

0xBADC0DE
C++Now 2014

Jens Weller
Meeting C++
info@codenode.de
info@meetingcpp.com
@meetingcpp #meetingcpp

About me



- * '81
- C++ since '98
- Vodafone '02-'07
- C++ Freelancer '07
- C++ UG NRW '11
- Meeting C++ '12

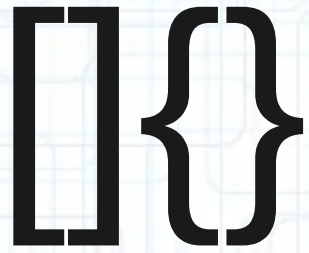
Meeting C++

- Conference
 - 2012: 150 Attendees
 - 2013: 200 Attendees
 - 2014: 300 Attendees
 - including 50 Students
- Website & Blog for C++
- Platform for C++ User Groups in Europe
- Goal
 - Building a (european) C++ Network

0xBADC0DE

<:]{%>

0xBADC0DE



- (empty) C++11 Lambda

0xBADC0DE

goto fail;

0xBADCODE

goto fail;

- Should not have happend
- But it did!
- Its a bug.
- A bug *can* result from bad code.

F*ck...



0xBADC0DE

- Your own code?
- Maybe the code of
 - the person next to you?
 - your boss?
 - ...

Weeks of coding can save you
hours of planning!

unknown programmer

0xBADC0DE

Poor mans C++

(What me originally made think about this topic)

Who is the poor man?

„A person whos main concern is not C++,
C++ is seen in the role of a tool“

Who is the poor man?

- This is just one category
- The poor man usually is not poor
 - just not a C++ Expert
 - basic („poor“) C++ knowledge
- Often is an expert
 - but in a different domain
 - e.g. scientists, other programminglanguages

Its maybe not even his fault

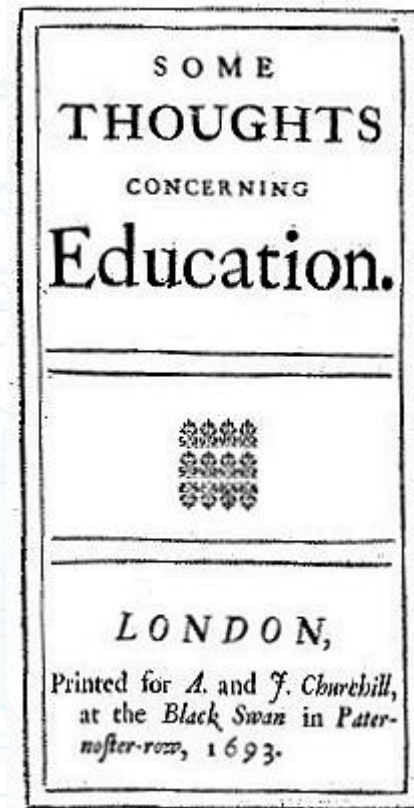
- As C++ is only seen as a tool
 - time to improve skills is limited
- „But this works too“
- Copy & Paste Evolution
 - C & P old solution
 - Mutate the things you need
 - Old code can live very long

Typical Problems

- poor design knowledge
- mixing old techniques and C into C++
- C with Classes
- Old C++ Books
- new Problems
 - aka memoryleaks
- clash of styles
 - loops vs. algorithms

There is hope!

- The 'poor man' can be educated!
- as experts, they're willing to learn



There is hope!



- Maybe hard to reach
 - due workload
 - C++ is not primary concern

Why fix it, if it ain't broke?

0xBADC0DE

Examples of bad code

Examples of 0xBADC0DE

- new more::Problems
- Layers of Engineering
- Classdesign
- Monster (classes | methods/functions)
- init 'patterns'
- Money \$ €
- switch
- ***that library***

Memoryleaks

- There is a certain overuse of new
- People forget often delete or delete[]
 - 'java' like C++ - no deletes
 - not always a show stopper
- Ownership concepts can reduce problem
 - smart pointers
 - objecthierachies (QObject e.g.)

Code Example (Qt)

```
void MainWindow::on_action()
{
    MyDialog* dlg = new MyDialog(0, "bad code");
    if(dlg->exec()) ...
}
```

```
void MainWindow::on_action()
{
    MyDialog* dlg = new MyDialog(this, "bad code");
    if(dlg->exec()) ...
}
```

- Memoryleak
- Resourceleak
- Parent delete?

```
void MainWindow::on_action()
{
    MyDialog dlg(this, "ok if parent lives longer");
    if(dlg.exec())...
}
```

```
void MainWindow::on_action()
{
    auto *dlg = new MyDialog(this, "noexcept");
    ...
    dlg->deleteLater(); // Qt Framework specific
    // pending events are processed
}
```

Memoryleaks

- What are smart pointers?
 - RAI and similar techniques are still often unknown
 - Pointerstyle
 - overusing pointers
 - overusing smartpointers
 - shared_ptr addiction

Stack > Smartpointer > raw owning pointer

Refactoring

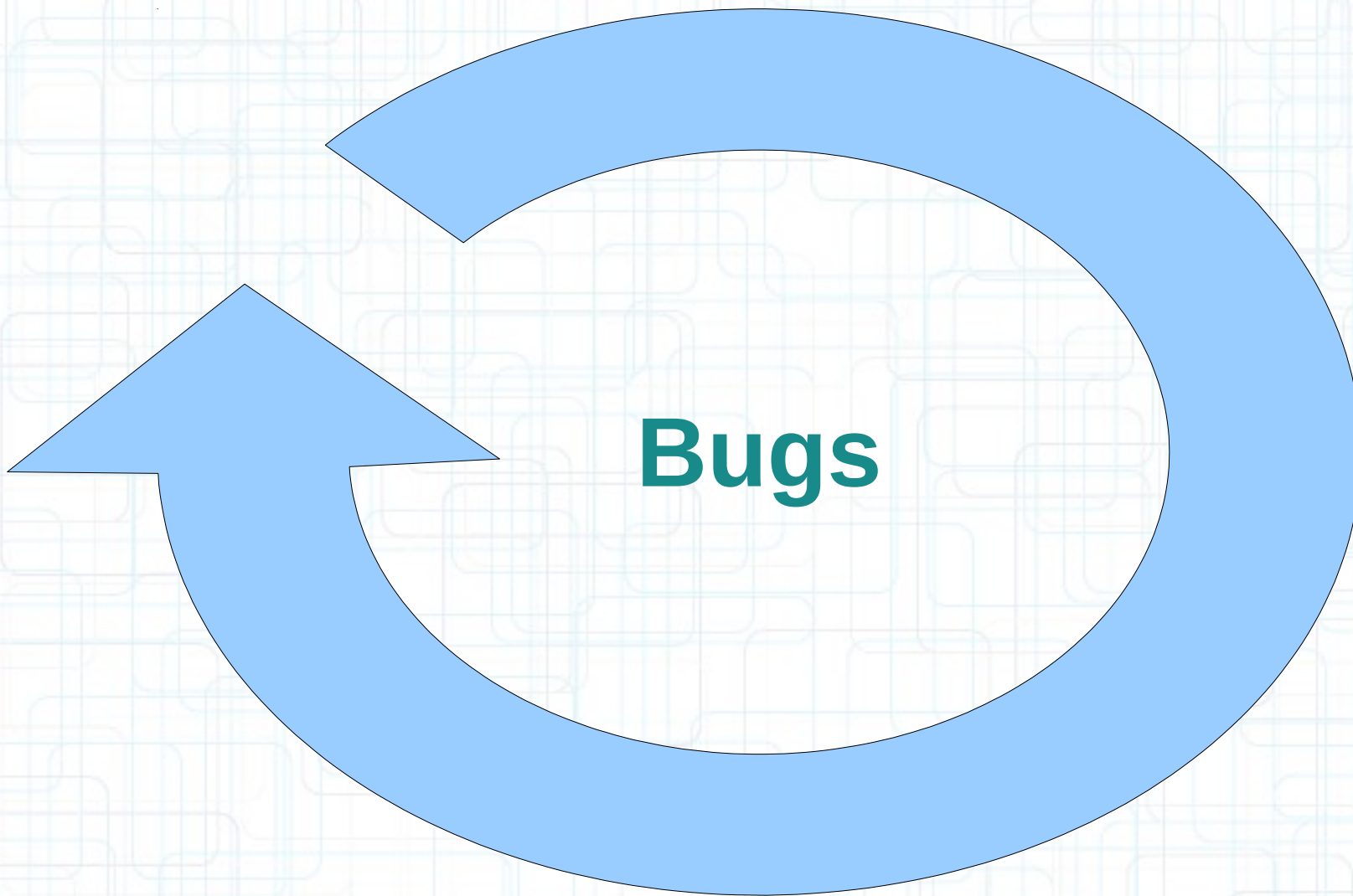
- Introducing smart pointers
 - Interdependencies can make this hard
 - Pointeroverusage vs. Smartpointeroverusage
- a rare case with delete
 - slowed my program
 - so importprogram was faster without.

Layers of Engineering

- Hiding code through layers
- Nice Surface & rotten hidden Parts.
- Example: projects with a longer history
- //Don't touch that code area
- Rather adding a new layer then doing proper refactoring
- Poor documentation

Layers of Engineering

- New Features > Bugfixes
- Bugfixes > Refactoring
- Refactoring > Documentation



Bugs

Classdesign

- Monsterclasses
- Dependency Hell
- OOP Overusage
- Interface vs. Implementation
 - example

Classdesign

```
class Parameter
...
public:
    virtual bool validate(FieldID id){return true;}; // FieldID is an enum
    virtual bool validate(QString fieldname){return true;}
...
```

```
class MyParameter : public Parameter // Problem
...
public:
    virtual bool validate(FieldID id){/*long validation*/}
    virtual bool validate(QString fieldname){return true;}
...
```

```
class Parameter { // Solution
public:
    virtual bool validate(FieldID id){/*long validation*/}
    virtual bool validate(QString fieldname){
        return validate(name2fieldID(fieldname));
    }
...
}
```


Classdesign

```
class Parameter
...
public:
    virtual bool validate(FieldID id){return true;}; // FieldID is an enum
    virtual bool validate(QString fieldname){return true;};
...
```

```
class MyParameter : public Parameter // Problem
...
public:
    virtual bool validate(FieldID id){/*long validation*/}
    // don't forget to fix your code!
...
```

```
class Parameter { // Solution
public:
    virtual bool validate(FieldID id){/*long validation*/}
    virtual bool validate(QString fieldname){
        return validate(name2fieldID(fieldname));
    }
...
}
```

Classdesign

- Non virtual Interfaces
 - good pattern for OOP
 - I've seen it rarely in application code
- Pattern (Gang of 4)
 - good knowledge
 - Patterns need to be correctly
 - implemented
 - used

Monsters

- Monsterclasses are quite common
 - layering can be a cause
 - adding new features to existing classes
- Monstermethods/functions
 - I'd love to get a tool for average and median method length in LoC.
 - switches + copy paste
- Refactoring needed (again)

Init 'Pattern'

- Often are init methods used
 - calling virtual functions
 - a valid object must call init after construction
- Example:
 - Bada SDK from Samsung
 - Some projects I've seen

Init 'Pattern'

- Use constructors properly...
- Avoid virtual function calls
 - for initializing your objects
- if you can't
 - force make functions or factories
 - make your constructors private
 - dont forget op=, move-op
 - rule of 0/5 defaults if no implementation

Money \$ €

- Using float for your cash
 - every now and then you loose a cent.
- Money should be a type
 - Store as cents in 1000

„It doesn't make sense, it makes you loose cents“

Switches

"I have a some 50kloc switch statement here that gcc doesn't even compile as it runs out of memory."

that library

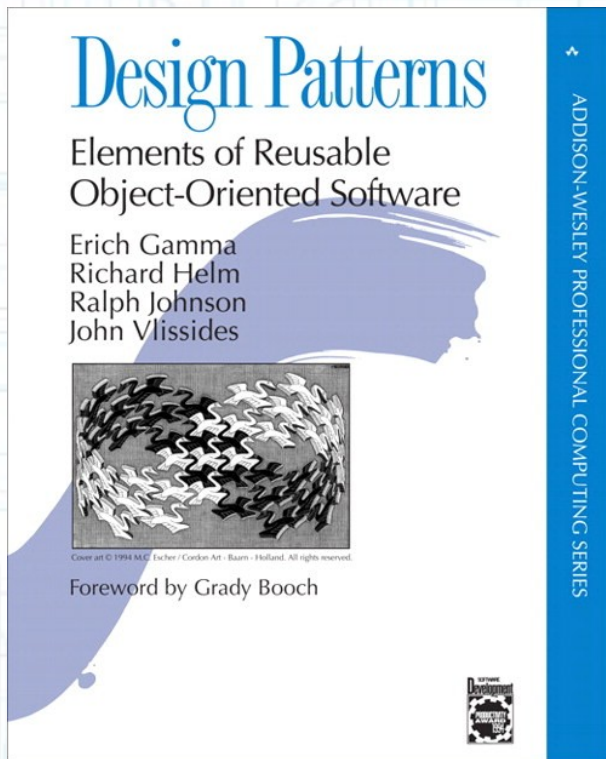
- Features:
 - implements a certain protocol
 - poorly documented
 - uses & brings boost 1.33
 - not intended to be used alone
 - Core users do VOIP, we only need the protocol

that library

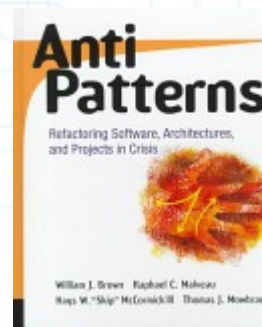
- Solution
 - Update to newer boost version not possible/available
 - Private implementation using the library
 - Public interface
 - does not expose any library details
 - No leaking into the project

Anti Patterns

- Design Patterns
 - Gang of 4



- Antipatterns



- Singleton
- God Objects
- Monsterclasses
- OO Overuse
- C++11/14:
new/delete

- Antipattern Catalog

MACROS are EVIL



More Examples?

- at a code base near you!
- Maybe in your next job

Dealing with / Using bad code

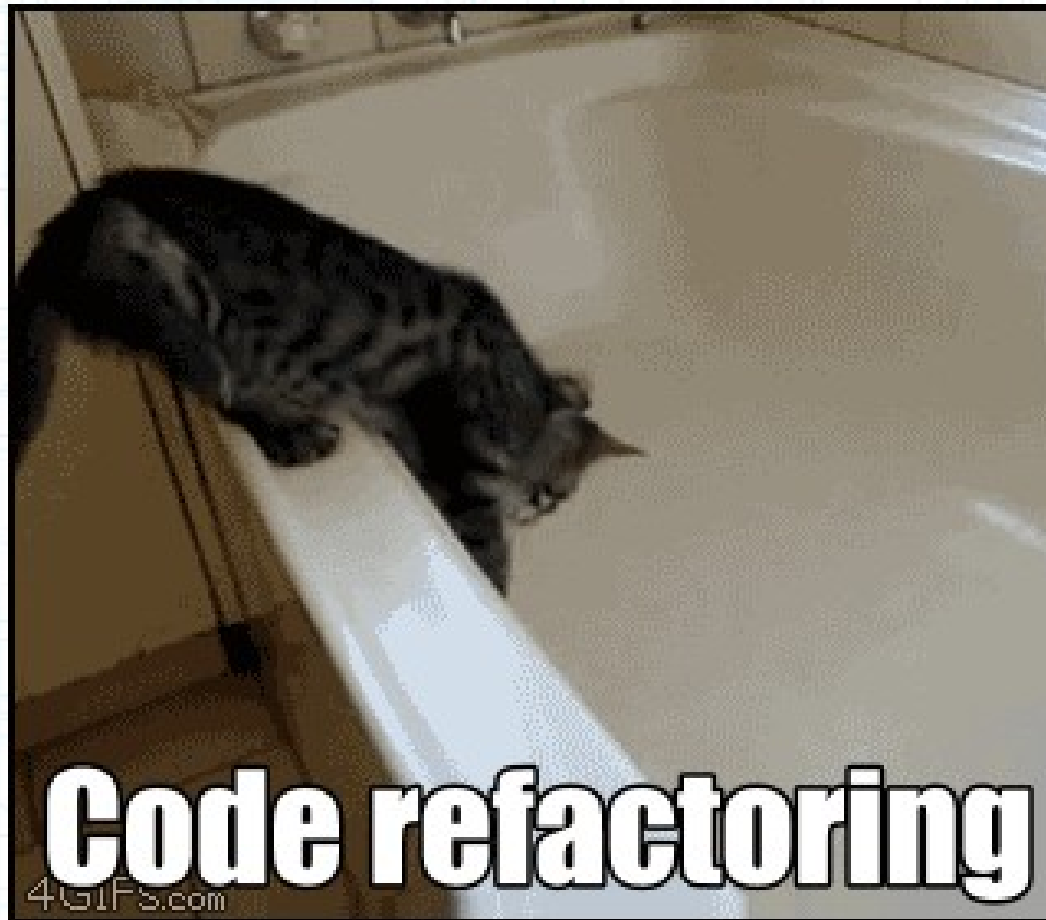
Fixing > Dealing > Using
Fixing < Dealing < Using

Fixing

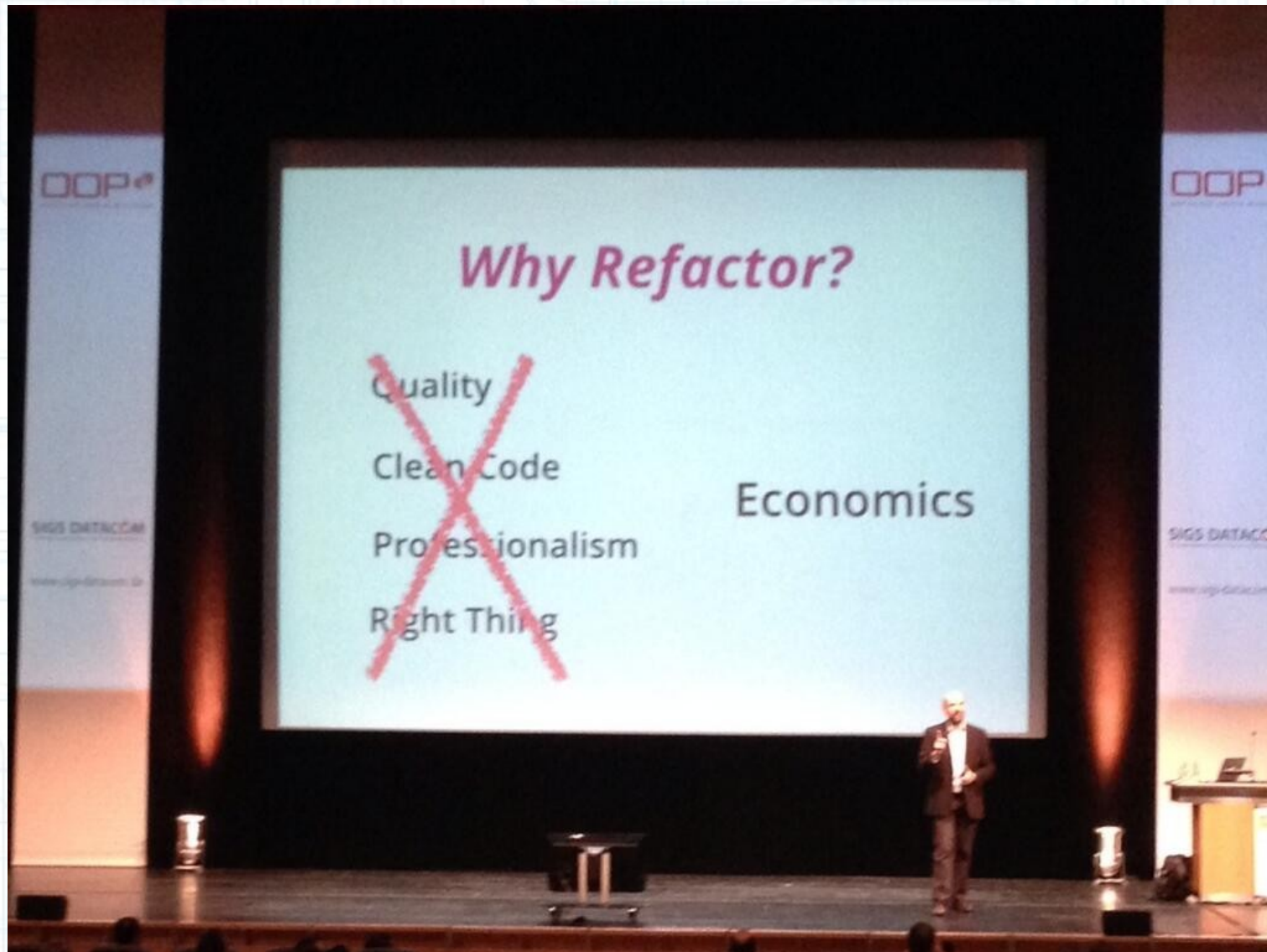
- When ever you can, fix!
 - but don't become Don Quijote!
- But is bad code the problem?
 - maybe its a symptom
- Maybe you can't fix it.
 - so deal with it?



On refactoring...



On Refactoring...



(Martin Fowler at OOP 2014)

Dealing with bad code

- Nobody has bad code that's not used
- Fixing bad code involves dealing with it.
- Refactoring or Rewriting
 - not always an option :/
- New or unknown parts of the code base

Dealing with bad code

- Static code analysis
 - use these Tools!
 - CppCheck, Clang static analyzer
 - commercial tools
 - gives you a first overview
 - you'll get a list of things to fix
 - Clang modernize
- Documentation
 - doxygen + graphviz

What if you can't fix it?



Dealing with using bad code

- Sometimes you can't fix it
- But you can deal with that
 - don't spread it yourself
- Contain it safely
- Try to fix later!



Prevention

- Educate your teams and coworkers!
 - and your Management
- Analyze how to improve your teams code quality!
- Update your companies C++ Books!!!
- Don't reinvent the wheel
 - use libraries

While I'm at Libraries...

- I think it is a good practice to develop in modules/libraries
- Even application code
- This forces at least a thought how to define an Interface

Instead of this

YOUR APPLICATION

Libraries
C++ Standard Library, Qt, boost, ...

Modularize your Application

YOUR APPLICATION Stub

YOUR UNIT TESTs

Application Layer of Libraries

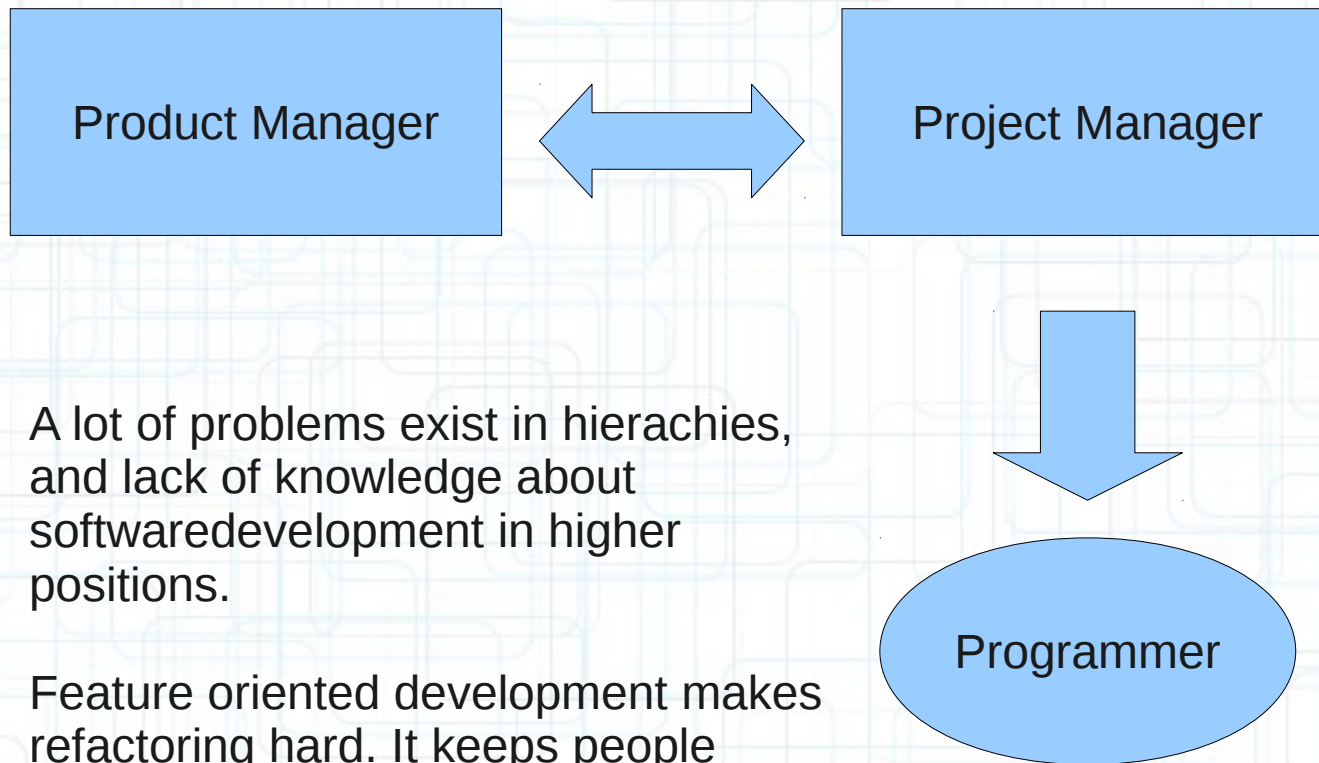
Libraries
C++ Standard Library, Qt, boost, ...

0xBADC0DE

Prefer library code over application code

All code decays.
(over time).

Bad Code Culture



A lot of problems exist in hierarchies, and lack of knowledge about software development in higher positions.

Feature oriented development makes refactoring hard. It keeps people busy with new features and new bug fixes.

Bad Code Culture

- Not always its the programmers fault!
 - bad „Work“environments
- Testing is not an industry standard
 - Testing is often not understood
 - Testcoverage is poor or 0
- Not all IDE/Tools produce good code

Layers of Engineering

- New Features > Bugfixes
- Bugfixes > Refactoring
- Refactoring > Documentation
- Tests?

Testing, lack of

- „Of course we do test“
- „No we don't write Unit Tests“
- Testing is not an industry standard
- Testlibraries
 - boost::test, google test/mock, CppUnit
 - C++11: Catch
 - And there is a lot more out there

Testtooling support

- IDEs do not support testing!
 - default projects should include tests!
 - default projects are often used...
- Testcoverage in Tutorials and Books is often rather poor
- Tests are heavily underused in the industry.

IT is not very healthy...

- Our industry „kills“ people every year
- Life is too short for bad work environments
 - if you can't change it
 - get out, get a new C++ job



A hamster wheel can look like a career ladder
from the inside...

Seeing the bigger picture

- We're moving towards newer, better standards
- Not all code can be rewritten
- C++ code base is huge
- With a constantly evolving C++ Standard
 - refactorings should occur more often

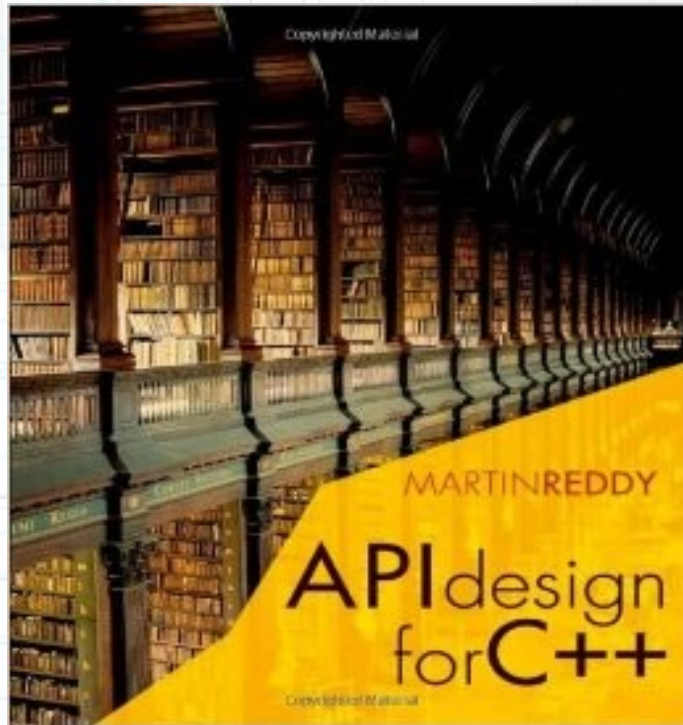
C++ Education

- You never finish learning C++
- You should never finish exchanging C++ knowledge
- Fixing bad code does not prevent it
- Educating and reaching more people who write C++ could achieve this.
 - Code Dojo

Books

- C++11
 - Bjarne
 - C++ Primer
- API/Design
 - Modern C++
 - API Design for C++
- My recommendations
- List on Stackoverflow

Books



- API design for C++
 - Martin Reddy
- Good, general overview on different development related practices.
- Testing, Scripting, API Design

Islands of C++

- C++ Community is very versatile
 - boost
 - Qt
 - wxWidgets
- C++11 brings a whole lot of new libraries
- Exchange and flow of knowledge

C++ User Groups

- Local active C++ Networks
 - Education
 - Jobs
 - Exchange of Knowledge
 - helping Talents
- Basic Building Block C++ Community
 - global
 - interconnected

IoT

- Internet of Things
 - (not) just a buzzword
 - a new interconnected world
 - driven by mobile and embedded
 - (maybe) a trend for the coming years
- Why not base it on...
 - C++14 & LLVM/clang?
 - Help create a better future with C++

(My) Conclusions I

- Goals
 - make you think of a solution that fits your needs
 - IMHO no global solution easily possible
- Bad Code
 - can mean a lot of things
 - it depends on your own knowledge
- Prefer library over application code

(My) Conclusions II

- dealing with bad code
 - fix/improve it if you can
 - don't spread it if you can't
 - contain it if you need to
- let Tools help you
 - static code analysis
 - doxygen & documentation
 - clang modernize

(My) Conclusions III

- Prevention
 - educate your teams and coworkers
 - AND management.
 - update C++ books
 - visit C++ Conferences! (scnr)
- be engaged in the community
 - share your knowledge
 - commit code to opensource projects
 - at least once a year (thanks to Eric)

(My) Conclusions

- C++11/14 are fundamental standards
 - Help create a better world with modern C++!
 - Modern C++ can prevent a lot of bad code

Last slide...

**Thank you &
Questions?**

info@codenode.de
info@meetingcpp.com
@meetingcpp