

# ODEMx-control

- Kontrollmechanismus in ODEMx
- ODEMx-Version 3.1
- Author: *Jonathan Schlue*
- **07.11.2016**

# Projektziel

Entwicklung einer ODEMX-Komponente für die Definition von Bedingungen, deren Wahrheitswert bei jeder Änderung einer der beteiligten Variablen automatisch neu evaluiert wird.

# Teilprojektschritte

1. Verwendung von Lambda-Ausdrücken zur *in-situ* Definition von Bedingungsausdrücken
2. Entwurf einer Komponente, mit der sich Variablen auszeichnen lassen, welche bei Zuweisungen automatisch registrierte Lambda-Ausdrücke informieren
3. Integration dieser Komponententen in ODEMX:
  - generische `wait()`-Methoden für Prozesse
  - generische Gewichtungsfunktionen für `waitQ`
  - generische Definition von [Stop-Bedingungen] von Differentialgleichungen von kontinuierlichen Prozessen
4. Beispielsimulator
5. Dokumentation

# (1.) Verwendung von Lambda-Ausdrücken

- **Idee:** Manipuliere globale Typdefinitionen in `TypeDefs.h` :

```
/* ... */  
typedef bool(Process::*Condition)();  
/* ... */  
typedef bool(Process::*Selection)(Process* partner);  
/* ... */  
typedef double(Process::*Weight)(Process* partner);  
/* ... */
```

# std::function

---

Defined in header `<functional>`

---

```
template< class >
class function; /* undefined */ (since C++11)
```

---

```
template< class R, class... Args >
class function<R(Args...)> (since C++11)
```

---

- "Class template `std::function` is a general-purpose polymorphic function wrapper. Instances of `std::function` can store, copy, and invoke any Callable target -- functions, **lambda expressions**, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members. [...]"
- <http://en.cppreference.com/w/cpp/utility/functional/function>

# TypeDefs.h

v3.0

```
/* ... */  
typedef bool(Process::*Condition)();  
/* ... */  
typedef bool(Process::*Selection)(Process* partner);  
/* ... */  
typedef double(Process::*Weight)(Process* partner);  
/* ... */
```

# TypeDefs.h

v3.1

```
/* ... */  
typedef  
    std::function<bool(Process* subject)>  
    Condition;  
/* ... */  
typedef  
    std::function<bool(Process* master, Process* slave)>  
    Selection;  
/* ... */  
typedef  
    std::function<double(Process* master, Process* slave)>  
    Weight;  
/* ... */
```

# Änderungen bei Funktionsaufrufen

v3.0

- in CondQ.cpp

```
(process->*cond)();
```

- in Continuous.cpp

```
(this->*stopCond)();
```

- in WaitQ.cpp

```
(master->*weightFct)(slave);
```

```
(master->*sel)(slave);
```



# Änderungen bei Funktionsaufrufen

## v3.1

- in CondQ.cpp

```
cond(process);
```

- in Continuous.cpp

```
stopCond(this);
```

- in WaitQ.cpp

```
weightFct(master, slave);
```

```
sel(master, slave);
```

# Teilprojektschritte

1. Verwendung von Lambda-Ausdrücken zur *in-situ* Definition von Bedingungsausdrücken -- **DONE**
2. Entwurf einer Komponente, mit der sich Variablen auszeichnen lassen, welche bei Zuweisungen automatisch registrierte Lambda-Ausdrücke informieren
3. Integration dieser Komponententen in ODEMX:
  - generische `wait()` -Methoden für Prozesse
  - generische Gewichtungsfunktionen für `waitq` -- **DONE**
  - generische Definition von [Stop-Bedingungen] von Differentialgleichungen von kontinuierlichen Prozessen -- **DONE**
4. Beispielsimulator
5. Dokumentation

## (2.) Kontrollvariablen

- eine Komponente, mit der sich Variablen auszeichnen lassen, welche bei Zuweisungen automatisch registrierte Lambda-Ausdrücke informieren, nennen wir **Kontrollvariable**

# Kontrollvariablen

- Idee: Variablen, die bei *Änderungen* wartende Prozesse alarmieren
  - i. Kontrollvariablen kennen die Prozesse, die auf sie warten
  - ii. bei Änderung werden wartende Prozesse aufgeweckt, um ihre Bedingungen neu zu evaluieren
  - iii. Handhabung genau wie übliche Variablen

# Kontrollvariablen

- Anwendung hauptsächlich für built-in-Datentypen
  - `int` , `long` , `unsigned` , ...
  - `float` , `double`
  - `bool`
- Daher: **Beschränkung auf built-in-Datentypen**
  - Vereinfachung der Benutzung
  - Steigerung der Usability

# Kontrollvariablen: Basisklasse

## Klasse `ControlBase`

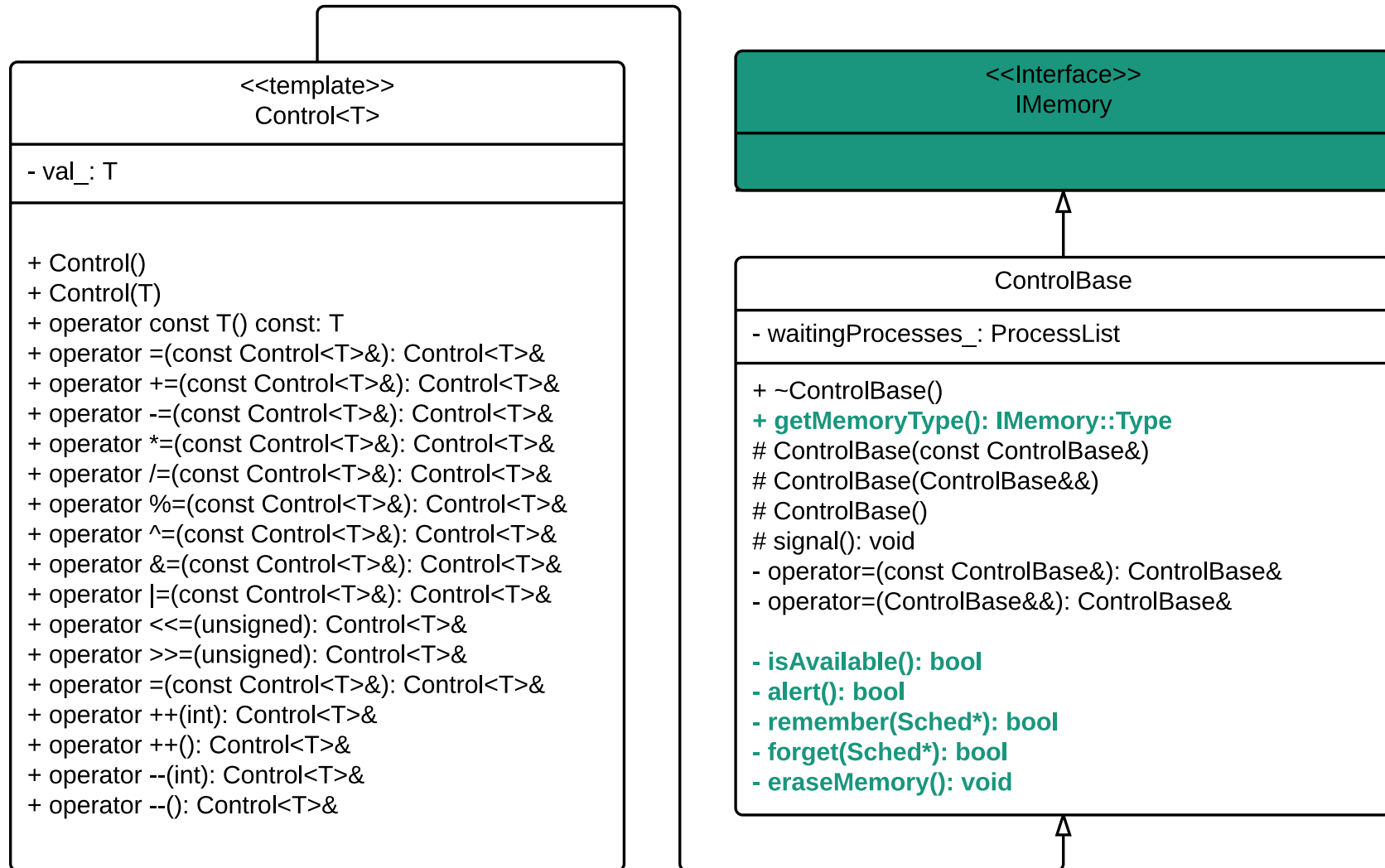
- Basisklasse aller Kontrollvariablen
- Beinhaltet Liste wartender Prozesse
- `signal()` -Methode zur alarmierung wartender Prozesse
- Implementiert Interface `IMemory`
  - ermöglich Integration in *Alert*-Mechanismus für Prozesse

# Kontrollvariablen: Klassentemplate

## Klassentemplate `Control<T>`

- Klassentemplate als Wrapper für Variablen von built-in-Typen `T`
- Subklasse von `ControlBase`
- Beinhaltet Variable vom Typ `T`
- überlädt alle ändernden Operatoren
  - **alle** Zuweisungsoperatoren
  - Inkrement, Dekrement
- Änderungen an Kontrollvariable werden direkt an innere Variable delegiert
- bei *tatsächlichen Änderungen* wird Signal ausgelöst
  - Aufruf von `ControlBase::signal()`

# Kontrollvariablen: Klassendiagramm





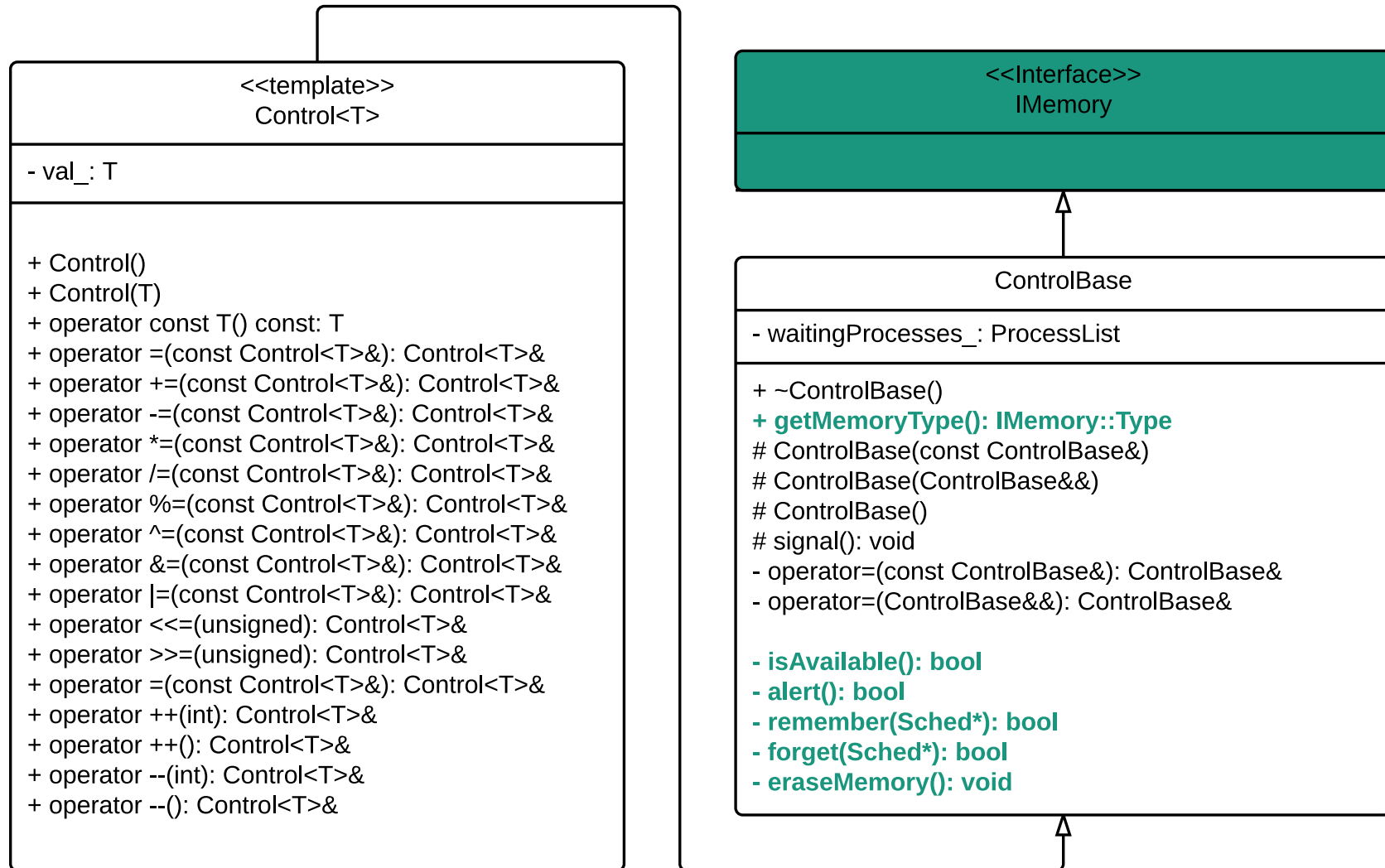
## Problem: Unterscheidung von built-in-Typen

- Denn:
  - Inkrement, Dekrement *nicht zugelassen* bzw. *deprecated* für `bool`
  - Bitwise Operationen *nicht zugelassen* für Fließkommazahlen

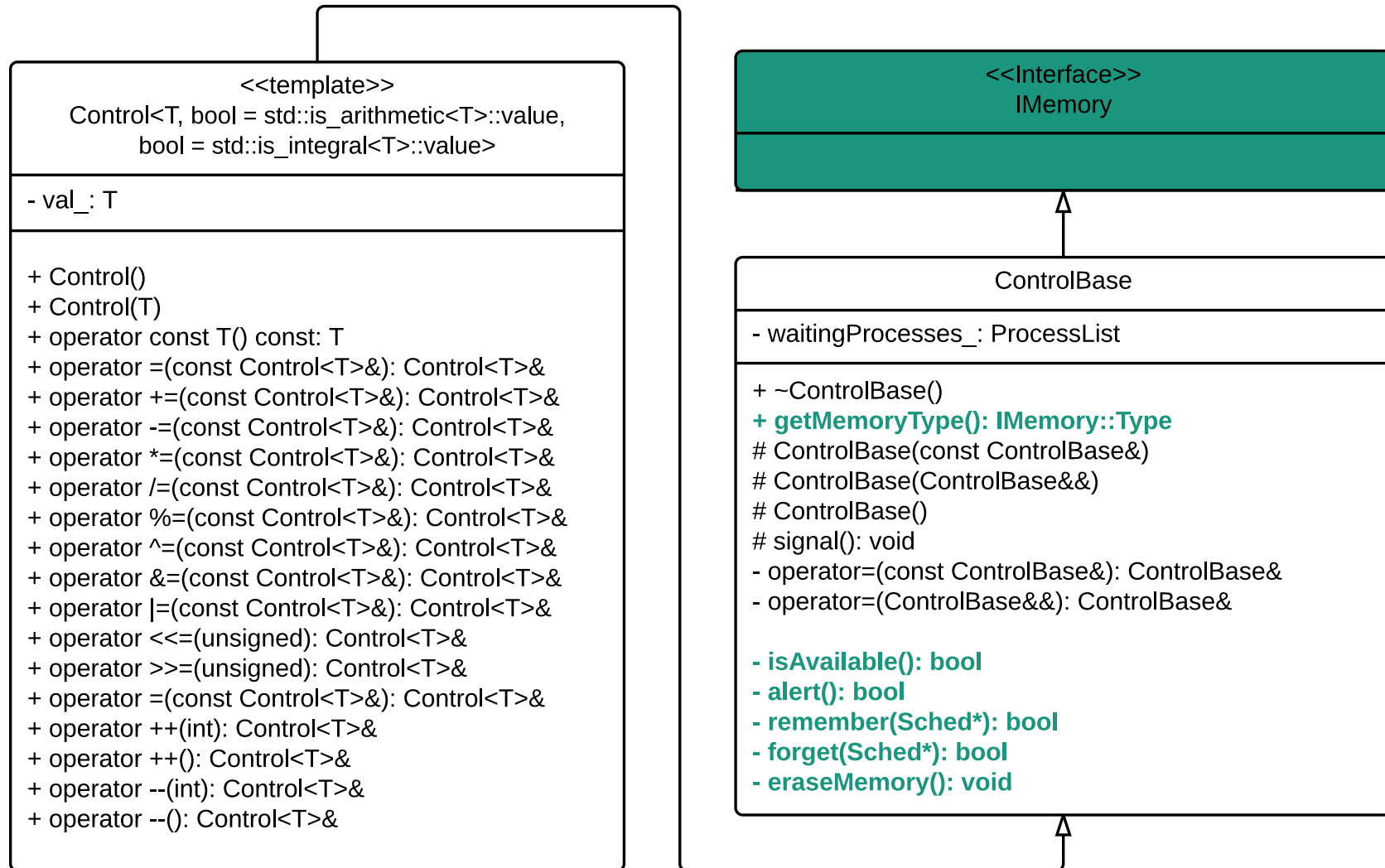
## Lösung

- Partielle Template-Spezialisierungen
- Type-Traits
  - `std::is_arithmetic`
  - `std::is_integral`

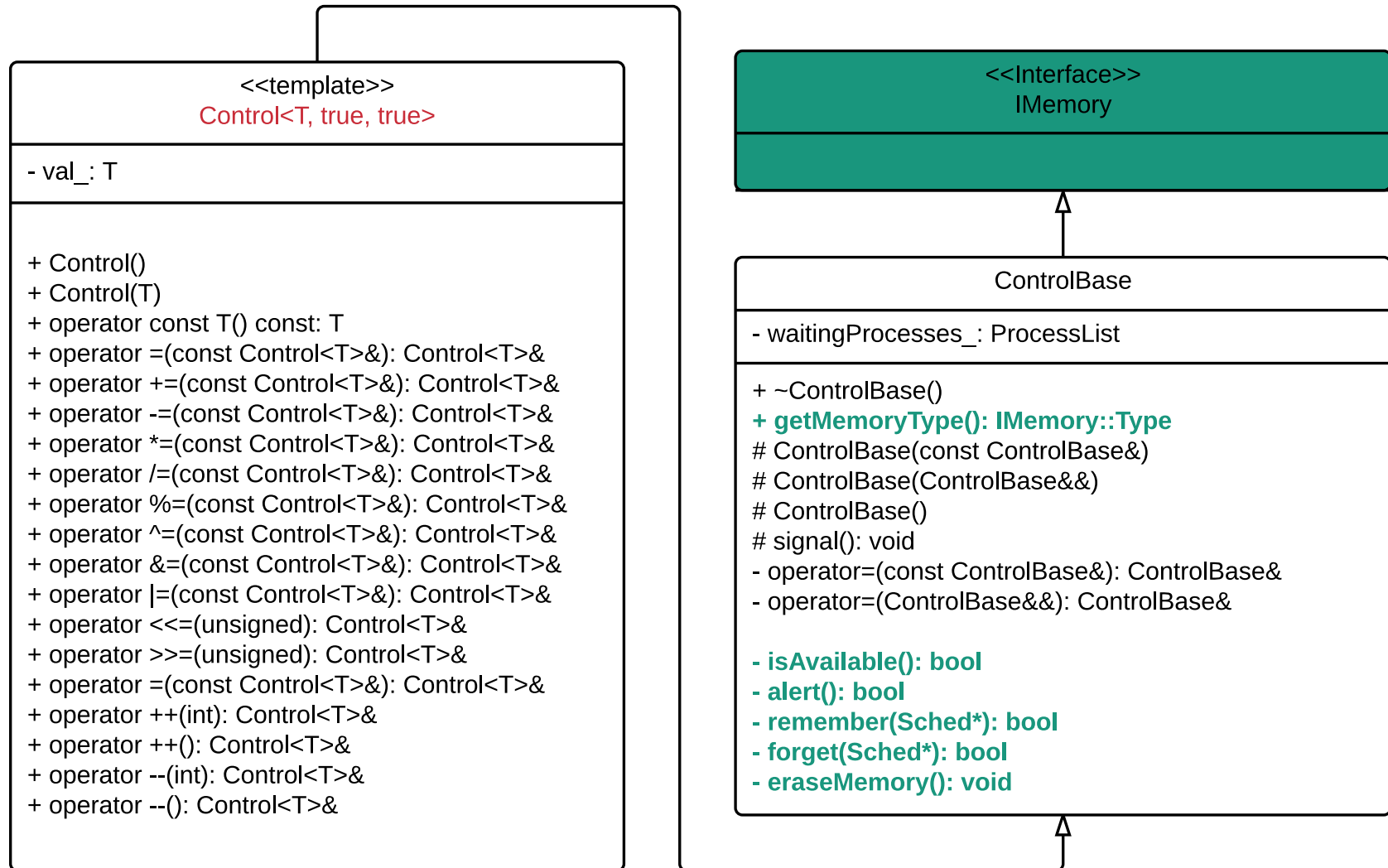
# Kontrollvariablen: Klassendiagramm



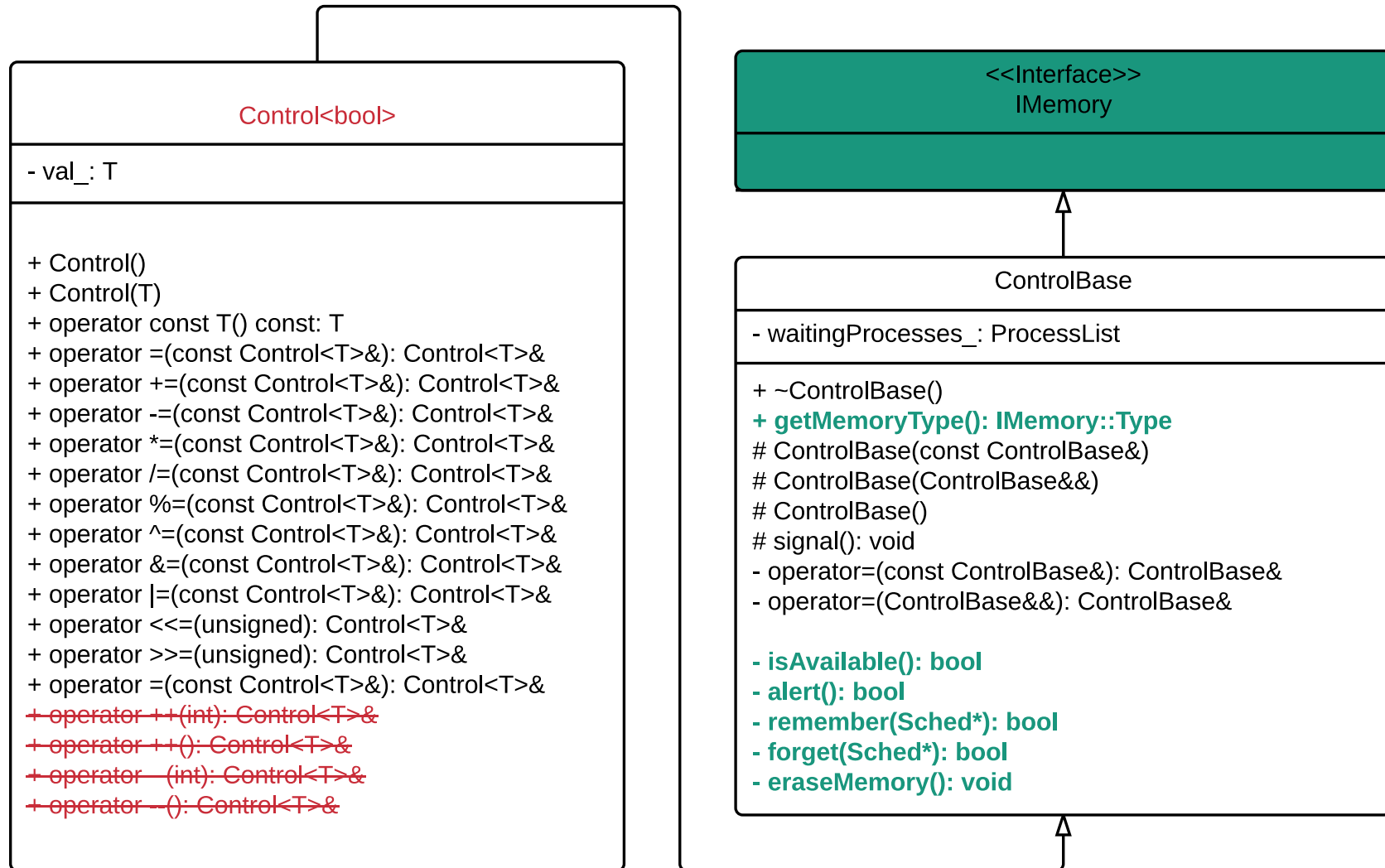
# Kontrollvariablen: Klassendiagramm



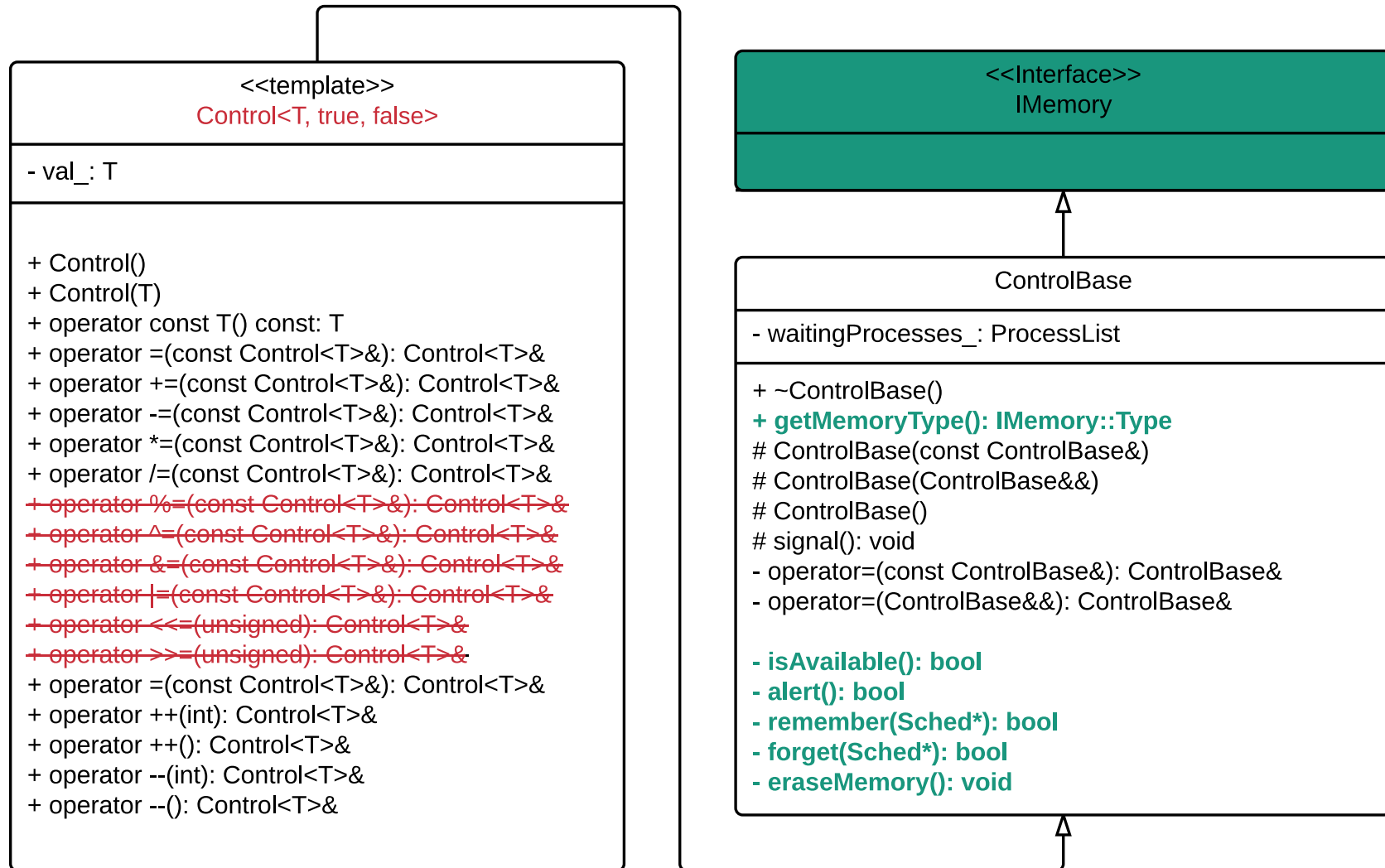
# Templatespezialisierung: T in { int , unsigned , long , ... }



# Templatespezialisierung: T ist bool



# Templatespezialisierung: T in { float , double , ... }



# Teilprojektschritte

1. Verwendung von Lambda-Ausdrücken zur *in-situ* Definition von Bedingungsausdrücken -- **DONE**
2. Entwurf einer Komponente, mit der sich Variablen auszeichnen lassen, welche bei Zuweisungen automatisch registrierte Lambda-Ausdrücke informieren -- **DONE (?)**
3. Integration dieser Komponententen in ODEMX:
  - generische `wait()` -Methoden für Prozesse
  - generische Gewichtungsfunktionen für `waitq` -- **DONE**
  - generische Definition von [Stop-Bedingungen] von Differentialgleichungen von kontinuierlichen Prozessen -- **DONE**
4. Beispielsimulator
5. Dokumentation



## (2.) Informieren von Lambda-Ausdrücken

- Idee: generische Methode `Process::waitUntil()`
  - bekommt Bedingung und die dafür relevanten Kontrollvariablen als Parameter übergeben
  - Orientierung an `CondQ::wait(Condition)`

## Process::waitUntil - Dokumentation

```
bool waitUntil ( const Condition & condition,  
                 const std::string label,  
                 const std::vector< std::reference_wrapper< ControlBase >> & controls  
                 )
```

This function can be used by processes to wait for arbitrary condition.

### Parameters

- condition** the condition to wait for to become true. This process will be passed as an argument to the condition.
- controls** a list of controlled variables, which inform the **Process** whenever they get accessed so that the condition turned true
- label** a label for all log entries related to this **wait()** call

### Returns

false if the process was interrupted, otherwise true

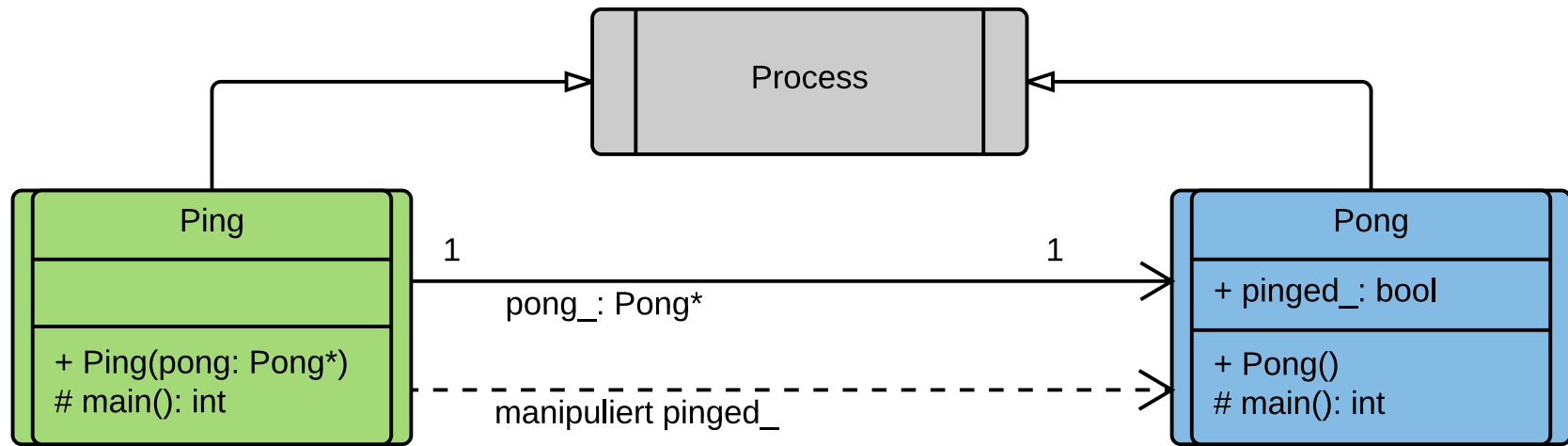
# Teilprojektschritte

1. Verwendung von Lambda-Ausdrücken zur *in-situ* Definition von Bedingungsausdrücken -- **DONE**
2. Entwurf einer Komponente, mit der sich Variablen auszeichnen lassen, welche bei Zuweisungen automatisch registrierte Lambda-Ausdrücke informieren -- **DONE**
3. Integration dieser Komponententen in ODEMX:
  - generische `wait()` -Methoden für Prozesse -- **DONE**
  - generische Gewichtungsfunktionen für `waitq` -- **DONE**
  - generische Definition von [Stop-Bedingungen] von Differentialgleichungen von kontinuierlichen Prozessen -- **DONE**
4. Beispielsimulator
5. Dokumentation

# Beispielsimulator: *Ping Pong*

- Zwei Prozesse:
  - Ping
    - pingt Pong *ab und zu* an
  - Pong
    - wartet auf eingehende Signale von Ping und gibt diese auf der Konsole aus

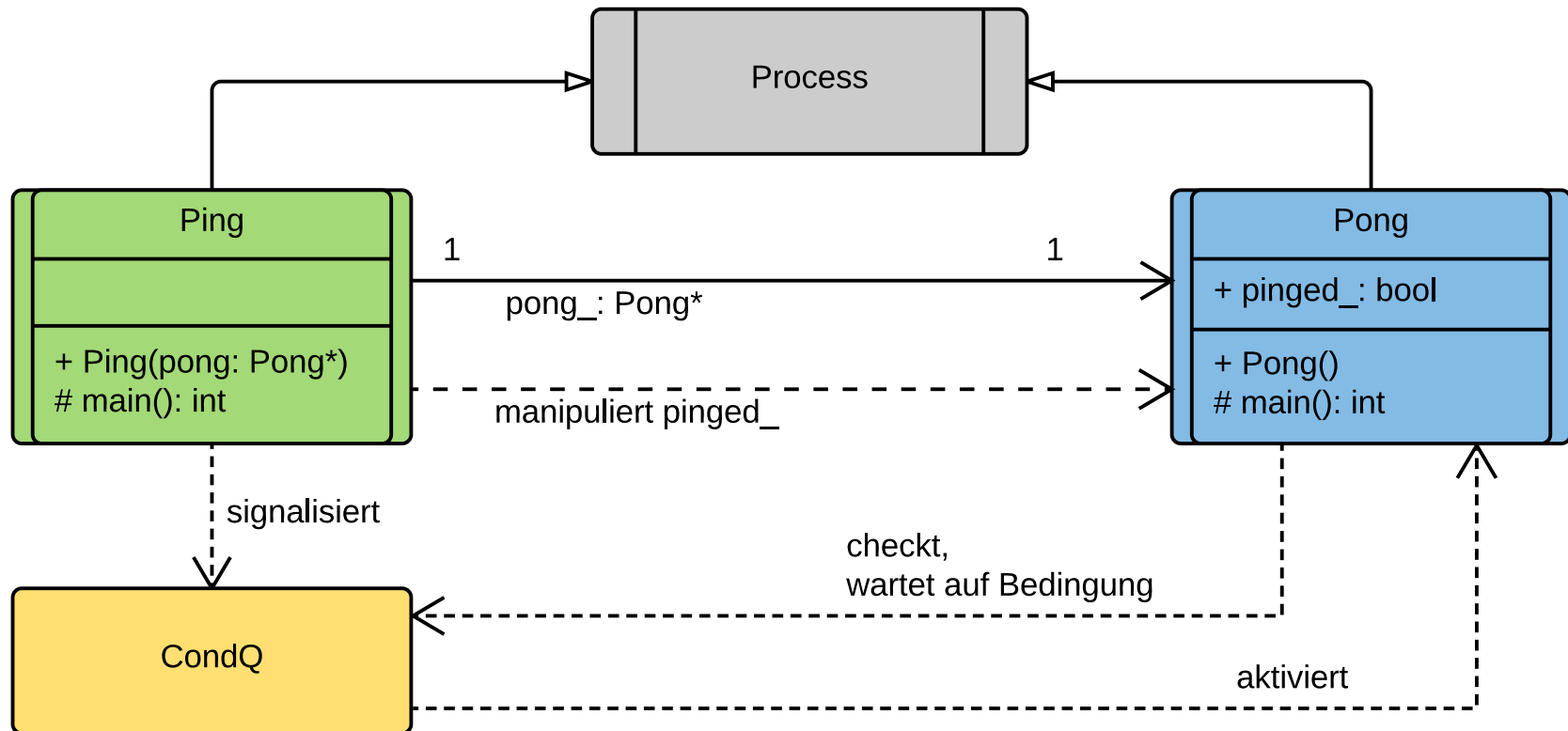
## Ping Pong - Klassendiagramm



# CondQ

"CondQ can be used to make a process wait for an arbitrary condition." - *Ralf Garstenberger, 2002*

## Ping Pong, CondQ - Klassendiagramm



# CondQ

## Public Member Functions

---

**CondQ** (**base::Simulation** &sim, const **data::Label** &label, **CondQObserver** \*obs=0)

Construction for user-defined Simulation. [More...](#)

**~CondQ** ()

Destruction. [More...](#)

const **base::ProcessList** & **getWaitingProcesses** () const

Get a list of blocked process objects. [More...](#)

### Synchronisation

bool **wait** (**base::Condition** cond)

**Wait** for cond. [More...](#)

void **signal** ()

Trigger condition check. [More...](#)



# CondQ

- `CondQ::wait(base::Condition cond)`
  - "Wait for `cond` . If the supplied condition `cond` is `false` the current process is blocked until the condition is `true` and `signal()` was called. [...]"
- `CondQ::signal()`
  - "Trigger condition check."

# Condition

```
typedef bool( Process::* Condition) ()
```

Member-pointer type for coding conditions.

This pointer to member-function type is used for coding conditions. The condition-function should return true if the condition is fulfilled.

Definition at line **66** of file **TypeDefs.h**.

# Entwurf

- Prozesse `Ping` und `Pong`
- Standardsimulation
- globale `CondQ` , an der `Pong` warten kann
- Bedingungsfunktion `Pong::isPinged()`

## Ping - Entwurf

```
struct Ping: Process {
    Pong* pong_;

    Ping(Pong* pong) :
        Process(odemx::getDefaultSimulation(), "Ping"),
        pong_{pong} {}

protected:

    virtual int main() override {
        for(int interval = 1;; ++interval) {
            holdFor(interval);
            pong_->pinged_ = true;
            condQ->signal();
        }
        return 0;
    }
}
```

## Pong - Entwurf

```
struct Pong: Process {  
    bool pinged_  
  
    Pong() :  
        Process(odemx::getDefaultSimulation(), "Pong"),  
        pinged_{false} {}  
  
protected:  
    /* ... */  
};
```

# Pong - Entwurf

```
struct Pong: Process {
    /* ... */
protected:

    virtual int main() override {
        int times_pinged = 0;
        for(;;) {
            condQ.wait((Condition) &Pong::isPinged);
            ++times_pinged;
            std::cout
                << "#" << times_pinged
                << ": \tgot pinged at time "
                << getTime() << std::endl;
            pinged_ = false;
        }
        return 0;
    }

    bool isPinged() {
        return pinged_;
    }
};
```

## Hauptroutine - Entwurf

```
odemx::synchronization::CondQ* condQ;  
/* ... */  
int main() {  
    Pong pong;  
    Ping ping{&pong};  
    condQ = new odemx::synchronization::CondQ  
    (  
        odemx::getDefaultSimulation(),  
        "waiting for a ping"  
    );  
    ping.hold();  
    pong.hold();  
    odemx::getDefaultSimulation().runUntil(101);  
    delete condQ;  
    return 0;  
}
```

# Ausgabe

```
#1:      got pinged at time 1
#2:      got pinged at time 3
#3:      got pinged at time 6
#4:      got pinged at time 10
#5:      got pinged at time 15
#6:      got pinged at time 21
#7:      got pinged at time 28
#8:      got pinged at time 36
#9:      got pinged at time 45
#10:     got pinged at time 55
#11:     got pinged at time 66
#12:     got pinged at time 78
#13:     got pinged at time 91
```



# Problem: Usability von CondQ

- *ex-situ* Definition von Bedingung `Pong::isPinged()`
- Typecast von `isPinged` nach `Condition`
  - `bool(Pong::*)()` nach `bool(Process::*)()`
- manueller Aufruf von `condQ->signal()` immer nach *potentieller Änderung*
  - **einer der beteiligten Variablen einer** Bedingung

## Pong - Entwurf (1)

```
struct Pong: Process {  
    bool pinged_  
  
    Pong() :  
        Process(odemx::getDefaultSimulation(), "Pong"),  
        pinged_{false} {}  
  
protected:  
    /* ... */  
};
```

# Pong - Entwurf (1)

```
struct Pong: Process {
    /* ... */
protected:

    virtual int main() override {
        int times_pinged = 0;
        for(;;) {
            condQ.wait((Condition) &Pong::isPinged);
            ++times_pinged;
            std::cout
                << "#" << times_pinged
                << ": \tgot pinged at time "
                << getTime() << std::endl;
            pinged_ = false;
        }
        return 0;
    }

    bool isPinged() {
        return pinged_;
    }
};
```

## Pong - Entwurf (2)

```
struct Pong: Process {
    /* ... */
protected:
    virtual int main() override {
        int times_pinged = 0;
        for(;;) {
            condQ.wait([](Process* p){
                return ((Pong*)p)->isPinged();
            });
            ++times_pinged;
            std::cout
                << "#" << times_pinged
                << ": \tgot pinged at time "
                << getTime() << std::endl;
            pinged_ = false;
        }
        return 0;
    }

    bool isPinged() {
        return pinged_;
    }
};
```

## Pong - Entwurf (3)

```
struct Pong: Process {
    /* ... */
protected:

    virtual int main() override {
        int times_pinged = 0;
        for(;;) {
            condQ.wait([&](Process*){
                return pinged_;
            });
            ++times_pinged;
            std::cout
                << "#" << times_pinged
                << ": \tgot pinged at time "
                << getTime() << std::endl;
            pinged_ = false;
        }
        return 0;
    }
};
```

## Pong - Entwurf (4, final)

```
struct Pong: Process {  
    Control<bool> pinged_  
  
    Pong() :  
        Process(odemx::getDefaultSimulation(), "Pong"),  
        pinged_{false} {}  
  
protected:  
    /* ... */  
};
```

## Pong - Entwurf (4, final)

```
struct Pong: Process {
    /* ... */
protected:

    virtual int main() override {
        int times_pinged = 0;
        for(;;) {
            waitUntil([&](Process*){
                return pinged_;
            }, "waiting for a ping", {pinged_});
            ++times_pinged;
            std::cout
                << "#" << times_pinged
                << ": \tgot pinged at time "
                << getTime() << std::endl;
            pinged_ = false;
        }
        return 0;
    }
};
```

## Ping - Entwurf (1)

```
struct Ping: Process {
    Pong* pong_;

    Ping(Pong* pong) :
        Process(odemx::getDefaultSimulation(), "Ping"),
        pong_{pong} {}

protected:

    virtual int main() override {
        for(int interval = 1;; ++interval) {
            holdFor(interval);
            pong_->pinged_ = true;
            condQ->signal();
        }
        return 0;
    }
}
```



## Ping - Entwurf (2)

```
struct Ping: Process {
    Pong* pong_;

    Ping(Pong* pong) :
        Process(odemx::getDefaultSimulation(), "Ping"),
        pong_{pong} {}

protected:

    virtual int main() override {
        for(int interval = 1;; ++interval) {
            holdFor(interval);
            pong_->pinged_ = true;
            // condQ->signal();
        }
        return 0;
    }
}
```

## Ping - Entwurf (2, final)

```
struct Ping: Process {
    Pong* pong_;

    Ping(Pong* pong) :
        Process(odemx::getDefaultSimulation(), "Ping"),
        pong_{pong} {}

protected:

    virtual int main() override {
        for(int interval = 1;; ++interval) {
            holdFor(interval);
            pong_->pinged_ = true;
        }
        return 0;
    }
}
```

## Hauptroutine - Entwurf (1)

```
odemx::synchronization::CondQ* condQ;  
/* ... */  
int main() {  
    Pong pong;  
    Ping ping{&pong};  
    condQ = new odemx::synchronization::CondQ  
    (  
        odemx::getDefaultSimulation(),  
        "waiting for a ping"  
    );  
    ping.hold();  
    pong.hold();  
    odemx::getDefaultSimulation().runUntil(101);  
    delete condQ;  
    return 0;  
}
```

## Hauptroutine - Entwurf (2)

```
odemx::synchronization::CondQ* condQ;  
/* ... */  
int main() {  
    Pong pong;  
    Ping ping{&pong};  
    /*condQ = new odemx::synchronization::CondQ  
    (  
        odemx::getDefaultSimulation(),  
        "waiting for a ping"  
    );*/  
    ping.hold();  
    pong.hold();  
    odemx::getDefaultSimulation().runUntil(101);  
    // delete condQ;  
    return 0;  
}
```

## Hauptroutine - Entwurf (2, final)

```
odemx::synchronization::CondQ* condQ;  
/* ... */  
int main() {  
    Pong pong;  
    Ping ping{&pong};  
    ping.hold();  
    pong.hold();  
    odemx::getDefaultSimulation().runUntil(101);  
    return 0;  
}
```

# Ausgabe

```
#1:      got pinged at time 1
#2:      got pinged at time 3
#3:      got pinged at time 6
#4:      got pinged at time 10
#5:      got pinged at time 15
#6:      got pinged at time 21
#7:      got pinged at time 28
#8:      got pinged at time 36
#9:      got pinged at time 45
#10:     got pinged at time 55
#11:     got pinged at time 66
#12:     got pinged at time 78
#13:     got pinged at time 91
```

# Problem: Usability von `CondQ` [gelöst!]

- *ex-situ* Definition von Bedingung `Pong::isPinged()`
  - Bedingungsdefinitionen jetzt *in-situ*
- Typecasts von Bedingungsfunktionen
  - eliminiert durch neue Typdefinition `Condition` mittels `std::function`
- manueller Aufruf von `condQ->signal()` immer nach *potentieller Änderung*
  - Alarmierungen geschehen jetzt automatisch bei Kontrollvariablenänderung
  - `CondQ` durch neuen Kontrollmechanismus überholt

# Trace & Observation

- Kontrollvariablen erben nicht von `Producer` , um Overhead gering zu halten
- Trace, Logging & Observation wartender Prozesse wird von `waitUntil` im Scope `Process` gesteuert



# Optimierung / Details

- keine Registrierung eines Prozesses bei sofort erfüllter Bedingung
- Zuweisungsoperatoren prüfen Argumente auf tatsächliche Änderung der Variablen
- Prozesse werden von `ControlBase::signal()` mit `hold()` , also mit FIFO-Semantik aufgeweckt
- Beispiel & Dokumentation ist Teil von **ODEMx 3.1**