Kurs OMSI
im WiSe 2013/14

Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer|ahrens|eveslage@informatik.hu-berlin.de

2. Prinzip der Next-Event-Simulation

1. Charakterisierung der Next-Event-Simulation
   - Rückblick
   - Zusammenhang von ereignisbasierter – prozessbasierter Modellbeschreibung

2. Umsetzung des Prinzips in ODEMx
   - Aufbau von ODEMx (erster Blick)
   - Trivialbeispiel

ODEMx = Object-oriented Discrete Event Modelling (extended)

Grundidee einer hierarchischen Prozessverwaltung

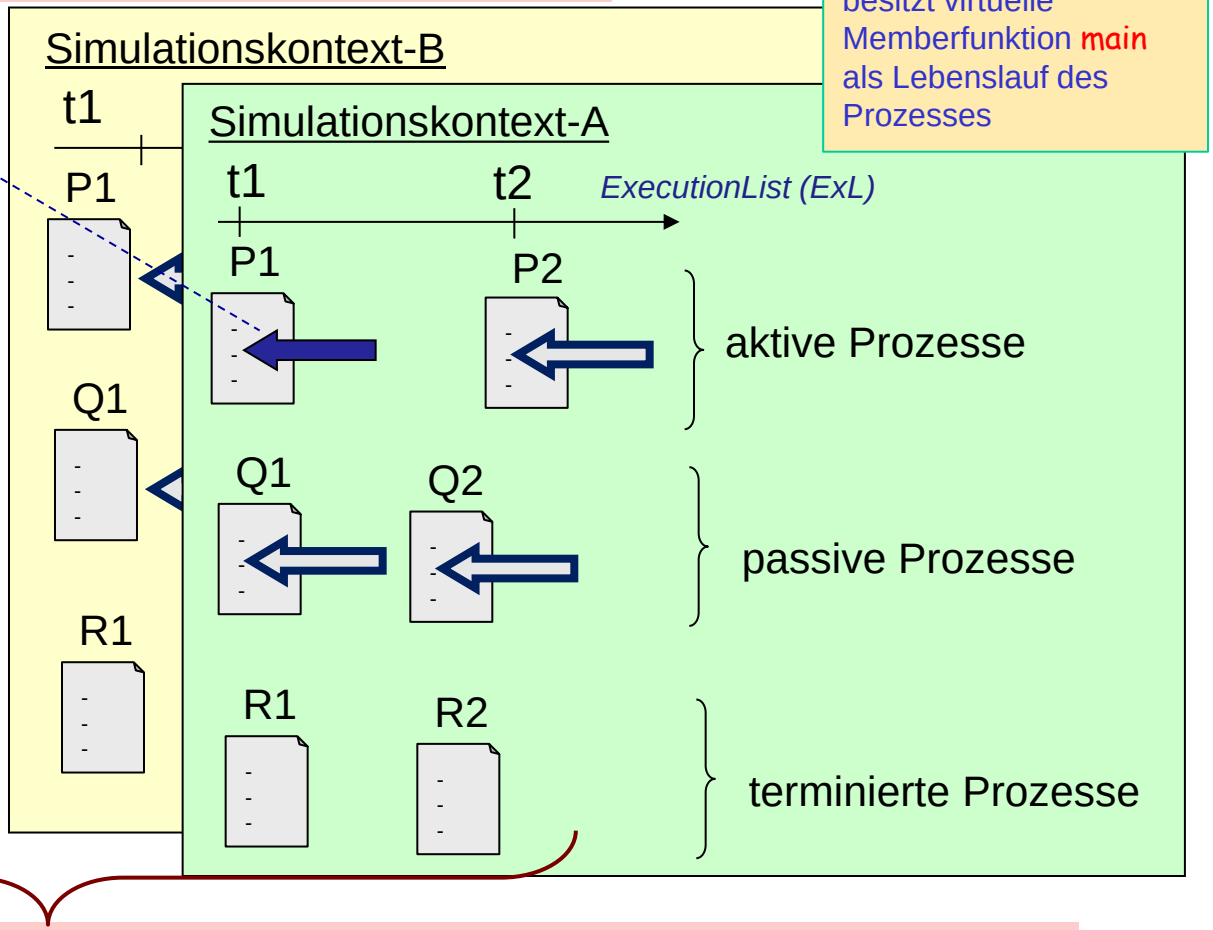
*Zu einem Zeitpunkt kann immer **nur ein Kontext** und in diesem **nur ein Prozess** aktiv sein (current)*

Klasse Process
besitzt virtuelle Memberfunktion main als Lebenslauf des Prozesses

Current- Prozess
- erster Eintrag
- kleinste Zeit
- höchste Priorität

C++ Hauptprogramm

```
int main ( ...) {
...
}
```

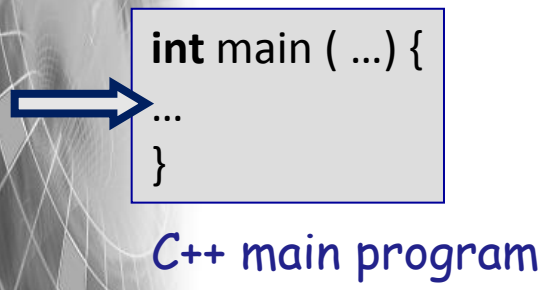


*Hauptprogramm (main-Fkt) und Prozesse (lokale main-Fkt) aller Simulationskontexte bilden ein **hierarchisches Koroutinensystem** auf einer Ein-Prozessor-Maschine*

Standardfall: nur ein Simulationskontext

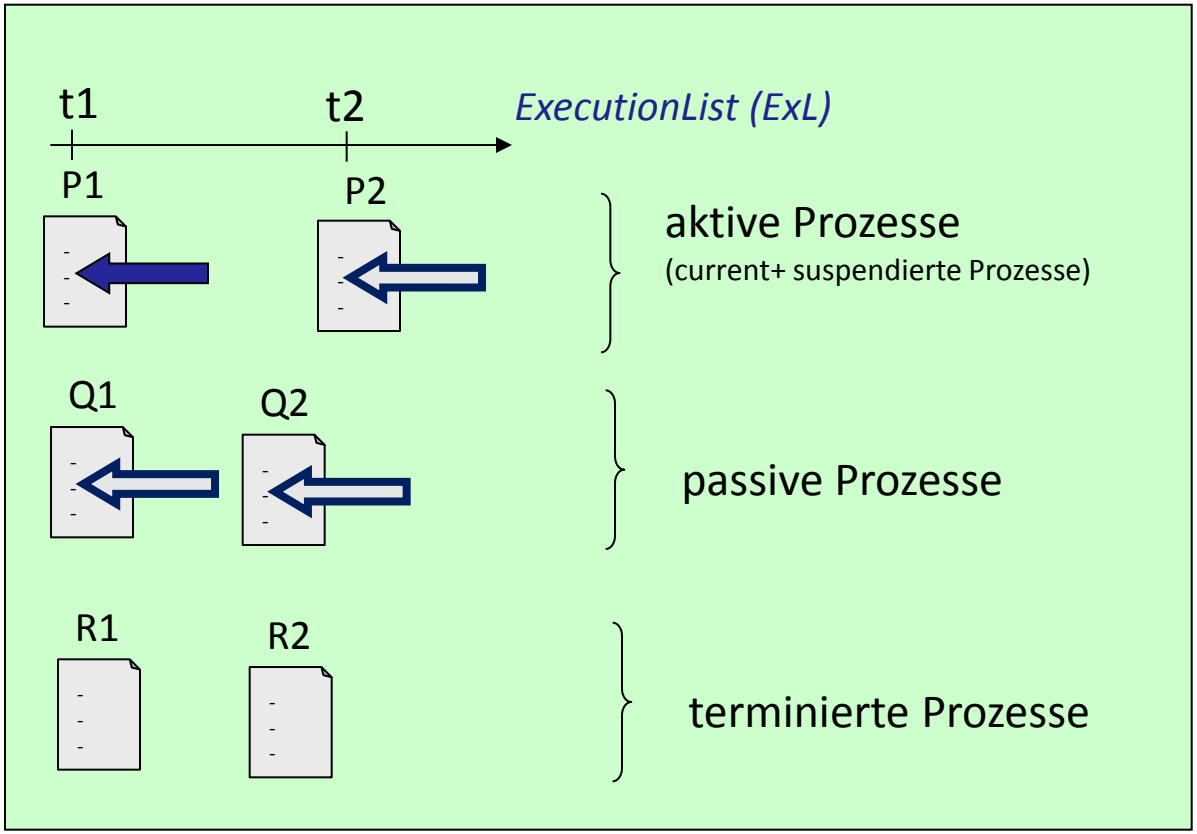
Zu einem Zeitpunkt ist entweder
- *das Hauptprogramm oder*
- *der Current-Prozess des Kontextes*

aktiv

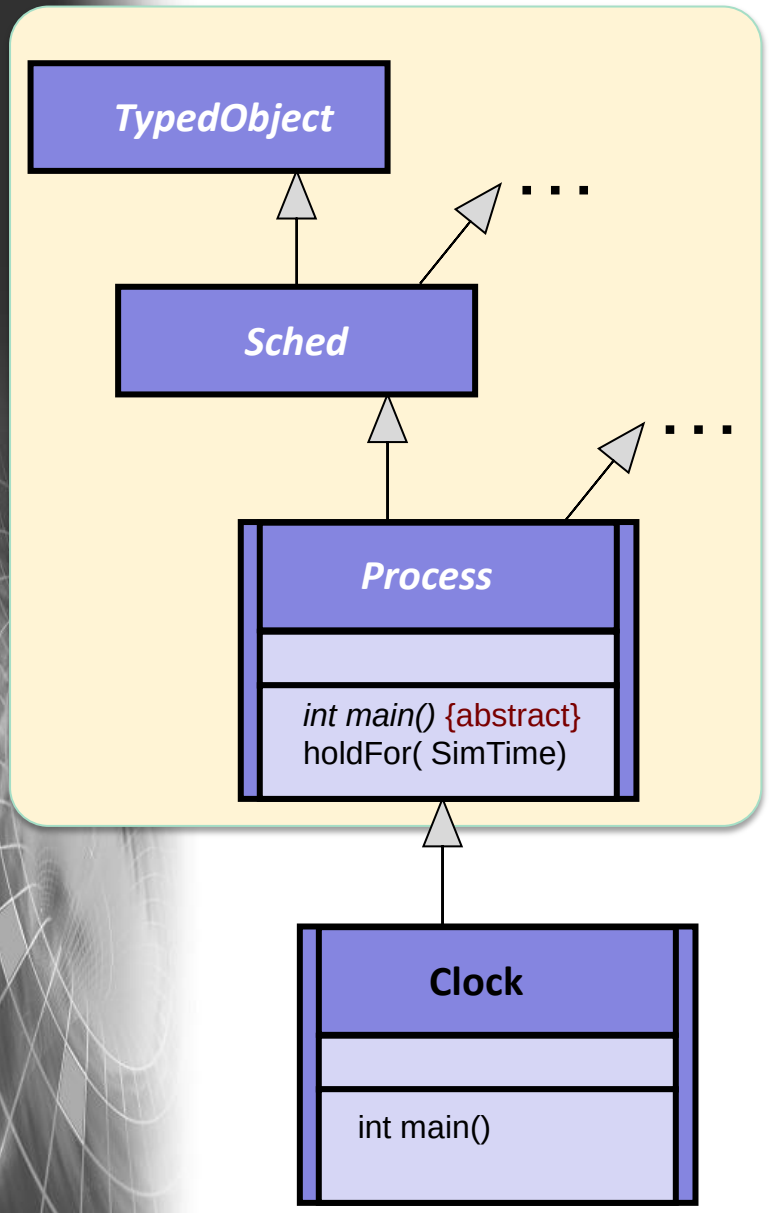


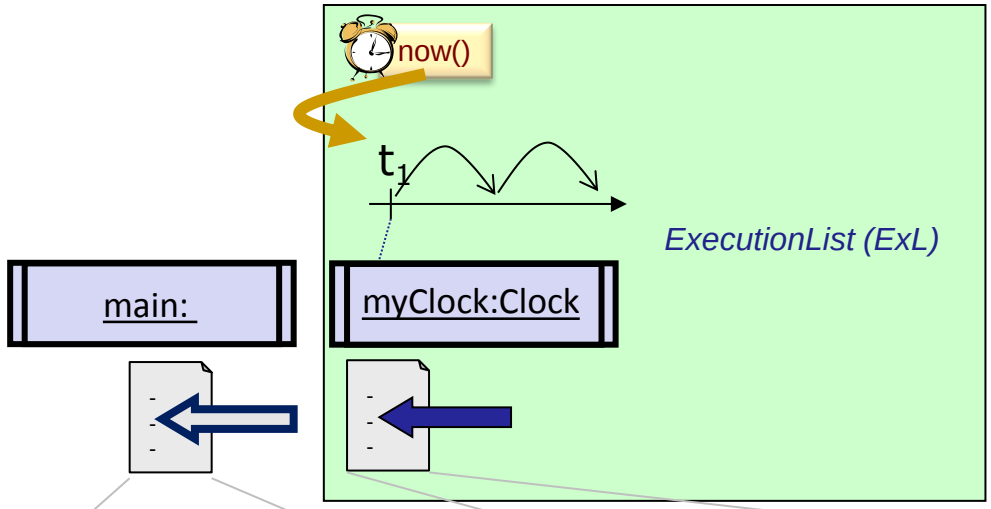
C++ main program

simulation context (DefaultSimulation-Objekt)



Einfaches Beispiel



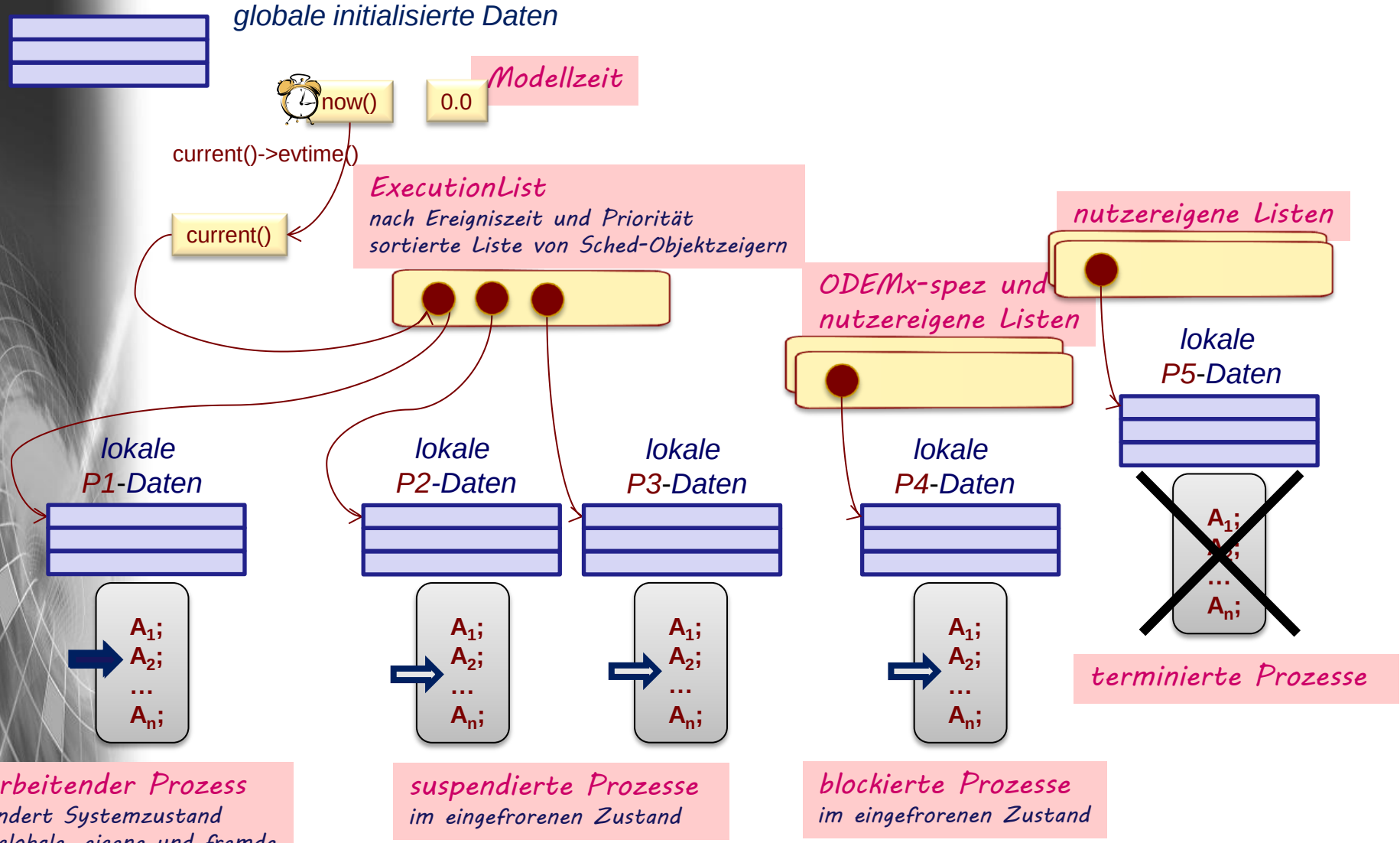


- Einrichten von myClock
- Laden der ExL
- AUSGABE
- Aktivieren des Kontextes
- AUSGABE

forever
- Zeitverbrauch
- AUSGABE
  Punkt-Zeichen

Varianten
a) step()
b) runUntil(simTime t)
c) run()

Schema der Zustandsänderung von ODEMx



Einfaches Beispiel: Quelltext

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
              Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                  getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or passed";
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

Einfaches Beispiel: Quelltext

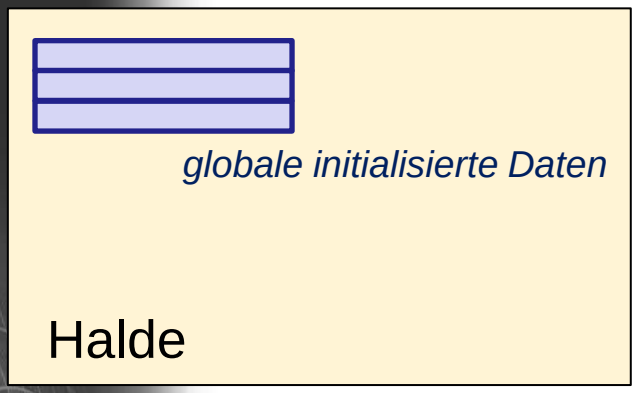
```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

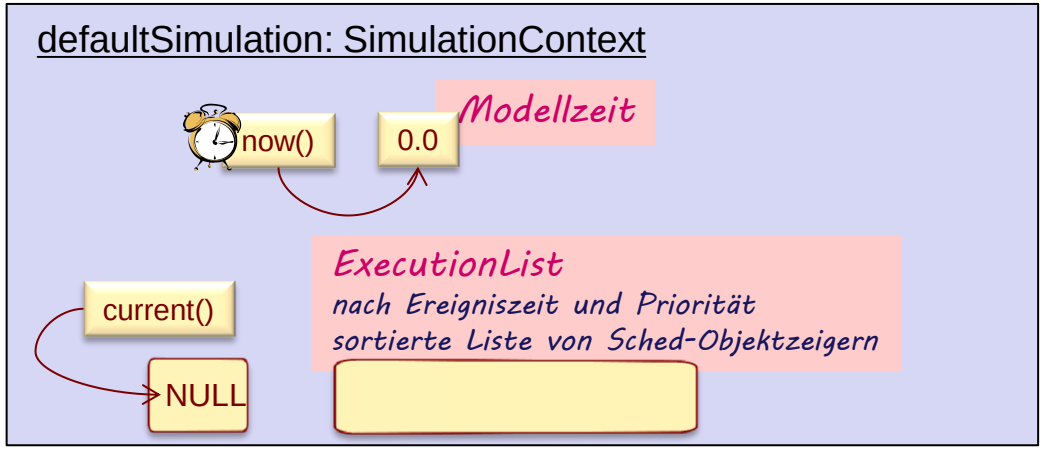
```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or passed";
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

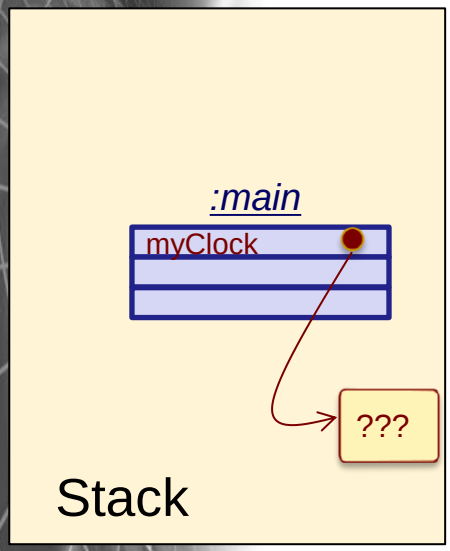
Simulationskontext

Scheduling-Operationen
Statusabfragen

Ablauf (1): Übergabe der Steuerung durch das Betriebssystem





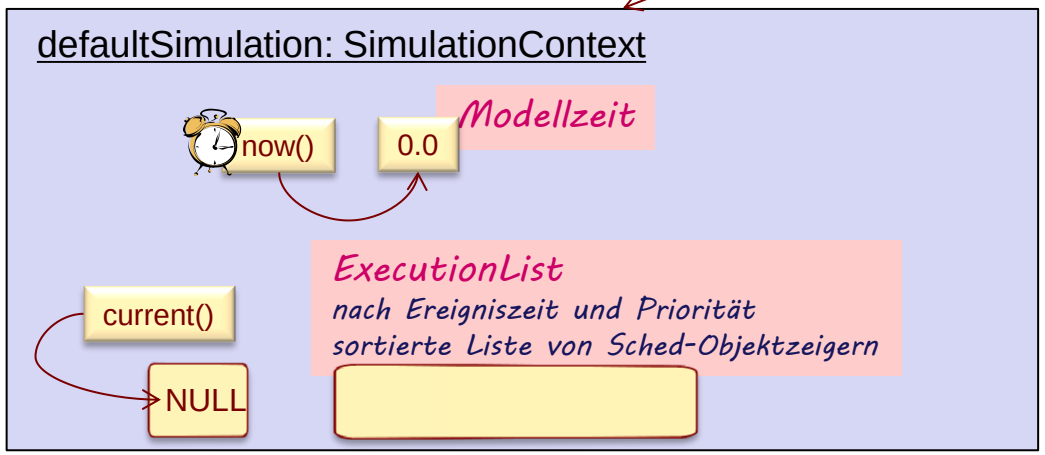



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or passed";
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

nichtsichtbarer InitialisierungsCode
von globalen Variablen, Objekten
a) C++ Laufzeitsystem
b) ODEMx-Laufzeitsystem
c) Anwenderprogramm

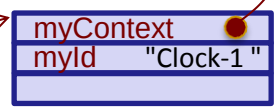
Ablauf (2): Objekt-Generierung

globale initialisierte Daten



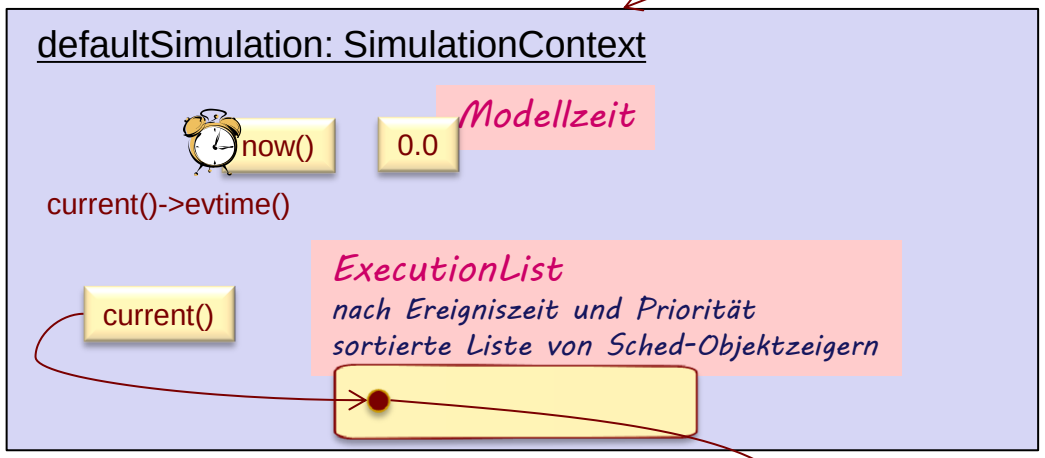

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Ex
        cout << "==========

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->s
            cout << endl << i << ". ste
                getDefaultSimulation
        }
        cout << endl;
        cout << "continue until SimT
        getDefaultSimulation()->run
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "======================" << endl;
        return 0;
}
```

```
class Clock : public Process {
public:
    Clock (Simulation* sim) :
        Process(sim, "Clock") {}

    virtual int main() {
            while (true) {
                holdFor(1.0);
                cout << '.';
            }
            return 0;
    }
};
```



Ablauf (3): Befüllen des Terminkalenders

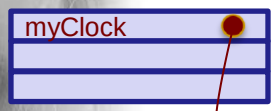
globale initialisierte Daten



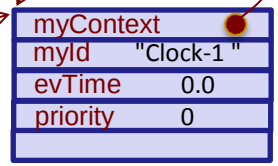

```
int main(int argc, char* argv[]) {
    Clock* myClock= new Clock (getDefaultSimulation());
    myClock->activate();

    cout << "Basic Simulation Example" << endl;
    cout << "========================" << endl;

    for (int i=1; i<5; ++i)
        getDefaultSimulat
        cout << endl << i <
            getDefaultSim
    }
    cout << endl;
    cout << "continue un
    getDefaultSimulation
    cout << endl  << "tim

    cout << "==========
    return 0;
}
```

```
class Clock : public Process {
public:
    Clock (Simulation* sim) :
        Process(sim, "Clock") {}

    virtual int main() {
        while (true) {
            holdFor(1.0);
            cout << '.';
        }
        return 0;
    }
};
```

:main

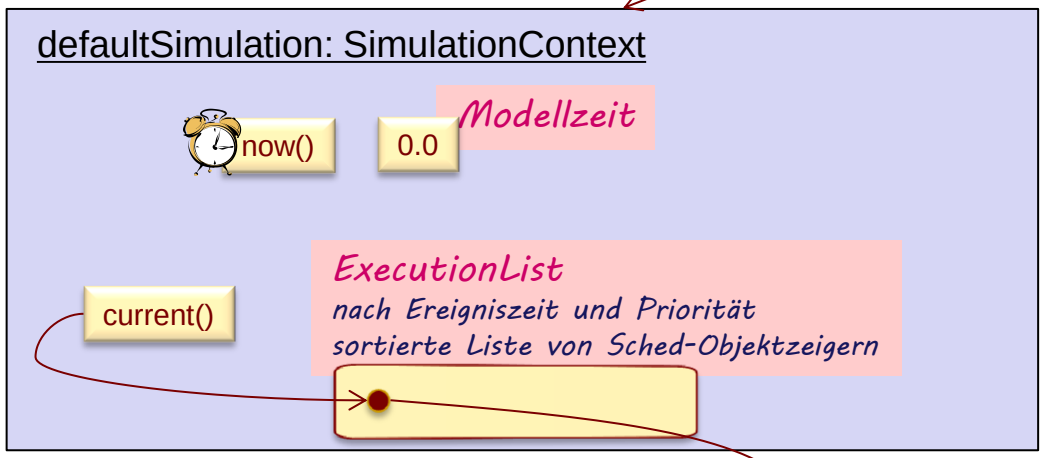


:Clock



Ablauf (4): Ausgabe

globale initialisierte Daten

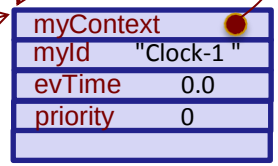



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "=======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulat
            cout << endl << i <
                getDefaultSim
        }
        cout << endl;
        cout << "continue un
        getDefaultSimulation
        cout << endl  << "tim

        cout << "==========
        return 0;
}
```

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

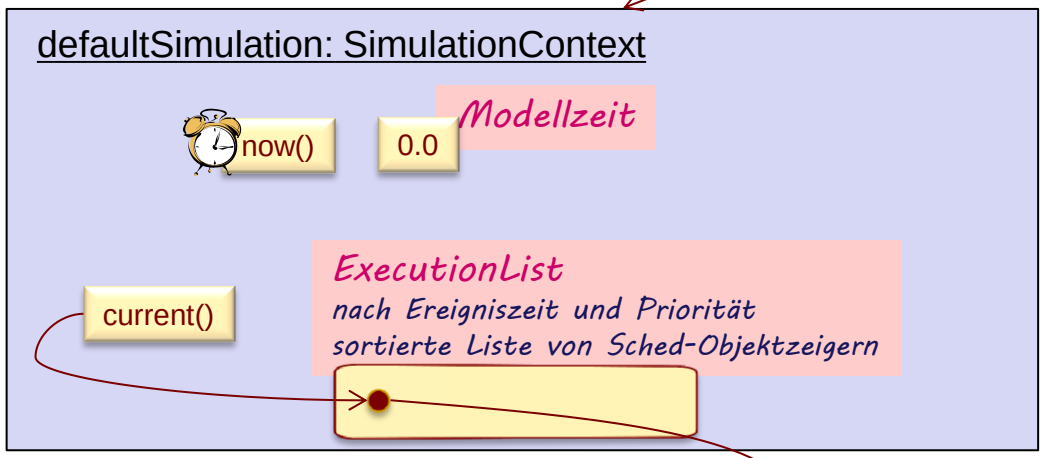
:main

:Clock



Ablauf (5): Steuerungsübergabe (Schrittmodus)

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "=======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "=======================" << endl;
        return 0;
}
```

:main

myClock

i= 1

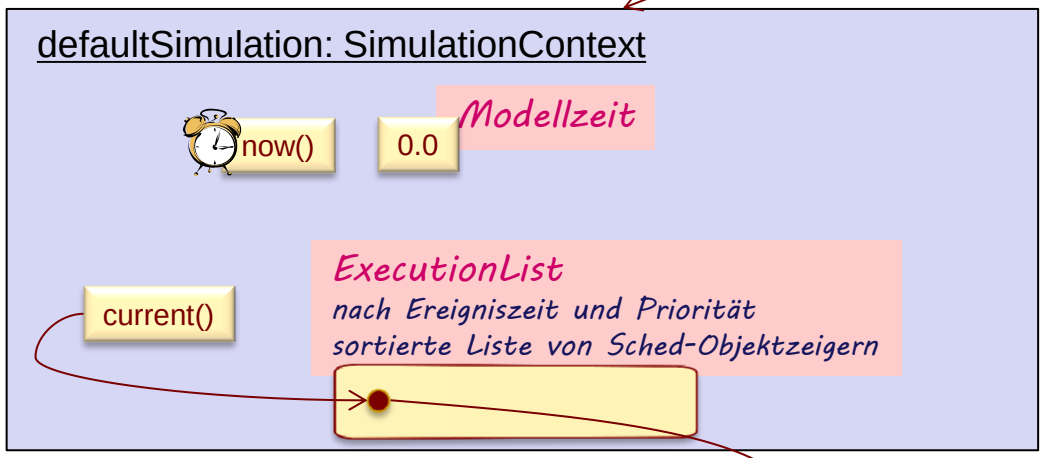
:Clock

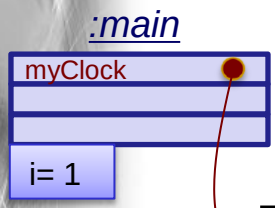
myContext

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

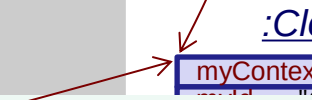
Ablauf (6): erfolgter Stack-Wechsel u. Sprung

globale initialisierte Daten





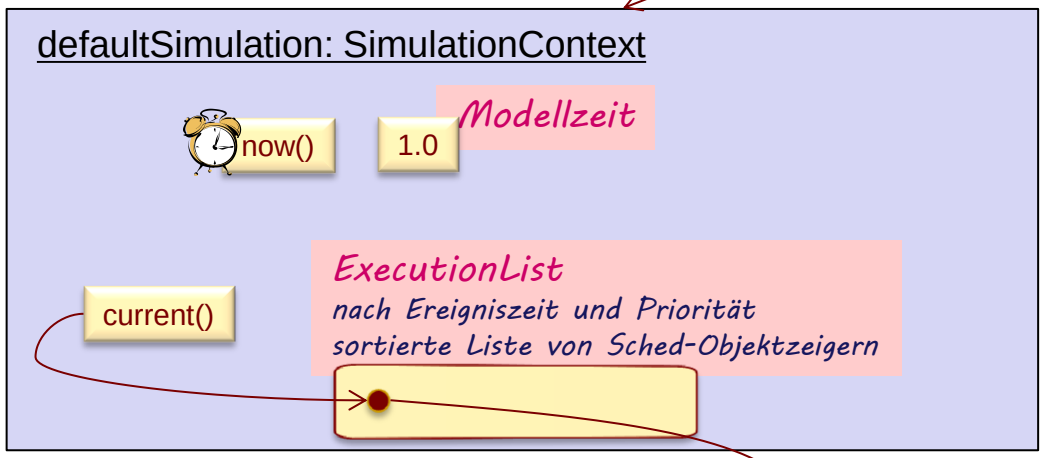

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "========================" << endl;
        return 0;
}
```



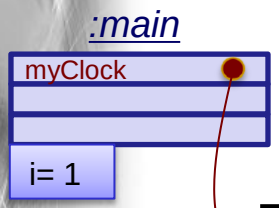

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (7): Verzögerung von Clock-1 um 1.0 ZE

globale initialisierte Daten



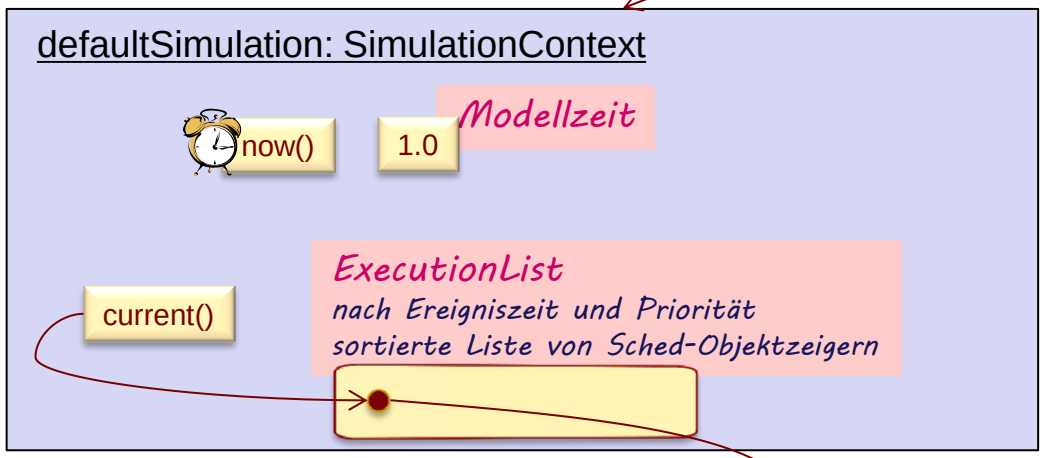

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "=======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". step time=" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "=======================" << endl;
        return 0;
}
```



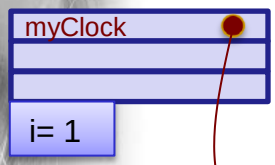

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

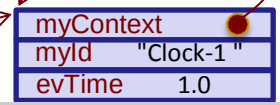
Ablauf (8): Rücksprung ins Hauptprogramm

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```



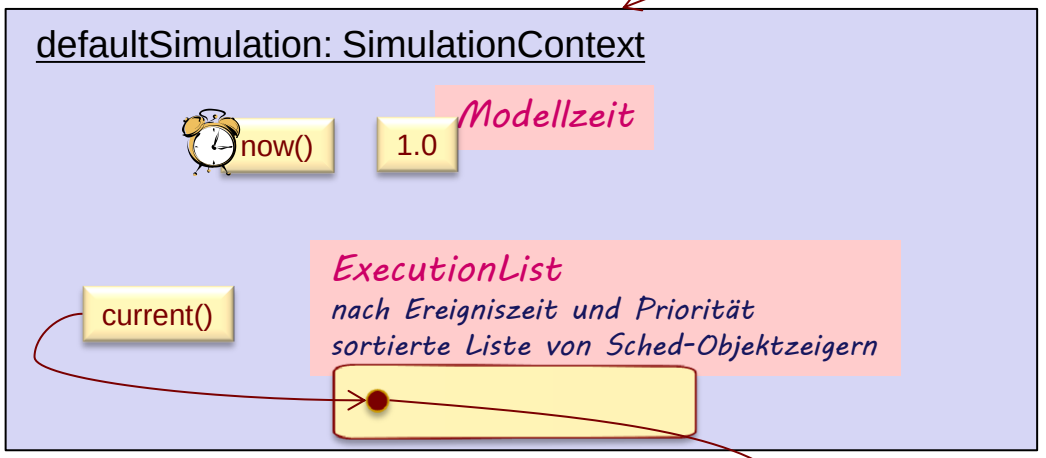



```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

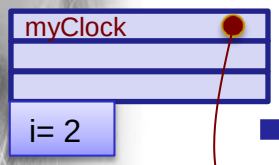
Ablauf (9): Hauptprogramm-Ausgabe

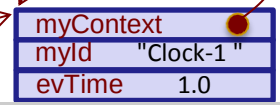
```
Basic Simulation Example
========================
1. time step at= 1.0
```

Ablauf (10): Steuerungsübergabe (Schrittmodus)




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```

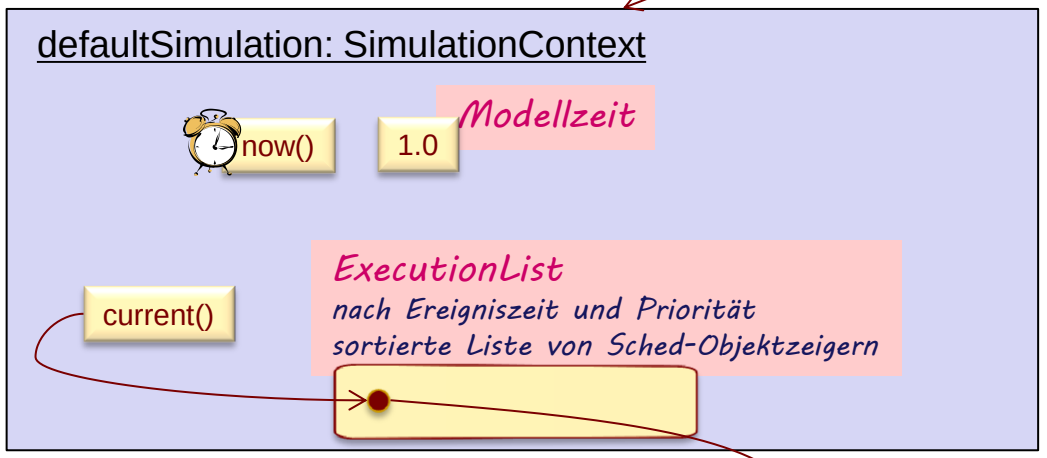




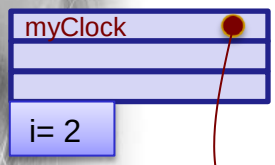

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

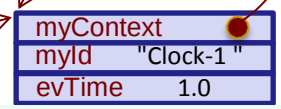
Ablauf (11): Clock-1 Ausgabe vom 1.Punkt

globale initialisierte Daten



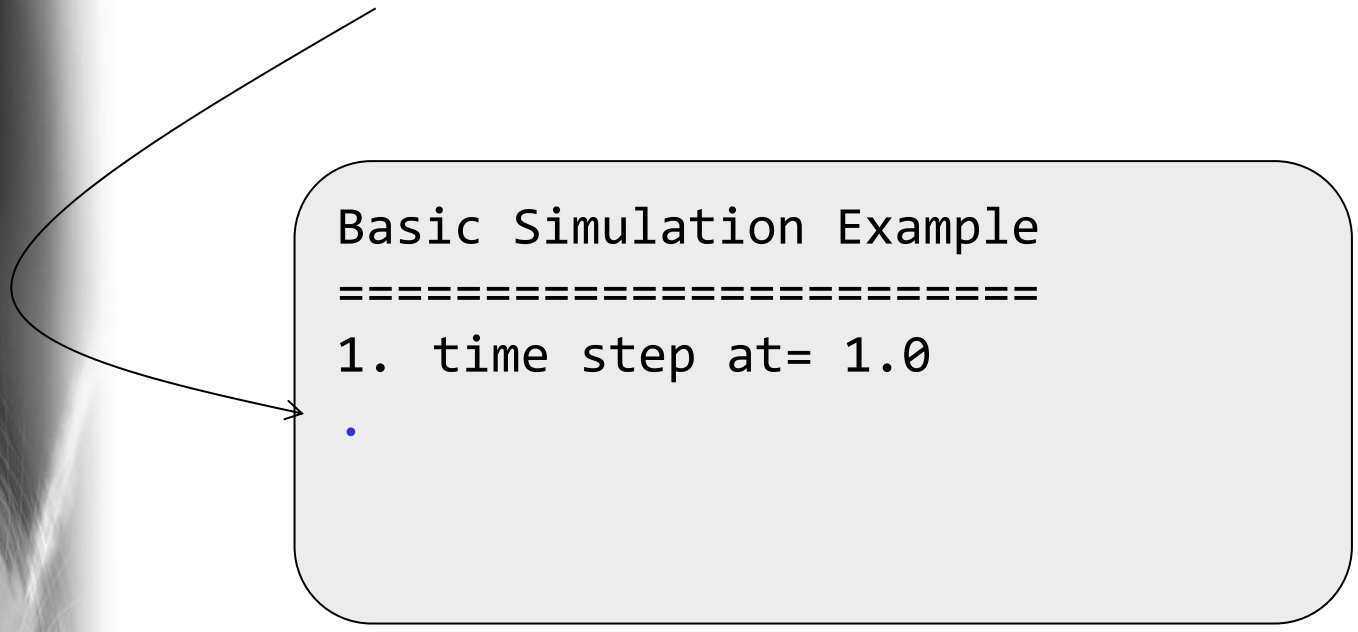

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```





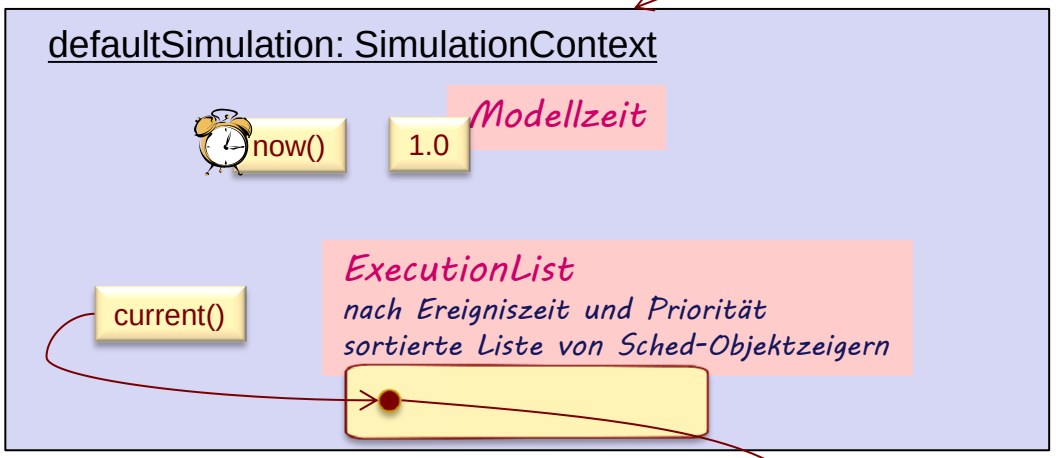

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (11): Clock-1 Ausgabe vom 1.Punkt

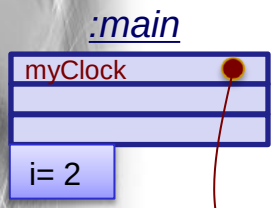


Ablauf (12): Verzögerung von Clock-1 um weitere ZE

globale initialisierte Daten



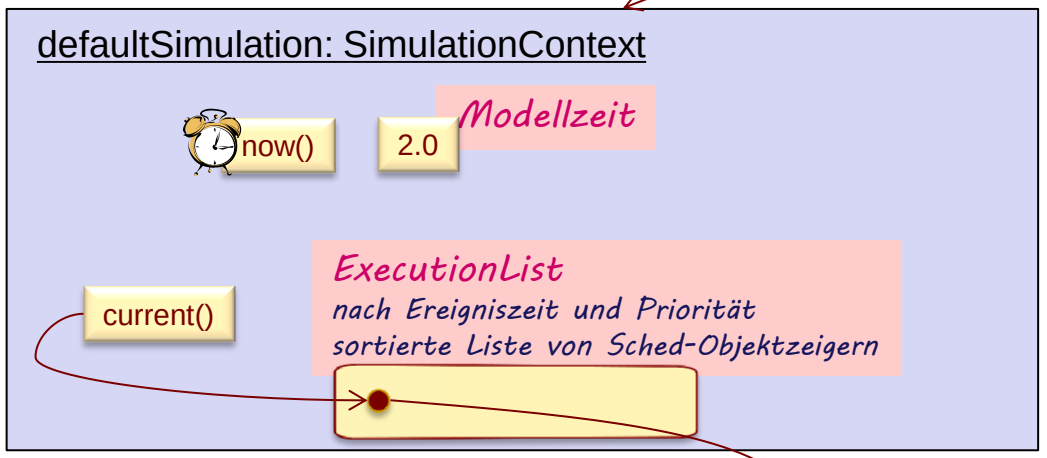

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```



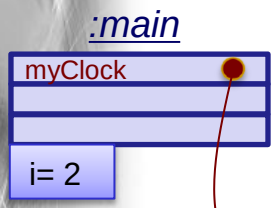

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (13): Rücksprung ins Hauptprogramm

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```




```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (14): Hauptprogramm Ausgabe

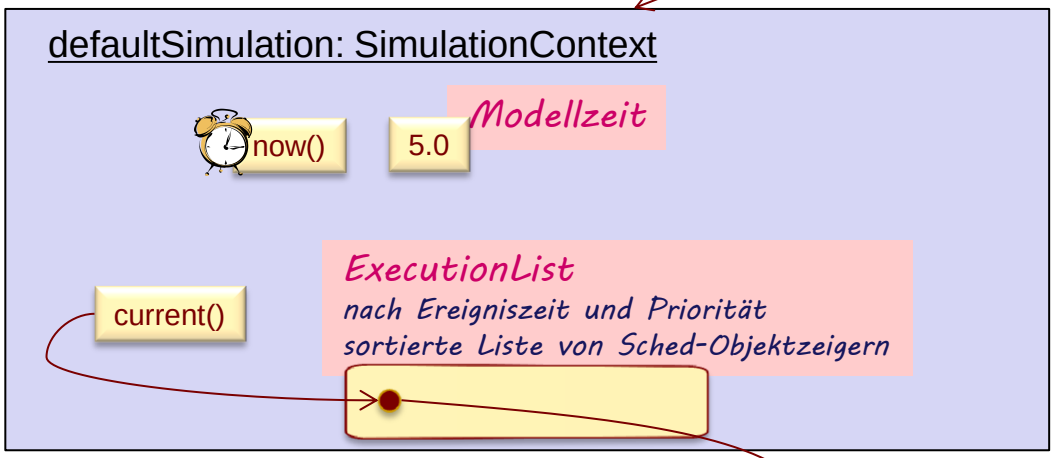
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
```

Ablauf (15): … bis zur Beendigung der For-Anweisung

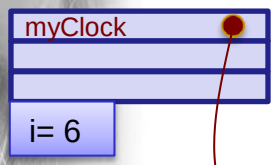
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
```

Ablauf (16): … inkl. letzter Punktausgabe

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```

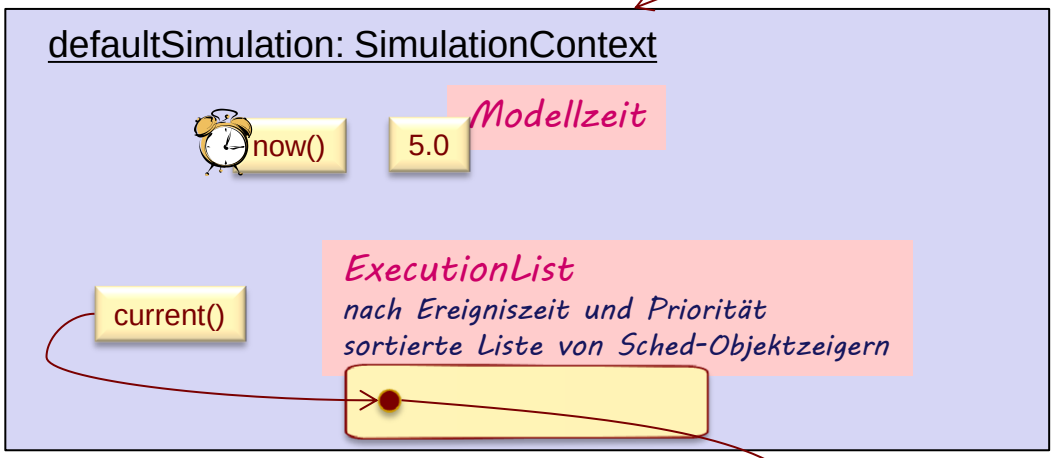



```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

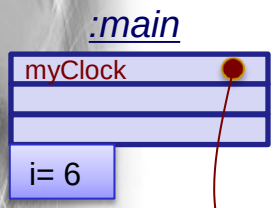
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
```

Ablauf (17): Verzögerung um 1 weitere ZE mit Rückkehr zu main

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```



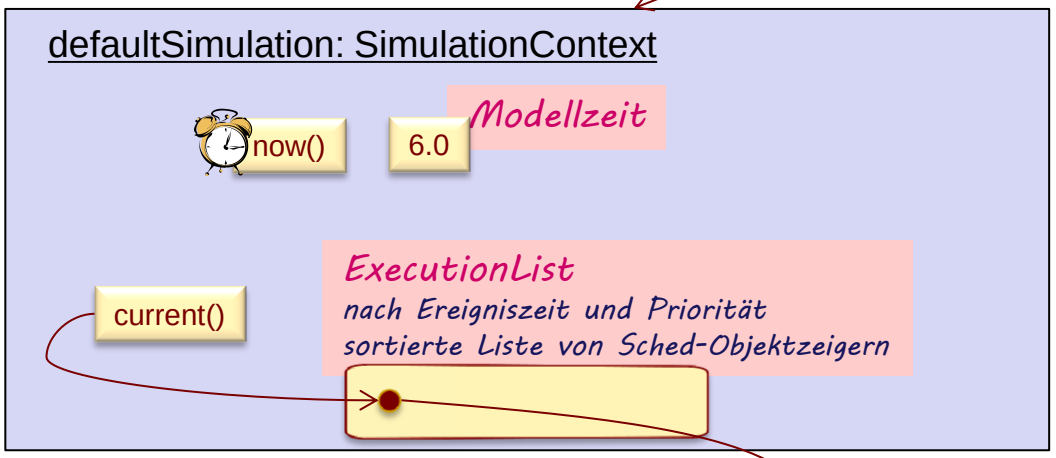

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (18): Ausgabe nach Beendigung der For-Anw.

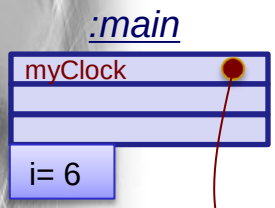
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
```

Ablauf (19): Steuerungsübergabe (Intervallmodus)

globale initialisierte Daten



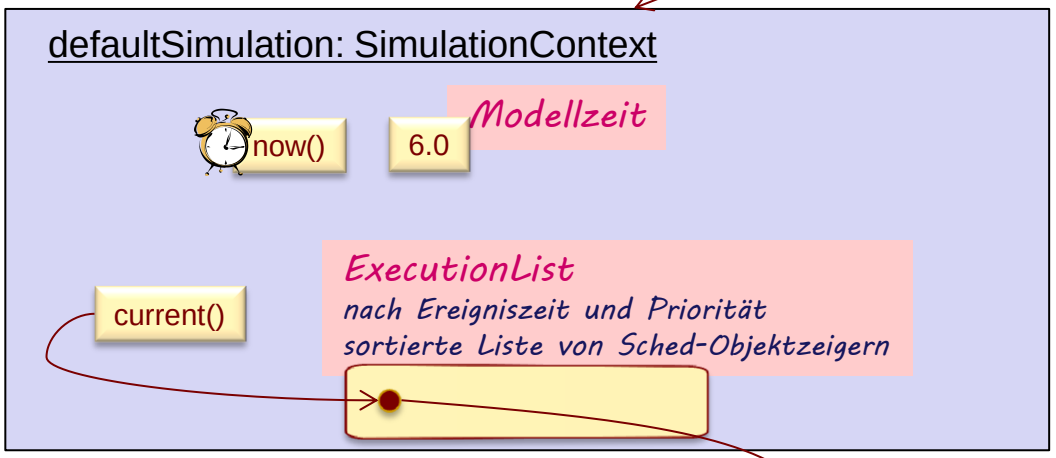

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "========================" << endl;
        return 0;
}
```



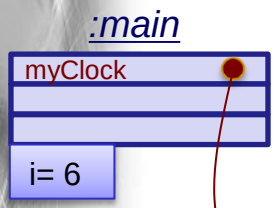

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ablauf (20): Fortsetzung mit Clock-1

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```




```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Kein Rücksprung

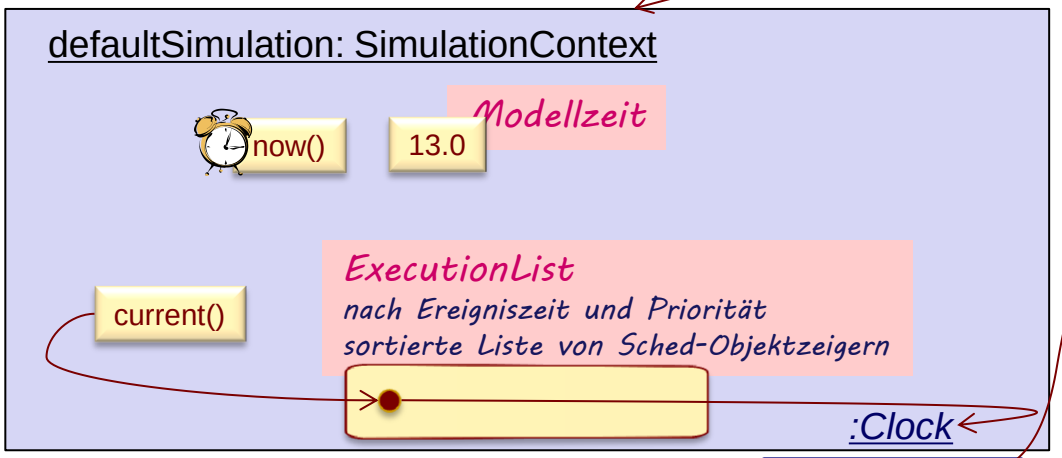
Ablauf (21): Ausgabe bis zur Modellzeit von 13.0

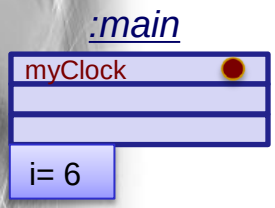
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
........
```

insgesamt 13 Punkte

Ablauf (22): Rücksprung aus Intervall-Modus

globale initialisierte Daten






```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

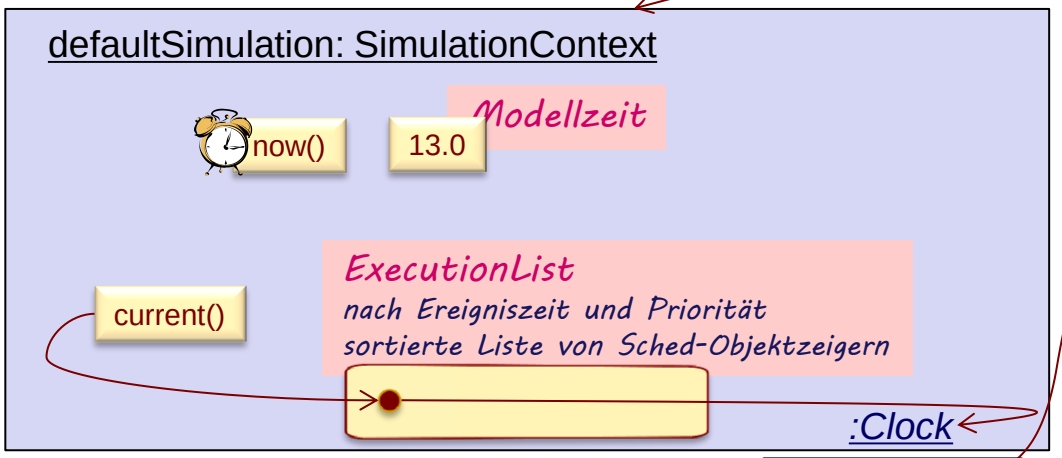
```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

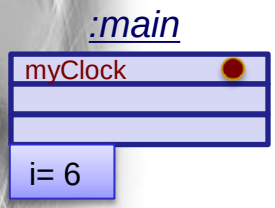
Ablauf (23): Komplettierung der Ausgabe

```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
........
time= 13.0
========================
```

Ablauf (24): Ende des Programms

globale initialisierte Daten






```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation()->getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

3. ODEMx-Prozess-Scheduling

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Varianten der Kontextaktivierung

Methoden der Klasse Simulation (Simulationskontext)
(aufgerufen vom C++ Hauptprogramm)

bisher eingesetzt

1. Einzelschrittausführung: step()

2. Lauf bis zum Erreichen/Überschreiten einer vorgegebenen Modellzeit (SimTime): runUntil(…)

3. Lauf bis zum Ende der Simulation: run()

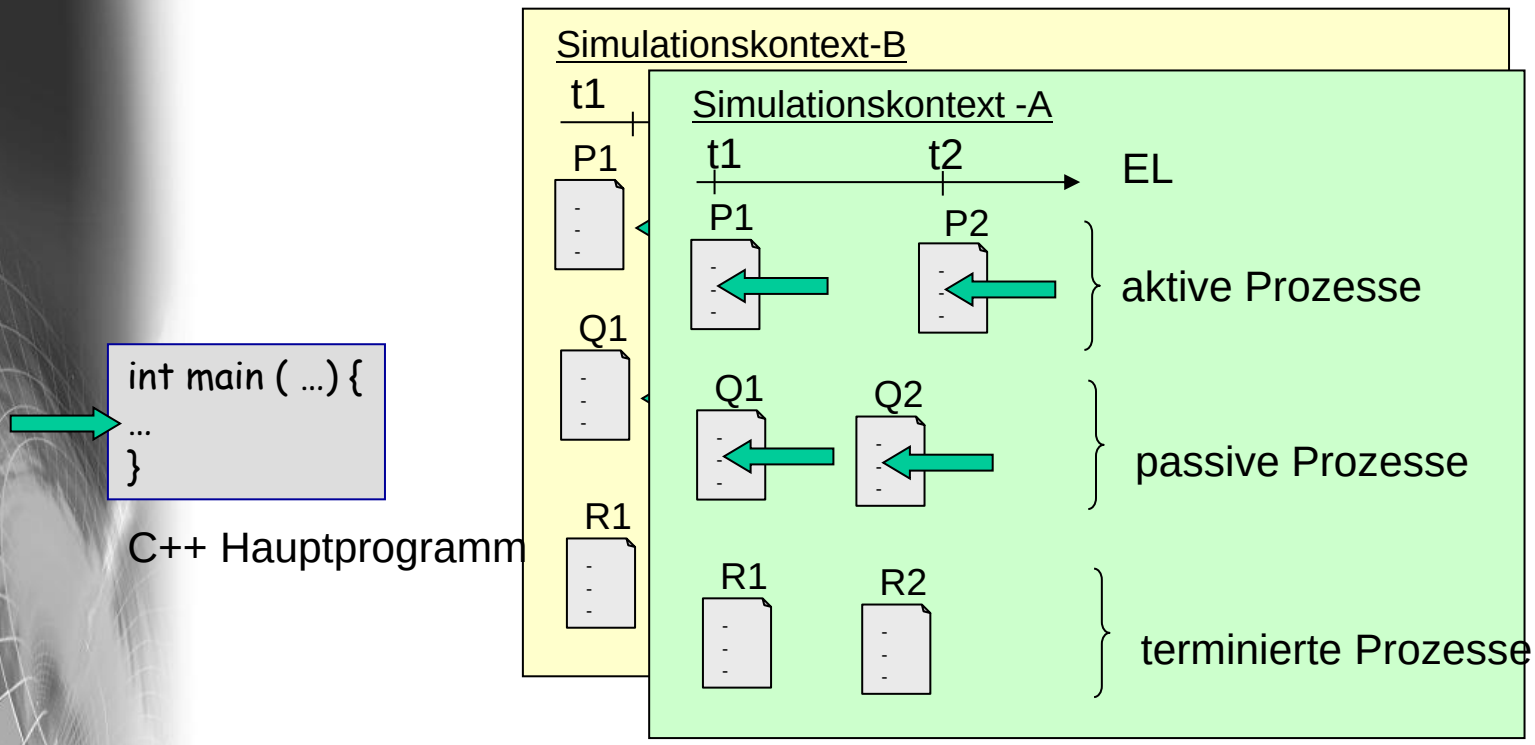
Rückkehr ins C++ Hauptprogramm:

- ***implizit***:
  es gibt keinen **aktiven** Prozess mehr im zugehörigen Simulationskontext (Kalender ist leer)

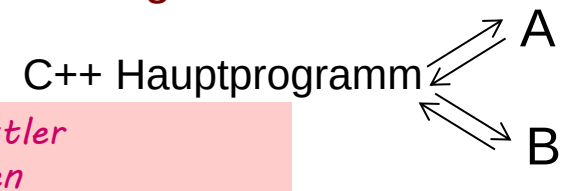
- ***explizit***: die Simulation wurde mit exitSimulation() durch einen beliebigen Prozesses des Simulationskontextes beendet

typisch für Arbeit DefaultSimulation-Kontext

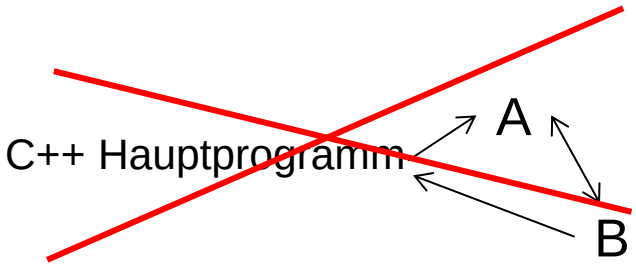
Verwaltung mehrerer Simulationskontexte



Steuerungszenarien:



als Mittler zwischen den Prozess-Systemen



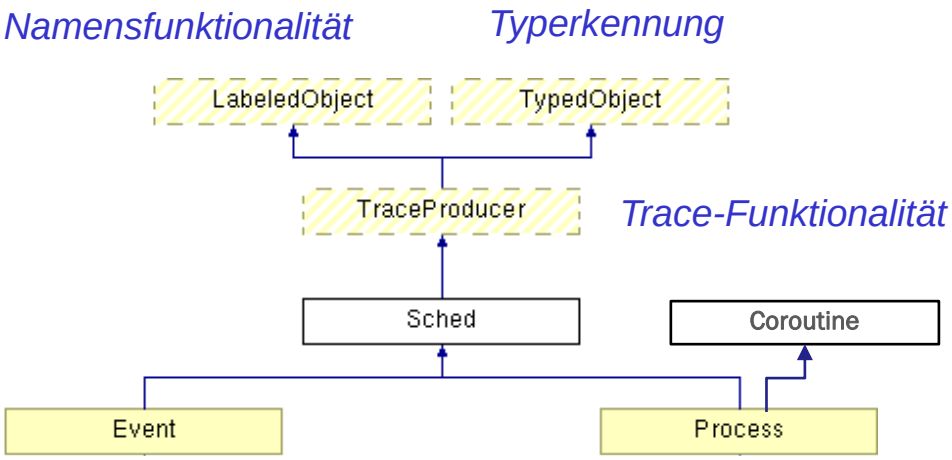
Klasse Sched, Event, Process

Sched

- abstrakte Klasse
- Objekte werden im Kalender in chronologischer Reihenfolge erfasst

Simulationslauf

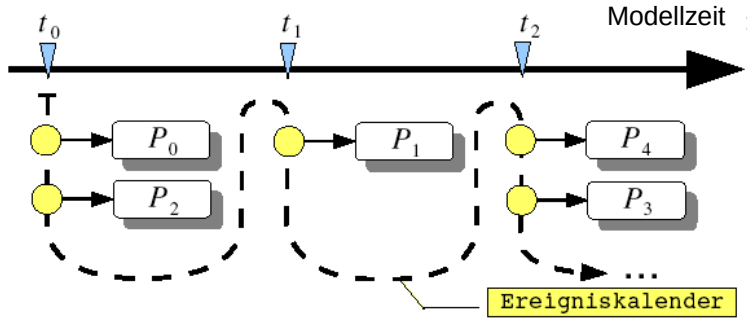
- realisiert durch Ausführung (execute) von Sched-Objekten
- in Abhängigkeit von
  - der jeweiligen Kalenderkonstellation und
  - des Typs der Sched-Objekte: Event, Process



dabei können neben **Zustandsänderungen** auch
- Eintragungen,
- Verschiebungen und Streichungen von **Sched-Objekten** vorgenommen werden

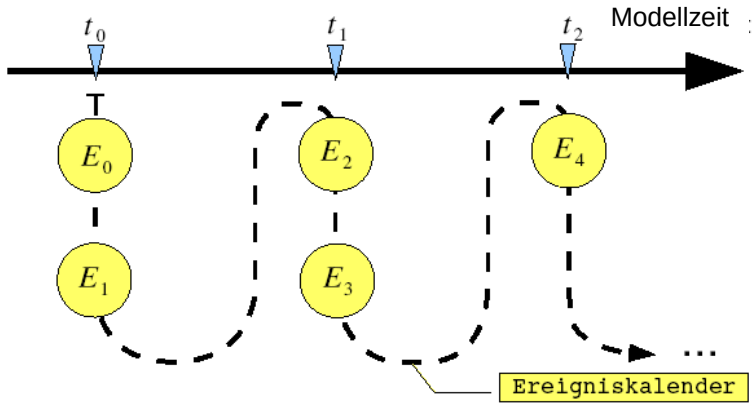
```
virtual SimTime getExecutionTime () const =0          // Get model time.
virtual SimTime setExecutionTime (SimTime time)=0     // Set model time.
virtual Priority getPriority () const =0
virtual Priority setPriority (Priority newPriority)=0 // Set new priority.
bool isScheduled () const                             // Check if Sched object is in schedule.
SchedType getSchedType () const                       // Determine the Sched object's type.
virtual void execute ()=0                             // Execution of Sched object.
```

Realisierungen der Next-Event-Simulation



Prozess-Scheduling

(Prozess als Folge von Ereignissen)



Ereignis-Scheduling

(Folge von Ereignissen)

ODEMx erlaubt beide Varianten (auch im Mix)

[Sched als abstrakte Basisklasse von Process und Event]

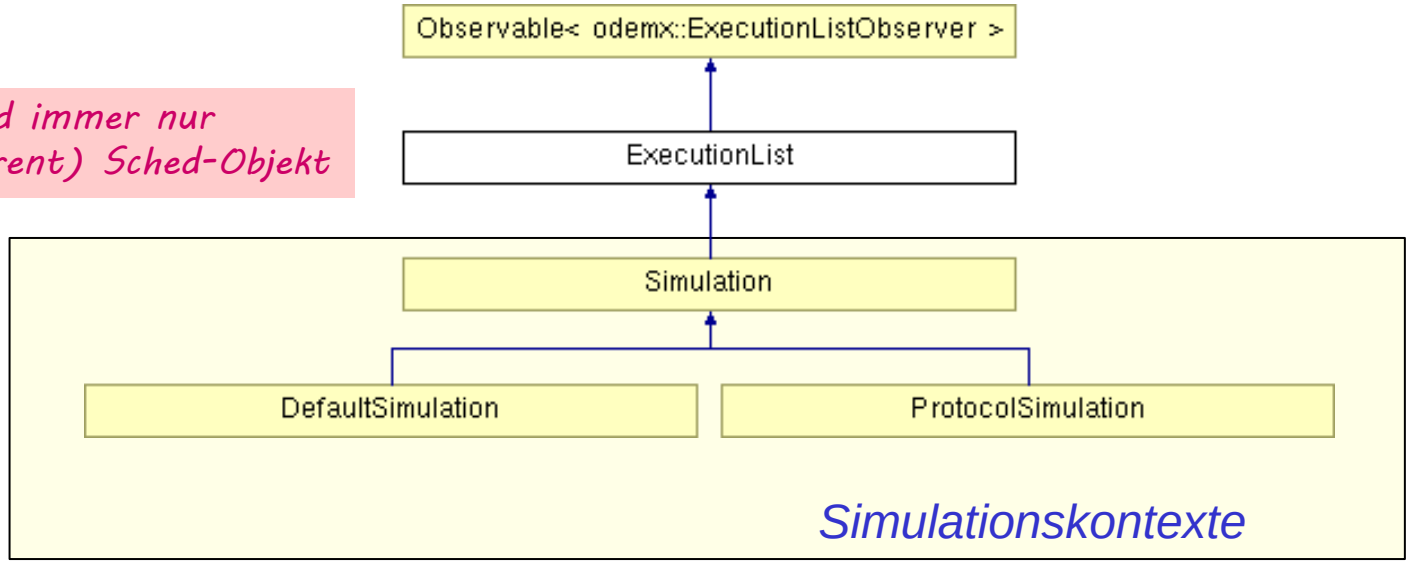
schöne Übungsaufgabe:
Clock-Beispiel als Event-Variante

Die Klasse ExecutionList (Ereignisliste, Kalender)

```
Sched * getNextSched ()    // top most Sched in ExecutionList
bool isEmpty ()            // check if ExecutionList is empty
virtual SimTime getTime () // const =0  get model time
```

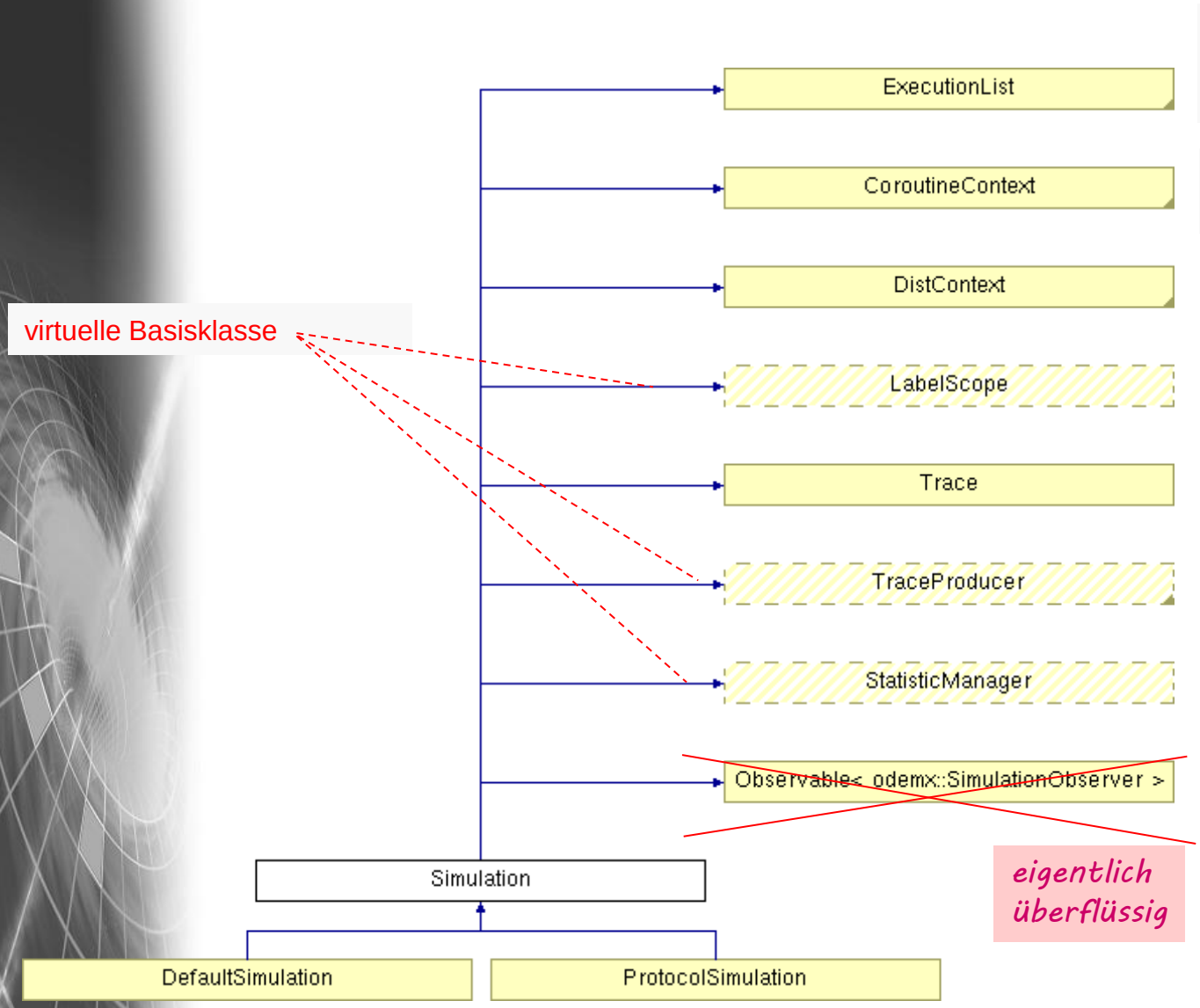
Vergangenheit wird nicht konserviert

ausgeführt wird immer nur das erste (current) Sched-Objekt



jeder Simulationskontext (Objekt von Simulation bzw. Ableitung) verfügt über eine eigene ExecutionList-Funktionalität

Simulationskontext



Prozess-Scheduling nach Zeit und Priorität
std::list<Sched*>

ExL

Koroutinen-Ensemble, inkl. C++ Hauptprogramm

Verwaltung aller erzeugten Zufallszahlengeneratoren

Objektnamenverwaltung

Trace-Manager um Ereignisse/Zustandsänderungen von Objekten zu registrieren

Erzeuger von Markierungen (marks) für Trace

Manager von Statistik-Objekten

Beobachtung registrierter Objekte

eigentlich überflüssig

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Grundstrategie

ein Prozess-Objekt

- bleibt in seinem gesamten Lebenslauf **einem einzigen Simulationskontext** zugeordnet

- ist während seines Lebenslaufes (in Abhängigkeit seines Grundzustandes) in vier unterschiedlichen **Listen** seines Simulationskontextes erfasst.

  Grundzustände
  - Created
  - Runable, dann auch in ExL
  - Idle, dann meist auch in dezentralen Synchronisationslisten
  - Terminated

Process-Member-Funktion

```
State getState() const;
```

zeitgleich kann ein blockierter Prozess (Idle) in weiteren Warteschlangen erfasst sein

Zugriffsfunktionen für Process-Listen

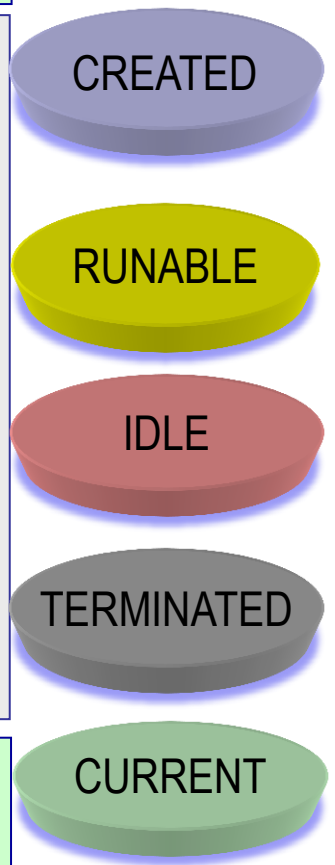
generell
Jeder Prozess wird in seinem gesamten Lebenslauf von seinem Simulationskontext verwaltet (ODEMx spezifischer Debugger)

```
std::list<Process*>& Simulation::getCreatedProcesses() {
      return created;
}
std::list<Process*>& Simulation::getRunnableProcesses() {
      return runnable;
}
std::list<Process*>& Simulation::getIdleProcesses() {
      return idle;
}
std::list<Process*>& Simulation::getTerminatedProcesses() {
      return terminated;
}
```

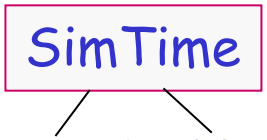
ExL

```
Process* Simulation::getCurrentProcess()
                        Get currently executed process.
Sched * getCurrentSched ()
                        Get currently executed Sched object.
```

Prozess-grundzustand



Zeitbezug



Modellzeit: Datentyp bestimmt Varianten von ODEMx: int, double

now

aktuelle Modellzeit (private Simulation Member-Variable)

getCurrentTime()
getSimulation()->getTime()

p->getExecutionTime()

geplante Aktivierungszeit eines beliebigen Prozesses p (in der ExL)

semantisch äquivalent:

now==
getCurrentTime()==
getCurrentProcess()->getExecutionTime() ==
getSimulation()->getTime()

Funktionssignaturen

Process-Member-Funktion

```
SimTime Process::getExecutionTime() const;
        // aktuelle Ereigniszeit
        //      0.0, falls Prozess nicht in ExL eingetragen ist
        // (Vorsicht: 0.0 legt allein noch nicht den Grundzustand fest)
```

Simulation-Member-Funktion

```
Process* Simulation::getCurrentProcess();
        // liefert Zeiger zum aktuellen Prozess der ExL

Simulation* getSimulation();
        // liefert Zeiger zum aktuellen Simulationskontext
```

globale Funktion

Prinzipielle Prozesslistenverarbeitung

```
typedef std::list<process*> LIST;

LIST&  myList= getSimulation().getCreatedProcesses();
                    // Übergabe der Referenz

for (LIST::iterator it= myList.begin();
      it != myList.end();  ++it)          {

    *it-> … // Zugriff auf Attribut/Memberfunktion
}
```

erster Eintrag

Nachfolger vom letzten Eintrag

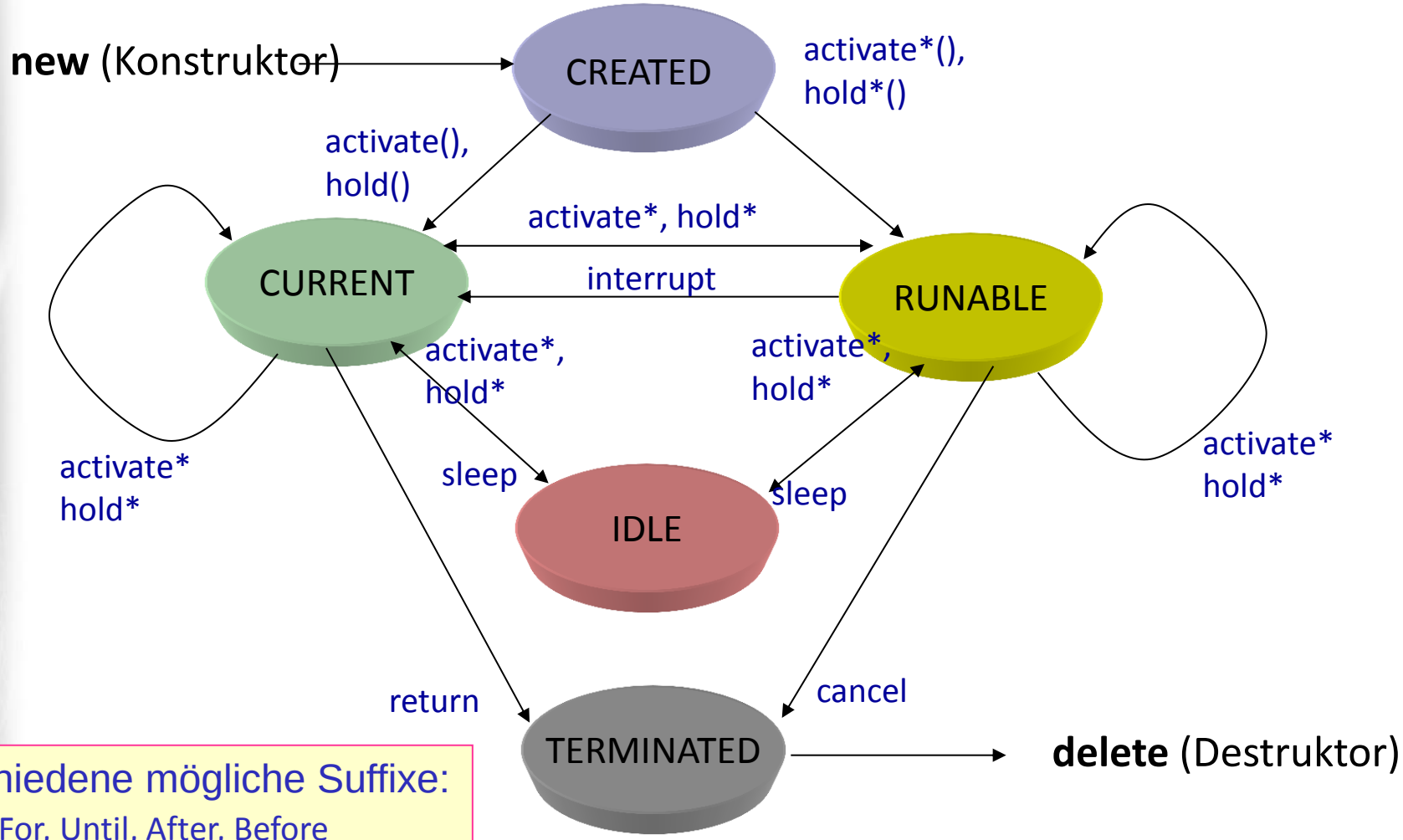
Wirkungsweise:
Gegenstand von OMSI-2

it++
verhält sich bzgl. it-Wert genauso, liefert aber alten Wert als zusätzliches Ergebnis
(n= it++), der hier nicht gebraucht wird
➔ Programm würde hier unnötig verlangsamt

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Überblick: Zustände und Scheduling-Operationen



* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before

Process: Scheduling-Operationen (1)

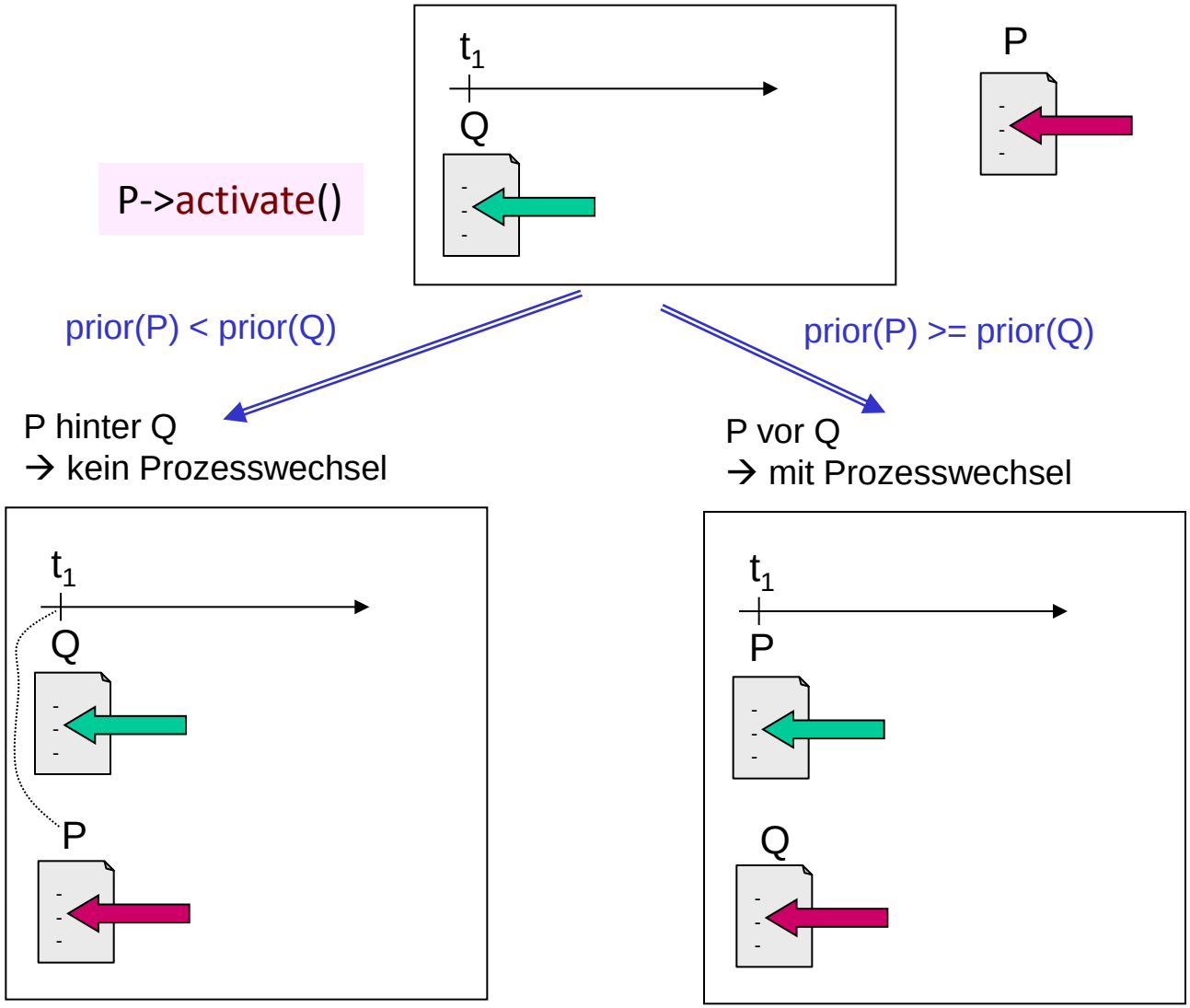
Prozessaktivierungen nach dem LIFO-Prinzip

```
void activate();          // Eintrag in ExL zur aktuellen Ereigniszeit now
                          // nach dem LIFO-Prinzip und Beachtung der Priorität
                          // i.d.R. Prozesswechsel
void activateIn (SimTime t);
                          // Eintrag in ExL zur Ereigniszeit now + t
                          // nach dem LIFO-Prinzip und Beachtung der Priorität
                          // falls t<0.0, dann t= 0.0

void activateAt (SimTime t);
                          // Eintrag in ExL zur absoluten Ereigniszeit t
                          // nach dem LIFO-Prinzip und Beachtung der Priorität
                          // falls t<now, dann t= now
```

Achtung:
nur aus dem Simulationskontext heraus,
nicht bei Aktivierung aus dem Hauptprogramm

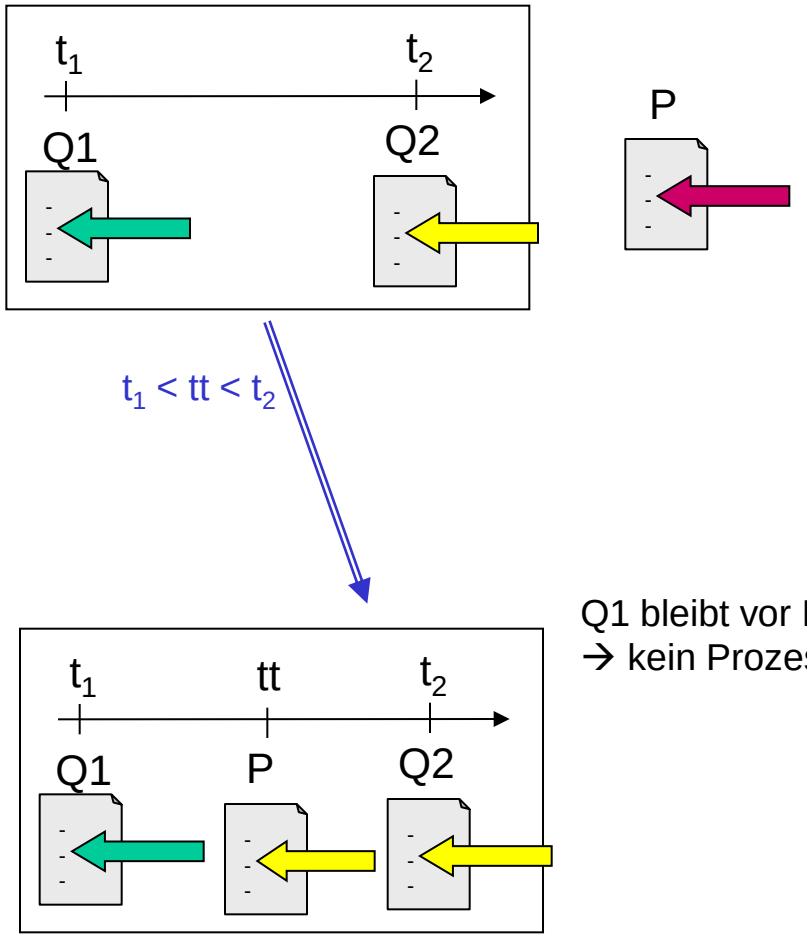
semantisch äquivalent:
P->activate() == P->activateIn(0.0) == P->activateAt(now)

Activate innerhalb eines Simulationskontextes



Activate innerhalb eines Simulationskontextes

P->activateAt(tt)

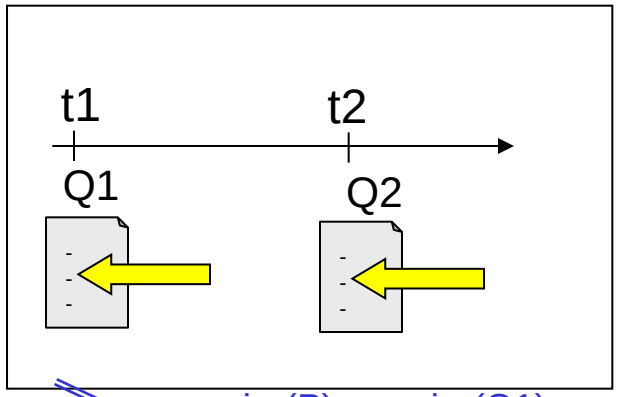


Q1 bleibt vor P
→ kein Prozesswechsel

Activate außerhalb eines Simulationskontextes

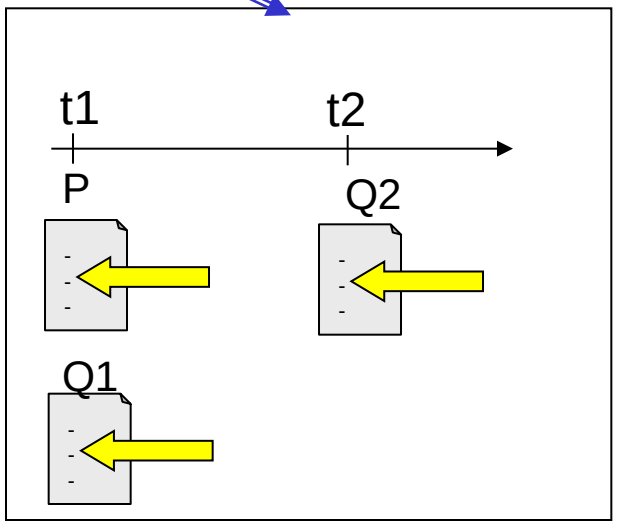
Simulationskontext (DefaultSimulation-Objekt)

```
int main ( ...) {
P->activate()
...
}
```



prior(P) >= prior(Q1)

```
int main ( ...) {
... P->activate()
...
}
```



trotzdem
noch kein
Prozesswechsel !
Hauptprogramm
setzt Ausführung fort

Process: Scheduling-Operationen (2)

Prozessverzögerungen nach dem FIFO-Prinzip

```
void hold();
                    // Eintrag zur aktuellen Ereigniszeit
                    // als letzter bei gleicher oder niedrigerer Priorität (FIFO)

void holdFor (SimTime t);
                    // Eintrag zur Ereigniszeit now + t
                    // als letzter bei gleicher oder niedrigerer Priorität (FIFO)
                    // falls t<0.0, dann t= 0.0

void holdUntil (SimTime t);
                    // Eintrag zur absoluten Ereigniszeit t
                    // als letzter bei gleicher oder niedrigerer Priorität (FIFO)
                    // falls t<now, dann t= now
```

semantisch äquivalent:

```
p->hold() == p->holdFor(0.0) == p->holdUntil(now)
p->holdUntil(t) == p->holdFor(t-now)
```

Process: Scheduling-Operationen (3)

Prozessaktivierungen nach dem Vorher/- Nachherprinzip

```
void activateBefore (Process* p);
                        // unmittelbarer Eintrag vor p mit Ereigniszeit von p,
                        // Übernahme der Priorität von p
                        // falls p == this: leere Anweisung
                        // falls p nicht in der ExL: Fehlermeldung, Abbruch

void activateAfter (Process* p);
                        // unmittelbarer Eintrag nach p mit Ereigniszeit von p
                        // Übernahme der Priorität von p
                        // falls p == this: leere Anweisung
                        // falls p nicht in der ExL: Fehlermeldung, Abbruch
```

Process: Scheduling-Operationen (4)

Prozessunterbrechungen

```
void sleep();
        // Entfernung von currentProcess()  / runable-Prozess aus der ExL
        // Wechsel in Zustand idle, Ereigniszeit wird 0.0
        // Aktivierung des ersten ExL-Eintrages
        //          falls Process, dann auch Prozesswechsel )
        //          falls ExL leer, dann Rückkehr ins Hauptprogramm

virtual void interrupt();
        // runable-Prozess wird in seiner hold/activate-Phase unterbrochen
        // wird evtl. zum neuen Current-Prozess (aus Zukunft zurückgeholt), falls kein
        // Prioritätskonflikt
        // und kann mit getInterrupter() die erfolgte Unterbrechung erkennen und
        // selbst behandeln

void cancel();
        // Prozessabbruch, Entfernung aus der ExL
        // Wechsel in Zustand terminated
        // erneute Aktivierung führt zum Fehler, Abbruch
```

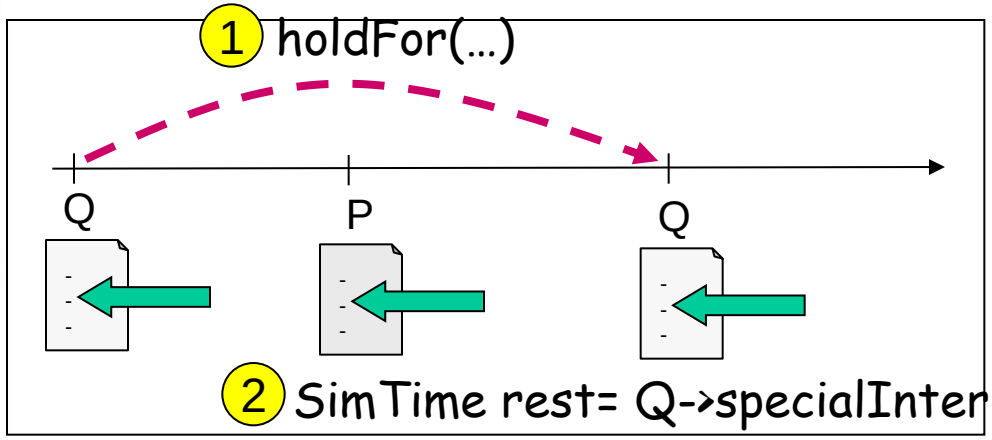
geplante Ereigniszeit des zu unterbrechenden Prozesses

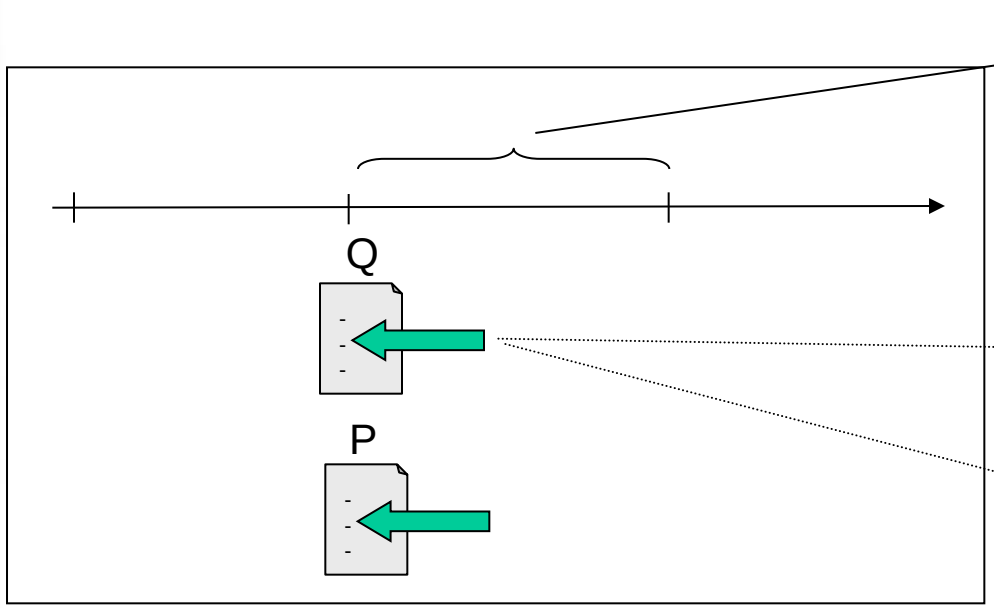
Kaskadierung von interrupt könnte die ausstehende Restzeit ermitteln:

```
SimTime specialInterrupt() {
        SimTime t= getExecutionTime() – getSimulation()->getTime();
        interrupt();
        return t;
}
```

Unterbrechungszeitpunkt, (aktuelle Modellzeit des interrupt-Rufers)

Process: Interrupt-Mechanismus (1)





Process: Interrupt-Mechanismus (2)

Unterbrechungsbehandlung

```
bool isInterrupted() const {return interrupted;}
        // Abfrage eines Interrupt-Zustandes (nach erfolgtem interrupt)
        // true, falls Unterbrechung erfolgte

Sched* getInterrupter() const {return interrupter;}
        // Anzeige des Prozesses/Ereignisses, der/das interrupt() gerufen hat
        // falls isInterrupted() == true und
        //       getInterrupter()==0: dann war Interrupter der
        //  Simulationskontext


void resetInterrupt() {interrupted=false; interrupter=0;}
        // löscht Interrupt-Zustandseinträge
        // implizit bei jeder Scheduling-Operation
```