Back To Basics
Refactoring

AMIR KIRSH

Cppcon
The C++ Conference

2025
September 13 - 19

About me

Lecturer
Academic College of Tel-Aviv-Yaffo and Tel-Aviv University

Co-Organizer of the **CoreCpp** conference and meetup group

Trainer and Advisor
(C++, but not only)

Let’s do some Code Review

What would you say about that:

What would you say about that:

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

What would you say about that:

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

- duplication: *players[i]*
- can use range-based for loop?

What would you say about that:

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

- negation makes conditions hard to read and maintain
- negation with || / && makes it even more complicated

What would you say about that:

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

it’s not so clear what do we check here

What would you say about that:

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

we access a data member,
better make it a function
(to protect the member and to
allow future logical changes)

Let’s refactor it!

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

Let’s refactor it! *Step 1*

```
auto& player = players[i];
if ( !player || !player.isAlive ) {
    // ...
}
```

if it’s in a loop, we may later change it to a range-based for

Let’s refactor it! *Step 2*

```
auto& player = players[i];
if ( !player || player.isDead() ) {
    // ...
}
```

make it a function call,
also avoiding negation
if possible

Let’s refactor it! *Step 3*

```
auto& player = players[i];
if ( player.isNotActive() || player.isDead() ) {
    // ...
}
```

prefer function with names over
operators overloading
(especially when the operator’s
action is not clear)

Let’s refactor it! *Step 4*

```
auto& player = players[i];
if ( player.isNotActiveOrDead() ) {
    // ...
}
```

in case isNotActive and isDead
are commonly checked together,
merge them into a single function

Let’s refactor it! *Step 4*

```
auto& player = players[i];
if ( player.isNotActiveOrDead() ) {
    // ...
}
```

in case isNotActive and isDead are commonly checked together, merge them into a single function

this may seem to contradict “Single Responsibility”, but if there is a continuous need for this check, it *is* a “Single Responsibility”

Original

```
if ( !players[i] || !players[i].isAlive ) {
    // ...
}
```

Refactored

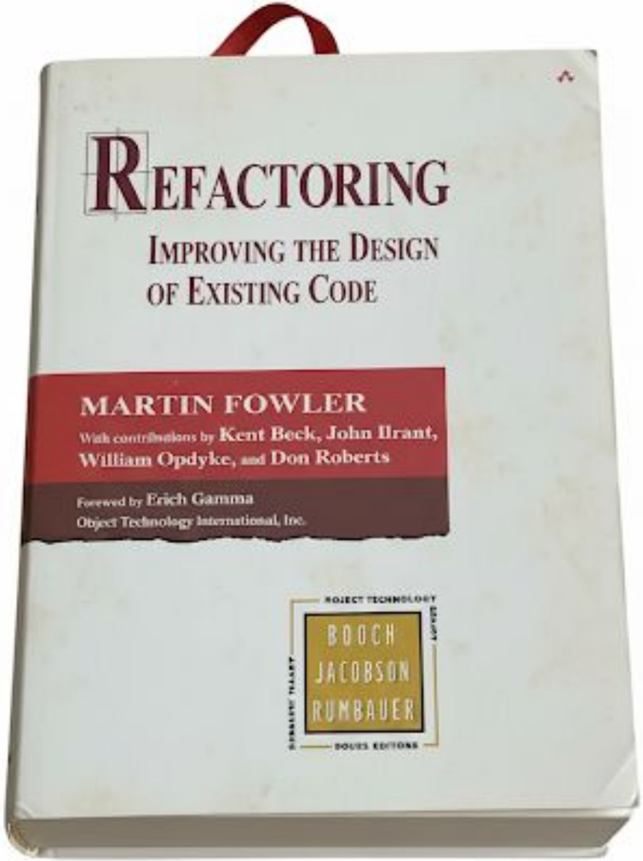
```
auto& player = players[i];
if ( player.isNotActiveOrDead() ) {
    // ...
}
```

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

https://refactoring.com

Martin Fowler


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BOOCH
JACOBSON
RUMBAUGH

The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION
REF
IMPRO
OF EX
MARTIN F
William Opdyke
Foreword by Erich Gamma
BOOCH
JACOBSON
RUMBAUGH

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure **without changing its external behavior.**

https://refactoring.com

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure **without changing its external behavior.**

https://refactoring.com

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure **without changing its external behavior.**

https://refactoring.com

How can we tell?

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure **without changing its external behavior.**

https://refactoring.com

How can we tell?

Regression Testing!

(Good Unit Testing and Coverage).

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure **without changing its external behavior.**

https://refactoring.com

How can we tell?

Regression Testing!

(Good Unit Testing and Coverage).

Testing is essential for good refactoring, but is not the topic of this talk.
There are great talks on testing, e.g.:
https://www.youtube.com/results?search_query=cppcon+testing

Refactoring - Why, When and How

Refactoring - Why, When and How

Refactoring is not another word for cleaning up code.
It's aimed for **improving the health of a code-base.**

Based on: https://refactoring.com

Refactoring - Why

Refactoring - Why

1. **Support for new features and future code changes**
   Adapt code structure for new requirements and upcoming code changes, **before** implementing the new changes.

Refactoring - Why

1. **Support for new features and future code changes**
   Adapt code structure for new requirements and upcoming code changes, **before** implementing the new changes.
2. **Improve readability and maintainability**
   Make the code easier to understand, modify, and extend by others and future self.

Refactoring - Why

1. **Support for new features and future code changes**
   Adapt code structure for new requirements and upcoming code changes, **before** implementing the new changes.
2. **Improve readability and maintainability**
   Make the code easier to understand, modify, and extend by others and future self.
3. **Performance improvements**

Refactoring - Why

1. **Support for new features and future code changes**
   Adapt code structure for new requirements and upcoming code changes, **before** implementing the new changes.
2. **Improve readability and maintainability**
   Make the code easier to understand, modify, and extend by others and future self.
3. **Performance improvements**
4. **Make the code less bug-prone and prevent future bugs**
   Identify areas where code invites bugs and prevent them through clearer, more consistent code, that better adheres to best practices.

Refactoring - Why

1. **Support for new features and future code changes**
   Adapt code structure for new requirements and upcoming code changes, **before** implementing the new changes.
2. **Improve readability and maintainability**
   Make the code easier to understand, modify, and extend by others and future self.
3. **Performance improvements**
4. **Make the code less bug-prone and prevent future bugs**
   Identify areas where code invites bugs and prevent them through clearer, more consistent code, that better adheres to best practices.

Refactoring - Why

Reduce Technical Debt

Technical debt incurs interest in the form of harder work and additional effort for every new feature or bug fix.

Refactoring - When

Refactoring - When (1)

Refactor on change, not on whim

When the code is about to change (e.g., adding features, fixing bugs).

Don’t polish stable code

If code just works and won’t change, leave it be. Even if it’s a bit smelly.

Refactoring - When (2)

Refactor based on actual benefits

Focus on eliminating code smells, reducing complexity, and improving maintainability, readability, or performance, as well as aligning with the project's conventions and guidelines.

Don't refactor based on subjective preferences

Avoid changes driven only by personal style or aesthetics.

Back to our code

Back to our code

```
for (size_t i = 0; i < players.size(); ++i) {
    auto& player = players[i];
    if ( player.isNotActiveOrDead() ) {
        continue;
    }
    action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(i);
    }
    // ...
}
```

Back to our code

```
for (size_t i = 0; i < players.size(); ++i) {
     auto& player = players[i];
     if ( player.isNotActiveOrDead() ) {
         continue;
     }
     action = player->getAction();
     if (action == ActionRequest::GetInfo) {
         handleInfoRequest(i); // we need i here
     }
     // ...
}
```

we prefer range-based-for,
but we need *i* below

Back to our code - with range-based for

```
size_t index = -1; // yes it's valid
for (auto& player : players) {
     ++index; // index count starts from 0
    if ( player.isNotActiveOrDead() ) {
        continue;
    }
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    // ...
}
```

Back to our code - with range-based for, C++20

```
// using C++20 init-statement in range-based for
for (size_t index = -1; auto& player : players) {
     ++index;
    if ( player.isNotActiveOrDead() ) {
        continue;
    }
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    // ...
}
```

better scoping for index

Back to our code - with C++23

```
for (auto&& [index, player] : players | std::views::enumerate) {
    if ( player.isNotActiveOrDead() ) {
        continue;
    }
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    // ...
}
```

using C++23 enumerate

Back to our code - with C++23, views::filter

```
for (auto&& [index, player] :
         players | std::views::enumerate
         | std::views::filter([](auto&& index_player) {
               auto&& [_, the_player] = index_player;
               return the_player.isAlive();
           })) {
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    // ...
}
```

Back to our code - with C++23, views::filter

```
for (auto&& [index, player] :
         players | std::views::enumerate
         | std::views::filter([](auto&& index_player) {
               auto&& [_, the_player] = index_player;
               return the_player.isAlive();
           })) {
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    // ...
}
```

don’t like it?
you don’t have to use it, but
you will need to be able to
read it when moving to C++20

Looping over all live players is common

Looping over all live players is common

Make it a function by itself!

Looping over all live players is common

```
template <typename Action>
void for_each_live_player(Action action) {
    for (auto&& [index, player] :
         players | std::views::enumerate
         | std::views::filter([](auto&& index_player) {
               auto&& [_, the_player] = index_player;
               return the_player.isAlive();
           })) {
        action(index, player);
    }
}
```

The loop machinery is now hidden in a function.
Calling the function creates much more declarative code.

Handling the actions

Handling the actions

```
// looping over all players performing their actions
for_each_live_player([](size_t index, auto& player) {
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    else if (action == ActionRequest::Shoot) {
        handleShootRequest(index);
    }
    // ...
});
```

what do you say about this?

Handling the actions

```
// looping over all players performing their actions
for_each_live_player([](size_t index, auto& player) {
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    else if (action == ActionRequest::Shoot) {
        handleShootRequest(index);
    }
    // ...
});
```

what do you say about this?

maybe a switch?

Handling the actions

```
// looping over all players performing their actions
for_each_live_player([](size_t index, auto& player) {
    ActionRequest action = player->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(index);
    }
    else if (action == ActionRequest::Shoot) {
        handleShootRequest(index);
    }
    // ...
});
```

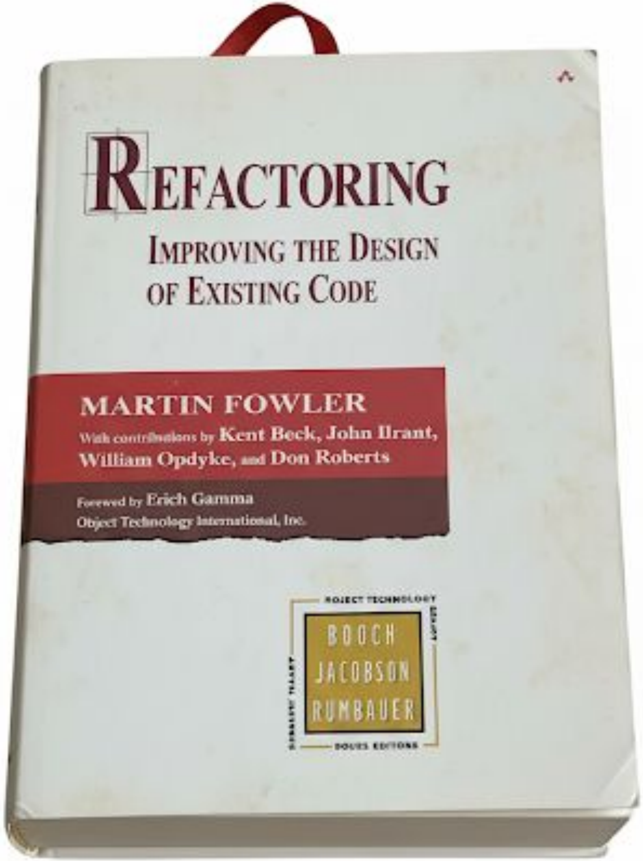
what do you say about this?

maybe a switch?
maybe command pattern?

Handling the actions - Command Pattern

```
// looping over all players performing their actions
for_each_live_player([](size_t index, auto& player) {
    auto action = player->getAction(); // a command instead of just an enum
     action.execute(index);
    // ...
});
```

code is more concise and declarative,
easier to read and maintain
move responsibility to other classes
(e.g. command class)
easier to handle future actions



REFACTORING

IMPROVING THE DESIGN
OF EXISTING CODE

MARTIN FOWLER
With contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

Catalog of Code Smells and Refactoring Options

Catalog of Code Smells and Refactoring Options

Why a catalog?

- Provides a **shared language** for discussing code quality.
- Connects each smell to **established refactoring options**.
- Makes problem detection **systematic and repeatable**.
- Provides **actionable remedies**, instead of reinventing the wheel.
- Supports **education, onboarding, and code reviews**.

Catalog of Code Smells and Refactoring Options

It’s a living catalog, not a finished book

Software practices and languages evolve - so do code smells and their mitigations.

Partial list of code smells and mitigations

1. Long Method

1. Long Method

Long methods are harder to maintain and tend to be more bug prone.

1. Long Method

Long methods are harder to maintain and tend to be more bug prone.

Common Refactorings:

- *Extract Method* (i.e. create helper methods)
- *Replace Temp with Query*
  (i.e. use functions return values in expressions instead of storing them in local variable)
- *Replace Method with Method Object*
  (i.e. turn a long or complex method into its own class, encapsulating state and logic)
- *Decompose Conditional*
  (i.e. break complex conditions into smaller, well-named methods, for clarity, reuse and conciseness)

Long Method Example

```
for (size_t i = 0; i < players.size(); ++i) {
    if ( !players[i] || !players[i].isAlive ) {
        continue;
    }
    action = players[i]->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(i);
    }
    else if (action == ActionRequest::Shoot) {
        handleShootRequest(i);
    }
    // ...
}
```

Long Method Example

Long Method Example

```
for_each_live_player([](size_t index, auto& player) {
    action = players[i]->getAction();
    if (action == ActionRequest::GetInfo) {
        handleInfoRequest(i);
    }
    else if (action == ActionRequest::Shoot) {
        handleShootRequest(i);
    }
    // ...
});
```

applying “Extract Method”

Long Method Example

```
for_each_live_player([](size_t index, auto& player) {
    auto action = player->getAction();
     action.execute(index);
    // ...
});
```

applying “Replace Method with Method Object” (using command pattern)

2. Long Parameter List

2. Long Parameter List

Long parameter lists are harder for both the caller and the function maintainer and tend to be more bug prone on both sides.

2. Long Parameter List

Long parameter lists are harder for both the caller and the function maintainer and tend to be more bug prone on both sides.

Common Refactorings:

- *Introduce Parameter Object* (i.e. pack a few parameters into a class)
- *Preserve Whole Object*
  (i.e. pass the entire object to a method instead of extracting and passing individual fields)
- *Replace Parameter with Method*
  (i.e. avoid sending the parameter and compute its value via a method in the called method)

2. Long Parameter List

Long parameter lists are harder for both the caller and the function maintainer and tend to be more bug prone on both sides.

Common Refactorings:

- *Introduce Parameter Object* (i.e. pack a few parameters into a class)
- *Preserve Whole Object*
  (i.e. pass the entire object to a method instead of extracting and passing individual fields)
- *Replace Parameter with Method*
  (i.e. avoid sending the parameter and compute its value via a method in the called method)

be very careful with these two

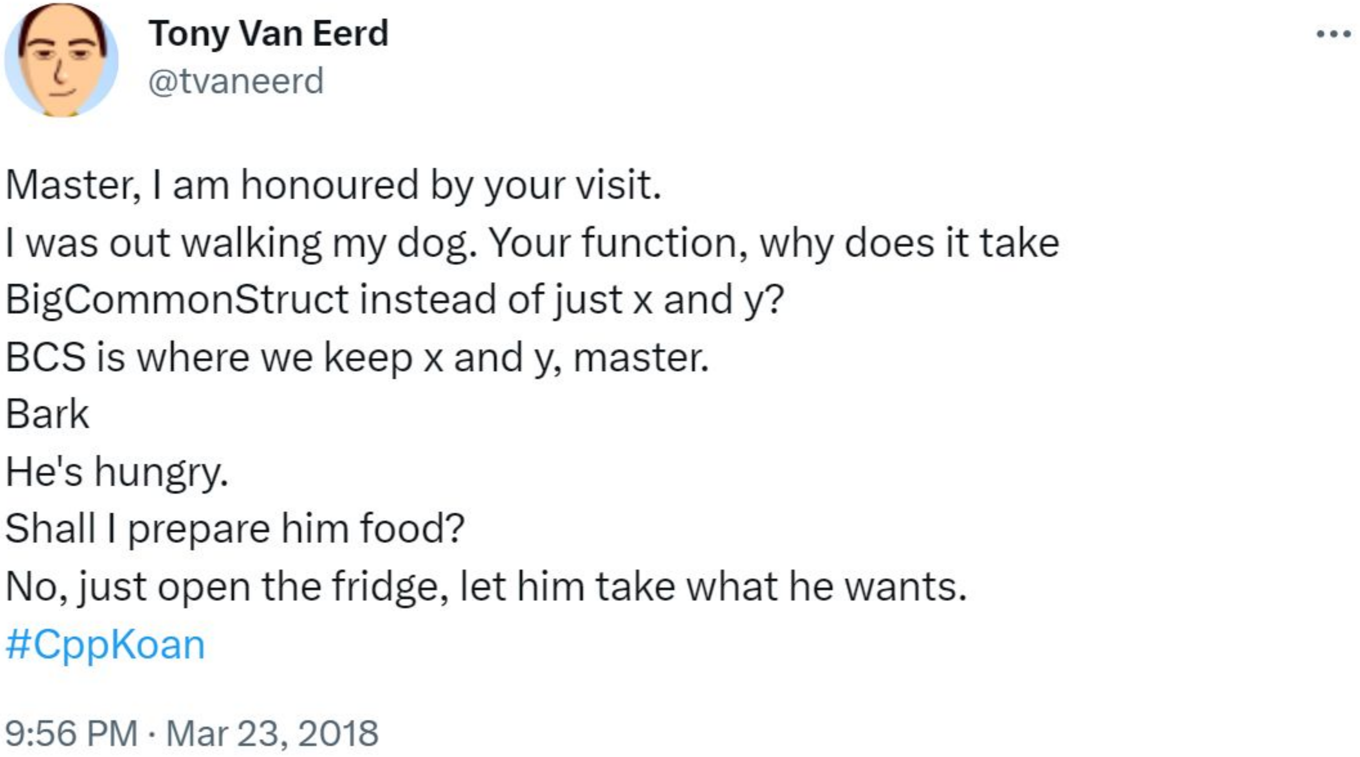
Preserve Whole Object

Might be a code smell by itself - why?

Preserve Whole Object

Might be a code smell by itself - hurts encapsulation and decoupling

Tony Van Eerd’s Koan - The Master and his Dog

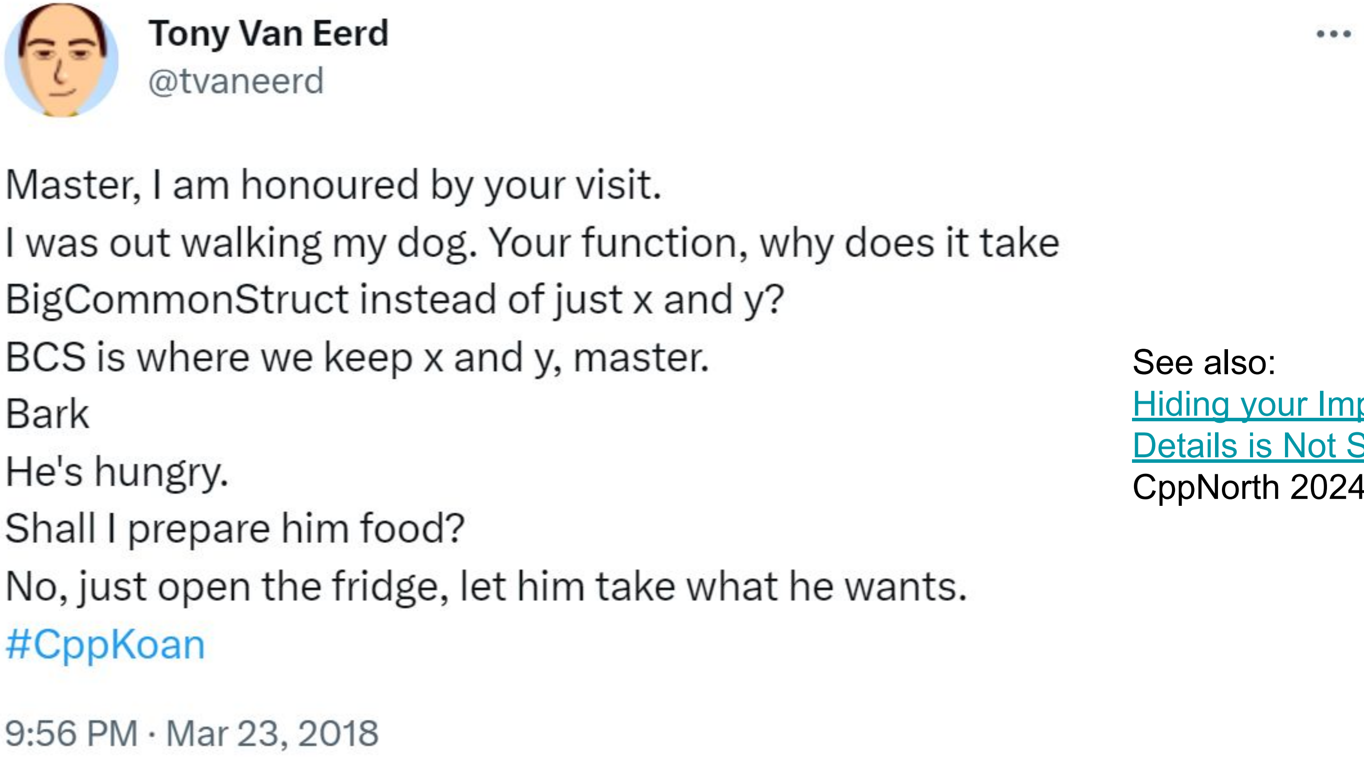


Tony Van Eerd
@tvaneerd

Master, I am honoured by your visit.
I was out walking my dog. Your function, why does it take BigCommonStruct instead of just x and y?
BCS is where we keep x and y, master.
Bark
He's hungry.
Shall I prepare him food?
No, just open the fridge, let him take what he wants.
#CppKoan

9:56 PM · Mar 23, 2018

Tony Van Eerd’s Koan - The Master and his Dog



Tony Van Eerd
@tvaneerd

Master, I am honoured by your visit.
I was out walking my dog. Your function, why does it take BigCommonStruct instead of just x and y?
BCS is where we keep x and y, master.
Bark
He's hungry.
Shall I prepare him food?
No, just open the fridge, let him take what he wants.
#CppKoan

9:56 PM · Mar 23, 2018

See also:
[Hiding your Implementation Details is Not So Simple -](#) CppNorth 2024

Replace Parameter with Method

Might be a code smell by itself - why?

Replace Parameter with Method

Might be a code smell by itself - may hurt encapsulation and decoupling

Proper way (doing it right): ***dependency injection***

3. Large Class

3. Large Class

A class that has grown too big, taking on too many responsibilities and becoming hard to maintain.

3. Large Class

A class that has grown too big, taking on too many responsibilities and becoming hard to maintain.

Common Refactorings:

- *Extract Class*
  (i.e. move part of a class’s fields and methods into a new class to separate responsibilities)
- *Extract Subclass* (i.e. a subclass may handle specialized responsibilities or behavior)
- *Extract Superclass* (i.e. create a superclass taking up some responsibilities)
- *Replace Data Value with Object* (i.e. turn a single field into an object)

4. Comments

4. Comments

Comments can be a code smell if better coding would remove the need for redundant documentation.

4. Comments

Comments can be a code smell if better coding would remove the need for redundant documentation.

Common Refactorings:

- *Extract Methods* (calling well-named methods reduces the need for comments)
- *Introduce Assertions*
  (replace comments documenting preconditions with actual assertions, or C++26 contracts)

5. Switch Statements and Complex Conditions

5. Switch Statements and Complex Conditions

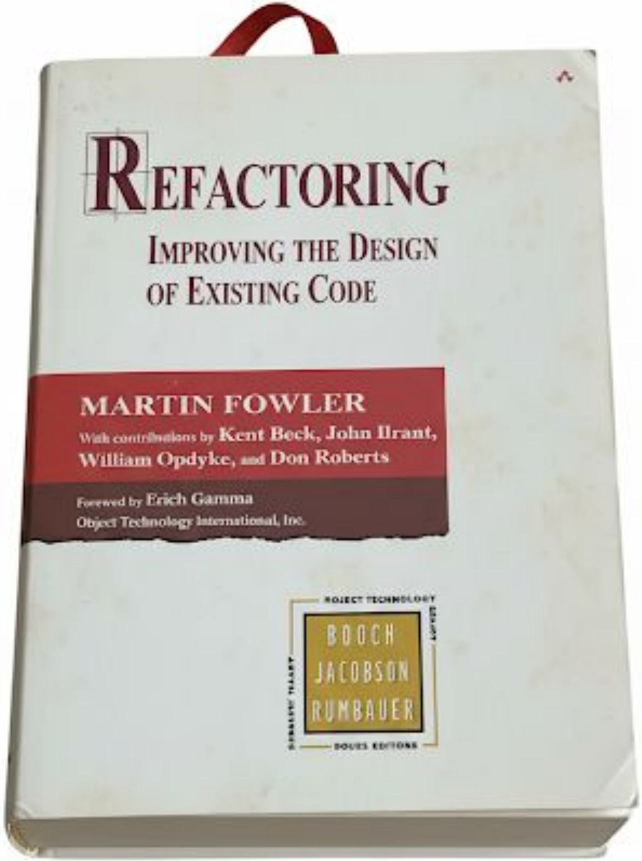
Complex conditional structures may become long and complicated to maintain, also violating the Open/Closed Principle, making the code hard to extend.

5. Switch Statements and Complex Conditions

Complex conditional structures may become long and complicated to maintain, also violating the Open/Closed Principle, making the code hard to extend.

Common Refactorings:

- *Replace Conditional with Polymorphism* (e.g. calling undo on an operation without the need to check for its type, using the command pattern)
- *Replace Type Code with State/Strategy* (i.e. instead of holding in a class an enum and following its type to select an operation, hold a state or a strategy)
- *Introduce Null Object* (i.e. instead of holding nullptr, hold an object that models the behavior in case of nullptr)


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BOOCH
JACOBSON
RUMBAUGH

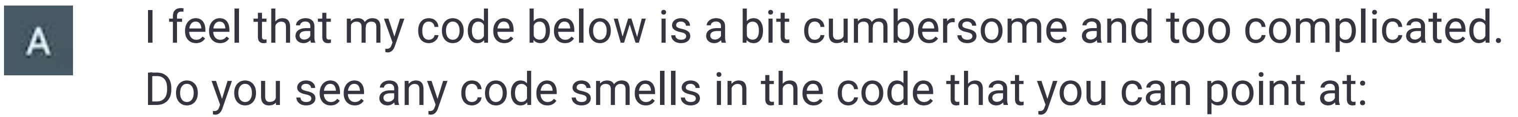
Above was just a partial list…

The Refactoring book lists:

- 22 code smells
- 72 refactorings

Getting AI to assist with identifying code smells

Getting AI to assist with identifying code smells



I feel that my code below is a bit cumbersome and too complicated.
Do you see any code smells in the code that you can point at:

```
void MyGameManager::processPlayerActions() {
 std::vector<ActionRequest> actions;
 ActionRequest action;
 roundActions.clear();
 // Get actions from all players
 for (size_t i = 0; i < players.size(); ++i) {
  if (!players[i] || !playerStates[i].isAlive) {
   actions.push_back(ActionRequest::DoNothing);
   if (!roundActions.empty()) {
    roundActions += ", ";
   }
   roundActions += alreadyKilledStr;
   continue;
  }
  action = players[i]->getAction();
  actions.push_back(action);
  // Handle GetBattleInfo action immediately
  if (action == ActionRequest::GetBattleInfo) {
   handleBattleInfoRequest(i);
  }
  // Format action for output
  if (!roundActions.empty()) {
   roundActions += ", ";
  }
  roundActions += UserCommon::actionToString(action);
 }
 // Process each action
 for (size_t i = 0; i < players.size(); ++i) {
  if (!players[i] || !playerStates[i].isAlive) {
   continue;
  }
```

Getting AI to assist with identifying code smells

Yes — there are some clear code smells in this function.

1. Mixed Responsibilities (God Method smell)

The method does *too many things at once*:

- Gathering actions
- Handling a special-case action (`GetBattleInfo`)
- Building an output string (`roundActions`)
- Processing actions

👉 This violates *Single Responsibility Principle*. Splitting into helper functions (`collectActions`, `formatRoundActions`, `processActions`) would improve readability.

Getting AI to assist with identifying code smells

2. Duplicate Conditional Logic

You repeat checks like:

```
    if (!players[i] || !playerStates[i].isAlive)
```

in both loops. That’s duplication and adds noise. Better to encapsulate:

```
bool isAlive(size_t i) const;
```

Getting AI to assist with identifying code smells

3. Temporary *action* Variable Reused

```
ActionRequest action;
...
action = players[i]->getAction();
```

Reusing the same variable across different contexts is confusing.
Prefer defining `action` closer to usage.

Getting AI to assist with identifying code smells

4. Hidden Magic (Unclear Constant)

`alreadyKilledStr` appears out of nowhere — a smell of hidden dependency. Either pass it in or clarify context.

Getting AI to assist with identifying code smells

6. Tight Coupling Between Collection and Processing

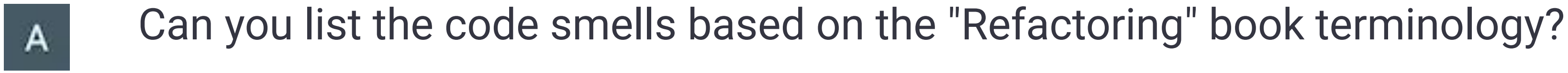
The vector `actions` is filled in one loop, then consumed in another. This coupling makes it hard to change one stage without breaking the other. Could use a pipeline style (collect → process immediately) unless deferred evaluation is needed.

Getting AI to assist with identifying code smells

...

< A few more notes >

Getting AI to assist with identifying code smells



Can you list the code smells based on the "Refactoring" book terminology?

Getting AI to assist with identifying code smells

Sure 👍 — if we map your function against the **code smells catalog** from Martin Fowler's *Refactoring* book (2nd edition, 2018), here's what shows up:

1. **Long Method –** `processPlayerActions` is doing multiple things (collecting actions, handling special cases, formatting strings, processing actions).
2. **Divergent Change –** If requirements change in *how actions are gathered*, *how strings are formatted*, or *how actions are processed*, you'll have to modify the same method in multiple, unrelated ways.
3. **Shotgun Surgery** (related to above) – Any change to action handling (new action type, new output format) risks changes scattered across this method and maybe others.
4. **Temporary Field (or Temporary Variable) –** The `ActionRequest action;` variable declared at the top is reused in multiple contexts. It exists longer than needed and obscures intent.
5. **Duplicated Code / Conditional Logic –** The check `if (!players[i] || !playerStates[i].isAlive)` appears in two places. This is a *duplicated conditional fragment*.
6. **Comments –** Several comments merely restate the code (`// Get actions from all players`, `// Process each action`). Fowler lists this as a smell: comments compensating for unclear code instead of improving code structure.
7. **Primitive Obsession / Magic String** – Building `roundActions` by hand with `", "` and `alreadyKilledStr` is a manual, error-prone approach. This falls under *primitive obsession* (using strings instead of proper domain abstractions).
8. **Feature Envy** (mild) – The loop both *asks* players for actions and *decides* how to format them, which could arguably belong more to the `Player` or a `RoundFormatter` than to `MyGameManager`.

Getting AI to assist with identifying code smells

Would you like me to **show a refactored version** of the code, annotated with which specific *refactorings* (from the book) would address each smell?

Getting AI to assist with identifying code smells

Refactored Version:

```
void MyGameManager::processPlayerActions() {
    auto actions = collectActions();
    processActions(actions);
}

// ----------------- Helpers -----------------

// Extract Method → reduces Long Method, improves readability
std::vector<ActionRequest> MyGameManager::collectActions() {

<…>
```

not always the best refactoring suggestions, apply your own logic at this stage!

Additional C++ Refactorings

1. Move from Weak Types to Strong Types

Old code:

```
void repeat(int count, int waitMs, Func&& action)
```

New code:

```
void repeat(int count, std::chrono::milliseconds wait, Func&& action)
```

Or:

```
void repeat(int count, std::chrono::duration<Rep, Period> wait, Func&& action)
```

2. Move from User Allocations to Lib Allocations

Do not use new and delete in user code. Use containers and smart pointers.

Prefer to hide your use of smart pointers as an internal design decision.

Old code:

```
auto e = new Sum (new Number(3), new Number(2));
cout << *e << " = " << e->eval() << endl;
delete e;
```

New code:

```
auto e = Sum(3, 2);
cout << e << " = " << e.eval() << endl;
```

See also:
[Six Ways for Implementing Math Expressions Calculator](#)
CppCon 2023

3. Move from Pointers and References to Values

Value semantics simplifies ownership and reduces risks of null / dangling resources.

Any example in mind?

3. Move from Pointers and References to Values

Value semantics simplify ownership and reduces risks of null / dangling resources.

Old code:

```
auto e = new Sum (new Number(3), new Number(2));
cout << *e << " = " << e->eval() << endl;
delete e;
```

New code:

```
auto e = Sum(3, 2);
cout << e << " = " << e.eval() << endl;
```

See also:
Six Ways for Implementing Math Expressions Calculator
CppCon 2023

4. Prefer Standard Algorithms Over Raw Loops

Standard algorithms usually better express intent, reduce boilerplate, and minimize errors. In some cases they may also offer better performance.

See:

GoingNative 2013 C++ Seasoning - Sean Parent famous talk presenting the motto “No Raw Loops”

On performance:

Raw loops for performance? | Sandor Dargo's Blog (and Sandor’s talk on Friday)

5. Avoid Macros If Possible

Replace Macros with Constants / Inline Functions to improves scoping, type safety, readability, and debugging.

6. Use RAII

Replace manual resource management and other finalization operations (such as releasing a lock) with RAII, to ensure the closing operation, avoiding leaks, dangling resources, deadlocks and other related issues.

Old code:

```
mtx.lock();
resource.write(something); // critical section
mtx.unlock();
```

New code:

```
std::lock_guard<std::mutex> lock(mtx); // RAII lock
resource.write(something); // critical section
```

Summary

Summary (1)

Continuous Refactoring is required to improve code readability, maintainability, and reliability and to reduce technical debt.

Summary (2)

Refactor code that you are actively working on - maintaining, adding features, fixing bugs.

Do not refactor code that is otherwise untouched.

Summary (3)

Refactor for a clear reason - not out of intuition, whim, or general aesthetic preferences.

- Code is hard to understand and maintain.
- You identify the code smells that need to be fixed.

Summary (4)

Refactor by small, incremental and testable changes.

- Apply well-known refactoring patterns.
- Follow modern C++ features and best practices:
  RAII, use of smart pointers, strong types, no raw loops.
- Try to make your code more declarative.
- Hide your complexities.

Summary (5)

Coding with AI - code quality and productivity doesn't necessarily improved:

[Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality]

[Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity]

A key message:

It's the programmer's responsibility to "own" the code and maintain its quality.

Refactoring becomes even more important in the AI era.

Summary (6)

You may use AI for refactoring but probably just as a helper tool.

The code is yours, you should decide which refactoring suggestions are good and which are not, and apply your own refactorings without relying only on AI.

Relying only on AI to refactor your code properly will (Sep-2025) result with messy code that still needs refactoring.

Any questions before we conclude?

Thank you!

```
void conclude(auto&& greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });


// Comments, feedback: kirshamir@gmail.com
```