Apex.AI®

Writing sustainable software.
The how and the what!

Meeting C++ 2021
Christian Eltzschig
https://github.com/elfenpiff
https://gitlab.com/el.chris

Motivation

- https://github.com/elfenpiff/meetingcpp_2021

Who am I
- C++ developer who works on iceoryx

What is iceoryx
- Open source inter process communication framework for safety critical systems
- Written in C++
- https://github.com/eclipse-iceoryx/iceoryx
- No heap, no undefined behavior and sadly no exceptions
- iceoryx_hoofs: STL types like optional, vector, string, expected

Why I am here
- iceoryx may run on machines and cars which will be involved in accidents
- I want to make sure that our software is never the cause
- Software certification is necessary but not sufficient to ensure software quality

The idea
- Can one improve the code quality by just asking the right questions?

Motivation

Something which should never happen in a safety critical environment

“When I wrote this, only God and I understood what I was doing. Now, God only knows.”
- maybe Karl Weierstrass

The talk presents ideas and concepts which are important in a safety context.
Some ideas may not apply for other domains - but maybe we can inspire each other.

Motivation

C++ principle

In general, C++ implementations obey the zero-overhead principle:
What you don’t use, you don’t pay for.

- *Bjarne Stroustrup*

C++ in a safety critical environment

Safety first, performance second. This means no undefined behavior, fail fast and the API should avoid fatal errors due to misuse.

The products in which our software runs will - sooner or later - be involved in accidents and we simply have to try that our software will be never the cause. Even if it costs us performance!

Writing software is like painting a picture

Well written software can be easily adapted

- The first lines of code are like a pencil sketch
- While coding you discover new use cases and improvements
- The abilities and structure of the final software product are unknown in the beginning

Why do we plan software projects like an architect who builds a bridge?

Writing software is like painting a picture

Well written software can be easily adapted

- The first lines of code are like a pencil sketch
- While coding you discover new use cases and improvements
- The abilities and structure of the final software product are unknown in the beginning

Why do we plan software projects like an architect who builds a bridge?

Choluteca Bridge, Honduras

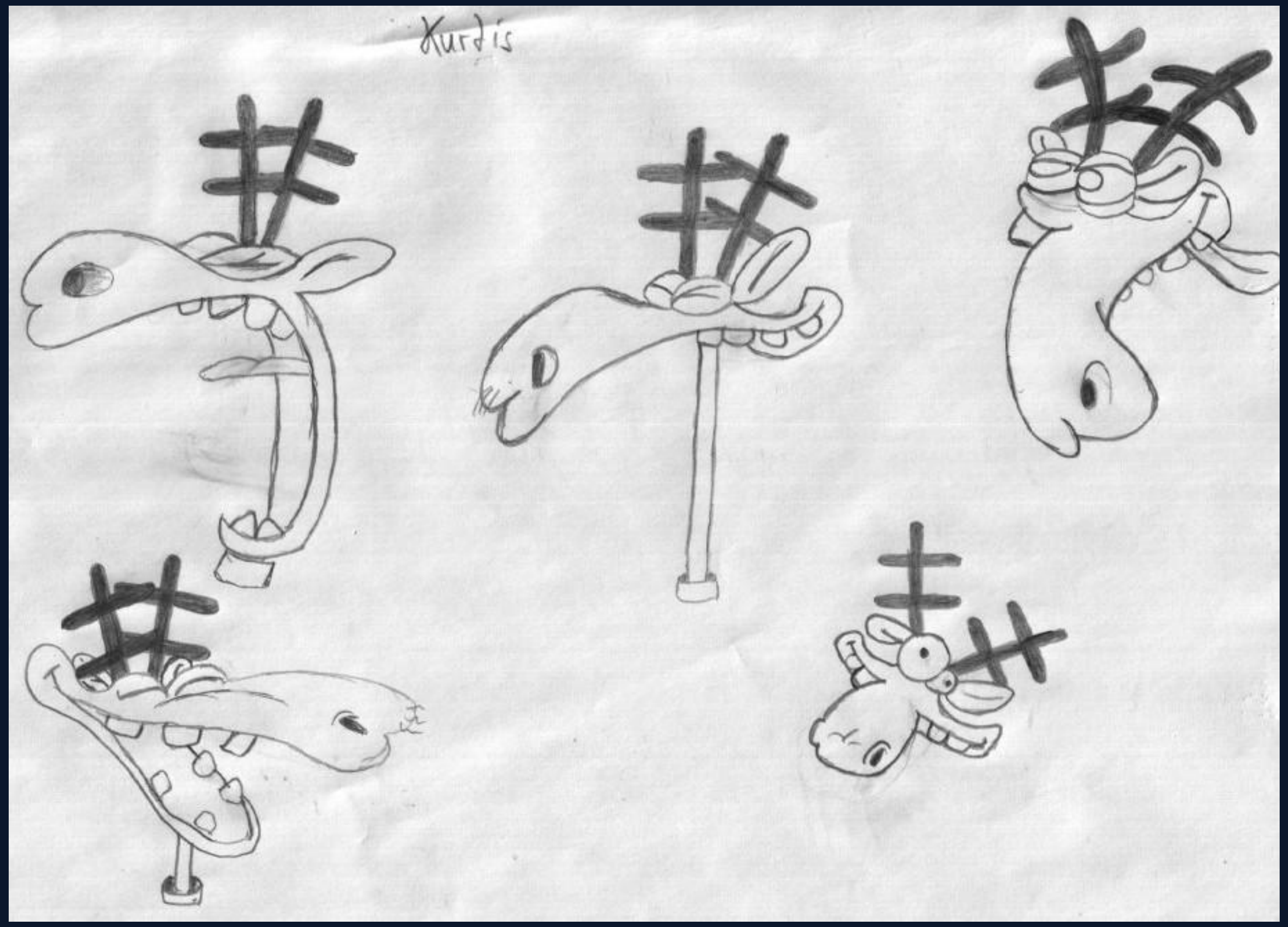
Writing software is like painting a picture

Well written software can be easily adapted

- The first lines of code are like a pencil sketch
- While coding you discover new use cases and improvements
- The abilities and structure of the final software product are unknown in the beginning

Why do we plan software projects like an architect who builds a bridge?

Choluteca Bridge, Honduras



- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is the design over-abstracted?

Design is always incomplete!

A software project and every feature starts with an idea

- It is hard to foresee if it is successful
- Like a living organism it will change and adapt over time
- Having a feature opens your eyes for its possibilities and applications

Beginner mistake

- Creates UML diagram (or design document) where all possible use cases are covered
- Classes are abstract as possible to cover extensibility
- Often leads to circular dependencies and bad code
- A design does not ensure that its described structure is actual implementable

How to avoid over abstraction

- Do small cycles of design and implementing a proof of concept
- Sometimes a design can be described by declaring classes and methods without implementation
- Doxygen can generate class diagrams
- Get the most basic use case working before extending the design

Is the design over-abstracted?

Design is always incomplete!

A software project and every feature starts with an idea
- It is hard to foresee if it is successful
- Like a living organism it will change and adapt over time
- Having a feature opens your eyes for its possibilities and applications

Beginner mistake
- Creates UML diagram (or design document) where all possible use cases are covered
- Classes are abstract as possible to cover extensibility
- Often leads to circular dependencies and bad code
- A design does not ensure that its described structure is actual implementable

How to avoid over abstraction
- Do small cycles of design and implementing a proof of concept
- Sometimes a design can be described by declaring classes and methods without implementation
- Doxygen can generate class diagrams
- Get the most basic use case working before extending the design

Is the design over-abstracted?

Design is always incomplete!

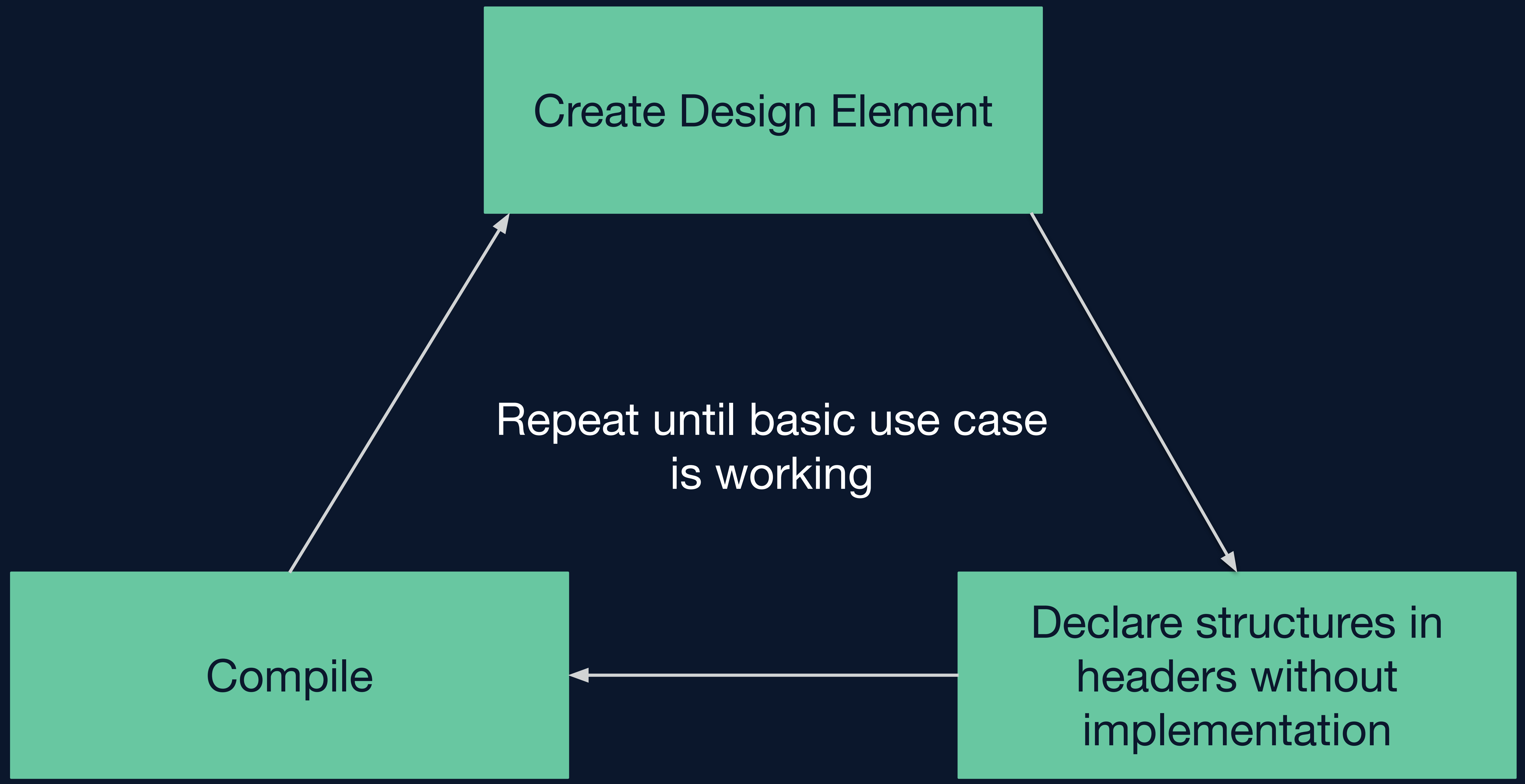
A software project and every feature starts with an idea
- It is hard to foresee if it is successful
- Like a living organism it will change and adapt over time
- Having a feature opens your eyes for its possibilities and applications

Beginner mistake
- Creates UML diagram (or design document) where all possible use cases are covered
- Classes are abstract as possible to cover extensibility
- Often leads to circular dependencies and bad code
- A design does not ensure that its described structure is actual implementable

How to avoid over abstraction
- Do small cycles of design and implementing a proof of concept
- Sometimes a design can be described by declaring classes and methods without implementation
- Doxygen can generate class diagrams
- Get the most basic use case working before extending the design

Is the design over-abstracted?

Design is always incomplete!



Jester - the professional fool

- In the middle ages the professional fool did not only entertain the nobles
- They read laws and explained their understanding to the lawmakers
  - If the professional fool understood the law as intended the law would be passed
  - If not the law was rewritten

Can we adapt the idea for a newly developed API?

- Provide the public API to a developer which is unfamiliar with all the details
- The developer does not get any documentation.
  “The compiler doesn't read comments and neither do I” - Bjarne Stroustrup
- If they would use the API intuitively right we can start roll it out

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is this API easy to use and hard to misuse?

This examples demonstrates a bad API.

```
class Receiver {
  public:
    Receiver();
    void init();
    void addConnection(ip_t ipAddress, port_t port);
    void connect();
    void disconnect();
```

The contract

- init has to be called first
- After init one can add any number of connections with addConnection
- After connect was called one is not allowed to add any more connections
- One can call disconnect only after connect
- init can be called only once

Is this API easy to use and hard to misuse?

This examples demonstrates a bad API.

```
class Receiver {
  public:
    Receiver();
    void init();
    void addConnection(ip_t ipAddress, port_t port);
    void connect();
    void disconnect();
```

The contract

- init has to be called first
- After init one can add any number of connections with addConnection
- After connect was called one is not allowed to add any more connections
- One can call disconnect only after connect
- init can be called only once

Is this API easy to use and hard to misuse?

This examples demonstrates a bad API.

```
class Receiver {
  public:
    Receiver();
    void init();
    void addConnection(ip_t ipAddress, port_t port);
    void connect();
    void disconnect();
```

```
  myReceiver.connect();       // <- violates contract, have to call init first

  myReceiver.addConnection(); // <- violates contract, no add after connect
                              //   * can cause bug reports: unable to add connection
  myReceiver.disconnect();

  myReceiver.init();          // <- violates contract, no init twice
  myReceiver.addConnection();
```

Is this API easy to use and hard to misuse?

This examples demonstrates a bad API.

```
class Receiver {
  public:
    Receiver();
    void init();
    void addConnection(ip_t ipAddress, port_t port);
    void connect();
    void disconnect();
```

- This object seems to have 3 states
  1. Constructed, before init was called
  2. Initialized, after init but before connect or after disconnect
  3. Connected

- In Rust we could use the typestate pattern
  - Short: Encode the objects runtime state in its compile time interface

Is this API easy to use and hard to misuse?

```
struct ReceiverState {};

class ReceiverConnected {
  public:
    ReceiverConnected(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized disconnect();
};

class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port);
    ReceiverConnected connect();
};

class Receiver {
  public:
    ReceiverInitialized init();
};
```

- All classes have a std::unique_ptr<ReceiverState> as member
- When a new state is created the ownership is moved to the next state

Is this API easy to use and hard to misuse?

```
struct ReceiverState {};

class ReceiverConnected {
  public:
    ReceiverConnected(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized disconnect();
};

class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port);
    ReceiverConnected connect();
};

class Receiver {
  public:
    ReceiverInitialized init();
};
```

- All classes have a std::unique_ptr<ReceiverState> as member
- When a new state is created the ownership is moved to the next state

Is this API easy to use and hard to misuse?

```
struct ReceiverState {};

class ReceiverConnected {
  public:
    ReceiverConnected(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized disconnect();
};

class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port);
    ReceiverConnected connect();
};

class Receiver {
  public:
    ReceiverInitialized init();
};
```

- All classes have a std::unique_ptr<ReceiverState> as member
- When a new state is created the ownership is moved to the next state

Is this API easy to use and hard to misuse?

```
struct ReceiverState {};

class ReceiverConnected {
  public:
    ReceiverConnected(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized disconnect();
};

class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port);
    ReceiverConnected connect();
};

class Receiver {
  public:
    ReceiverInitialized init();
};
```

- All classes have a std::unique_ptr<ReceiverState> as member
- When a new state is created the ownership is moved to the next state

Is this API easy to use and hard to misuse?

```
auto receiver = Receiver()
                    .init()
                    .addConnection(..) // <- contract violations will lead to compile
                    .addConnection(..) //    time error
                    .connect();

receiver.disconnect();
```

- This is also known as Design by contract, when the contract is enforced via the interface
  - Your IDE will tell you how to use it correctly via auto completion
- Useful pattern when one function handles all the states consecutively
- Cumbersome when the state transitions are handled in multiple functions

Is this API easy to use and hard to misuse?

```
Receiver receiver;

auto receiverInitialized = receiver.init();

receiverInitialized.addConnection(..);
receiverInitialized.addConnection(..);

auto receiverConnected = receiverInitialized.connect();

receiverConnected.disconnect();
```

- It is hard to violate the API but it is also cumbersome to use
- Furthermore, the ownership of the state is silently transferred which could lead to further mistakes like use after move

```
auto receiverInitialized2 = receiver.init();
```

Is this API easy to use and hard to misuse?

```
Receiver receiver;

auto receiverInitialized = receiver.init();

receiverInitialized.addConnection(..);
receiverInitialized.addConnection(..);

auto receiverConnected = receiverInitialized.connect();

receiverConnected.disconnect();
```

- It is hard to violate the API but it is also cumbersome to use
- Furthermore, the ownership of the state is silently transferred which could lead to further mistakes like use after move

```
auto receiverInitialized2 = receiver.init();
```

Is this API easy to use and hard to misuse?

```
struct ReceiverState {};

class ReceiverConnected {
  public:
    ReceiverConnected(std::unique_ptr<ReceiverState>&& state);
    static ReceiverInitialized disconnect(ReceiverConnected && self);
};

class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port);
    static ReceiverConnected connect(ReceiverInitialized && self);
};

class Receiver {
  public:
    static ReceiverInitialized init(Receiver && self);
};
```

- The ownership transfer is directly visible and static code analysis is able to catch a use after move

Is this API easy to use and hard to misuse?

```
auto receiverInitialized = Receiver::init(std::move(receiver));

receiverInitialized.addConnection(..);
receiverInitialized.addConnection(..);

auto receiverConnected = ReceiverInitialized::connect(std::move(receiverInitialized));
// do some stuff
ReceiverConnected::disconnect(std::move(receiverConnected));
```

- Moving the previous state object into the factory of the next state makes the ownership clear
- Creating a design which enforces the contract can also reduce the error handling

```
// creates a use after move warning
auto receiverInitialized2 = Receiver::init(std::move(receiver));
```

Is this API easy to use and hard to misuse?

```
auto receiverInitialized = Receiver::init(std::move(receiver));

receiverInitialized.addConnection(..);
receiverInitialized.addConnection(..);

auto receiverConnected = ReceiverInitialized::connect(std::move(receiverInitialized));
// do some stuff
ReceiverConnected::disconnect(std::move(receiverConnected));
```

- Moving the previous state object into the factory of the next state makes the ownership clear
- Creating a design which enforces the contract can also reduce the error handling

```
// creates a use after move warning
auto receiverInitialized2 = Receiver::init(std::move(receiver));
```

Is this API easy to use and hard to misuse?

```
auto receiver = Receiver()
                    .init()
                    .addConnection(..)
                    .addConnection(..)
                    .connect();
```

- To ensure that the API is via method chaining one can restrict all methods to rvalues.

```
class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port) &&;
    ReceiverConnected connect() &&;
};
```

- One can still misuse it but it's easy to spot.

```
auto receiverInitialized = Receiver().init();
std::move(receiverInitialized).addConnection();
```

Is this API easy to use and hard to misuse?

```
auto receiver = Receiver()
                    .init()
                    .addConnection(..)
                    .addConnection(..)
                    .connect();
```

- To ensure that the API is via method chaining one can restrict all methods to rvalues.

```
class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port) &&;
    ReceiverConnected connect() &&;
};
```

- One can still misuse it but it’s easy to spot.

```
auto receiverInitialized = Receiver().init();
std::move(receiverInitialized).addConnection();
```

Is this API easy to use and hard to misuse?

```
auto receiver = Receiver()
                    .init()
                    .addConnection(..)
                    .addConnection(..)
                    .connect();
```

- To ensure that the API is via method chaining one can restrict all methods to rvalues.

```
class ReceiverInitialized {
  public:
    ReceiverInitialized(std::unique_ptr<ReceiverState>&& state);
    ReceiverInitialized & addConnection(ip_t ipAddress, port_t port) &&;
    ReceiverConnected connect() &&;
};
```

- One can still misuse it but it’s easy to spot.

```
auto receiverInitialized = Receiver().init();
std::move(receiverInitialized).addConnection();
```

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Can we reduce the error handling?

```
class Receiver {
  public:
    Receiver();
    bool init();
    bool addConnection(ip_t ipAddress, port_t port);
    bool connect();
    bool disconnect();
```

- When we forge a design which can easily be misused we may have to increase the error handling.
- Every method returns a bool to inform the user about the success.
- In an application this can lead to error propagation and to excessive unnecessary error handling

In a safety critical system we have to handle every single error!

Can we reduce the error handling?

```
class Receiver {
  public:

    bool addConnection(ip_t ipAddress, port_t port);

    /// @brief returns true if the connection existed and was removed, otherwise false
    bool removeConnection(ip_t ipAddress, port_t port);
```

```
if ( !myReceiver.removeConnection(..) )
  // one may think remove failed
```

- After removeConnection the connection does not exist anymore, so why returning a bool?
- Error handling can accumulate and multiply and lead to a top level API where thousands of avoidable error handlings clutter the code

Can we reduce the error handling?

```
class Receiver {
  public:

    bool addConnection(ip_t ipAddress, port_t port);

    /// @brief returns true if the connection existed and was removed, otherwise false
    bool removeConnection(ip_t ipAddress, port_t port);


if ( !myReceiver.removeConnection(..) )
  // one may think remove failed
```

- After removeConnection the connection does not exist anymore, so why returning a bool?
- Error handling can accumulate and multiply and lead to a top level API where thousands of avoidable error handlings clutter the code

Can we reduce the error handling?

```
auto receiver = Receiver()
                    .init()
                    .addConnection(..)
                    .addConnection(..)
                    .connect();
```

```
if ( !receiver.init() ) {
  // cleanup code
  return ReceiverError::INIT_FAILED;
}

if ( !receiver.addConnection(..) ) {
  // cleanup code
  return ReceiverError::ADD_CONNECTION_FAILED;
}

if ( !receiver.connect() ) {
  // cleanup code
  return ReceiverError::CONNECT_FAILED;
}
```

- Since the documentation is not read by the compilers and others, most will perform this kind of error handling in a safety critical system
- Increases lines of code and the complexity of unit tests - especially in a safety critical system where we have to verify every possible branch

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is this API easy to use? Concurrent case.

- API for concurrent structures has to be designed with care
  - The wrong interface/functionality can open the door for a wide variety of bugs

```
class ThreadsafeIntegerVector {
  public:
    int & operator[](size_t pos);
    bool empty() const;
}
```

```
if ( !myVector.empty() )
  // the OS can interrupt the thread at this position for some time
  std::cout << myVector[0] << std::endl
```

- Is the method empty useful in a concurrent context?
  - As soon as one acquired the information it could be out of date - opens the door for race conditions.

- A functional API for concurrent constructs can reduce bugs

Is this API easy to use? Concurrent case.

- API for concurrent structures has to be designed with care
  - The wrong interface/functionality can open the door for a wide variety of bugs

```
class ThreadsafeIntegerVector {
  public:
    int & operator[](size_t pos);
    bool empty() const;
}
```

```
if ( !myVector.empty() )
  // the OS can interrupt the thread at this position for some time
  std::cout << myVector[0] << std::endl
```

- Is the method empty useful in a concurrent context?
  - As soon as one acquired the information it could be out of date - opens the door for race conditions.

- A functional API for concurrent constructs can reduce bugs

Is this API easy to use? Concurrent case.

```
class ThreadsafeIntegerVector {
  public:
    //...

    void for(size_t pos, std::function<void(int & value)> action);
    void for_each(std::function<void(int & value)> action);
}

//if ( !myVector.empty() )
//  std::cout << myVector[0] << std::endl;

myVector.for(0, [](auto & value){ std::cout << value << std::endl; });
```

- This functional approach eliminates the possible race condition and makes the code future proof
- Future proof in a sense that users can write better and safer code

Is this API easy to use? Concurrent case.

```
class ThreadsafeIntegerVector {
  public:
    //...

    void for(size_t pos, std::function<void(int & value)> action);
    void for_each(std::function<void(int & value)> action);
}

//if ( !myVector.empty() )
//  std::cout << myVector[0] << std::endl;

myVector.for(0, [](auto & value){ std::cout << value << std::endl; });
```

- This functional approach eliminates the possible race condition and makes the code future proof
- Future proof in a sense that users can write better and safer code

Is this API easy to use? Concurrent case.

```
class ThreadsafeIntegerVector {
  public:
    // ...

    size_t push_back(int value);
    void   remove_if(size_t                                          positionHint,
                     std::function<bool(size_t pos, int & value)> removeCondition);
}
```

```
size_t positionOfValue = myVector.push_back(1234);
// ...
myVector.remove_if(positionOfValue, [](auto, auto & v){ return v == 1234; });
```

- A thread safe API which also minimize the chance for error.
  - If one would use erase on a thread safe vector one would have to make sure that the iterator itself is thread safe and a potential race condition is avoided

Is this API easy to use? Concurrent case.

```
class ThreadsafeIntegerVector {
  public:
    // ...

    size_t push_back(int value);
    void   remove_if(size_t                                           positionHint,
                     std::function<bool(size_t pos, int & value)> removeCondition);
}
```

```
size_t positionOfValue = myVector.push_back(1234);
// ...
myVector.remove_if(positionOfValue, [](auto, auto & v){ return v == 1234; });
```

- A thread safe API which also minimize the chance for error.
  - If one would use erase on a thread safe vector one would have to make sure that the iterator itself is thread safe and a potential race condition is avoided

Is this API easy to use? Concurrent case.

- Creating thread safe constructs have two challenges
  - The actual creation of that construct
  - The design of an API which allows a thread safe usage

- Avoid returning values which can change concurrently like: `empty(), size(), operator[]`
- Offer functional alternatives like: `when_empty(std::function<void()> action),`

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Does the name fit?

- A class/struct name should reflect the <u>purpose</u> of the object
  - It can contain a design pattern specific role
  - If an object has multiple purposes it may be an indicator to split it up
- A function/method name should reflect the <u>task</u> the function is performing
  - When one requires “and” or “or” to describe the task it may be an indicator that the function should be split up
- A variable name should reflect its <u>content</u>

Maybe the naming is hard since we broke

- the Single Responsibility Principle
- Separation of Concerns
- to have a clear design idea

- Bad naming can also lead to bugs

Does the name fit?

- A class/struct name should reflect the <u>purpose</u> of the object
  - It can contain a design pattern specific role
  - If an object has multiple purposes it may be an indicator to split it up
- A function/method name should reflect the <u>task</u> the function is performing
  - When one requires “and” or “or” to describe the task it may be an indicator that the function should be split up
- A variable name should reflect its <u>content</u>

  Maybe the naming is hard since we broke
  - the Single Responsibility Principle
  - Separation of Concerns
  - to have a clear design idea

- Bad naming can also lead to bugs

Does the name fit?

- A class/struct name should reflect the purpose of the object
  - It can contain a design pattern specific role
  - If an object has multiple purposes it may be an indicator to split it up

- A function/method name should reflect the task the function is performing
  - When one requires “and” or “or” to describe the task it may be an indicator that the function should be split up

- A variable name should reflect its content

Maybe the naming is hard since we broke

- the Single Responsibility Principle
- Separation of Concerns
- to have a clear design idea

- Bad naming can also lead to bugs

Does the name fit?

- A class/struct name should reflect the purpose of the object
  - It can contain a design pattern specific role
  - If an object has multiple purposes it may be an indicator to split it up

- A function/method name should reflect the task the function is performing
  - When one requires “and” or “or” to describe the task it may be an indicator that the function should be split up

- A variable name should reflect its content

Maybe the naming is hard since we broke

- the Single Responsibility Principle
- Separation of Concerns
- to have a clear design idea

- Bad naming can also lead to bugs

Does the name fit?

- In iceoryx we implemented a string for safety critical applications called: cxx::string
    - API is similar to the STL string
    - Resides on the stack, capacity has to be known at compile time
    - No exceptions

- We renamed assign(const char * s) to unsafe_assign(const char * s)
    - The idea is to make the developer and the reviewer aware that the string has to be null terminated
        - Potential for memory issues

- When assigning a cxx::string one can use assign(const cxx::string & s) without the unsafe prefix
    - We have control over underlying memory and can ensure the safety

https://github.com/eclipse-iceoryx/iceoryx/blob/master/iceoryx_hoofs/include/iceoryx_hoofs/cxx/string.hpp

Does the name fit?

- In iceoryx we implemented a string for safety critical applications called: cxx::string
  - API is similar to the STL string
  - Resides on the stack, capacity has to be known at compile time
  - No exceptions

- We renamed assign(const char * s) to unsafe_assign(const char * s)
  - The idea is to make the developer and the reviewer aware that the string has to be null terminated
    - Potential for memory issues

- When assigning a cxx::string one can use assign(const cxx::string & s) without the unsafe prefix
  - We have control over underlying memory and can ensure the safety

https://github.com/eclipse-iceoryx/iceoryx/blob/master/iceoryx_hoofs/include/iceoryx_hoofs/cxx/string.hpp

Does the name fit?

- In iceoryx we implemented a string for safety critical applications called: cxx::string
  - API is similar to the STL string
  - Resides on the stack, capacity has to be known at compile time
  - No exceptions

- We renamed assign(const char * s) to unsafe_assign(const char * s)
  - The idea is to make the developer and the reviewer aware that the string has to be null terminated
    - Potential for memory issues

- When assigning a cxx::string one can use assign(const cxx::string & s) without the unsafe prefix
  - We have control over underlying memory and can ensure the safety

https://github.com/eclipse-iceoryx/iceoryx/blob/master/iceoryx_hoofs/include/iceoryx_hoofs/cxx/string.hpp

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Can one produce readable code with this API?

```
void Receiver::connect(bool useTcp,
                       bool reconnectWhenDisconnected,
                       bool receiveHistory,
                       bool printReceivedMessagesToConsole,
                       bool writeReceivedMessagesInLogFile)
{
  // perfectly readable implementation
}
```

- Functions with boolean arguments can lead very fast to unreadable code
- One requires more lines of code to use it cleanly

Can one produce readable code with this API?

```
// bad, please don't do this
myReceiver.connect(true, false, true, true, false);
```

```
// readable, but a lot of overhead
constexpr bool DO_USE_TCP = false;
constexpr bool DO_NOT_RECONNECT = false;
constexpr bool DO_RECEIVE_HISTORY = true;
constexpr bool DO_PRINT_MESSAGES = true;
constexpr bool DO_NOT_LOG_MESSAGES = false;

myReceiver.connect(DO_USE_TCP,
                   DO_NOT_RECONNECT,
                   DO_RECEIVE_HISTORY,
                   DO_NOT_LOG_MESSAGES,
                   DO_PRINT_MESSAGES);
```

Can you spot the two bugs?

Can one produce readable code with this API?

```
// bad, please don't do this
myReceiver.connect(true, false, true, true, false);


// readable, but a lot of overhead
constexpr bool DO_USE_TCP = false;
constexpr bool DO_NOT_RECONNECT = false;
constexpr bool DO_RECEIVE_HISTORY = true;
constexpr bool DO_PRINT_MESSAGES = true;
constexpr bool DO_NOT_LOG_MESSAGES = false;

myReceiver.connect(DO_USE_TCP,
                   DO_NOT_RECONNECT,
                   DO_RECEIVE_HISTORY,
                   DO_NOT_LOG_MESSAGES,
                   DO_PRINT_MESSAGES);
```

Can you spot the two bugs?

Can one produce readable code with this API?

```
// bad, please don't do this
myReceiver.connect(true, false, true, true, false);


// readable, but a lot of overhead
constexpr bool DO_USE_TCP = false;
constexpr bool DO_NOT_RECONNECT = false;
constexpr bool DO_RECEIVE_HISTORY = true;
constexpr bool DO_PRINT_MESSAGES = true;
constexpr bool DO_NOT_LOG_MESSAGES = false;

myReceiver.connect(DO_USE_TCP,
                   DO_NOT_RECONNECT,
                   DO_RECEIVE_HISTORY,
                   DO_NOT_LOG_MESSAGES,
                   DO_PRINT_MESSAGES);
```

Can you spot the two bugs?

1. DO_USE_TCP is false but it should be true
2. DO_NOT_LOG_MESSAGES and DO_PRINT_MESSAGES are switched

So why not use enum classes to ensure that such bugs never repeat again.

Can one produce readable code with this API?

```
myReceiver.connect(Protocol::TCP,
                   OnFailure::NO_RECONNECT,
                   ReceiveHistory::ENABLE,
                   ConsolePrinting::ENABLE,
                   Logging::DISABLE);
```

- If one mixes up the arguments we get a compile time error thanks to the strong type safety of C++
- Clean and readable code
- Enums are extendable booleans are not

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Can the API be used without documentation?

```
/// @brief …
/// ...
/// @param[in] onReceive optional argument, when set
///                  onReceive is called for every received message
/// @param[in] onConnection optional argument, when set …
/// ...
Receiver::Receiver(unsigned int historyCacheSize,
                   std::function<void()> onReceive,
                   std::function<void()> onConnection,
                   Acknowledge acknowledge)
{
  // implementation
}
```

Why use documentation when you can use types?

Can the API be used without documentation?

```
Receiver::Receiver(unsigned int historyCacheSize,
                   std::optional<std::function<void()>> onReceive,
                   std::optional<std::function<void()>> onConnect,
                   Acknowledge acknowledge)
{
  // implementation
}
```

Try to avoid to rely on the nullability of nullable types - use std::optional instead

- Can reduce error handling
- Makes the code future proof

Provide `optional` as prefix for optional variables.

- Provide the reviewer additional insights

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

What bugs does the API allow?

- During a review a developer may spots mistakes more easily when the constructs have descriptive names
  - Remember: The variable name should reflect the contents.

```
void Receiver::receive()
{
  this->onReceive();
}
```

<- Whoopsie, this can throw an exception.

```
void Receiver::receive()
{
  (*this->optionalOnReceive)();
}
```

<- A reviewer may realize that we have to verify first if it actually contains a value.

What bugs does the API allow?

- During a review a developer may spots mistakes more easily when the constructs have descriptive names
    - Remember: The variable name should reflect the contents.

```
void Receiver::receive()
{
  this->onReceive();
}
```

<- Whoopsie, this can throw an exception.

```
void Receiver::receive()
{
  (*this->optionalOnReceive)();
}
```

<- A reviewer may realize that we have to verify first if it actually contains a value.

What bugs does the API allow?

- C++23 will introduce monadic std::optional

```
std::optional<int> optionalValue;

// old
if ( optionalValue )
  std::cout << *optionalValue << std::endl;
else
  /* do something else */


// new
optionalValue
   .and_then([](auto & v){ std::cout << v << std::endl; })
   .or_else([]{ /* no value */ });
```

What bugs does the API allow?

- C++23 will introduce monadic std::optional

```
std::optional<int> optionalValue;

// old
if ( optionalValue )
  std::cout << *optionalValue << std::endl;
else
  /* do something else */


// new
optionalValue
   .and_then([](auto & v){ std::cout << v << std::endl; })
   .or_else([]{ /* no value */ });
```

What bugs does the API allow?

- Why not use <u>pure</u> monadic access in std::optional, std::unique_ptr, std::shared_ptr and other nullable constructs in a safety environment?
    - Maybe it is not suitable if you want to have maximum performance but it reduces the probability of bugs.

```
std::unique_ptr<int> intPointer;

intPointer
   .and_then([](auto & v){
        std::cout << v << std::endl;
   });
```

- has defined behavior
- excludes error when accessing unset value

```
std::unique_ptr<int> intPointer;

std::cout << *intPointer << std::endl;
```

- undefined behavior when intPointer is empty

What bugs does the API allow?

- In a safety environment we can remove operator* and provide access solely via and_then and or_else
- This eliminates bugs like

```
void Receiver::receive()
{
  this->optionalOnReceive();
}
```

- We always access it in a safe manner

```
void Receiver::receive()
{
  this->optionalOnReceive()
    .and_then([](auto & f){ f(); })
    .or_else([]{ /* no optional value */ });
}
```

If you cannot wait, an optional with monadic operations is already available:

https://github.com/eclipse-iceoryx/iceoryx/blob/master/iceoryx_hoofs/include/iceoryx_hoofs/cxx/optional.hpp

What bugs does the API allow?

- In a safety environment we can remove operator* and provide access solely via and_then and or_else
- This eliminates bugs like

```
void Receiver::receive()
{
  this->optionalOnReceive();
}
```

- We always access it in a safe manner

```
void Receiver::receive()
{
  this->optionalOnReceive()
    .and_then([](auto & f){ f(); })
    .or_else([]{ /* no optional value */ });
}
```

If you cannot wait, an optional with monadic operations is already available:

https://github.com/eclipse-iceoryx/iceoryx/blob/master/iceoryx_hoofs/include/iceoryx_hoofs/cxx/optional.hpp

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is the API easy to use?

```
class Receiver {
  public:
    Receiver(unsigned int                            historyCacheSize,
             std::optional<std::function<void()>> onReceive,
             std::optional<std::function<void()>> onConnection,
             Acknowledge                             acknowledge);
```

- Let us assume that every argument has a default value

```
class Receiver {
  public:
    Receiver(unsigned int                            historyCacheSize = 10,
             std::optional<std::function<void()>> onReceive        = std::nullopt,
             std::optional<std::function<void()>> onConnection     = std::nullopt,
             Acknowledge                             acknowledge      = Acknowledge::SEND);
```

- What if the user would like to adapt only the first and the last argument?

Is the API easy to use?

```
class Receiver {
  public:
    Receiver(unsigned int                            historyCacheSize,
             std::optional<std::function<void()>> onReceive,
             std::optional<std::function<void()>> onConnection,
             Acknowledge                           acknowledge);
```

- Let us assume that every argument has a default value

```
class Receiver {
  public:
    Receiver(unsigned int                            historyCacheSize = 10,
             std::optional<std::function<void()>> onReceive        = std::nullopt,
             std::optional<std::function<void()>> onConnection     = std::nullopt,
             Acknowledge                           acknowledge      = Acknowledge::SEND);
```

- What if the user would like to adapt only the first and the last argument?

Is the API easy to use?

- First, pack all arguments into a struct

```
struct ReceiverInfo {
  unsigned int                          historyCacheSize = 10;
  std::optional<std::function<void()>> onReceive        = std::nullopt;
  std::optional<std::function<void()>> onConnection     = std::nullopt;
  Acknowledge                           acknowledge      = Acknowledge::SEND;
};
```

- Use the struct in the constructor

```
class Receiver {
  public:
    Receiver(ReceiverInfo constructorInfo);
```

Is the API easy to use?

- First, pack all arguments into a struct

```
struct ReceiverInfo {
  unsigned int                          historyCacheSize = 10;
  std::optional<std::function<void()>> onReceive        = std::nullopt;
  std::optional<std::function<void()>> onConnection     = std::nullopt;
  Acknowledge                           acknowledge      = Acknowledge::SEND;
};
```

- Use the struct in the constructor

```
class Receiver {
  public:
    Receiver(ReceiverInfo constructorInfo);
```

Is the API easy to use?

- We use the C++20 feature aggregate initialization to set only the first and last argument.

- Initialize the struct with uniform initialization {}
  - A member can be assigned by using a dot prefix and the member name

```
StructName{
  .memberName = VALUE1,
  .anotherMemberName = VALUE2,
}
```

- Applied to our Receiver constructor

```
Receiver myReceiver{ReceiverInfo{
    .historyCacheSize = 20,
    .acknowledge = Acknowledge::DISCARD
  }};
```

Is the API easy to use?

- We use the C++20 feature aggregate initialization to set only the first and last argument.

- Initialize the struct with uniform initialization {}
  - A member can be assigned by using a dot prefix and the member name

```
StructName{
  .memberName = VALUE1,
  .anotherMemberName = VALUE2,
}
```

- Applied to our Receiver constructor

```
Receiver myReceiver{ReceiverInfo{
    .historyCacheSize = 20,
    .acknowledge = Acknowledge::DISCARD
  }};
```

Is the API easy to use?

```
Receiver myReceiver{ReceiverInfo{
    .historyCacheSize = 20,
    .acknowledge = Acknowledge::DISCARD
  }};
```

```
Receiver myReceiver{20, std::nullopt,
                        std::nullopt,
                        Acknowledge::DISCARD};
```

- When a method or function has more than one argument the struct approach in combination with C++20 aggregate initialization can improve the readability
- Whenever you have more than one argument consider using a struct with C++ aggregate initialization to improve the readability

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is the task the code performs clear?

- Let's add two additional constructors to our Receiver

```
class Receiver {
  public:
    // init, adds connection and connects
    Receiver(const connection_t port);

    // init, adds connections but does not connect
    Receiver(const std::vector<connection_t> & connections);
};
```

```
    Receiver myReceiver1({127.0.0.1, 4117});     // connects automatically

    Receiver myReceiver2({{127.0.0.1, 313}});    // does not connect
```

Is the task the code performs clear?

- Let's add two additional constructors to our Receiver

```
class Receiver {
  public:
    // init, adds connection and connects
    Receiver(const connection_t port);

    // init, adds connections but does not connect
    Receiver(const std::vector<connection_t> & connections);
};
```

```
    Receiver myReceiver1({127.0.0.1, 4117});      // connects automatically

    Receiver myReceiver2({{127.0.0.1, 313}});     // does not connect
```

Is the task the code performs clear?

- A trick from the implementation of std::variant could help.
  - Define two empty structs to distinguish the constructors

```
struct connectOnCreation_t {};
struct addConnectionsOnly_t {};

constexpr connectOnCreation_t  connectOnCreation{};
constexpr addConnectionsOnly_t addConnectionsOnly{};


class Receiver {
  public:
    // init, adds connection and connects
    Receiver(connectOnCreation_t, const connection_t& connection);

    // init, adds connections but does not connect
    Receiver(addConnectionsOnly_t, const std::vector<connection_t> & connections);
};
```

Is the task the code performs clear?

- A trick from the implementation of std::variant could help.
  - Define two empty structs to distinguish the constructors

```
struct connectOnCreation_t {};
struct addConnectionsOnly_t {};

constexpr connectOnCreation_t  connectOnCreation{};
constexpr addConnectionsOnly_t addConnectionsOnly{};


class Receiver {
  public:
    // init, adds connection and connects
    Receiver(connectOnCreation_t, const connection_t& connection);

    // init, adds connections but does not connect
    Receiver(addConnectionsOnly_t, const std::vector<connection_t> & connections);
};
```

Is the task the code performs clear?

```
Receiver myReceiver1(connectOnCreation,
                     {127.0.0.1, 4117});

Receiver myReceiver2(addConnectionsOnly,
                     {{127.0.0.1, 313}});
```

```
Receiver myReceiver1({127.0.0.1, 4117});

Receiver myReceiver2({{127.0.0.1, 313}});
```

Is the task the code performs clear?

- Whenever possible, try to use factory methods over constructor specialization.
  - Can improve testability with mocks.
  - Allows better error handling when the constructor fails

```
class Receiver {
  public:
    static std::optional<Receiver> CreateAndConnect(const connection_t & connection);

    static std::optional<Receiver>
              CreateAndAddConnections(const std::vector<connection_t> & connections);
};
```

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- **Ownership & lifetime - do we use RAII?**
- Is this code extendable?

Ownership & lifetime - do we use RAII?

- Method pairs like
  - `AddX()`, `RemoveX()`
  - `CreateX()`, `DestroyX()`
  - `AttachX()`, `DetachX()`
  - `Allocate()`, `Free()`
  - `Open()`, `Close()`

  are an indicator that we should use RAII

- RAII = Resource Acquisition Is Initialization, binds the resource lifetime to the objects lifetime
  - examples in the STL are std::unique_ptr or std::shared_ptr

- We would like to forward a message to a sender whenever our Receiver receives a message

Ownership & lifetime - do we use RAII?

```
class Receiver {
  public:
    callbackId_t AttachOnReceiveCallback(const std::function<void()> & f);

    void RemoveCallback(const callbackId_t id);
};
```

Ownership & lifetime - do we use RAII?

```
class Receiver {
  public:
    callbackId_t AttachOnReceiveCallback(const std::function<void()> & f);

    void RemoveCallback(const callbackId_t id);
};
```

```
{
  Sender mySender;
  mySender.callbackId = myReceiver.AttachOnReceiveCallback([&]{
     auto message = myReceiver.receive();
     mySender.send(message);
     });
  // forgot to call RemoveCallback before out of scope -> dangling reference
}
```

Ownership & lifetime - do we use RAII?

- Realize RAII with a std::unique_ptr in combination with a custom deleter

```
class Receiver {
  public:
    // use a unique_ptr with custom deleter to remove callback
    std::unique_ptr<callbackId_t, std::function<void()>
    AttachOnReceiveCallback(const std::function<void()> & f) {
      // some code
      return {new callbackId_t(callbackId),
              // custom deleter, remove callback
              [this](callbackId_t * id){
                this->RemoveCallback(id);
                delete id;
              }};
    }
  private:
    void RemoveCallback(const callbackId_t id);
};
```

Ownership & lifetime - do we use RAII?

- Realize RAII with a std::unique_ptr in combination with a custom deleter

```
class Receiver {
  public:
    // use a unique_ptr with custom deleter to remove callback
    std::unique_ptr<callbackId_t, std::function<void()>
    AttachOnReceiveCallback(const std::function<void()> & f) {
      // some code
      return {new callbackId_t(callbackId),
              // custom deleter, remove callback
              [this](callbackId_t * id){
                this->RemoveCallback(id);
                delete id;
              }};
    }
  private:
    void RemoveCallback(const callbackId_t id);
};
```

Ownership & lifetime - do we use RAII?

```
{
  Sender mySender;
  mySender.callbackId = myReceiver.AttachOnReceiveCallback([&]{
     auto message = myReceiver.receive();
     mySender.send(message);
     });
  // sender goes out of scope, with it callbackId and the destructor calls
  // removeCallback for us
}
```

Ownership & lifetime - do we use RAII?

```
std::unique_ptr<callbackId_t, std::function<void()>
AttachOnReceiveCallback(const std::function<void()> & f) {
  return {new callbackId_t(callbackId),

   // as soon as Receiver is moved or copied the this pointer becomes invalid
          [this](callbackId_t * id){
            this->RemoveCallback(id);
            delete id;
          }};
}
```

- When using this technique, copy and move operations should be deleted
  - Or ensure that the object is not moved or copied anymore - error prone.
  - Did you know that objects inside a vector are moved/copied when elements are deleted or added?

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

Is this code extendable?

```
class Sender {
  public:
    void Attach(Receiver & receiver);

    void Send(int message) {
      for(auto & r : this->receivers)
        r->Deliver(message);
    }

  private:
    std::vector<std::shared_ptr<Receiver>>
       receivers;
};
```

- To be able to send to a wide variety of Receiver one may introduce a Receiver base class.
- The Sender requires a Deliver method to deliver a message to any kind of receiver.
- In object oriented programming objects are communicating via messages.
- Isn’t therefore not every class which provides some kind of method with an integer argument a receiver of a message send by a Sender?
- Wouldn’t it be awesome to get rid of inheritance and be able to attach any object which has a Deliver method?

Is this code extendable?

```
class Sender {
  public:
    void Attach(Receiver & receiver);

    void Send(int message) {
      for(auto & r : this->receivers)
        r->Deliver(message);
    }

  private:
    std::vector<std::shared_ptr<Receiver>>
       receivers;
};
```

- To be able to send to a wide variety of Receiver one may introduce a Receiver base class.
- The Sender requires a Deliver method to deliver a message to any kind of receiver.
- In object oriented programming objects are communicating via messages.
- Isn't therefore not every class which provides some kind of method with an integer argument a receiver of a message send by a Sender?
- Wouldn't it be awesome to get rid of inheritance and be able to attach any object which has a Deliver method?

Is this code extendable?

```
class Sender {
  public:
    void Attach(Receiver & receiver);

    void Send(int message) {
      for(auto & r : this->receivers)
        r->Deliver(message);
    }

  private:
    std::vector<std::shared_ptr<Receiver>>
       receivers;
};
```

- To be able to send to a wide variety of Receiver one may introduce a Receiver base class.
- The Sender requires a Deliver method to deliver a message to any kind of receiver.
- In object oriented programming objects are communicating via messages.
- Isn't therefore not every class which provides some kind of method with an integer argument a receiver of a message send by a Sender?
- Wouldn't it be awesome to get rid of inheritance and be able to attach any object which has a Deliver method?

Is this code extendable?

```
class Sender {
  public:
    void Attach(Receiver & receiver);

    void Send(int message) {
      for(auto & r : this->receivers)
        r->Deliver(message);
    }

  private:
    std::vector<std::shared_ptr<Receiver>>
       receivers;
};
```

- To be able to send to a wide variety of Receiver one may introduce a Receiver base class.
- The Sender requires a Deliver method to deliver a message to any kind of receiver.
- In object oriented programming objects are communicating via messages.
- Isn’t therefore not every class which provides some kind of method with an integer argument a receiver of a message send by a Sender?
- Wouldn’t it be awesome to get rid of inheritance and be able to attach any object which has a Deliver method?

Is this code extendable?

```
class Sender {
  public:
    void Attach(Receiver & receiver);

    void Send(int message) {
      for(auto & r : this->receivers)
        r->Deliver(message);
    }

  private:
    std::vector<std::shared_ptr<Receiver>>
       receivers;
};
```

- To be able to send to a wide variety of Receiver one may introduce a Receiver base class.
- The Sender requires a Deliver method to deliver a message to any kind of receiver.
- In object oriented programming objects are communicating via messages.
- Isn’t therefore not every class which provides some kind of method with an integer argument a receiver of a message send by a Sender?
- Wouldn’t it be awesome to get rid of inheritance and be able to attach any object which has a Deliver method?

Is this code extendable?

```
class Sender {
  public:
    template<typename ReceiverType>
    void Attach(const std::shared_ptr<ReceiverType> & receiver) {
      // instead of storing a pointer to the receiver and call Deliver directly
      // we store a callable which indirectly stores the type and calls Deliver
      // for us
      this->receivers.emplace_back(
        [receiver](int message){
           receiver->Deliver(message);
        }
      );
    }

  private:
    std::vector<std::function<void()>> receivers;
};
```

Summary

- Is the design over-abstracted?
- Is this API easy to use and hard to misuse?
- Can we reduce the error handling?
- Is this API easy to use? Concurrent case.
- Does the name fit?
- Can one produce readable code with this API?
- Can the API be used without documentation?
- What bugs does the API allow?
- Is the API easy to use?
- Is the task the code performs clear?
- Ownership & lifetime - do we use RAII?
- Is this code extendable?

- The slides, the questions and links to actual production source code, which use the described techniques, can be found here:

  https://github.com/elfenpiff/meetingcpp_2021