Bringing Clean Code to Large Scale Legacy Applications

@arne_mertz

Bringing Clean Code to Large Scale Legacy C++ Applications

@arne_mertz

"What do you mean?"

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

...the speaker/writer consider outdated and/or
...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


@arne_mertz

"I've read the book, but there is not much in it that we can use for C++."

- KISS
- S.O.L.I.D.
- DRY

Performance is important.

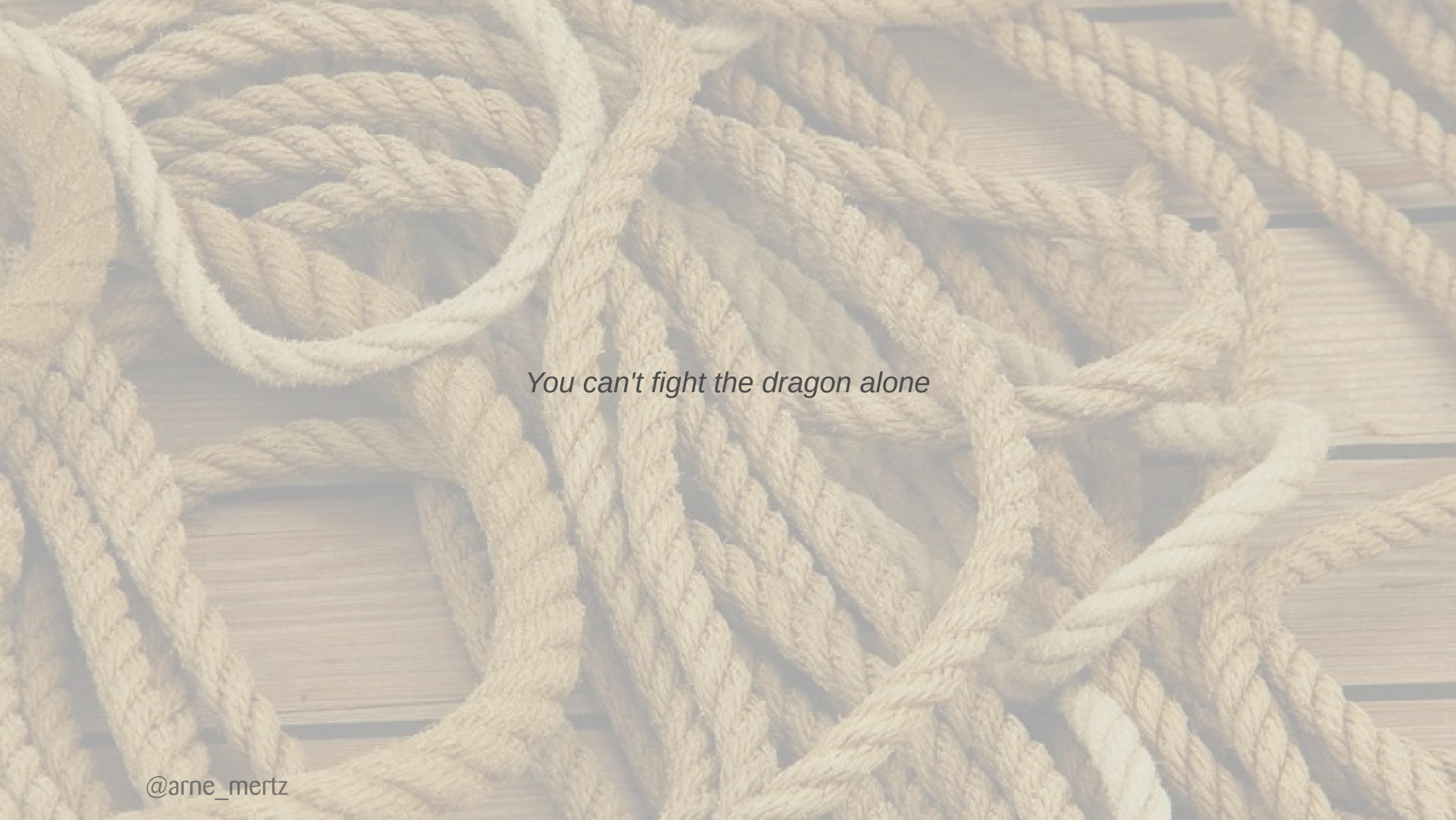
that does not mean we have to optimize every little piece of code.

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

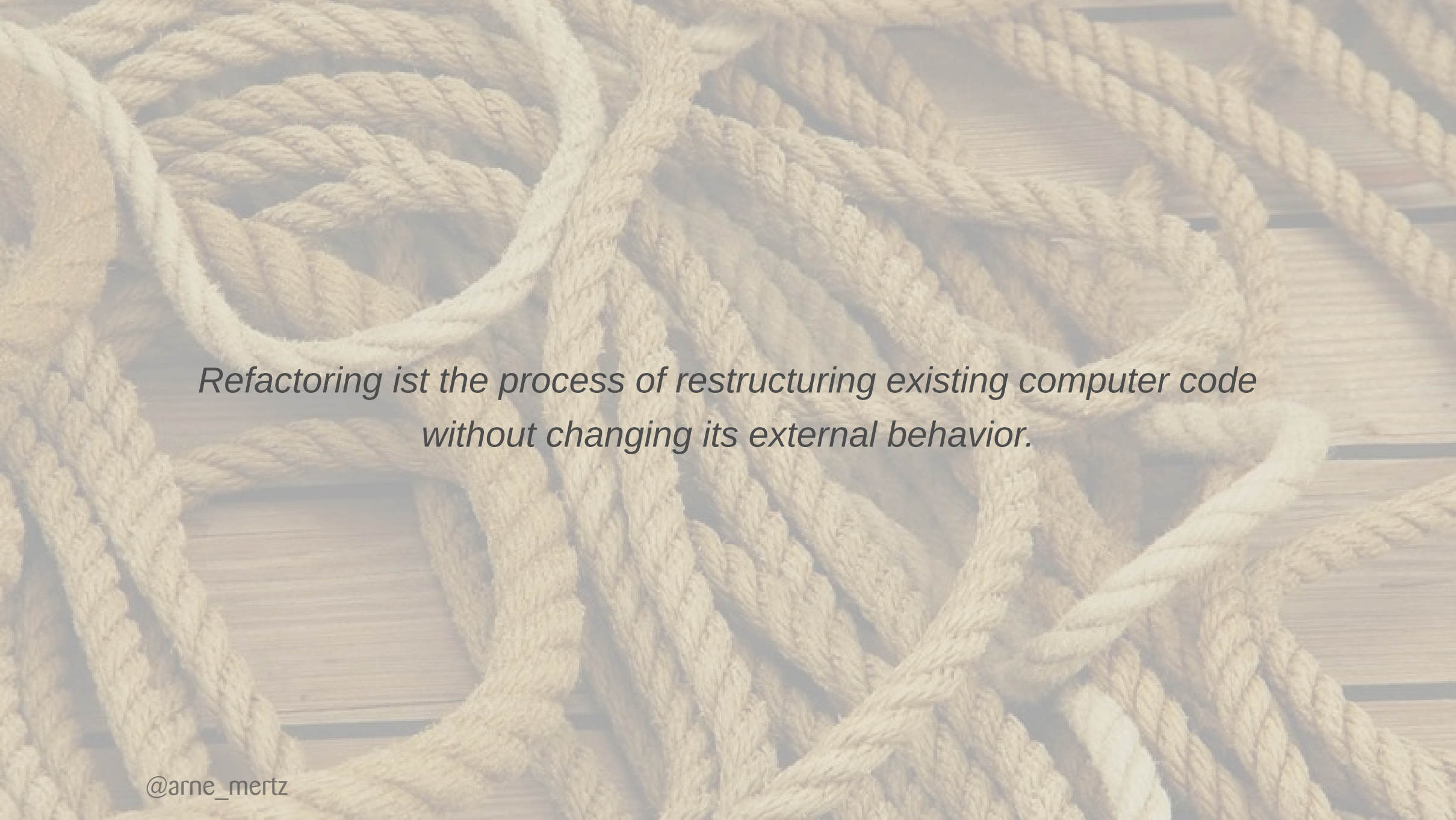
- ... use the right data structures and algorithms.
- ... trust the optimizer.

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

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


You can't fight the dragon alone
@arne_mertz


@anne_pierz


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

Determine the "hot spots" of the codebase

Determine the "hot spots" of the codebase

Adam Tornhill: *Your code as a crime scene*

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

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

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

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

The legacy codebase may have grown without refactoring the architecture

- High coupling
- Original architecture is only present as misleading names

- If there are no modules, don't pretend to have them
- Remove misleading namespace borders
- Take apart collections of functionality that is not related
- Disassemble before reassembling the parts

- *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 can be an option for smaller components.

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

- 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

- 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)

- Builtin IDE tooling
- Static analyzers
- Refactoring aides

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

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


ONE DOES NOT SIMPLY
SWITCH TO ANOTHER COMPILER

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

- E.g. when renaming functions and variables
- `override` & `final`
- Strong types with explicit conversions
- Warnings and errors

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

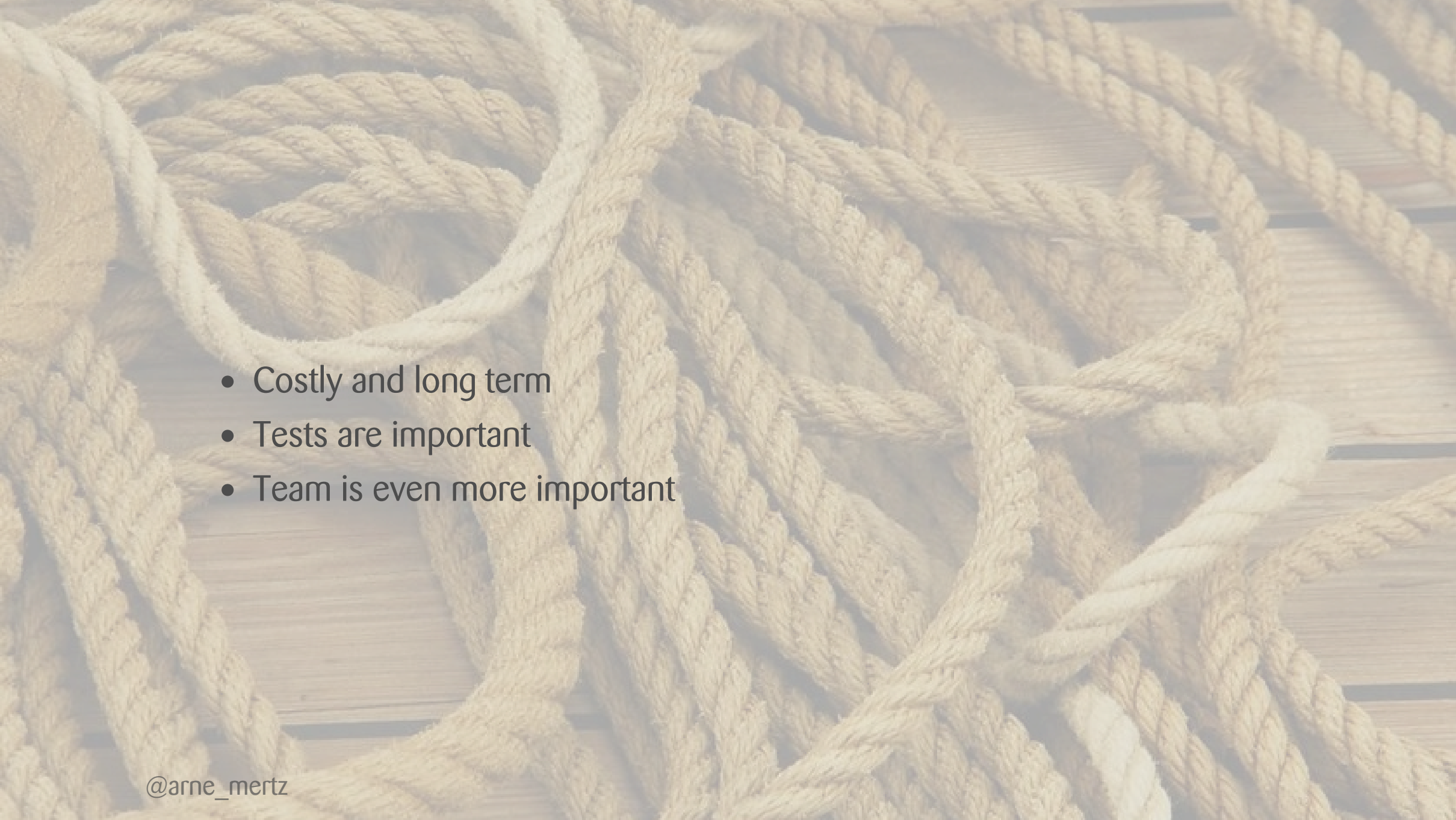
```
shared_ptr<Node> createTree(TreeData const& data) {
  auto createNode = [](TreeData const& data){
    auto rootData = data.root();
    auto newNode = make_shared<Node>();
    newNode->configure(rootData);
    return newNode;
  };
  auto newNode = createNode(data);
  for (auto&& subTreeData : data.children())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



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

zühlke
empowering ideas

- Simplify C++! - www.arne-mertz.de
- @arne_mertz
- arne.mertz@zuehlke.com
- #include<C++> Discord (www.includecpp.org)