



Bringing Clean Code to Large Scale Legacy C++ Applications

Arne Mertz

The logo for zühlke is centered on a solid purple square. The word "zühlke" is written in a white, lowercase, sans-serif font. Below it, the phrase "empowering ideas" is written in a smaller, white, lowercase, sans-serif font.

zühlke
empowering ideas

Simplify C#!

arne-mertz.de

Bringing Clean Code to Large Scale Legacy C++ Applications

"What do you mean?"

Quote

"Legacy code" is a term often used derogatorily to characterize code that is written in a language or style that

(1) ...the speaker/writer consider outdated and/or

(2) ...is competing with something sold/promoted by the speaker/writer.

"Legacy code" often differs from its suggested alternative by actually working and scaling.

Bjarne Stroustrup



Clean Code **in** C++ Applications

"Is that even possible?"

Make use of C++ features that make the code more readable and safer

For example:

- range based for
- references vs. pointers
- typedefs
- smart pointers

"But performance!"

"But performance!"

Performance is important.

"But performance!"

Performance is important.

But that does not mean we have to optimize every little piece of code.

Before you manually optimize...

- ... use the right data structures and algorithms.

Before you manually optimize...

- ... use the right data structures and algorithms.
- ... trust the optimizer.

Before you manually optimize...

- ... use the right data structures and algorithms.
- ... trust the optimizer.
- ... find the actual bottleneck.

Use. A. Profiler.

</OptimizationRant>

Bringing Clean Code to Large Scale Legacy C++ Applications

Means fighting for maintainability and against code rot in a sea of old C++ code, usually while simultaneously fixing bugs and adding new features.



It's a team game

You can't fight the dragon alone

Practice

Meeting resistance





Refactoring

Refactoring ist the process of restructuring existing computer code without changing its external behavior.

**Good refactoring is the key to
legacy code**

Planned refactoring: Where?

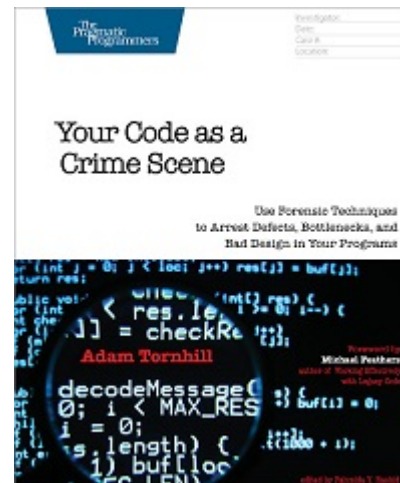
Planned refactoring: Where?

Determine the "hot spots" of the codebase

Planned refactoring: Where?

Determine the "hot spots" of the codebase

Adam Tornhill: *Your code as a crime scene*



Planned refactoring: Pick a goal

There is *still* more than enough to do. Pick a goal and work towards it.

- Determine main pain points
- No cosmetic refactoring
- No sidetracking
- Timebox or narrow down the goal

Possible goals

- Automated testing
- Less bugs
- Faster development
- Faster onboarding
- Shorter compile times
- Scaling architecture

Separate refactoring from daily maintenance

Safe refactoring

- Safe refactoring needs test coverage
- Unit tests need modularization
- Modularization is achieved through refactoring

Small, provably correct steps

Start with large scale decoupling

1. Bring a larger part of code under test
2. Refactor for decoupling using small steps
3. Repeat with finer granularity

Reassemble

- *Consciously* design a new architecture
- Fit the previously decoupled classes into that architecture
- Grow core(s) around which the new architecture can be evolved

Rewriting instead of refactoring

Rewriting can be an option for smaller components.

Consider having old and new components in parallel until all references are migrated.

Cons

- Errors that had been removed in the old version can be committed again
- Double maintenance while the old component is in place
- Complete decoupling of the component needed first

Pros

- Can start with clean code practices from scratch
- No legacy design to cope with, only the interface matters
- Can use other techniques (e.g. DSLs)

Tooling

- Builtin IDE tooling
- Static analyzers
- Refactoring aides

Problem: Tools may not be present for older compilers/IDEs.

Get help from the compiler

- E.g. when renaming functions and variables
- override & final
- "Strong typedefs"
- Warnings

Consider using a newer IDE and compiler

Apart from the tooling they also support modern C++ standards

ONE DOES NOT SIMPLY

SWITCH TO ANOTHER COMPILER

Switching the compiler

- A refactoring goal on its own
- Usually smaller refactorings

Switching the compiler

- A refactoring goal on its own
- Usually smaller refactorings
- ... unless you have to get rid of proprietary frameworks

Wrap up

- Costly and long term
- Tests are important
- Team is even more important

Questions?

Thank you!

- **Blog:** Simplify C++! - www.arne-mertz.de
- **Twitter:** @arne_mertz
- **Mail:** arne.mertz@zuehlke.com

Backup

"Factor out function guided by the compiler"

```
shared_ptr<Node> createTree(TreeData const& data)
{
    auto rootData = data.root();
    auto newNode = make_shared<Node>();
    newNode->configure(rootData);
    for (auto&& subTreeData : data.children()) {
        newNode->add(createTree(subTreeData));
    }
    return newNode;
}
```

```
shared_ptr<Node> createTree(TreeData const& data) {  
    {  
        auto rootData = data.root();  
        auto newNode = make_shared<Node>();  
        newNode->configure(rootData);  
    }  
    for (auto&& subTreeData : data.children()) {  
        newNode->add(createTree(subTreeData));  
    }  
    return newNode;  
}
```



```
shared_ptr<Node> createTree(TreeData const& data) {  
    auto createNode = [&]() {  
        auto rootData = data.root();  
        auto newNode = make_shared<Node>();  
        newNode->configure(rootData);  
        return newNode;  
    };  
    auto newNode = createNode();  
    for (auto&& subTreeData : data.children()) {  
        newNode->add(createTree(subTreeData));  
    }  
    return newNode;  
}
```

```
shared_ptr<Node> createTree(TreeData const& data) {  
    auto createNode = [](){  
        auto rootData = data.root();  
        auto newNode = make_shared<Node>();  
        newNode->configure(rootData);  
        return newNode;  
    };  
    auto newNode = createNode();  
    for (auto&& subTreeData : data.children()) {  
        newNode->add(createTree(subTreeData));  
    }  
    return newNode;  
}
```

```
shared_ptr<Node> createTree(TreeData const& data) {  
    auto createNode = [](TreeData const& data){  
        auto rootData = data.root();  
        auto newNode = make_shared<Node>();  
        newNode->configure(rootData);  
        return newNode;  
    };  
    auto newNode = createNode(data);  
    for (auto&& subTreeData : data.children()) {  
        newNode->add(createTree(subTreeData);  
    }  
    return newNode;  
}
```

```
auto createNode(TreeData const& data) {  
    auto rootData = data.root();  
    auto newNode = make_shared<Node>();  
    newNode->configure(rootData);  
    return newNode;  
}  
  
shared_ptr<Node> createTree(TreeData const& data) {  
    auto newNode = createNode(data);  
    for (auto&& subTreeData : data.children()) {  
        newNode->add(createTree(subTreeData));  
    }  
    return newNode;  
}
```

```
auto createNode(NodeData const& data) {  
    auto newNode = make_shared<Node>( );  
    newNode->configure(data);  
    return newNode;  
}
```

```
shared_ptr<Node> createTree(TreeData const& data) {  
    auto newNode = createNode(data.root( ));  
    for (auto&& subTreeData : data.children( )) {  
        newNode->add(createTree(subTreeData));  
    }  
    return newNode;  
}
```