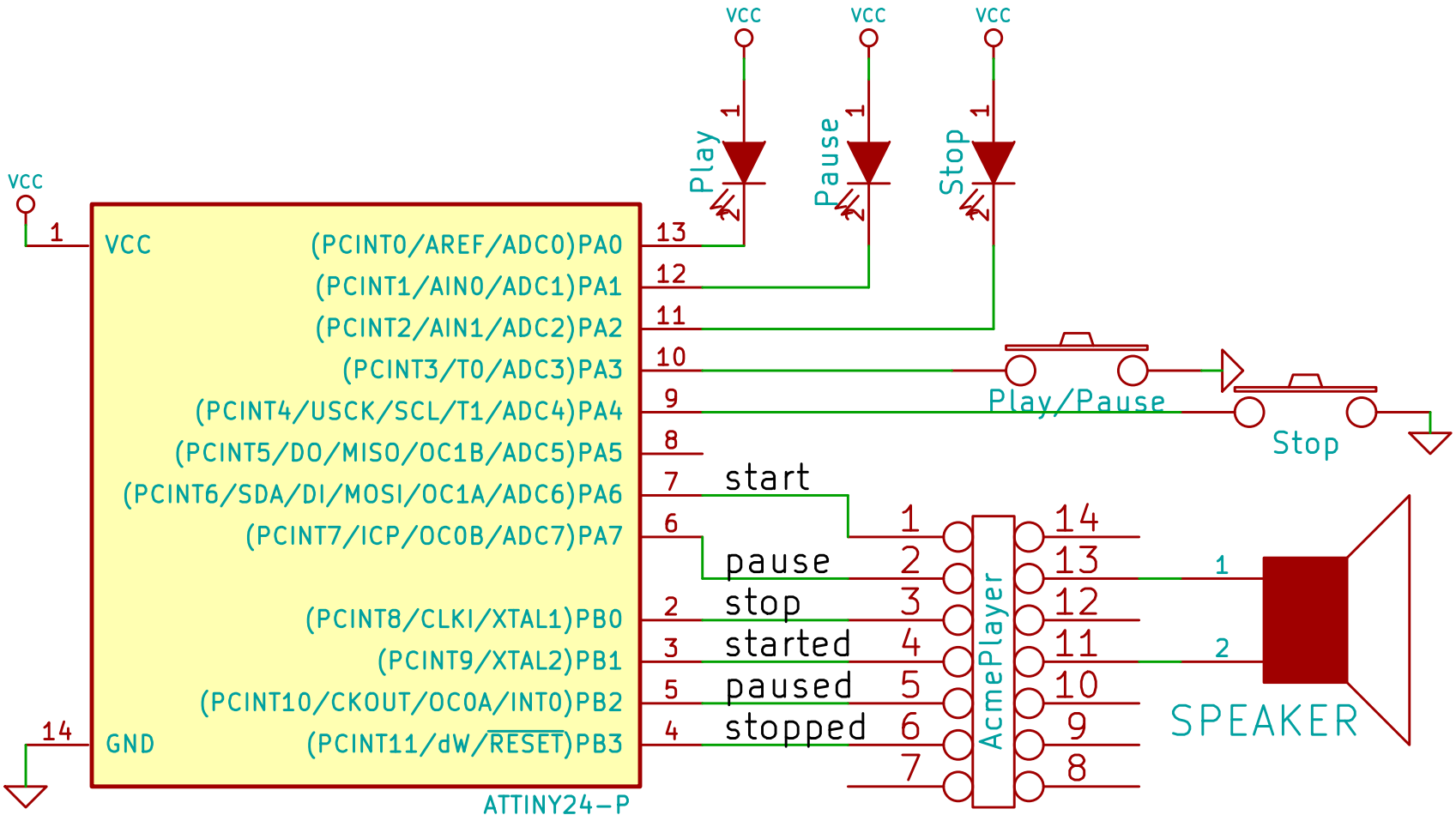
Beispiel Playersteuerung

- User-Input: play/pause, stop
- User Benachrichtigung: playing, paused, stopped
- Player-Input: started, paused, stopped
- Player Steuerung: play, pause, stop

Beispiel Playersteuerung Linux

```
urs@ares:~$ ./player
AcmePlayer UI
Control:  [p] play/pause
          [s] stop
play
pause
play
stop
```

Beispiel Playersteuerung Microcontroller



Prozedurale Programmiersprache

```
extern void doStop(int plFd);
extern void doStart(int plFd);
void payerControl(int plFd, int uiFd)
{
  Status status = STOP;
  int ret;
  while((ret = select(plFd,uiFd)) != -1){
    switch(getc(uiFd)){
      case 'p':
        switch(status){
          case PAUSE:
            doStart(plFd);
            break;
          case PLAY:
            doPause(plFd);
            break;
```

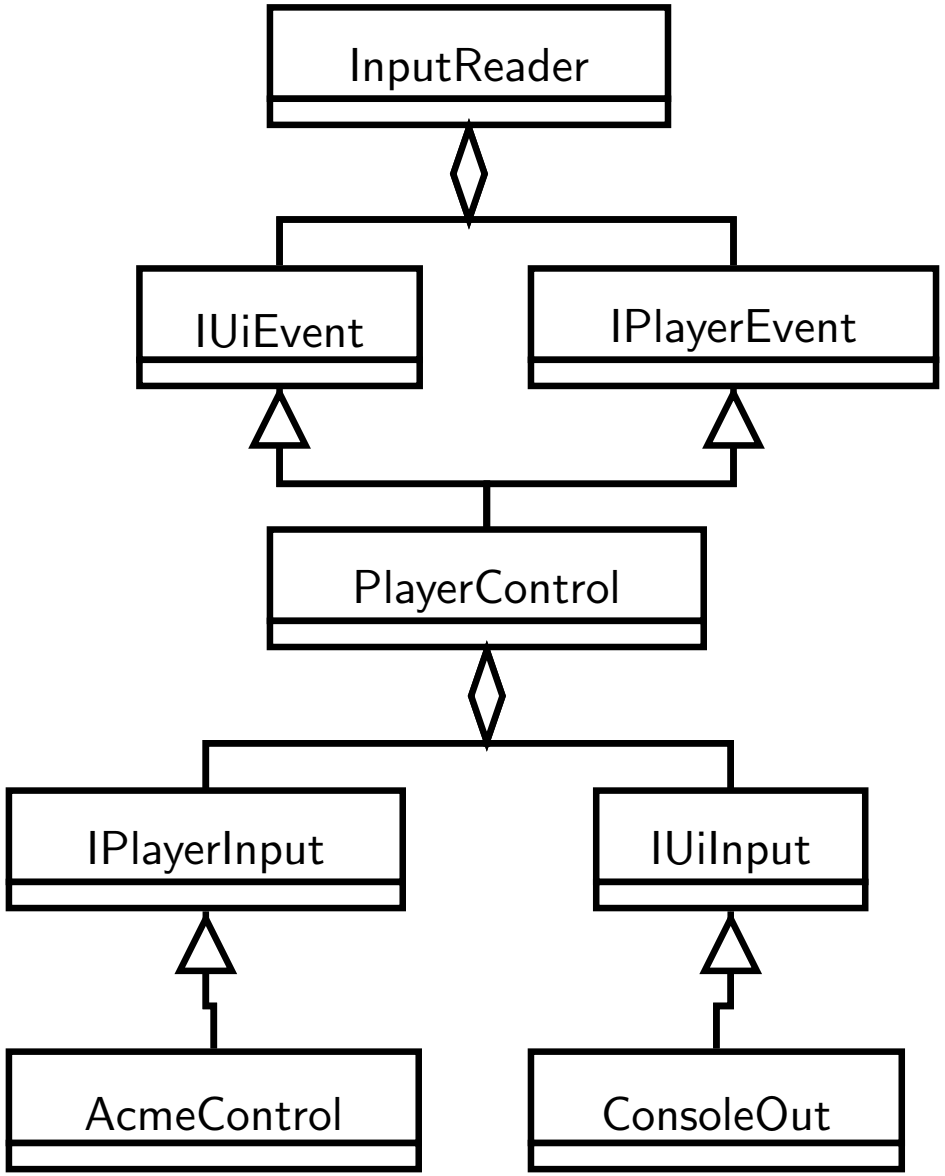
Analyse Prozedural

- warten auf mehrere Ereignisse ist kompliziert
- unübersichtlich
- Callback-Funktionen unschön/Ressourcen verschwendend

Objektorientiert Programmierung

```
#include "interfaces.hpp"

class PlayerControl:
    public IUiEvent,
    public IPlayerEvent
{
  public:
    PlayerControl(IPlayerInput *cntrl,
                  IUiInput *ui);
    ~PlayerControl();

    void playPause();
    void stop();

    void started();
    void stopped();
```

Objektorientiert Programmierung



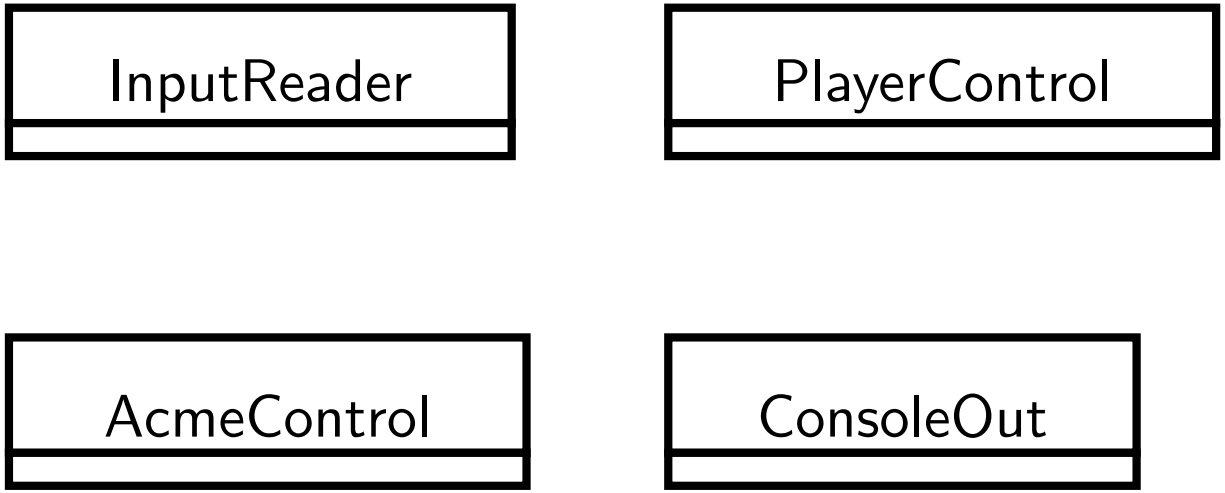
Analyse Objektorientiert

+ Kapselung
+ warten auf mehrere Ereignisse ist ausgelagert
+ übersichtlicher Code
- Klassenabhängigkeit ist kompliziert
- Problem mit Callback-Funktionen nicht gelöst

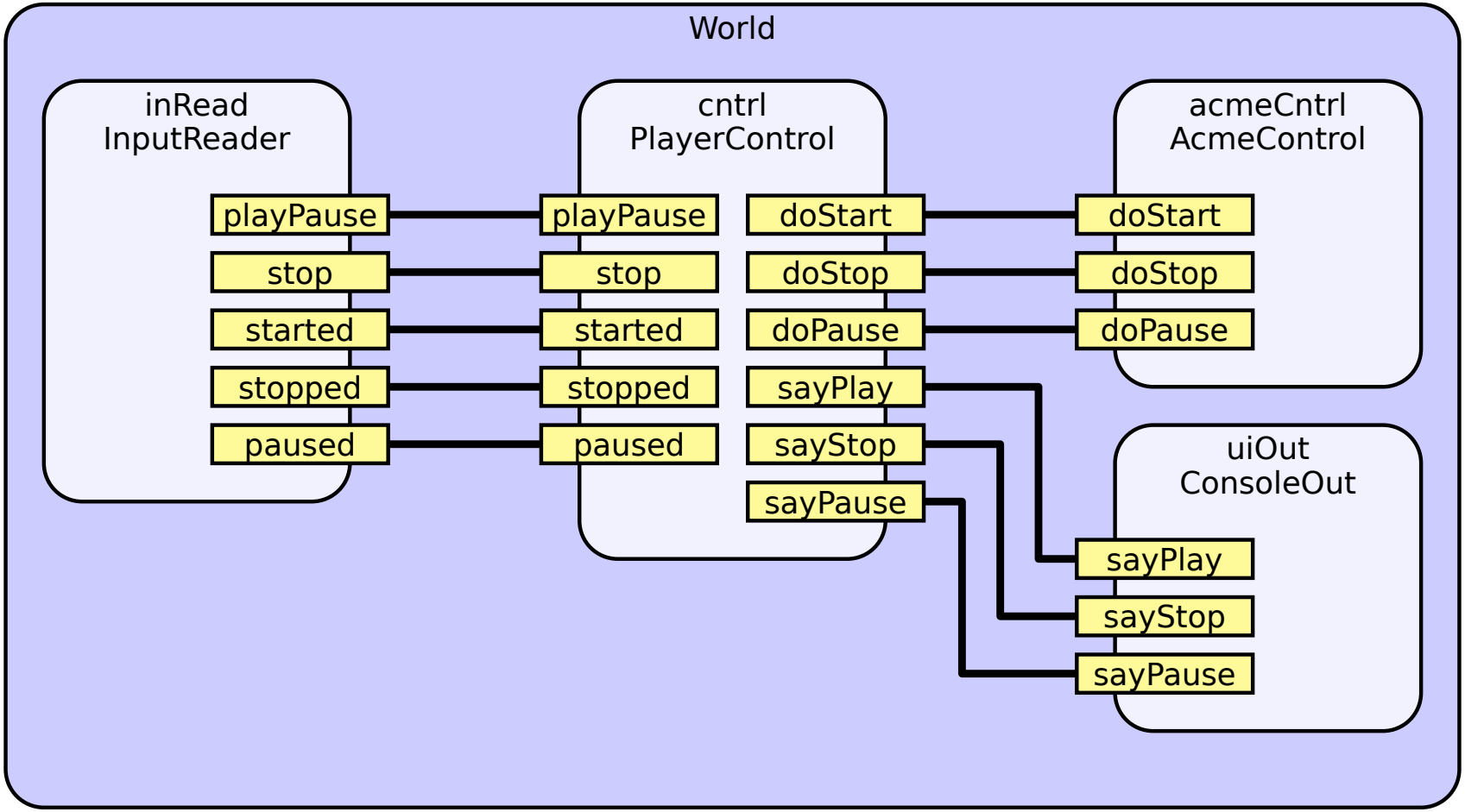
Qt PlayerControl

```
class PlayerControl: public QObject
{
    Q_OBJECT
  signals:
    void doStart();    void sayPlay();
    void doStop();     void sayStop();
    void doPause();    void sayPause();

  public slots:
    void playPause();
    void stop();

    void started();
    void stopped();
    void paused();

  public:
```

Qt Klassendiagram



Signal Diagram



Analyse Qt

+ Supercool
+ übersichtlicher Code
+ Klassen unabhängig voneinander
+ Problem mit Callback-Funktionen gelöst
+ einfaches warten in separaten Threads
- läuft nicht auf Microcontroller
- lässt alle C++ Schweinereien zu

Literature I

[1] Adafruit trinket.
URL: http://www.tandyonline.co.uk/adafruit-trinket-mini-microcontroller-5v.html.

[2] Arduinouno r3 front.
URL: http://arduino.cc/en/uploads/Main/ArduinoUno_R3_Front.jpg.

[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.
Linux Device Drivers.

[4] Quantum Leaps.
Qp rom-ram footprint.
URL: http://www.state-machine.com/qp/index.php.

Literature II

[5] Qt Project.
Signals & slots.
URL: http://qt-project.org/doc/qt-5/signalsandslots.html.

[6] Marco Siniscalco.
Asp.net e pattern observer in pratica – quarta parte.
URL: http://marcosiniscalco.wordpress.com/.

[7] Wikipedia.
Komponentenbasierte entwicklung — wikipedia, die freie enzyklopädie, 2012.
[Online; Stand 9. November 2014].
URL: http://de.wikipedia.org/w/index.php?title=Komponentenbasierte_Entwicklung&oldid=108459381.

Literature III

[8] Wikipedia.
Inversion of control — wikipedia, die freie enzyklopädie, 2013.
[Online; Stand 11. November 2014].
URL: http://de.wikipedia.org/w/index.php?title=Inversion_of_Control&oldid=125646165.

[9] Wikipedia.
Foreach loop — wikipedia, the free encyclopedia, 2014.
[Online; accessed 8-November-2014].
URL: http://en.wikipedia.org/w/index.php?title=Foreach_loop&oldid=626987004.

Literature IV

[10] Wikipedia.
Hollywood principle — wikipedia, the free encyclopedia, 2014.
[Online; accessed 11-November-2014].
URL: http://en.wikipedia.org/w/index.php?title=Hollywood_principle&oldid=622118443.

[11] Wikipedia.
Type inference — wikipedia, the free encyclopedia, 2014.
[Online; accessed 8-November-2014].
URL: http://en.wikipedia.org/w/index.php?title=Type_inference&oldid=629356576.