SIEMENS Gamesa
RENEWABLE ENERGY

Variable monitoring with declarative interfaces

About me

Background

- Electronics engineering / process control
- Embedded C
- 10+ years working with C++

Siemens Gamesa Renewable Energy

What do we do?

Wind turbines

Backgrounds

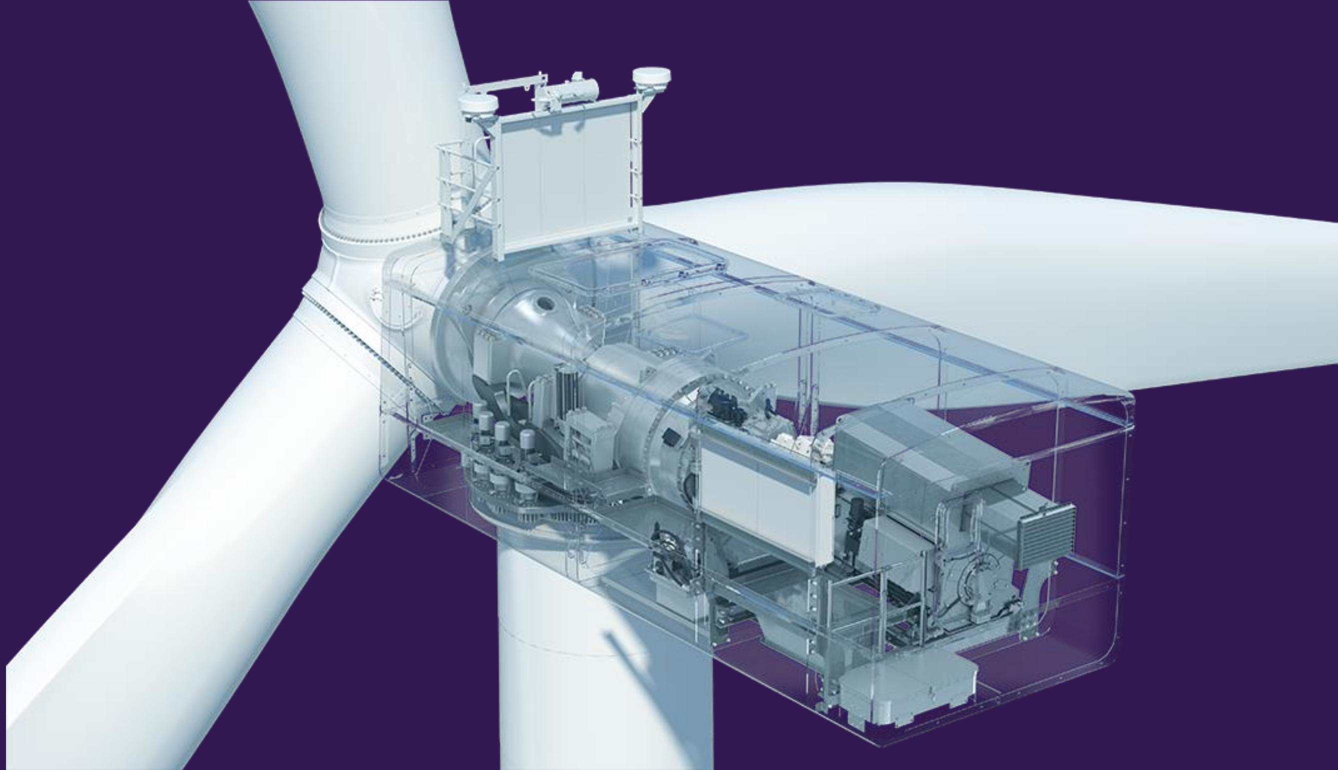
We have a large range of backgrounds

- Math / physics
- Process control
- Mechanics
- Hardware
- …
- SW

Not everyone is equally interested in the SW part

- For some, SW just is the means to arrive at a solution
- Poses challenges in itself (which can be interesting and rewarding)

What goes on in the software in a wind turbine



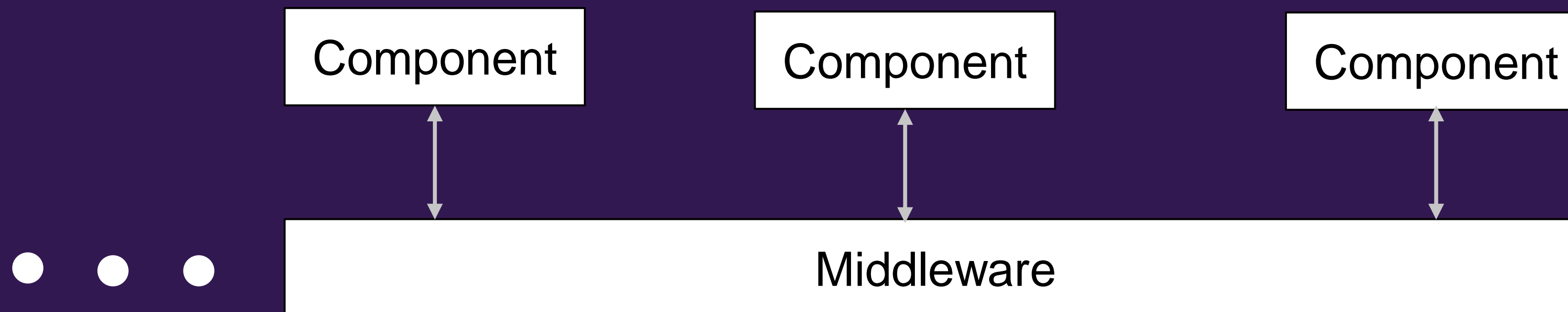
Facts (typical 7 MW turbine):

• Sensors (inputs):	~450
• Actuators (outputs):	~150
• Code size:	~2.5MLOC

Our software platform

A pub-sub middleware

- HW access
- State machines
- Algorithms for control
- Monitoring



Monitoring

- Measurement
- Derived value
- Manual input

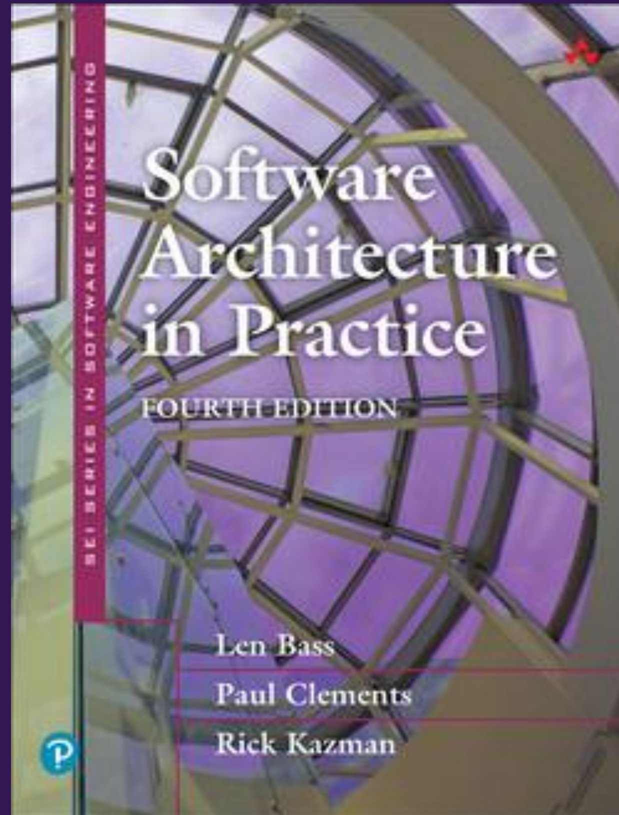
- In range?
- Status good?

- Control actuator
- Set an alarm
- Stop the turbine

Problem statement

”How do we give our developers some good tools to create monitors?”

Quality attributes



- Scalability
- Usability
- Modifyability

Let’s get started

Ad-hoc approach

```
Subscribe(mSignal).SetHandler([this]
{
  if (InRange(mSignal.Value(), 0.0, 1.0))
    DisableAlarm(mAlarm);
  else
    SetAlarm(mAlarm);
});
```

- Subscribe() is a function that subscribes to a signal
- The handler will be called whenever the signal changes

Ad-hoc approach

```
Subscribe(mSignal).SetHandler([this]
{
  if (!InRange(mSignal.Value(), 0.0, 1.0))
  {
    mTimer.Start(10s).SetNotify([this]{ SetAlarm(mAlarm); });
  }
  else
  {
    mTimer.Stop();
    DisableAlarm(mAlarm);
  }
});
```

- Gets complicated quickly
- Easy to make mistakes (DRY)

Quality attributes

	Scalability	Usability	Modifyability
Ad-hoc			✓

Monitor classes

- We can make some classes that contains the business logic

```
struct Component
{
    ...
  private:
    MonitorSignalQuality mWaterPressureMonitor;
    MonitorSignalQuality mOilPressureMonitor;
    MonitorLimits        mWindSpeedMonitor;
}
```

```
Component() :
  mWaterPressureMonitor(o, WaterPressA(), CSignal::BAD, INSTANT_ALARM, PressAlarm()),
  mOilPressureMonitor(o, OilPressA(), CSignal::BAD, INSTANT_ALARM, PressAlarm()),
  mWindSpeedMonitor(o, WindSpeed(), HIGHER_THAN, 10.0, WindSpeedHigh()) {}
```

- Declarative interface

Monitor classes

- There is only one problem: The passage of time

- People wanted some new functionality.

No problem, we just add it to this monitoring class…

It doesn't quite fit the bill = new monitoring class

CMonitor.cpp
CMonitor.h
CMonitorAlarmActivated.cpp
CMonitorAlarmActivated.h
CMonitorAlarmRepeatedlyActivated.cpp
CMonitorAlarmRepeatedlyActivated.h
CMonitorAlarmStaysActivated.cpp
CMonitorAlarmStaysActivated.h
CMonitorContainer.cpp
CMonitorContainer.h
CMonitorContainerInterface.h
CMonitorDelta.cpp
CMonitorDelta.h
CMonitorDifference.cpp
CMonitorDifference.h
CMonitorInterface.h
CMonitorLimit.cpp
CMonitorLimit.h
CMonitorLimitBase.cpp
CMonitorLimitBase.h
CMonitorLimitDelayedAlarm.h
CMonitorLimitDelayedAlarmWithDeactivationDela
CMonitorMultipleValues.cpp
CMonitorMultipleValues.h
CMonitorOscillations.cpp
CMonitorOscillations.h
CMonitorOscillationsPercent.h
CMonitorPercentageLimit.h
CMonitorSignalQuality.cpp
CMonitorSignalQuality.h

- We now have 18 different monitoring classes
- Some of them, a complete rewrite

Quality attributes

	Scalability	Usability	Modifyability
Ad-hoc			✓
Utility classes	✓	✓	

Let’s try a different approach

Adhere to SOLID principles:

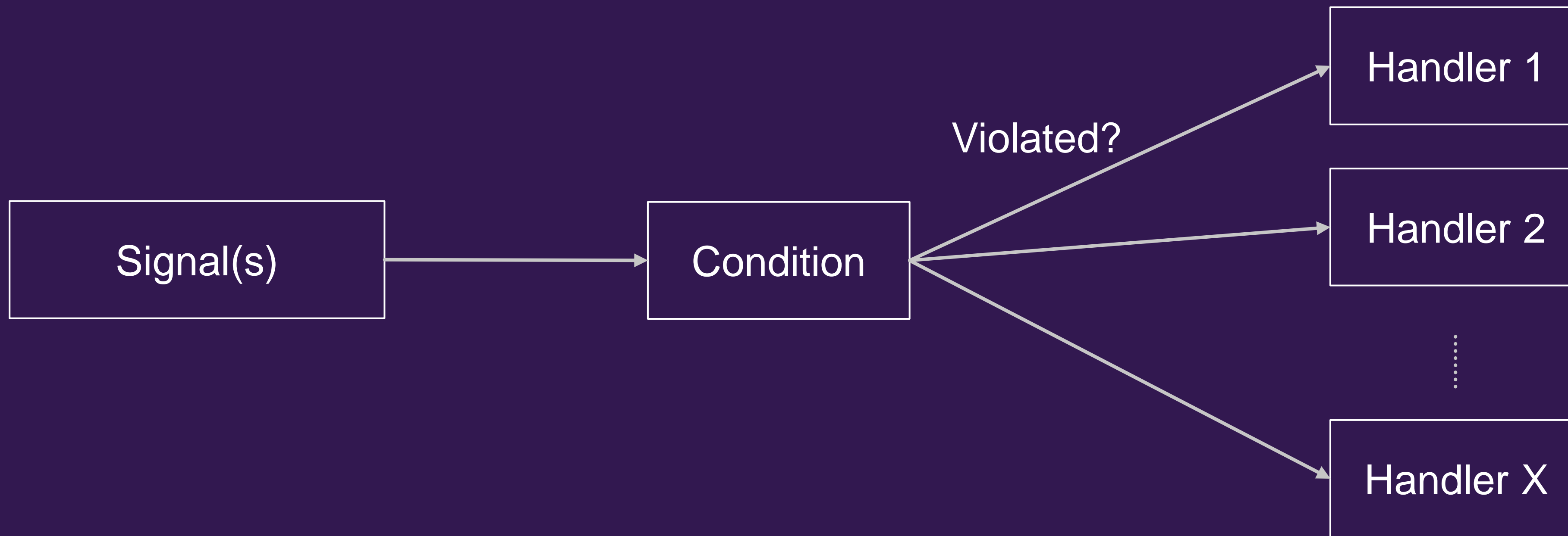
- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Seggregation
- Dependency Inversion

Let’s try a different approach

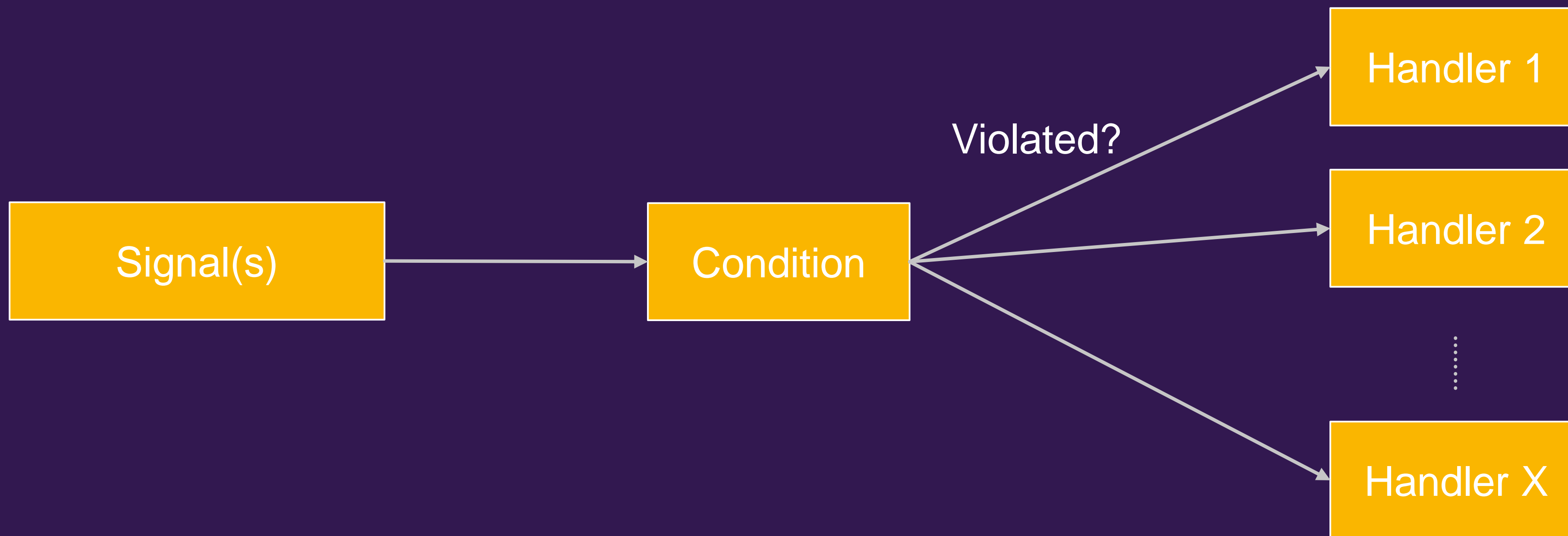
Adhere to SOLID principles:

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Seggregation
- Dependency Inversion

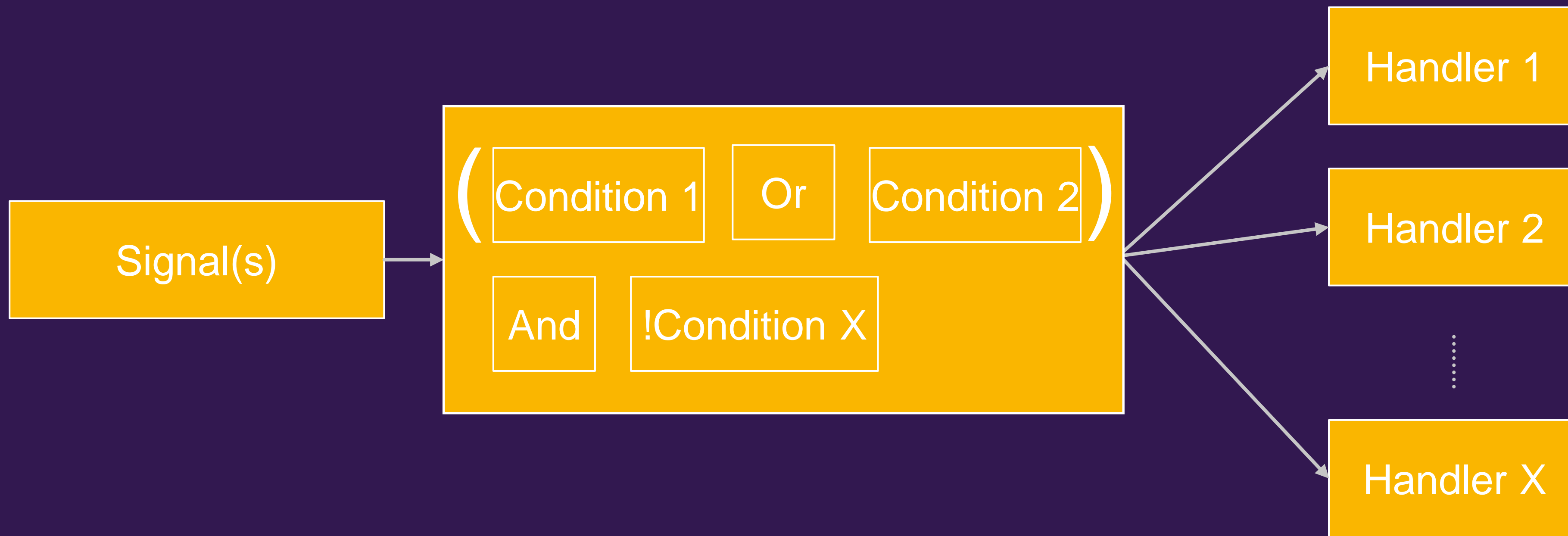
Lets break it down a bit



We would like to be able to customize all of this



We would like to be able to customize all of this



The following monitor graph:

Expressed in code:

```
auto monitor = Monitor<MySignal>(o)
                .Condition(LargerThan(2) && SmallerThan(10))
                .Handler(PrintViolation());
```

```
Publish(MySignal(1)); // Prints “Violated: true”
Publish(MySignal(2)); // Prints “Violated: false”
Publish(MySignal(5)); // Prints “Violated: true”
```

We can get creative with the naming:

```
auto monitor = Monitor<MySignal>(o)
                 .MustBe(LargerThan(2) && SmallerThan(10))
                 .WhenNot(PrintViolation());
```

Reads:

”MySignal must be larger than 2 and must be smaller than 10. When not, print the violation.”

How can we do this?

```
template<typename Conditions, typename Handlers, typename... Types>
struct MonitorBuilder
{
  template<typename ConditionT>
  auto Condition(ConditionT condition);

  template<typename HandlerT>
  auto Handler(HandlerT handler);

  void operator()(const Types&... ts);

  Conditions mConditions; // Tuple of conditions
  Handlers   mHandlers;   // Tuple of handlers
};
```

- Condition returns a new MonitorBuilder that has added the condition
- Handler returns a new MonitorBuilder that has added the new handler to its handlers
- operator() checks a set of types “ts” for the condition(s) and calls the handlers

Then we create a Monitor() function that creates a default monitor builder

```
class MonitorBuilder
{
  auto Condition (ConditionT condition);
  auto Handler   (HandlerT handler);
  void operator()(const Ts&... ts);
};
```

```
template<typename... Types>
auto Monitor()
{
  return MonitorBuilder<std::tuple<>, std::tuple<>, Types...>({},{});
}
```

We can now build our monitor like this:

```
auto monitor = Monitor<MySignal>()
                 .Condition(LargerThan(2))
                 .Condition(SmallerThan(10))
                 .Handler(PrintViolation());
```

Then we create a Monitor() function that creates a default monitor builder

```
class MonitorBuilder
{
  auto Condition (ConditionT condition);
  auto Handler   (HandlerT handler);
  void operator()(const Ts&... ts);
};
```

```
template<typename... Types>
auto Monitor()
{
  return MonitorBuilder<std::tuple<>, std::tuple<>, Types...>({},{});
}
```

We can now build our monitor like this:

```
auto monitor = Monitor<MySignal>()
                 .Condition(LargerThan(2) && SmallerThan(10) && MyOddCondition())
                 .Handler(ImmediateAlarmActivation(mAlarm))
                 .Handler(DelayedAlarmDeactivation(mAlarm, 10s));
```

We also need to implement the logic dealing with subscribing and calling the monitor.

```
auto monitor = Monitor<MySignal>(o)
                .MustBe(LargerThan(2) && SmallerThan(10))
                .WhenNot(PrintViolation());
```

So the MonitorBuilder would look like this

```
class MonitorBuilder
{
  MonitorBuilder (PubSubT& pubsub);
  auto Condition (ConditionT condition);
  auto Handler   (HandlerT handler);
};
```

How to write new conditions:

Just a callable (returns true if condition violated)

```
bool Larger(const int a, const int b)
{
    return a <= b;
}
```

With parameters

```
template<typename T>
auto Minimum(const T& minimum)
{
    return [minimum](const T& val) { return val < minimum; };
}
```

Generic version

```
auto Even()
{
  return [](const auto& val) { return val % 2 != 0; };
}
```

You can go crazy with conditions, without having to influence others code:

```
template<int WindowLength, typename T, bool initValue = false>
auto MaxVarianceWindow(const T& maxVariance)
{
    return [var, window = std::array<T, WindowLength>(), windowIndex = 0, initialized = false](const T& val) mutable
    {
        window[windowIndex] = val;
        windowIndex = (windowIndex + 1) % WindowLength;
        initialized = initialized or windowIndex == 0;
        if (initialized)
        {
            const auto mean = std::accumulate(window.cbegin(), window.cend(), 0) / WindowLength;
            const auto calcVar = std::accumulate(window.cbegin(), window.cend(), 0,
                                        [mean](int rhs, int lhs) { return rhs + std::pow(lhs - mean, 2); }) / (WindowLength - 1);
            return calcVar > maxVariance;
        }
        else
        {
            return initValue;
        }
    };
}
```

```
auto monitor = Monitor<int>()
                .Condition(MaxVarianceWindow<20, int>(10))
                .Handler(ActivateAlarmImmediately(mAlarm))
```

Now, we do need to do something to be able to do boolean logic on conditions

```
auto condition = Condition1 && (Condition2 || !Condition3)
```

A condition is a callable, taking some parameters and returning a boolean

```
bool Larger(const int a, const int b)
{
    return a <= b;
}
```

If we have a collection of callables, all returning bool, and taking the same parameters, would it not make sense to be able to use boolean logic on them?

So let's try to overload && and ||, even though everyone warns about it.

First, we wrap our callable in a class:

```cpp
template<typename Func>
struct LogicalF
{
  LogicalF(Func func) : mFunc(std::move(func))
  {
  }

  template<typename... Args>
  auto operator()(const Args&... args)
  {
    return mFunc(args...);
  }

  private:
    Func mFunc;
};
```

Then we override the &&, || and ! operators for that class

```
struct LogicalF
{
  template<typename OtherT>
  auto operator&&(const OtherT& other)
  {
    auto lamb = [thisFunc = this->mFunc, otherFunc = other](const auto&... args)
    {
      const auto res1 = thisFunc(args...);
      const auto res2 = other(args...);
      return res1 && res2;
    };
    return LogicalF<decltype(lamb)>(lamb);
  }
};
```

Now our conditions just need to be wrapped

Before:

```
auto Even()
{
  return [](const auto& val) { return val % 2 != 0; };
}
```

After:

```
auto Even()
{
  return LogicalF([](const auto& value) { return value % 2 != 0; });
}
```

How to write new handlers:

Simple handler

```
void PrintStatus(const bool violated)
{
    if (!violated)
        std::cout << "Monitor passed\n";
    else
        std::cout << "Monitor didn't pass\n";
}
```

Handler with state

```
void ImmediateAlarmActivation(CAlarmInterface& alarm)
{
    return [&alarm](const bool violated)
    {
        if (violated)
        {
            alarm.Activate();
        }
    };
}
```

We can instantiate the members of our component as before:

```
Component() :
  mWPM(Monitor<WaterPressA>(o).Condition(StateGood()).Handler(ImmediateAlarmActivation(PressAlarm())),
  mOPM(Monitor<OilPressA>(o).Condition(StateGood()).Handler(ImmediateAlarmActivation(PressAlarm())),
  mWSM(Monitor<WindSpeed>(o).Condition(LessThan(10.0)).Handler(SetSignal<WindSpeedHigh>()
{}
```

Previous approach:

```
Component() :
  mWaterPressureMonitor(o, WaterPressA(), CSignal::BAD, INSTANT_ALARM, PressAlarm()),
  mOilPressureMonitor(o, OilPressA(), CSignal::BAD, INSTANT_ALARM, PressAlarm()),
  mWindSpeedMonitor(o, WindSpeed(), HIGHER_THAN, 10.0, WindSpeedHigh()) {}
```

Quality attributes

	Scalability	Usability	Modifyability
Ad-hoc			✓
Utility classes	✓	✓	
Builder	✓	✓	✓

Implementation

In 3 slides

Implementation

MonitorBuilder constructor

```
template<typename Conditions, typename Handlers, typename... Types>
struct MonitorBuilder
{
  MonitorBuilder(Conditions conditions, Handlers handlers) :
    mConditions(conditions), mHandlers(handlers) {}

  Conditions mConditions;
  Handlers mHandlers;
};
```

Implementation

Builder methods:

```
template<typename ConditionT>
auto Condition(ConditionT condition)
{
  auto newConditions = std::tuple_cat(mConditions, std::make_tuple(condition));
  return MonitorBuilder<decltype(newConditions), Handlers, Types...>(newConditions, mHandlers);
}
```

```
template<typename HandlerT>
auto Handler(HandlerT handler)
{
  auto newHandlers = std::tuple_cat(mHandlers, std::make_tuple(handler));
  return MonitorBuilder<Conditions, decltype(newHandlers), Types...>(mConditions, newHandlers);
}
```

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated = intern::any_true(results);
  intern::call_functions(mHandlers, violated);
}
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated =
  intern::call_functior
}
```

```
    results<0> = mConditions<0>(args...);
    results<1> = mConditions<1>(args...)
       ...
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated = intern::any_true(results);
  intern::call_functions(mHandlers, violated);
}
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.
- any_true, returns true if any elements in a tuple evaluates to true.

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated = intern::any_true(results);
  intern::call_functions(mHandlers, violated);
}
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.
- any_true, returns true if any elements in a tuple evaluates to true.
- call_functions calls all callables in a tuple with the given parameter.

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated = intern::any_true(results);
  intern::call_functions(mHandlers, violated);
}
```

```
    mHandlers<0>(violated);
    mHandlers<1>(violated)
       ...
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.
- any_true, returns true if any elements in a tuple evaluates to true.
- call_functions calls all callables in a tuple with the given parameter.

Implementation

Checker method:

```
void operator()(const Types&... args)
{
  const auto results = intern::apply_functions(mConditions, args...);
  const auto violated = intern::any_true(results);
  intern::call_functions(mHandlers, violated);
}
```

- apply_functions calls all callables in a tuple with the given parameters and returns a tuple of results.
- any_true, returns true if any elements in a tuple evaluates to true.
- call_functions calls all callables in a tuple with the given parameter.

Implementation

Note, that I have not included any pub/sub framework specific code here.

This is because every pub/sub framework is different, so we cannot do it in a generic way.

In essence, it boils down to subscribing to the signals, and calling operator() when they change.

But we want to make the conditions depend on some other signal (parameters)

```
auto monitor = Monitor<MySignal>(o)
                .MustBe(LargerThan(MyLowerBoundParam) && SmallerThan(MyHigherBoundParam))
                .WhenNot(PrintViolation());
```

We could make a specialization taking references to parameter types.

```
template<>
auto LargerThan(const MyLowerBoundParamType& minimum)
{
    return [&minimum](const auto&) { return val < minimum.GetValue(); };
}
```

But then the class owning the monitor needs to have an instance of the parameter, which we would like to avoid, if possible.

Also, if the parameter changes during execution, the owner needs to explicitly make the monitor check whether the conditions are violated. So, it needs to subscribe to the parameter, and call the monitors operator() on change.

A possible solution:

Make all conditions take an implicit first parameter, which is a ”handle” to the pub/sub framework:

```
template<typename T>
auto LargerThan(T minimum) {
  return [minimum](auto&, const auto& val) {
    return val < minimum;
  };
}

template<>
auto LargerThan(MyLowerBoundParamType minimum) {
  return [minimum, subscribed = false](auto& pubsub, const auto& val) mutable {
    if (!subscribed) {
      pubsub.Subscribe(minimum);
      subscribed = true;
    }
    return val < minimum.GetValue();
  };
}
```

Should we override && for the handlers?

```
auto monitor = Monitor<int>()
                .Condition((LargerThan(2) && SmallerThan(10)) || MyOddCondition())
                .Handler(ImmediateAlarmActivation(mAlarm) && DelayedAlarmDeactivation(mAlarm, 10s));
```

Maybe this is a case of ”just because you could, doesn’t mean you should”.

Also, handlers are not really ”logical” types

SIEMENS Gamesa
RENEWABLE ENERGY

Thank you. Questions?

https://github.com/nfogh/monitoring