C++ Unit Testing

C++ Unit Testing

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad@mastodon.social (🦣)

Slides:

https://github.com/PeterSommerlad/CPPCourseUnitTesting/tree/main/

My philosophy

Less Code

=

More Software

Speaker notes

I borrowed this philosophy from Kevlin Henney.

© 2024 Peter Sommerlad

What is in C++

Speaker notes

In the C++ Introduction we look solely on parts of C++ that are relevant for App development.

This course we start to look at C++ parts that cross over to the library language, but mostly to cover things that might have been used in existing applications without need (or that are no longer needed in more recent C++ versions). C++ Expert will dive more deeply into the “dirty” language features that might be needed for library code, but that requires careful attention to detail to not be misused.

C++ Resources

- ISO C++ standardization
- C++ Reference
- Compiler Explorer
- C++ Core Guidelines
- Hacking C++ reference sheets
- Our Introduction Exercises
- Our Advanced Exercises
- Our Expert Exercises

Speaker notes

complete link texts, because hyperlinks vanish from PDF

- https://isocpp.org/
- https://en.cppreference.com/w/
- https://compiler-explorer.com/
- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- https://hackingcpp.com/
- https://github.com/PeterSommerlad/CPPCourseIntroduction
- https://github.com/PeterSommerlad/CPPCourseAdvanced
- https://github.com/PeterSommerlad/CPPCourseExpert

© 2024 Peter Sommerlad

C++ Genealogy

C++98
initial standardized version

C++03
bug-fix of C++98, no new features

C++11
major release (known as C++0x): lambdas, constexpr, threads, variadic templates

C++14
fixes and extends C++11 features: variable templates, generic lambdas

C++17
(almost) completes C++11 features: CTAD, better lambdas

C++20
new major extension: concepts, coroutines, ranges, modules, constexpr “heap”

C++23
fixes/extends C++20, generator, mdspan, import std, std::print/ln, expected

Speaker notes

The ISO standardization process now uses a three year release cycle. However, for major releases it takes time for implementors to provide the new language features and library. Most C++ compilers do not yet have fully implemented C++20 and some implementation diverge in subtle details, because the specification is inaccurate. This is a typical chicken-egg problem: * compilers will only implement language features in production quality, when they are part of the standard * specification in the standard is only scrutinized when independent compiler/library authors implement them

C++20 modules and coroutines are not yet generally usable across compilers. Concepts are.

C++23 support is also still lacking. May be 2025 we can start using modules in a portable way.

C++26 is in its making.

BTW, I have contributed features for all C++ standards since 2011.

© 2024 Peter Sommerlad

In this course

- Modern C++17 and some of C++20/23
- C++ Unit Testing Easier library (CUTE)

Speaker notes

While you might prefer other IDEs, I chose the Eclipse-CDT-based IDE Cevelop that my former team at IFS Institute for Software created.

Cevelop provides some checkers for typical beginner mistakas as well as good support for writing Unit Tests with my test framework CUTE (https::/cute-test.com). The latter is important, because CUTE relies on the IDE to automatically generate test registration code.

The principles taught using CUTE will port well to all other unit testing frameworks.

© 2024 Peter Sommerlad

Not in this course

- *All* of C++
- Building C++ on the command line
- C++ build systems (cmake, scons, make)
- C++ package manager (conan, vcpkg)
- Other C++ IDEs (vscode, clion)

Speaker notes

You can observe the command line used by Cevelop in its Console window.

While we look also at features of C++20, we will ignore *modules* and *coroutines*. Also not part of the introduction are the limitations of previous C++ language standards.

© 2024 Peter Sommerlad

A minimal valid C++ program

```
int main(){}
```

https://godbolt.org/z/7eM9ja5dj

Speaker notes

In C++ we code the functionality of our program within functions.

Each function has a

- return type (`int`)
- name (`main`) - where `main` is special
- pair of parentheses (`()`) enclosing parameters - even if none is defined
- body (`{}`(,cpp)) - statements enclosed in curly braces

The `main()` function is special, because

- it is the only one, where the return statement can be omitted,
- it defines the code that makes the whole program, and
- it is the only one that cannot be called from, therefore,
- it cannot be tested from within C++ code.

see also https://godbolt.org/z/7eM9ja5dj

Hello world

```cpp
//============================================================================
// Name        : hello.cpp
// Author      : Peter Sommerlad
// Version     : 0.0
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

int main() {
        cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
        return 0;
}
```

https://godbolt.org/z/bqdevaK8x

Speaker notes

What is wrong with this hello world program?

https://godbolt.org/z/bqdevaK8x

Less Code

=

More Software

Kevlin Henney and Peter Sommerlad

Speaker notes

Remember my philosophy!

© 2024 Peter Sommerlad

Hello world simplified

```
#include <iostream>

int main() {
        std::cout << "!!!Hello World!!!\n";
}
```

What is still bad with this code?

https://godbolt.org/z/c16cPdjsW

Speaker notes

The essence of the code is not easily testable, only program observation will tell its behavior

While obvious in this case, it is easily non-obvious in normal code

Also dependency to global variable makes it untestable

https://godbolt.org/z/c16cPdjsW

© 2024 Peter Sommerlad

Producing output

```
        out << "!!!Hello World!!!\n";
```

- Output is via `std::ostream` objects and uses the `<<` operator.
- `<<` can be chained to output multiple values:

```
  out << "Hello " << name << ", how are you?\n";
```


Speaker notes

Overloading operators is an essential features of C++ that we will look at. The order of the arguments is significant. For now, the built-in meaning of `<<` is “left bit shift” but for stream objects it means output.

Hello world alternative

```cpp
#include <fmt/core.h>

int main() {
        fmt::print("Hello {}!\n", "Cpp");
}
```

the fmt library provides a more modern alternative for output

Compiler Explorer: https://godbolt.org/z/av884e1Kr

C++23 version with `std::print`: https://godbolt.org/z/5Ke3GW89W

Speaker notes

C++20 provides the format library as part of its standard via

`#include <format>`

However, today, the format library provides a better alternative even for earlier C++ versions.

see https://github.com/fmtlib/fmt

Note that in Compiler Explorer you need to select the fmt library to be able to use it.

https://godbolt.org/z/av884e1Kr

A C++23 version using `std::print` looks like

https://godbolt.org/z/5Ke3GW89W

© 2024 Peter Sommerlad

Testable Code

- no hard-coded dependencies to globals (`std::cout`)
  - except for **`constexpr`** compile-time constants
- pass globals down the call chain from `main()`
- unit-test your functions
  - test also for error conditions and handling
  - test the good cases
  - but only test what can go wrong
  - always test for what went wrong once

Speaker notes

While my explanations might omit corresponding tests, all code that we write should be accompanied by automated test.

Starting with a test case, makes it easier to understand what we want to achieve with a function and as a side effect tends to reduce dependencies.

We will see and exercise how to create a testable hello world program after we looked at how C++ compilation works.

https://godbolt.org/z/YrT4sxaTa

© 2024 Peter Sommerlad

Untestable Hello world

```cpp
#include <iostream>

int main() {
        std::cout << "!!!Hello World!!!\n";
}
```

What is still bad with this code?

We cannot test anything in `main()`

Speaker notes

The essence of the code is not easily testable, only program observation will tell its behavior

While obvious in this case, it is easily non-obvious in normal code

Also dependency to global variable makes it untestable

© 2024 Peter Sommerlad

Testing Output

use a function parameter ***`std::ostream &out`***

```
void sayhello(std::ostream &out) {
        out << "!!!Hello World!!!\n";
}
```

`&` *in front of the parameter means* ***pass by reference***.
This is used for ***side effects***.

Speaker notes

We cannot test output using `std::cout`. However, `std::cout` is a specialized version of `std::ostream`. The type `std::ostringstream` can be used in tests to collect output written to an `ostream`.

Use *reference parameters* only, when there must be a side effect on the function call argument. When a function has a (non-const lvalue) reference parameter, the argument at the call site must be a variable (mutable lvalue). Since output is a side effect on a stream object and because we cannot pass a stream object to a function otherwise, it is OK. Normal Parameters of the form `type name` are “pass by value”.

The output created by the output operator `<<` depends on the type of the variable. There is no need for a matching formatting character as in C, for example.

© 2024 Peter Sommerlad

Hello world: extract function

hellomain.cpp

```cpp
#include "sayhello.h"
#include <iostream>

int main() {
        sayhello(std::cout);
}
```

hellotest.cpp

```cpp
#include "sayhello.h"

void testSayHello() {
  std::ostringstream out{};
  sayhello(out);
  ASSERT_EQUAL("Hello World\n",
    out.str());
}
```

sayhello.h

```cpp
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
#include <iosfwd>
void sayhello(std::ostream &out);
#endif /* SAYHELLO_H_ */
```

sayhello.cpp

```cpp
#include "sayhello.h"
#include <ostream>
void sayhello(std::ostream &out) {
        out << "!!!Hello World!!!\n";
}
```

https://godbolt.org/z/WWzocn9j5

Speaker notes

here we have separated the functionality from main into a function that we put into a separate translation unit and library. This way, our hello world program becomes testable.

Note, that the global variable is only used within main.

This parameter passing is sometimes referred to as “dependency injection” to confuse people

https://godbolt.org/z/WWzocn9j5

© 2024 Peter Sommerlad

Hello world: Other Frameworks

CUTE

```
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"

#include <sstream>
#include "sayhello.h"

void testSayHello() {
  std::ostringstream out{};
  sayhello(out);
  ASSERT_EQUAL("Hello World\n",
    out.str());
}

bool runAllTests(int argc, char const *argv[]) {
        cute::suite s { };
        //TODO add your test here
        s.push_back(CUTE(testSayHello));
        cute::xml_file_opener xmlfile(argc, argv);
        cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);
        auto runner = cute::makeRunner(lis, argc, argv);
        bool success = runner(s, "AllTests");
        return success;
}

int main(int argc, char const *argv[]) {
    return runAllTests(argc, argv) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Google Testhttps://godbolt.org/z/jc44x7c9M

```
#include "sayhello.h"
#include "gtest/gtest.h"
#include <sstream>
namespace {
TEST(HelloTest, sayHelloSaysHello) {
    std::ostringstream out{};
    sayhello(out);
    ASSERT_EQ("Hello, world!", out.str());
}
}
```

Catch2https://godbolt.org/z/n64q5vs4j

```
#include "sayhello.h"
#include "catch2/catch_all.hpp"
#include <sstream>
namespace {
TEST_CASE("HelloTest", "sayHelloSaysHello") {
    std::ostringstream out{};
    sayhello(out);
    REQUIRE(out.str() == "Hello, world!" );
}
}
```

Doctesthttps://godbolt.org/z/o3zjq1fnM

```
#include "sayhello.h"
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest/doctest.h"
#include <sstream>
namespace {
TEST_CASE("sayHelloSaysHello") {
    std::ostringstream out{};
    sayhello(out);
    REQUIRE(out.str() == "Hello, world!" );
}
}
```


Speaker notes

While CUTE requires manual test registration, this has an advantage to avoid the static initialization order fiasco that can hurt one when test frameworks use automatic registration through static storage duration data that is dynamically initialized.

In addition the automatic provisioning of a `main()` function by the testing frameworks requires either linking with a corresponding library (gtest main, Catch2Main) or a macro-controlled conditional compilation in the header-only library Doctest.

Regardless if `main()` is provided or not, google test and catch2 require linking to a static library (gtest, Catch2) provided by the frameworks. CUTE and doctest are header-only libraries.

CUTE as header-only testing framework relies on automatic code generation of test scaffolding and test registration by the Cevelop IDE. Nevertheless, the required test registration and main function can easily provided by hand and are visible for adaptation, i.e. by providing a different reporting syntax. Writing your own main functions for the other testing frameworks requires looking things up and adhering to the testing frameworks assumptions

What is Unit Testing

- Test anything that might break
- Test everything that does break
- New code is guilty until proven innocent
- Write at least as much test code as production code
- Run local tests with each compile
- Run all tests before check-in to repository

Speaker notes

Writing tests for your code not only improves quality, it also provides immediate feedback on the interface design decisions you take.

If it is hard writing tests for a piece of functionality, that functionality is also hard to use by its intended users.

If you don’t know what to test, you might not actually know what to implement (yet).

Whenever you think about a feature or a piece of code, also think: how can I test that I am done with it and how can I test that it does what is intended?

Whenever you get a bug report, first write tests that reproduce the bug, so that you know when you have fixed it.

That way, those test will also help to avoid re-introducing the bug again with future changes.

© 2024 Peter Sommerlad

Why Unit Testing / Test Automation

- reduce need for interactive debugging
- enable change and evolution
- document functionality
- improve design

I haven't used an interactive debugger for C++ code for decades!

Speaker notes

I know that may people get introduced to embedded programming via a debugger. This is sad.

For me and many others, adhering to Test-First/Test-driven Design helps to get better at designing and writing code.

I personally haven't used an interactive debugger for C++ code for decades (while still writing C++ code :-), because I write unit tests.

© 2024 Peter Sommerlad

When to Write Tests

- Before writing any piece of new production code (TDD)
- Before fixing a reported bug (reproducability)
- Before changing a piece of legacy code (enable change)
- When in doubt about functionality of an API (explorative)

Speaker notes

for your own notes

© 2024 Peter Sommerlad

What not to Test at run-time

- Employ the C++ type system
- Compile-time assumptions checks with **`static_assert`**
- Automated unit tests
- Run-time `assert()`ions as a last resort

Speaker notes

Assertions triggering only at run-time of the production system are the second worst, the worst is a crashing or misbehaving system.

Tests executed at run-time of the test executable give quick feedback if run with every successful compile, but tests that the compiler already does while compiling give even quicker feedback.

The C++ type system and `static_assert` can help to reduce the need for run-time tests by letting the compiler prevent ridiculous code to be written.

© 2024 Peter Sommerlad

Excursion: Type System

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Stop Worrying and Love the C++ Type System

> *“I observed that type errors almost invariably reflected either a silly programming error or a conceptual flaw in the design.”*
> *– Bjarne Stroustrup, The Design and Evolution of C++*

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Type = ({values}, {operations})

- **A Type denotes**
  - a set of possible values and
  - a set of possible operations on these values
- operations define the meaning of the values
  - also define possible conversions (implicit or explicit)

"Types provide meaning to programs"

Speaker notes

- Cardinality(=size) of the first set denotes the least number of bits required for representing all values
  - languages and hardware might impose additional overhead (e.g. alignment, run-time type information)
- Operations include operators, functions, in general all possible uses of the values

Type Theory often taught far away from use of types in programming

Type Theory associates a term (expression) with a type

- **This is often only taught implicitly from using a programming language**
  - it took me decades to actually learn about it well enough and I am still learning...

A Type in C++ determines

- `sizeof(T)`
- interpretation of memory
- built-in operators
- parameter passing
- overload selection
- instantiation of templates
- meta programming
- ABI mechanics

`typedef` does NOT define a type

Speaker notes

- ABI specification often limits to adjust types after a library has been published and used by third parties

Remember: typedef does not define a type but an alias

© 2024 Peter Sommerlad

Type System

- provides meanings to programs
  - prevents mis-interpretation of data bits
- associates types with expressions and entities
  - statements do not have a type
- raises type errors
  - if wrong operations are attempted

Speaker notes

An entity can be a function, an object, a variable, or a reference.

statements do not have a type.

the only thing you can do with statements is sequencing (`;`)

© 2024 Peter Sommerlad

C++ Type System

- detects type errors (mostly strong)
  - supports implicit 🥺 and explicit conversion
- deduces types -> magic & less code
  - **`auto`** and templates
- selects overloads
  - functions and operators
- instantiates and selects templates
  - SFINAE/concepts
- is static, except when dynamic
  - **`virtual`**, **`dynamic_cast`**

Speaker notes

C++ has a mostly strong, mostly static type system, except for the weak parts inherited from C, which got them from B.

Only when using the keyword **`virtual`** C++ supports dynamic typing (and with `std::variant`).

C++ template mechanism and type system can perform magic at compile time.

Reflection will add a new dimension to the type system of C++26(?)

© 2024 Peter Sommerlad

When types don’t fit

- garbage (ASM)
- crash
- undefined behavior
- invalid values
  - `NaN`
  - `""`
  - `nullptr`
- exceptions
- ignore and continue
- runtime error

C++ static typing provides

- safety: compile error ✅
- implicit conversions ❌
- type deduction ✅
- overload resolution ⁉️
- template instantiation ✅
- meta programming ⁉️
- runtime efficiency ✅

Speaker notes

A type system prevents ridiculous code, either to execute (dynamically typed languages like Python) or better to compile (C++).

Employ the type system so that bad code won’t compile.

© 2024 Peter Sommerlad

C++ Type System Weaknesses

- integral promotion, including `bool` as integer
- usual arithmetic conversion
- implicit conversion of built-in types
- array to pointer decay
- implementation-defined types `int`
- undefined behavior on “normal” arithmetic
- casts (less arbitrary than in C)
- type punning (mostly illegal)
- C-string convention

static analysis, compiler warnings, follow guidelines!

Speaker notes

Many of the C++ type system weaknesses can be programmed/designed around, but that requires diligence and some effort.

C++ Type System Strengths

*classes make a **strong** type system*
(Bjarne Stroustrup, D&E)

- User-defined types & templates are *first class citizens*
- Type safety
- Type deduction (`auto`, template arguments)
- Compile-time polymorphism
  - Overload resolution
  - Templates
- Computation with types at compile time
- Run-time efficiency

*For integer-like types `enum class` can be **strong types** P.S.*

Speaker notes

Compile-time computation allows for meta-programming and types representing values.

Compilers employ the type system for optimization, cheating can either limit optimization or optimization can break code that attempts to “cheat” and thus has undefined behavior.

© 2024 Peter Sommerlad

Summary C++ Type System

- Employ the C++ type system to prevent bad/ridiculous code from compiling
- Favor compile-time checks like (`static_assert`) over run-time assertions and writing tests
- Learn how to create and use strong types for your domain!

Speaker notes

I couldn't fit more about strong typing in this course.

You might start out with writing unit tests and then later refactor to make code that might violate the test not even compile.

© 2024 Peter Sommerlad

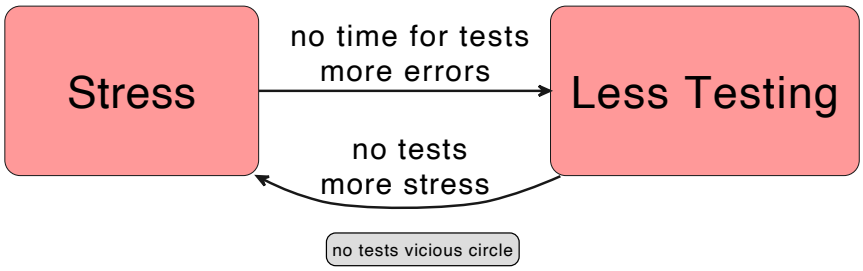
More on Writing Tests

Speaker notes

for your own notes

© 2024 Peter Sommerlad

“No Tests” Vicious Circle



Speaker notes

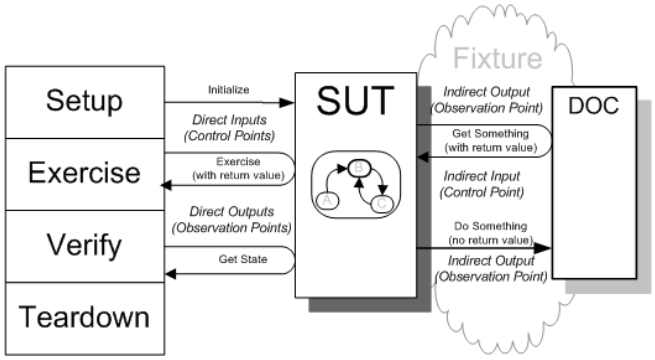
Stress while developing software leeds to taking shortcuts, such as not properly writing automated tests. This in turn leads to worse quality, leading to more stress.

Software is unintuitive: making it cheap, isn’t.

Key to keep going at a steady pace are unit testing and refactoring as hygiene practices.

© 2024 Peter Sommerlad

Testing Terminology



- Refactoring
- SUT - System Under Test
- DOC - Dependent Other Component
  - target for Test Stub / Mock

Speaker notes

Refactoring is an important development practice that continuously strives to improve and simplify the design of a piece of code we are working on. Indicated need for refactoring is called “design smell” (according to Kent Beck: If it stinks change it - also relating to the diapers of his then small kids :-)

Depending on the granularity or level of test cases, the system under test (SUT) might be a single function, an object of a class, or a whole subsystem.

Where the lattern often means more complicated interaction that goes beyond what is commonly known as unit testing.

© 2024 Peter Sommerlad

Refactoring

Refactoring is a controlled technique for improving the design of an existing code base without changing its behaviour.

Speaker notes

Refactoring consists of very small steps.

Some refactorings are the reverse of others, so be aware to not walk in circles.

There should never be a “refactoring” entry in the project plan, but an ongoing practice like writing unit tests.

If there is a big “Refactoring” needed, this is called **Redesign** !

© 2024 Peter Sommerlad

Major Goal of Refactoring

OAOO - Once And Only Once (Kent Beck)

DRY - Don’t Repeat Yourself (Pragmatic Programmers)

- reduce duplication
- allow chagnes of

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Getting Started

- How would I know that code is performing as intended?
- How can I test that?
- What could go wrong with the code?
- Did we have a similar failure/bug that we can check for

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Conceptual Test Structure

Arrange prerequisites	`std::ostringstream out{};`
Act: perform SUT	`sayhello(out);`
Assert results	`ASSERT_EQUAL("Hello World\n", out.str());`

*This is called the **AAA** test schema*

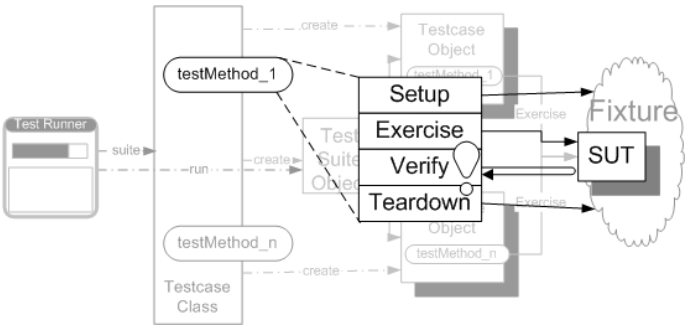
Speaker notes

SUT = system under test = the functionionality that you want to test

Some unit testing literature (e.g. http://xunitpatterns.com/) and frameworks also calls out a 4 Phase Test:

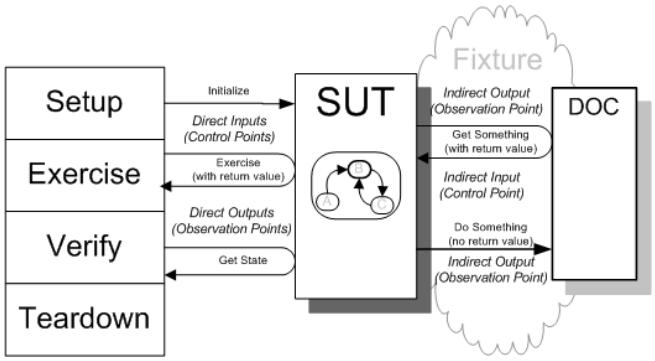
- Setup
- Exercise
- Verify
- Teardown

The last phase that needs to be explicit in other languages is not needed in C++, because any C++ object that is created for a test execution will automatically be destroyed when the test scope is left. C++ unit testing frameworks that provide a teardown hook are not adhering to the C++ language design (such as google test :-).



used with permission from xunitpatterns.com

xunitpatterns terminology



Speaker notes

used with permission from xunitpatterns.com

© 2024 Peter Sommerlad

What are GUTs?

Good Unit Tests

Good, DRY, and Simple

- no control structures
  - linear control flow (AAA)
  - test assertion at the end
- test one thing at a time
  - a GUT has one reason to fail and is named after it
  - not a test per function (SUT), usually multiple
  - tests per equivalence class of input values

Speaker notes

for your own notes

© 2024 Peter Sommerlad

What are GUTs (2)?

- no inter-test (order) dependency
  - tests don’t leave traces for others to depend upon
- all tests run successfully when you deliver (to source control)
- tests provide a good coverage of production code
- are often created Test-First

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Getting started with writing tests

- Right BICEP 💪
- CORRECT Boundary Conditions 👍
- GUTs are A TRIP 🗺️
- Testing with ZOMBIES 🧟‍♂️🧟‍♀️🧟

Speaker notes

These mnemonics are taken from the books “Pragmatic Unit Testing” (variations exist for different languages) by Andrew Hund and David Thomas.

© 2024 Peter Sommerlad

Right BICEP 💪

- Is the result **Right**?
- **B** - Are all **boundary** conditions *CORRECT*?
- **I** - Can you check **inverse** relationships?
- **C** - Can you **cross-check** results by other means?
- **E** - Can you force **error conditions**?
- **P** - Are **performance** characteristics within bounds?

https://godbolt.org/z/jrK8TWjYx

Speaker notes

Righ BICEP is a mnemonic to help you think about what tests to write for a function/unit.

https://godbolt.org/z/jrK8TWjYx

© 2024 Peter Sommerlad

CORRECT Boundary Conditions 

- **C**onformance
- **O**rdering
- **R**ange
- **R**eference
- **E**xistence
- **C**ardinality
- **T**ime

Speaker notes

CORRECT is a mnemonic to help you think how to check results for correctness when writing tests.

© 2024 Peter Sommerlad

GUTs are A TRIP 🗺️

or a PIRATe 🏴‍☠️

- **A**tomatic
- **T**hourough
- **R**epeatable
- **I**ndependent
- **P**rofessional

Speaker notes

A TRIP is a mnemonic for describing what makes good tests.

- Manual tests are often not repeatable or can be burden for testers. Automate as much as possible/reasonable. Especially for bug reports create automated tests demonstrating the bug, before attempting to fix it. Those tests will help to not reintroduce the same bug again.
- Good tests tests everything that is likely to be wrong. Coverage tools can help measuring, but don't put coverage goals first, because that can lead to bad tests. I have seen outsourced projects with 100% test coverage, where most tests where checking nothing and just executed code. There are even tools that generate test cases causing good coverage, where the generated tests are prohibiting changing production code, including its bugs, because they are manifested in the generated tests. However, if you have parts of the system with very low test coverage those tend to contain the most problems and are hard to refactor, because of the lack of automated testing.
- Repeatable tests mean that they produce the same results regardless when or where they run. Having a build system running all the tests and running them locally for every change help with that. Repeatability also implies **I**ndependence of tests.
- Independent tests can run in isolation and have a single reason to fail. I often enforce this through only allowing a single ASSERT/REQUIRE per test case. While not always the simplest way, it helps to get to the practice of having many independent tests.
- Professional unit tests mean that you put the same care and quality into the test cases as you would into production code. Tests also must have good design such as encapsulation, adhering the DRY principle, low coupling, and so on. If test code starts to look unobvious or messy, refactor your tests as well, like you would do with production code.

TDD guided by 

- **Z**ero
- **O**ne
- **M**any (or More complex)
- **B**oundary/Behaviors
- **I**nterface definition
- **E**xercise/Exceptional behavior
- **S**imple Scenarios, Simple Solutions

Speaker notes

ZOMBIES is an abbreviation invented by J.W. Grenning. There is some overlap to the concepts introduced by the pragmatic starter kit on unit testing.

from http://blog.wingman-sw.com/tdd-guided-by-zombies

“When test-driving, guided by ZOMBIES, the first test Scenarios are for Simple post-conditions of a just created object. These are the Zero cases. While defining the Zero cases, take care to design the Interface and capture the Boundary Behaviors in your test Scenarios. Keep it Simple, both Solutions and Scenarios. You’ll find that hard. Once progress is made on the Zero cases, move to the next special Boundary case, testing the Behavior desired when transitioning from Zero to One. To do so there are likely other Interfaces to define and use in new test Scenarios. Once the Boundary Behaviors between Zero and One (and possibly back to Zero from One) have been captured in tests, move on to start to generalize your design now dealing with More complex Scenarios and Many items being managed. Often there are new Boundary conditions to be concerned with. Finally review your work and make sure you consider and Exercise the Exceptional things that might happen.”

Testing the Tests

Quis custodiet ipsos custodes?

- Improve tests when fixing bugs
- Prove tests by introducing bugs

Any bug reported should be reproduced by a test first

Any test that you write should fail at least once

Speaker notes

Who will guard the guards themselves?

In addition: when writing a test case for a bug, think where in the system you might have similar problems and add tests for these situations.

© 2024 Peter Sommerlad

Exercise Test Concepts

Exercise

Speaker notes

https://github.com/PeterSommerlad/CPPCourseUnitTesting/blob/main/exercises/exercise_lasercutter.md

© 2024 Peter Sommerlad

What hinders Testability

- Dependencies
  - worst:
    - Singletons
    - global variables
    - (hidden) side effects
- Duplication (not DRY)
- Developer Lazyness or Ignorance
- Shyness to Refactor

Speaker notes

Take the courage to clean up your system, especially when you can then write better and simpler tests.

© 2024 Peter Sommerlad

Dealing with Dependencies

What to do, when a dependency cannot be eliminated

- dependent other component (DOC)
  - too slow for unit tests
  - doesn't exist yet
  - hard to trigger test behavior (errors)
- Refactor to allow parameterization
- Substitute DOC with a special case for a test(s): create a test fixture

Speaker notes

I do not recommend to use Mocking Frameworks in general, because they can lead to sidestep the process to loosen or eliminate dependencies, leading to very rigid architectures that are hard to change. See my CPPCon 2017 talk "Mocking Frameworks Considered Harmful" https://youtu.be/uhuHZXTRfH4

Mocking should be expensive, so it is only used, when needed, not unnecessarily manifesting dependency connections.

© 2024 Peter Sommerlad

Test Doubles

Different (but similar) approaches for test fixtures replacing a DOC

- Dummy Object
- Test Stub
- Test Spy
- Mock Object
- Fake Object

Speaker notes

Most of the time Test Doubles for a DOC are relevant when specific information needs to be delivered, or a side effect on the DOC needs to be checked.

A Dummy Object is used as a placeholder for a DOC that is not actually needed for the test, but that is required to construct the object to be tested.

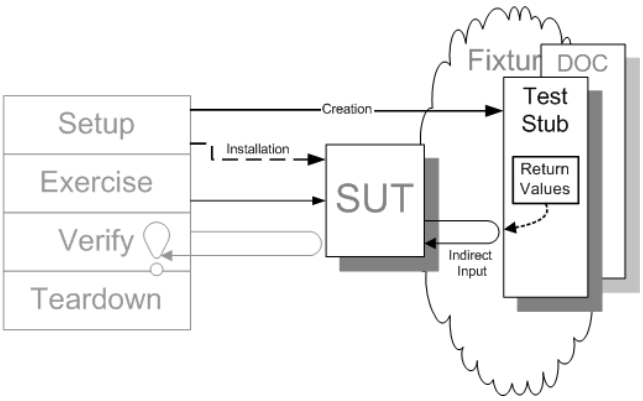
A Test Stub for a DOC provides indirect inputs to the SUT that are used to guarantee specific behaviour expected by a test.

A Fake Object replaces the DOC with alternative implementation of the behavior as required by tests. This is a bit more elaborate thatn a Test Stub.

A Test Spy for a DOC allows to collect usages (indirect output) of the DOC and check those from the test.

A Mock Object replaces the DOC and allows controlling the SUT employs the expected protocol on the DOC (more fine granular than the Test Spy).

Test Stub

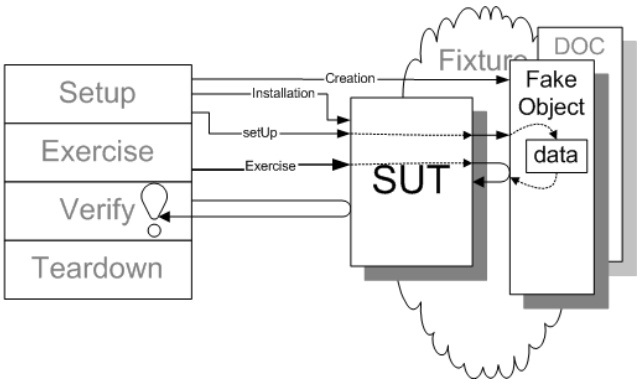


Speaker notes

A Test Stub often contains pre-configured test data to be delivered when called from the SUT during the test.

© 2024 Peter Sommerlad

Fake Object



Speaker notes

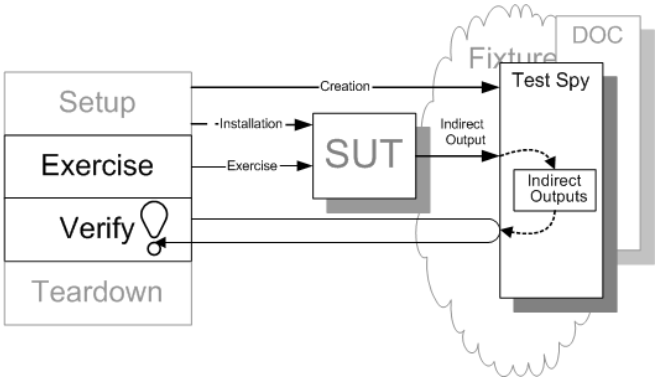
Whereas a test stub has hard coded results to be delivered to the DOC when called, a Fake can be configured during the setup.

I usually don't distinguish between Test Stubs and Fakes, because where the actual results returned to the SUT come from is an implementation detail.

For example, a Mocking Framework can provide elaborate means to set up returned values, whereas a DIY solution would just write hard coded values in (member-)function(s) to return.

© 2024 Peter Sommerlad

Test Spy

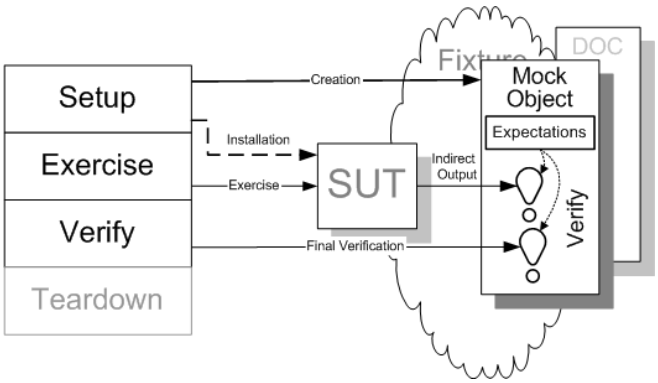


Speaker notes

In our initial example the `std::ostringstream` passed to the sayHello() function serves the role of a test spy to collect all the output written, so that it can later be checked in the test case.

```
void testSayHello() {
  std::ostringstream out{}; // Test Spy
  sayhello(out);
  ASSERT_EQUAL("Hello World\n", out.str());
}
```

Mock Object



Speaker notes

Another way to implement Behavior Verification is to install a Mock Object in place of the target of the indirect outputs. As the SUT makes calls to the DOC, the Mock Object uses assertions. to compare them with the expected calls and arguments.

Again Mocking Framworks can provide very elaborate solutions to set up expectations from a test's setup and implicitly check those after the SUT was exercised.

I personally tend to prefer simpler direct solutions if mocking/spying is really necessary. Usually such a need only arises when the DOC has a complex API protocol to adhere to, which is often an indicator for a too complex design.

Too Much Mocking?

- **White box tests** couple Tests, SUT and DOC tight
  - hard to refactor for tools and manually, design flexibility is lost
- **Fosters stateful APIs**, especially when used with TDD of SUT and DOC
  - `setWiggle(Wiggle), setWaggle(Waggle), WiggleTheWaggle()` in the mocked object, called in sequence from SUT
- Tendency for single/no Parameter Functions and required sequence of when being called
  - Uncle Bob (“Clean Code”) misunderstood: favor “niladic” methods
  - he says: “avoid temporal coupling”

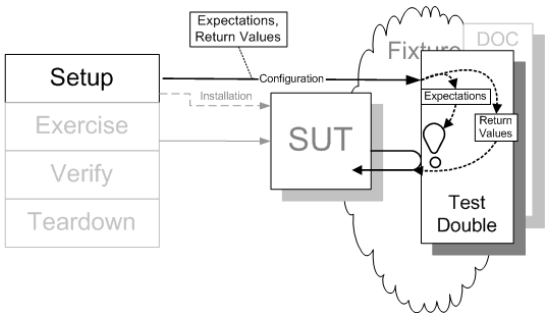
Speaker notes

niladic member functions still have a parameter, the `this` object.

For the reasons above, I recommend to use mocking carefully only after design choices have been considered. If existing use of mock objects hinder change, it is an indication that the design between SUT and DOC is too tightly coupled.

© 2024 Peter Sommerlad

Configurable Test Double




```
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```

- Elaborated setup code
- Implicit checking of assumptions during destruction
- Danger of manifesting tight coupling and fragile tests

Speaker notes

The Mocking Framework DSL easily leads to complex specifications that hinder future change and refactoring.

GoogleMock DSL allows specification of behaviour and expectations.

In general C++ mocking frameworks tend to be problematic with respect to modern C++ design:

- Design tend to follow JMock/EasyMock - keep Java Design- lack C++ strengths
- No useful reflection in C++: Macros for generating names, defining function stubs
- Subclassing, virtual member functions, sometimes fiddling via undefined behavior or low-level machine-specific ABI, e.g., replacing vtable entries (Hippomocks)
- DSL to specify (expected) behavior, not plain code, MACRO magic
  - with implicit matching of actual behavior vs expected in destructors
  - fragile with refactoring, hard to reuse across tests, bloated tests

DRY Tests

```
void FizzBuzzOneIs1() {
    ASSERT_EQUAL("1",fizzbuzz(1));
}
void FizzBuzzTwoIs2(){
    ASSERT_EQUAL("2",fizzbuzz(2));
}
void FizzBuzzThreeIsFizz(){
    ASSERT_EQUAL("Fizz",fizzbuzz(3));
}
```

copy-paste tests and change arguments

consider data-driven testing

Speaker notes

For such simple cases actually copy-paste and change can still be beneficial, because the names of the tests can give important information when one fails.

© 2024 Peter Sommerlad

Data-driven Tests (CUTE)

```
struct fizzbuzz_table {
    std::string  expected;
    int  input;
    cute::test_failure  fail; // location
};
// .. table
void FizzBuzzAllTable(){
    for(auto const &test:theFizzBuzzTests){
        ASSERT_EQUAL_DDT(test.expected,
      fizzbuzz(test.input), test.fail);
    }
}
```

```
fizzbuzz_table const
theFizzBuzzTests[]={
        {"1",    1,  DDT()},
        {"2",    2,  DDT()},
        {"Fizz", 3,  DDT()},
        {"4",    4,  DDT()},
        {"Buzz", 5,  DDT()},
        {"Fizz", 6,  DDT()},
        {"7",    7,  DDT()},
        {"8",    8,  DDT()},
        {"Fizz", 9,  DDT()},
        {"Buzz", 10, DDT()},
        {"11",   11, DDT()},
        {"Fizz", 12, DDT()},
        {"13",   13, DDT()},
        {"14",   14, DDT()},
        {"FizzBuzz", 15, DDT()}
};
```


Speaker notes

The `DDT()` macro generates information comparable to `std::source_location` so that failures can refer to the actual test data instead of the loop of the test.

The underlying principle:

- create a data structure for input, expected result, and possibly a reference to the position in the table.
- create a table of test data (plain array is OK here, otherwise a `std::vector` or `std::to_array` can be used)
- have a test function iterating the table

© 2024 Peter Sommerlad

Data-driven Tests (GoogleTest)

```
struct fizzbuzz_test_entry {
    std::string expected;
    int input;
};
fizzbuzz_test_entry const
fizzbuzz_tests[]={
    {"1",     1},
    {"2",     2},
    {"Fizz",  3},
    {"4",     4},
    {"Buzz",  5},
    {"Fizz",  6},
    {"7",     7},
    {"8",     8},
    {"Fizz",  9},
    {"Buzz",  10},
    {"11",    11},
    {"Fizz",  12},
    {"13",    13},
    {"14",    14},
    {"FizzBuzz", 15}
};
```

```
struct FizzBuzzDataDrivenTest
  : testing::TestWithParam<fizzbuzz_test_entry> {};

TEST_P(FizzBuzzDataDrivenTest, FizzBuzzTest) {
    auto entry = GetParam();
    ASSERT_EQ(fizzbuzz(entry.input),entry.expected);
}

INSTANTIATE_TEST_SUITE_P(
        FizzBuzzGroup,FizzBuzzDataDrivenTest,
        testing::ValuesIn(fizzbuzz_tests),
        [](auto &param)->std::string{
            return  std::to_string(param.param.input)
                    + "_should_be_" + param.param.expected; }
        );
```

https://godbolt.org/z/shsxEhvK6

Speaker notes

While the principle is the same, GoogleTest is a bit more involved to get parameterized tests. Iteration is hidden with `.ValuesIn()`. Test reference is not via source location but using a generated name that must look like a valid C++ identifier.

https://godbolt.org/z/shsxEhvK6

© 2024 Peter Sommerlad

(Bad?) Alternative: non-stopping Assertions

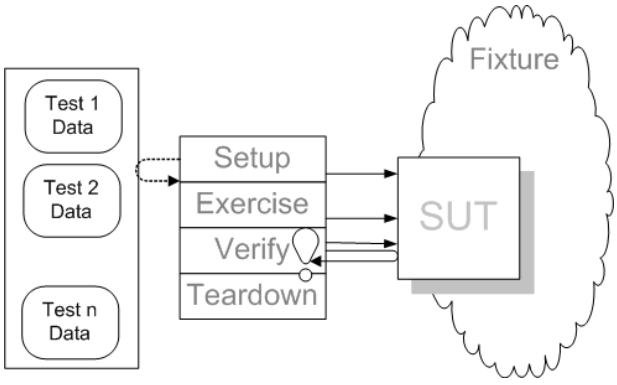
```
TEST(FizzBuzzBulkTest, FizzBuzzTest){
    EXPECT_EQ("1",   fizzbuzz(1));
    EXPECT_EQ("2",   fizzbuzz(2));
    EXPECT_EQ("Fizz",fizzbuzz(3));
    EXPECT_EQ("4",   fizzbuzz(4));
    EXPECT_EQ("Buzz",fizzbuzz(5));
    EXPECT_EQ("Fizz",fizzbuzz(6));
    EXPECT_EQ("7",   fizzbuzz(7));
    EXPECT_EQ("8",   fizzbuzz(8));
    EXPECT_EQ("Fizz",fizzbuzz(9));
    EXPECT_EQ("Buzz",fizzbuzz(10));
    EXPECT_EQ("11",  fizzbuzz(11));
    EXPECT_EQ("Fizz",fizzbuzz(12));
    EXPECT_EQ("13",  fizzbuzz(13));
    EXPECT_EQ("14",  fizzbuzz(14));
    EXPECT_EQ("FizzBuzz", fizzbuzz(15));
}
```


Speaker notes

In cases, where programmer lazyiness happens, or when failing a test it will be very obvious what and how it went wrong, I have seen GoogleTest users to use EXPECT macros in a row. This violate the principle that a test should have one reason to fail. But one can squeeze a bit and say that each EXPECT is actually a test case in its own, just an anonymous one.

© 2024 Peter Sommerlad

Summary Data-driven Tests



- Group Tests with a common fixture, i.e., by putting the test functions in a class.
- When there is a need for many similar tests, consider data-driven testing

Speaker notes

Not shown, but obvious is the use of a test fixture to share common initialization.

© 2024 Peter Sommerlad

Exercise Controller class

Exercise Controller Class

Speaker notes

https://github.com/PeterSommerlad/CPPCourseUnitTesting/blob/main/exercises/exercise_controller.md

© 2024 Peter Sommerlad

Software Architecture

Consciously manage dependencies between software artifacts

Speaker notes

Too many dependencies are the major roadblock to testing

© 2024 Peter Sommerlad

Software Architecture

*A **software architecture** is a description of the subsystems and components of a software system and the relationships between them.*

Views:

- physical: mapping to hardware
- logical: e.g. layering
- process: concurrency and synchronisation
- development: organization, e.g., file structure: `#include`, libs, object files, build

Speaker notes

This definition and some of the following ones is taken from “Pattern-oriented Software Architecture: A System of Patterns” [POSA1] co-authored by me.

The dependencies created by what a component contains and whot other components it depends upon or what other components depend on it have a great influence on testability of code.

© 2024 Peter Sommerlad

Relationship

*A **relationship** denotes a connection between components. A relationship may be static or dynamic. Static relationships show directly in source code. They deal with the placement of components within an architecture. Dynamic relationships deal with temporal connections and dynamic interaction between components. They may not be easily visible from the static structure of source code.*

- be aware of hidden relationships
- make relationships obvious through parameterization
- minimize and loosen coupling

Speaker notes

Relationships between components are needed for a system to function. However, when not consciously managed they can lead to tangled systems that are hard to test.

Often, managing dependencies comes as an afterthought, or tight coupling is a given due to used infrastructure/frameworks.

© 2024 Peter Sommerlad

Some Important Architectural Patterns

- Layers
- Pipes and Filters

Speaker notes

While there exist many more architectural styles or patterns to derive from, these two are fundamental to structuring code and are often misunderstood or badly applied.]

© 2024 Peter Sommerlad

Layers 

The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

- Levels of abstraction
- Dependencies in one direction
- Exchangeability of implementation

Speaker notes

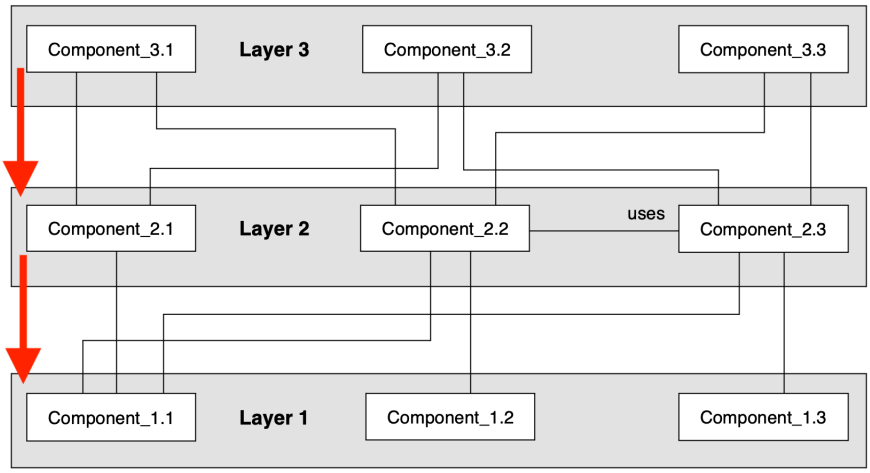
An important point is raising the level of abstraction. Just reimplementing the same level of abstraction by delegating to lower layers just adds overhead without gaining much.

There always will be fundamental types/functions that are useful in all layers. However, to manage dependencies, it is best to just rely on the next layer below to achieve exchangability and testability. Fakes for testing a layer can be for one layer below, or sometimes for the layer above.

Control flow is not necessarily from higher to lower layer, in an event-based system, control flow often goes from lower layers to higher layers (callbacks).

© 2024 Peter Sommerlad

Layers



Speaker notes

Note that inter-layer dependencies should always be directed towards the lower layer to avoid circular dependencies.

Often there are components useful in all layers. Those should be considered platform library that all layers can depend on. There is no need to have a layer-specific string class for example.

On the other hand, system services should only be used by the lowest layer and provided in a more abstract and convenient way to a higher layer.

Pipes and Filters

The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

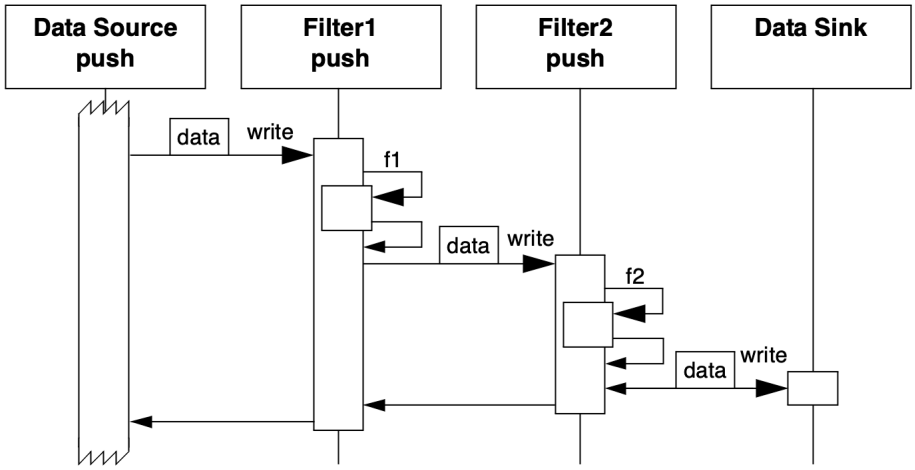
C++20 ranges views combination via overloaded **`operator|`** applies P&F.

Speaker notes

While using the vertical bar | as a combination symbol is common (UNIX, C++ ranges), the underlying architectural principle also works with regular function calls, either via pull (lazy - previous filter), or via push (eager - following filter) when calling a function in sequence.🧹🧹

© 2024 Peter Sommerlad

P & F: push pipeline

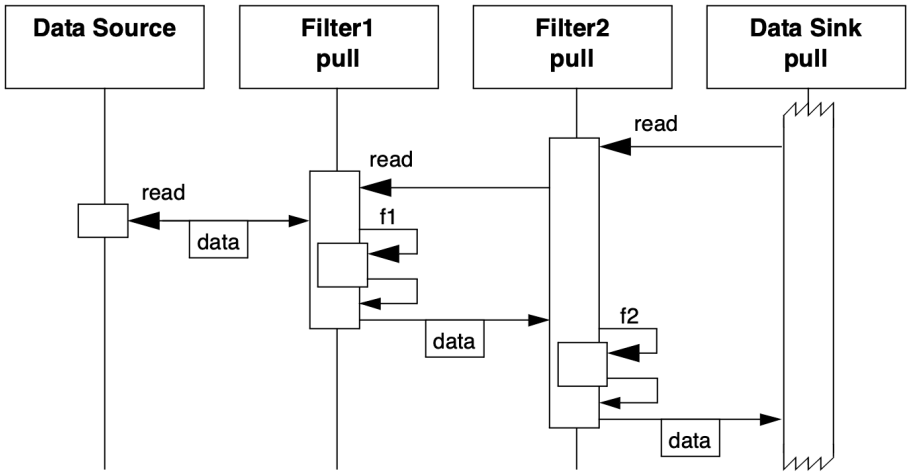


eager evaluation

Speaker notes

This can be implemented as a standard call hierarchy from the data source.

P & F: pull pipeline



lazy evaluation

Speaker notes

This is the kind of combination the standard ranges library views provide.

Dependencies in C++

- file structure: `#include` dependencies
- functions: declaration visible
- namespaces: caution ADL
- type usage: definition visible
  - variables, parameters, class members
  - inheritance: tightest coupling

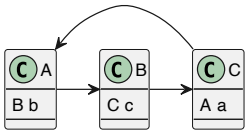
Dependencies glue a system together

Speaker notes

While some glue is important, pouring superglue between all parts of a system through circular dependencies and other thight coupling makes it almost impossible to test individual components and thus write good unit tests.

© 2024 Peter Sommerlad

Circular dependencies 🔃



- type forward declarations in C++ can indicate circular dependencies

using one element in a circular dependencie implies using all from the circle

- Tests for one element in a dependency circle depend on all parts of the circle

Speaker notes

From code examples I received some seem to glue together many other classes by creating a circular dependency.

© 2024 Peter Sommerlad

Circular depency example 🔃

```
void initHelper::InitializeACLoadCycle(sttdACLoadCycleDef & loadCycle,
                                       const EMaterialTypes & eMatType,
                                       bool bSetDefaultMaterialVolume)
{
    loadCycle.eMaterialType     = eMatType;
//...
}

sttdACLoadCycleDef::sttdACLoadCycleDef()
{
    initHelper::InitializeACLoadCycle(  *this, EXMT_SAMPLE, true);
}
```

- Put what belongs into a constructor into the constructor
- member initializers should be used instead, especially with constants
- Ideally all members are initialized before the constructor body runs

Speaker notes

Initializing a class non-static data members in a constructor body through assignmen is considered too late and potentially inefficient, since for data members with default constructors those constructors run before the constructor body reassigns a value to them.

I couldn't understand for the initialization helper infrastructure. Can you enlighten me with its rationale.

Unlucky assignment of functionality can easily lead to much tighter coupling than needed.

© 2024 Peter Sommerlad

Strength of Coupling

- `#include`: all declared (and used) entities
- Inheritance: -> base classes
- global variable usage: all usage points
- type usage: data member, function signature type
- function usage: all signature types, all (visible) overloads
- reference/pointer to type: incomplete type possible
- template argument: only instantiation point

list is incomplete

Speaker notes

Often changing a dependency to a concrete element through parameterization can help reducing dependencies

© 2024 Peter Sommerlad

Architecture and Test Summary

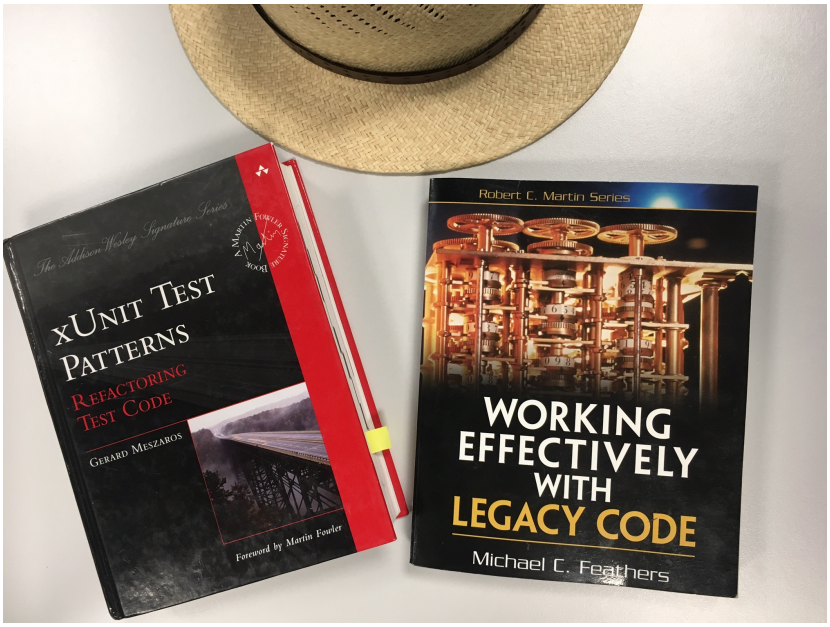
- Design and Architecture heavily influence Testability
- Improving Testability also improves Design
- Better Testability means less tight coupling
- Refactoring and Redesign might be needed

Speaker notes

Changing design requires courage. Having good tests eases change. Tests can make you a braver developer.

© 2024 Peter Sommerlad

Testing Legacy Code



Speaker notes

Michael Feathers wrote a whole book about “Working Effectively with Existing Code” that deals with the situation that you want to get existing code unter test so that you can refactor and change it safely.

© 2024 Peter Sommerlad

Dilemma

- Refactoring should be safe by having tests
- Adding tests requires refactoring

Speaker notes

Being brave and systematic, it is possible to make schematic global changes to a system when you have all its source code and can recompile everything. This can be used to replace an interface type, for example. I have change a larger C++ framework from using `char const *` to a string class in the 1990s before we introduced unit tests for the code base.

Having an IDE with a project centric view helps to search-replace all usages of a function that you want to change.

Small refactoring steps are always possible and less risky, even without having good automated tests yet.

© 2024 Peter Sommerlad

Separate New Code

Sprout Class *and* ***Sprout Function***

- Add new code as a separate unit that you can develop with tests/TDD
- Don’t change existing code inline
- Only add calls to new separately tested code

Speaker notes

This approach is a means to get started when having inherited a legacy codebase.

It can be assisted with having system-level (end-to-end) tests that you can perform as a sanity check.

When you don’t have tests, you want to make minimal intrusive changes so that you don’t risk breaking working stuff without knowing.

© 2024 Peter Sommerlad

Dependencies in Legacy Code

Old applications are glued together tightly

- Global variables or Singletons
- Data members of fixed foreign types
- Other uses of fixed foreign types
- worst are circular dependencies between components, avoid!

also

- global functions with outside side effects, e.g., printf()

Speaker notes

Dependencies come in different strengths and between different programming elements. Examples are:

- Calling a global function that incurs side effects is a hard to test dependency.
  - In general functions with side effects on non-parameters are very hard to impossible to test.
  - Functions with side-effects on parameters can still be a nuisance to call.
- Inheriting from a base class is one of the strongest dependencies.
- Using a non-const global variable glues together all pieces of code implicitly where this variable is accessed.

© 2024 Peter Sommerlad

Legacy Code Change Algorithm

1. Identify Change Points
2. Identify Test Points
3. Break Dependencies
4. Write Tests
5. Refactor and/or Change

Speaker notes

Taken from Michael Feathers' "Working Effectively With Legacy Code"

© 2024 Peter Sommerlad

Breaking Dependencies

- breaking a dependency often means to introduce a means to substitute the dependent component by a Test Stub/Mock for writing tests

Blindly replacing any dependent classes by mocks is not helping with really breaking dependencies

Speaker notes

First of all, you need to make yourself aware of the dependencies of a piece you want to test.

© 2024 Peter Sommerlad

Example: GameFourWins

GameFourWins.h

```
#include "Die.h"
#include <iosfwd>

struct GameFourWins {
  void play(std::ostream& os) ;
private:
  Die die{};
};
```

Die.h

```
#ifndef DIE_H_
#define DIE_H_

struct Die {
  int roll() const;
};

#endif /* DIE_H_ */
```

GameFourWins.cpp

```
#include "GameFourWins.h"
#include <ostream>
void GameFourWins::play(std::ostream& os) {
  if (die.roll() == 4) {
    os << "You won!\n" ;
  } else {
    os << "You lost!\n" ;
  }
}
```

Die.cpp

```
#include "Die.h"
#include <cstdlib>

int Die::roll() const {
  return rand() % 6 + 1;
}
```

https://godbolt.org/z/4ffhz1dcj

Speaker notes

External dependency to random number generator makes written tests fragile

https://godbolt.org/z/4ffhz1dcj GameFourWins Test example starting point

© 2024 Peter Sommerlad

Introducing **Seams**

- Extract Interface Refactoring
- Introduce Parameter Refactoring
- Introduce Template Parameter
- Preprocessor Seams
- Linker Seams

Speaker notes

Seams losen dependencies, so that tests can substitute the DOCs. While it might be hard to rip into a tightly knit fabric of code, cutting for seams is a good metaphor for managing dependencies in legacy code.

While the techniques shown here are not without risk, they can be important starting points for further improving the structure of a legacy system.

© 2024 Peter Sommerlad

Extract Interface Refactoring

- identify target class dependency you need to decouple in SUT
- idendify target class member functions called
- name the interface and make it a base class of the target
- check for unintended overrides in target class subclasses
- replace target class with interface in SUT
- lean on the compiler to find the member functions needed in the interface
- copy function declarations from target class to interface as pure virtual

Speaker notes

Interface: A class with only pure virtual member functions.

Unintended Override: the target class that you want to extract an interface from has subclasses that implement a member function that is now virtual and not the original class. This case is a bad design anyway, but a corner case to be aware of.

When the target class already has (non-pure) virtual member functions one might to actually push down their implementations to create the interface in the target class.

© 2024 Peter Sommerlad

Example: After Extract Interface

GameFourWins.h

```
#include "IDie.h"
#include <iosfwd>

struct GameFourWins {
  void play(std::ostream& os);
  GameFourWins(IDie &theDie)
    :die{theDie}{}
private:
  IDie &die;
};
```

GameFourWins.cpp

```
#include "GameFourWins.h"
#include <ostream>
void GameFourWins::play(std::ostream& os) {
  if (die.roll() == 4) {
    os << "You won!\n";
  } else {
    os << "You lost!\n";
  }
}
```

IDie.h

```
#ifndef IDIE_H_
#define IDIE_H_
struct IDie {
  virtual int roll() = 0;
  virtual ~IDie() = default;
  IDie& operator=(IDie&&) & = delete;
};
#endif /* IDIE_H_ */
```

Die.h

```
#ifndef DIE_H_
#define DIE_H_
#include "IDie.h"
struct Die : IDie {
   int roll() override;
};

#endif /* DIE_H_ */
```

https://godbolt.org/z/vo69K9Y1j

Speaker notes

This is the classic object-oriented approach by extracting an explicit interface class. While common in languages like Java, it is kind of intrusive in C++ code. Both the SUT and the DOC need adaptation.

Die.cpp remains unchanged:

```
#include "Die.h"
#include <cstdlib>

int Die::roll()  {
  return rand() % 6 + 1;
}
```

https://godbolt.org/z/vo69K9Y1j GameFourWins Test with Object Seam

Example: Test provides Mock types

```
struct LosingDie1 : IDie{
    int roll() override { return
        1;}
};
TEST(GameTest, gameLosesWith1) {
    std::ostringstream out{};
    LosingDie1 theDie{};
    GameFourWins game{theDie};
    game.play(out);
    ASSERT_EQ("You lost!\n",
        out.str());
}
```

```
struct WinningDie : IDie{
    int roll() override { return
        4;}
};
TEST(GameTest, gameWins) {
    std::ostringstream out{};
    WinningDie theDie{};
    GameFourWins game{theDie};
    game.play(out);
    ASSERT_EQ("You won!\n",
        out.str());
}
```


Speaker notes

If you want to test multiple scenarios, you might need multiple mock implementations.

However, if the variation is just with a simple value, either constructor parameter and a data member or a value template parameter can limit the amount of source code to write and reduce the duplication.

© 2024 Peter Sommerlad

Example: With GoogleMock tests

```
struct MockDie: IDie {
    MOCK_METHOD(int,roll,(),(override));
};
using testing::Return;
TEST(GameTest, gameLosesWith1) {
    std::ostringstream out{};
    MockDie theDie;
    EXPECT_CALL(theDie,roll()).WillOnce(Return(1));
    GameFourWins game{theDie};
    game.play(out);
    ASSERT_EQ("You lost!\n", out.str());
}
```


Speaker notes

GoogleMock allows to “configure” the behavior of the mocked IDie implementation.

While it looks attractive, I find the indirect specification can lead to non-obious tests and also to tighter coupling between tests and SUT, hindering refactoring and evolution.

It can also lead to testing the mock instead of testing the actual SUT.

© 2024 Peter Sommerlad

Compile Seam

Introduce Template Parameter Refactoring

- Identify undesirable dependency to concrete type CT
- Add a template parameter T to the class/function
- Replace all uses of CT with T
- Move dependent code into header (before C++20 modules)
- Provide type alias with default argument T=CT

Speaker notes

Still the big change is that now the dependent code must all be put into a header file. C++20 modules make this seam more attractive, since a module can export templates like types.

For C++, it might be much less intrusive than the Object seam resulting from Extract Interface.

© 2024 Peter Sommerlad

Example: After Introduce Template Parameter Refactoring

GameFourWins.h

```
#ifdef GAME_FOUR_WINS_H_
#define GAME_FOUR_WINS_H_
#include "Die.h"
#include <ostream> // need full definition
template<typename DIE=Die>
struct GameFourWins {
  void play(std::ostream& os) {
    if (die.roll() == 4) {
      os << "You won!\n";
    } else {
      os << "You lost!\n";
    }
  }
private:
  DIE die;
};
#endif
```

Die.h unchanged:

```
#ifndef DIE_H_
#define DIE_H_
struct Die {
  int roll() ;
};
#endif /* DIE_H_ */
```

Die.cpp unchanged:

```
#include "Die.h"
#include <cstdlib>

int Die::roll() const {
  return rand() % 6 + 1;
}
```

https://godbolt.org/z/h6Y8dq4jM

Speaker notes

In real-world cases one might actually chose a slightly different name for the template and then provide an alias for the old class name with the default template parameter:

```
template<typename DIE>
struct GameFourWinsT{...};
#include "Die.h"
using GameFourWins=GameFourWinsT<Die>;
```

Still the big change is that now the dependent code must all be put into a header file. C++20 modules make this seam more attractive, since a module can export templates like types.

an example test employing the compile seem looks like:

```
struct LosingDie1{
  int roll() { return 1;}
};
TEST(GameTest, gameLosesWith1) {
  std::ostringstream out{};
  GameFourWins<LosingDie1> game{};
  game.play(out);
  ASSERT_EQ("You lost!\n", out.str());
}
```

https://godbolt.org/z/h6Y8dq4jM GameFourWins Test with Compile Seam

Example Test with Compile Seam

```
struct LosingDie1{
  int roll() { return 1;}
};
TEST(GameTest, gameLosesWith1) {
  std::ostringstream out{};
  GameFourWins<LosingDie1> game{};
  game.play(out);
  ASSERT_EQ("You lost!\n", out.str());
}
```

```
struct WinningDie {
    int roll() { return 4;}
};
TEST(GameTest, gameWins) {
    std::ostringstream out{};
    GameFourWins<WinningDie> game{};
    game.play(out);
    ASSERT_EQ("You won!\n", out.str());
}
```

https://godbolt.org/z/h6Y8dq4jM

Speaker notes

With the compile seam test cases can provide minimal template argument types that just implement what the test case needs. This is in contrast to the Object seam, where all pure virtual member functions of the extracted interface must be implemented by each concrete subclass used in the tests.

© 2024 Peter Sommerlad

Preprocessor Seam

measure of last resort, very error prone

- figure hard dependency on a function that could be replaced with a macro call
- prepare substitute function delegating to original function
- create macro substituting original `#define original(x) substitute(x)`
- manipulate compile command `-include=mockheader.h`
- provide implementation of substitute function for test

Speaker notes

We were able to automate that with our Eclipse CDT plugin “Mockator”:

Video: https://www.youtube.com/watch?v=uv-AC_-QwKs

Unfortunately, Eclipse CDT changed to much so that Mockator no longer works with current releases.

© 2024 Peter Sommerlad

Example: Preprocessor Seam

leapyear.h

```
#ifndef TODAYSTIME_H_
#define TODAYSTIME_H_
bool isLeapYear();
#endif /* TODAY_H_ */
```

leapyear.cpp

```
#include "leapyear.h"
#include <ctime>
static unsigned getYear() {
        time_t now = time(0);// replace?
        tm* z = localtime(&now);
        return z->tm_year + 1900;
}
bool isLeapYear() {
        auto year = getYear();
        if ((year % 400) == 0) {
    //...
```

-i mockit_time.h pre-loaded

```
#ifndef MOCKIT_TIME_H_
#define MOCKIT_TIME_H_
#include <ctime>
time_t mockit_time(time_t*, const char*, int);
#define time(_timer_) \
  mockit_time((_timer_), __FILE__, __LINE__)
#endif
```

mockit_time.cpp

```
#include "mockit_time.h"
#undef time
time_t mockit_time(time_t* _timer_,
  const char* file, int line) {
  //time(_timer_); // production implementation
  return 986725194; // day in 2001
}
```


Speaker notes

Different implementations of the substitute function can be used for test and for production.

Note that because the preprocessor is not discriminating on C++ syntax and not type safe, this is error prone when the same identifier that has to be used for the macro will be used in other contexts as well, e.g., as a member function.

© 2024 Peter Sommerlad

Linker Seam

measure of last resort, tedious to get right

- re-implement (system) libary function in separate object file
- change the linker command to include that object file first/early
  - or use `LD_PRELOAD` trick
- violates the “One Definition Rule” of C++

deliberately no example given

Speaker notes

This is a trick for experts and can easily render a program unusable, when the replaced system function is essential in other parts.

It usually only works for C-linkage library functions. Even for system call functions it might fail, because they might actually be macros disguising the real function name.

In the above examples it could be used to replace `rand()` or `time()`, but like the preprocessor seam it should be considered a measure of last resort to enable special cases in testing.

© 2024 Peter Sommerlad

Further Improving Design for Testability

- eliminate circular dependencies 🔃
- reduce non-essential dependencies
  - `#include` what you use
  - member variable -> local variable/parameter of member function
  - move/split responsibility/functionality
- loosen coupling through parameterization

Speaker notes

All of these dependencies come with a compile-time and a run-time possiblity.

Often circular dependencies stem from misplaced abstractions or misplaced parts into the wrong component.

© 2024 Peter Sommerlad

`#include` what you use (IWYU)

Goals

- make header-files self-contained (easier to use)
- don't introduce unnecessary dependencies (minimal)

Guidelines

- *`#include`* own header in translation unit first
- *`#include`* system/std headers after project headers
- only *`#include`* headers that are really needed
- forward declarations of class types indicate potential circular dependencies

Speaker notes

Good IDEs should help with optimizing *`#include`* structures.

Making header files incoherent can lead to bloat. This is always a balancing action.

A class type definition should go together with free function declarations depending on the class type into the same namespace and best also into the same header.

© 2024 Peter Sommerlad

eliminate needless member (variables)

member variables should denote the state of the object and not just be accidental communication mechanism between member functions.

functions without implicit dependencies are easier to test

- make member variables only used within a member function a local variable
- consider passing (member-)variables as function arguments
- consider moving member function to a free (friend) function

Speaker notes

Some design approaches seem to favour using member variables over explicit parameters of member functions (IIRC “Uncle Bob Martin: Clean Code” is a bit guilty about having parameterless methods in Java code).

This couples functions to the class and thus lead to potential code duplication.

© 2024 Peter Sommerlad

Split class / extract function

inability to find good names is an indicator of non-cohesive functionality

- small is beautiful, unless it is too small
- look for the worst and see if you can extract useful parts
- in-line comments indicate potential sub-functions to extract
- long functions are impossible to test with good path coverage

Speaker notes

I observed: Developers addicted to interactive debugging love large functions, because they can step through without losing context. This is unfortunately a big time waster not only for the person needing to debug, but also for every programmer touching such code. Refactor large classes and especially long functions into smaller ones. Observe potential for reuse, so that in the end you might have less code through combining smaller pieces. Refactor mercilessly, especially when you have unit tests showing that you don't break stuff.

From example code given to me:

`void mmMaterial::SendAlarmsAndLogs()` has at least a McCabe complexity of 25 (hand counted…), 255 LoC, with visible duplication.

Parameterization { } ( ) <>

C++ elements can have parameters

- initialization: `var{value}`
- functions: `f(params)`
- templates: `T<tparams>`

arguments: compile time {}()<>, run time (){}

Speaker notes

This is a very rough overview.

© 2024 Peter Sommerlad

C++ Parameterization

We can parameterize several things:

- functions with function parameters
- lambdas with captures
- template with template parameters
  - class templates
  - function templates
  - variable templates
- template parameters:
  - class templates
  - types
  - compile-time values
  - global references
- argument deduction
  - function templates
  - class templates/constructo

Speaker notes

Parameterization is what makes code composable, testable, and reusable.

Relying on global state syntactically, e.g., writing to `std::cout`, makes code untestable and hard to reuse.

Remove unnecessary dependencies to objects/values/types by introducing parameters for them.

© 2024 Peter Sommerlad

Introduce Parameter Refactoring

Break up directly referred dependents

- values, e.g. literals: `20`
- variables (globals)
- functions (functions are first class objects in C++)
- types (as template parameters)

Speaker notes

In C++ you can add run-time parameters for functions and constructors to break a dependency to a concrete object, and you can have compile-time parameters for templates to break dependencies to types and global objects.

In C++ we can pass functions as arguments to other functions. This is easiest to specify when the target parameter is a deduced template parameter, but can be harder to call when the function is overloaded. Otherwise, directly specifying the parameter as a function pointer or the use of `std::function<>` template is possible.

© 2024 Peter Sommerlad

Introduce function parameter

Instead of relying on a global or member variable from a function body, pass the dependency as a function paramter

```
void sayhello() {
    std::cout << "!!!Hello World!!!\n";
}
```

refactor to

```
void sayhello(std::ostream &out) {
    out << "!!!Hello World!!!\n";
}
```

allows testing

Speaker notes

and also reduces dependency from `<iostream>` to `<ostream>` and `<iosfwd>` for declaration

That stream classes already have polymorphic behavior makes it easier to rely on testability.

© 2024 Peter Sommerlad

Introduce template value parameter

```
void turnOnLED(){
  auto const portd = reinterpret_cast<volatile uint8_t *> (0x2b);
  *portd = 0xFF;
}
```

->

```
template<auto portaddr = 0x2b>
void turnOnLEDT(){
  auto const portd = reinterpret_cast<volatile uint8_t *> (portaddr);
  *portd = 0xFF;
}
```

see https://godbolt.org/z/qenb735jr

Speaker notes

especially for hardware addresses it can be useful to make them replacable for tests. Template parameters help doing so without any run-time overhead.

Disadvantage: function body must be visible = defined in header (or use C++20 modules).

Advantage: template function body can more easily be inlined by the compiler (not only at link time optimization)

see https://godbolt.org/z/qenb735jr

© 2024 Peter Sommerlad

Summary

- Refactor to Seams
  - enables DOC replacement
  - can losen coupling
- Losen Coupling
  - eliminate hard-coded dependencies
  - global objects, global functions
  - parametrize

Speaker notes

Except for the Extract Interface refactoring, refactoring to seams can also losen coupling and enable DOC replacement.

© 2024 Peter Sommerlad

TDD Patterns

TDD: Test-driven Development

- TDD is not a testing technique, but a coding and design technique
  - TDD patterns help you writing tests
- TDD relies heavily on Refactoring

Speaker notes

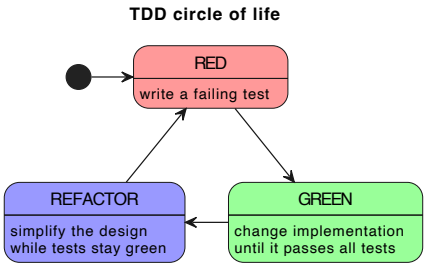
- There are several books on test-driven design (or TDD)
  - Kent Beck, Dave Astels, Gerard Meszaros, J.W. Grenning, Jeff Langr

Those authors books form the basis of this part and are used even when not directly cited.

[TDDAPG]
David Astels: Test-Driven Development: A Practical Guide
[TDDBE]
Kent Beck: Test-Driven Development By Example
[TDDFEC]
James W. Grenning: Test-Driven Development for Embedded C
[MCPWTDD]
Jeff Langr: Modern C++ Programming with Test-Driven Development
[XUP]
Gerard Meszaros: XUnit Patterns

TDD patterns help you writing tests, regardless if you follow TDD or not

TDD Circle



RED -> GREEN -> Refactor -> ...

all in very tiny steps 👣

Speaker notes

TDD writes production code only to make a failing test pass.

If you already think you use tiny steps, use smaller steps.

However, use “Obvious Implementation” when the problem to be solved is clear.

Being Overwhelmed?

Ward Cunningham:

Do the simplest thing that could possibly work!

when you don't know what to do.

Speaker notes

Image source:

https://commons.wikimedia.org/wiki/File:Ward_Cunningham_-_Commons-1.jpg

Carrigg Photography for the Wikimedia Foundation, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons

© 2024 Peter Sommerlad

Three Rules of TDD

1. Only write production code to pass a failing test.
2. Write no more of a unit tests than sufficient to fail.
   - compilation failures are failures.
3. Write no more production code than necessary to pass the one failing unit test.

Speaker notes

These rules were spelled by Robert C. Martin (Uncle Bob) and are taken from [MCPWTDD].

corollaries:

1. write tests first
2. proceed incrementally in steps as small as possible
3. do not run ahead with implementing production code without appropriate tests (would violate rule 1)

© 2024 Peter Sommerlad

Getting GREEN on RED 

- running the wrong tests
- testing the wrong code
- unfortunate test specification
- invalid assumptions
- suboptimal test order
- linked production code
- overcoding (violation of TDD rule 3)
- testing for confidence

Speaker notes

This is taken from [MCPWTDD].

wrong tests
: you used test filtering to omit the new test

wrong code
: linking the wrong library or object files, did you rebuild?

bad test
: logic of assertion wrong way around? this can be useful to ensure a test can fail and is executed, but should be changed afterwards

assumptions wrong
: behaviour already implemented?

test order
: you already have a test indicating the behaviour that you check with the new test. You didn't follow "Do the simplest thing that could possibly work." principle

TDD Patterns

- Practices
- Red Bar
- Green Bar
- Testing

Speaker notes

We are looking at some of the patterns presented in [TDDBE]

© 2024 Peter Sommerlad

Practices Patterns

Isolated Test

This is the I of A TRIP/PIRAT: a test should never depend on another test

Test List

Write down ideas for further tests, but only implement the next test

Test First

see Rule 1 of TDD

Assert First

when writing a test start with formulating the assertion at its end

Speaker notes

Assert First is also helpful for naming the test.

In general good test names help.

© 2024 Peter Sommerlad

Red Bar Patterns

One Step Test
Choose a test from the Test List that is not dependent on other functionality still on the List. ZOM from Zombies can help for selecting the next text

Starter Test
Z from ZOMBIES. start with the simplest test scenario.

Explanation Test
Use tests to communicate with peers. discuss design by formulating a test.

Learning Test
Familirize yourself with foreign code/API with tests checking your assumptions

Regression Test
Manifest bug reports as failing tests first

Break
Take enough breaks. Being tired is counterproductive.

Do Over
Delete code and start over, if feeling lost.

Speaker notes

These patterns adress the situation on how to get to the next “Red Bar” situation: what test to write next and how.

Do Over might feel uninituitive. Remember that you have a time machine (git/vcs) to revisit your deeds later. Not throwing out bad designs and bad tests costs more than the redoing. But consider refactoring first!

© 2024 Peter Sommerlad

Green Bar Patterns

Fake It ('Til You Make It)
just return a hard coded result

Triangulate
derive abstractions from at least two concrete examples

Obvious Implementation
for simple cases, don't fake first

One To Many
ZOM from ZOMBIES

Speaker notes

These patterns address the situation when you don't know enough yet to create a working solution.

They follow the principle "Do the simplest thing that could possibly work."

If the problem is too complicated to grasp it from one test, then faking the solution to just suit that single test is OK, because it allows you to write the next test. With two or more tests you might be able to triangulate the solution. Nevertheless, when you know the solution *Obvious Implementation* is OK, but beware to not over-engineer then.

Zero-One-Many help with getting started and have a set of tests and (fake) solutions to triangulate from.

© 2024 Peter Sommerlad

Testing Patterns

Child Test
if test case gets big, extract core, “remove the rest” and add after getting green
Mock Object
Decouple SUT from environment (DOCs)
Self Shunt
make the test case class the mock object
Log String
collect mocked call sequence
Chrash Test Dummy
have a Mock/Dummy as DOC to simulate error conditions
Broken Test
Keep a failing test before a break to get started again
Clean Check-in
Never ever check in code and tests that fail

Speaker notes

Log String is the simple DIY version, where Mocking Frameworks often provide corresponding recording and matching infrastructure by a DSL.

© 2024 Peter Sommerlad

TDD interactive Example

- FizzBuzz
  - count numbers from 1
  - for a number divisible by 3 emit “Fizz”
  - for a number divisible by 5 emit “Buzz”
  - for a number divisible by 15 emit “FizzBuzz”
- Roman Numbers
  - 2024 -> “MMXXIV”
- Expression Evaluator
  - “2 + 1” -> 3

Start: https://godbolt.org/z/TEGad9Gvx

Speaker notes

We chose one example based on the time we have and then mob-program.

Start: https://godbolt.org/z/TEGad9Gvx

© 2024 Peter Sommerlad

TDD summary

- Test first makes you a victim of interface design and gives immediate feedback
- rehearsing advice given earlier
- Really work in small steps.
  - If you thinks your steps are small: Take even smaller steps!

Do the simplest thing that could possibly work!

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Testing C++ Templates

- Be clear about the concepts of the template parameters
- Write “normal” tests for representative template argument(s)
- Consider compilability tests for unwanted template arguments
- To ensure later changes don’t break assumptions, special template arguments for tests can be used

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Testing Compilability

- SFINAE can be used, but requires compilation failure in an expression

```
template<typename FROM, typename=void>
constexpr bool
from_int_compiles=false;
template<typename FROM>
constexpr bool
from_int_compiles<FROM,std::void_t<decltype(pssodin::from_int(FROM{}))>> =
        true;
static_assert(! from_int_compiles<bool>);
static_assert(! from_int_compiles<char>);
static_assert(from_int_compiles<unsigned char>);
```


Speaker notes

Unfortunately, even so a lambda expression is an expression, compilers are not looking into a lambda's body for compile errors during template substitution. This can lead to hard errors that are not easily exploitable in a testing setting.

© 2024 Peter Sommerlad

Testing Compilabilty Expressions

```
namespace compile_checks {
using namespace pssodin;
template<auto ...value>
using consume_value = void;
#define concat_line_impl(A, B) A##_##B
#define concat_line(A, B) concat_line_impl(A,B)
#define check_does_compile(NOT, FROM, oper) \
namespace concat_line(NOT##_test, __LINE__) { \
        template<typename T, typename=void>\
        constexpr bool\
        expression_compiles{false};\
template<typename T> \
constexpr bool \
expression_compiles<T, consume_value<(T{} oper T{})> > {true};\
static_assert(NOT expression_compiles<FROM>, "should " #NOT " compile: " #FROM "{}" #oper #FROM "{}");\
} // namespace tag
// need to be on separate lines for disambiguation
check_does_compile(not, csi8 , <<  )
check_does_compile(not, cui8 , + (1_cui8 << 010_cui8) + ) // too wide shift
check_does_compile(   , cui8 , + (1_cui8 << 7_cui8) + ) // not too wide shift
```


Speaker notes

If this looks to convoluted, yes it is. Using macros for code generation to create distinguishible C++ names.

The underlying principle is the same as in the previous example, by using SFINAE and an expression that is dependent on a template parameter. The expression_compiles variable template comes with a base template with a defaulted second typename parameter and then a specialization that checks for the validity of a dependent expression. Most operations are checking for combining values with the given operator, but because oper is a macro parameter, we can provide more complicated expressions as part of the checking.

The `consume_value` alias template works similarly to `std::void_t<T...>` by consuming value template parameters. Alternatively, one could have used `std::void_t<decltype(expression)>` like in the previous example.

If/when C++ obtains universal template parameters `std::void_t<>` can be adapted to accept both, value and typename parameters.

Limits of compilability Tests

- SFINAE only works for expressions and template instantiations
- Instantiation failure due to concept mismatch is a hard compile error
- One cannot tests with templates for (example):
  - structured bindings
  - **`requires`** failure

Speaker notes

Unfortunately the body of a lambda expression is not considered when SFINAE happens. Even so being an expression, lambda bodies contain statements and that is still considered too much to check for a compiler, or may be there are additional reasons beyond my grasp.

© 2024 Peter Sommerlad

Special template arguments for testing

- check for copy or move support
- check for expected allocations
- check for compilability with move-only types
- check for behavioral error handling
- check for out-of-memory handling
- check for weirdly throwing operations

https://github.com/PeterSommerlad/CPPCourseExpert/tree/main/exercises

Speaker notes

For example see https://github.com/PeterSommerlad/CPPCourseExpert/tree/main/exercises

© 2024 Peter Sommerlad

Example: non-copy-non-move Type

```
struct NonCopyableNonMovable {
    NonCopyableNonMovable(int i , double d):i{i},d{d} {}
    // Rule of DesDeMovA:
    NonCopyableNonMovable&
    operator=(NonCopyableNonMovable&&)noexcept = delete;
    int i;
    double d;
};
void test_emplace_works(){
    BoundedBuffer<NonCopyableNonMovable> buffer{5};
    buffer.push_emplace(42,3.14);
    ASSERT_EQUAL(42,buffer.front().i);
}
```


Speaker notes

for your own notes

© 2024 Peter Sommerlad

Example: container element for non-default constructible

```
struct NonDefaultConstructible {
    explicit NonDefaultConstructible(int) : value{1234}
        {}
    ~NonDefaultConstructible() {
        throwIfNotInitialized();
        nOfDtorCalls++;
    }
    NonDefaultConstructible(NonDefaultConstructible
        const & other)
  : value{other.value} {
        throwIfNotInitialized();
        nOfCopyConstructions++;
    }
    NonDefaultConstructible& operator=
        (NonDefaultConstructible const & other) {
        throwIfNotInitialized();
        other.throwIfNotInitialized();
        value = other.value;
        nOfCopyAssignments++;
        return *this;
    }
    NonDefaultConstructible& operator=
        (NonDefaultConstructible && other) {
        throwIfNotInitialized();
        other.throwIfNotInitialized();
        std::swap(value, other.value);
        nOfMoveAssignments++;
        return *this;
    }
    NonDefaultConstructible(NonDefaultConstructible && other)
  : value{1234} {
        other.throwIfNotInitialized();
        std::swap(value, other.value);
        nOfMoveConstructions++;
    }
    static inline unsigned nOfCopyConstructions{0};
    static inline unsigned nOfCopyAssignments{0};
    static inline unsigned nOfMoveConstructions{0};
    static inline unsigned nOfMoveAssignments{0};
    static inline unsigned nOfDtorCalls{0};
private:
    void throwIfNotInitialized() const {
        if (value != 1234) {
            throw std::logic_error{"Operation on NDC with value:
        " + std::to_string(value)};
        }
    }
    volatile unsigned value;
};
void resetCounters() {
    NonDefaultConstructible::nOfCopyConstructions = 0;
    NonDefaultConstructible::nOfCopyAssignments = 0;
    NonDefaultConstructible::nOfMoveConstructions = 0;
    NonDefaultConstructible::nOfMoveAssignments = 0;
    NonDefaultConstructible::nOfDtorCalls = 0;
}
```


Speaker notes

A typical test case with that looks like:

```
void test_element_in_buffer_is_destroyed_once() {
    {
        BoundedBuffer<NonDefaultConstructible> buffer{5};
        buffer.push(NonDefaultConstructible{23});
        resetCounters();
    }
    ASSERT_EQUAL(1, NonDefaultConstructible::nOfDtorCalls);
}
```

Example Test for Container with non-default constructible elements

```
void test_lvalue_push_copies_element() {
    resetCounters();
    BoundedBuffer<NonDefaultConstructible> buffer{5};
    NonDefaultConstructible element{23};
    buffer.push(element);
    ASSERT_EQUAL(1, NonDefaultConstructible::nOfCopyConstructions);
}
void test_rvalue_push_moves_element() {
    resetCounters();
    BoundedBuffer<NonDefaultConstructible> buffer{5};
    NonDefaultConstructible element{23};
    buffer.push(std::move(element));
    ASSERT_EQUAL(1, NonDefaultConstructible::nOfMoveConstructions);
}
```


Speaker notes

above examples taken from `bounded_buffer_non_default_constructible_suite.cpp` available at https://github.com/PeterSommerlad/CPPCourseExpert/raw/refs/heads/main/exercises/exercise03/BoundedBufferRawMem
or as permalink:
https://github.com/PeterSommerlad/CPPCourseExpert/blob/2d2be766b7f453d5d2d48ee7534d98ee9102127a/exercises/e

© 2024 Peter Sommerlad

More?

- AMA: Ask me anything?

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Done...

Feel free to contact me @PeterSommerlad@mastodon.social (🦣) or peter.cpp@sommerlad.ch in case of further questions

Speaker notes

stay tuned for C++Expert if you really want to know how to create modern good C++ libraries.

Unfortunately, even 12 days of intensive training with a lot of exercises in between can not cover all of the interesting details of C++ one can encounter.

© 2024 Peter Sommerlad

Appendix: Why I don't like Google Test:

```
TEST(HelloTest, sayHelloSaysHello) {
```

becomes

```
static_assert(sizeof("HelloTest") > 1,                \
              "test_suite_name must not be empty");                     \
  static_assert(sizeof("sayHelloSaysHello") > 1,                    \
                "test_name must not be empty");                         \
  class HelloTest_sayHelloSaysHello_Test                        \
      : public ::testing::Test {                                          \
   public:                                                                \
    HelloTest_sayHelloSaysHello_Test() = default;               \
    ~HelloTest_sayHelloSaysHello_Test() override = default;     \
    HelloTest_sayHelloSaysHello_Test                            \
    (const HelloTest_sayHelloSaysHello_Test &) = delete;        \
    HelloTest_sayHelloSaysHello_Test & operator=(               \
        const HelloTest_sayHelloSaysHello_Test &) = delete; /* NOLINT */      \
    HelloTest_sayHelloSaysHello_Test                            \
    (HelloTest_sayHelloSaysHello_Test &&) noexcept = delete;    \
    HelloTest_sayHelloSaysHello_Test & operator=(               \
        HelloTest_sayHelloSaysHello_Test &&) noexcept = delete; /* NOLINT */  \
                                                                            \
   private:                                                             \
    void TestBody() override;                                           \
    static ::testing::TestInfo* const    \
        test_info_;                                                     \
  };                                                                    \
                                                                        \
  ::testing::TestInfo* const HelloTest_sayHelloSaysHello_Test::test_info_ =   \
      ::testing::internal::MakeAndRegisterTestInfo(                     \
          "HelloTest", "sayHelloSaysHello", nullptr, nullptr,                \
          ::testing::internal::CodeLocation("hellogtest.cpp", 5), (::testing::internal::GetTestTypeId()),  \
          ::testing::internal::SuiteApiResolver<                        \
              ::testing::Test>::GetSetUpCaseOrSuite("hellogtest.cpp", 5),       \
          ::testing::internal::SuiteApiResolver<                        \
              ::testing::Test>::GetTearDownCaseOrSuite("hellogtest.cpp", 5),    \
          new ::testing::internal::TestFactoryImpl<HelloTest_sayHelloSaysHello_Test>);                              \
  void HelloTest_sayHelloSaysHello_Test::TestBody()
```


Speaker notes

There are more reasons than given here, but just one of the "strange" design decisions made by GoogleTest.

One of the few goodies of GoogleTest are the crash-tests, that actually expect a program to crash under circumstances.

© 2024 Peter Sommerlad

TEST() macro generates tight coupling

```
static_assert(sizeof("HelloTest") > 1,                \
              "test_suite_name must not be empty");
        \
  static_assert(sizeof("sayHelloSaysHello") > 1,                    \
              "test_name must not be empty");
        \
  class HelloTest_sayHelloSaysHello_Test                   \
      : public ::testing::Test {
        \
```


Speaker notes

Making the test case inherit from a framework's base class tightly couples tests with the framework.

© 2024 Peter Sommerlad

TEST macro generates redundant definitions

```
public:
     \
 HelloTest_sayHelloSaysHello_Test() = default;            \
 ~HelloTest_sayHelloSaysHello_Test() override = default;  \
 HelloTest_sayHelloSaysHello_Test                         \
 (const HelloTest_sayHelloSaysHello_Test &) = delete;     \
 HelloTest_sayHelloSaysHello_Test & operator=(            \
     const HelloTest_sayHelloSaysHello_Test &) = delete; /* NOLINT */
     \
 HelloTest_sayHelloSaysHello_Test                         \
 (HelloTest_sayHelloSaysHello_Test &&) noexcept = delete; \
 HelloTest_sayHelloSaysHello_Test & operator=(            \
     HelloTest_sayHelloSaysHello_Test &&) noexcept = delete; /* NOLINT
     */  \
```


Speaker notes

If the base class with virtual functions doesn't prevent copying and provides a virtual destructor then it is broken anyway.

All the special member functions would be provided by the compiler as specified without further code - Rule of Zero.

I cannot understand, why Rule of Zero is not even adhered to in generated code that a user wouldn't normally see.

May be Google has a style checker to enforcing Rule of Six, but that is ridiculous, because such a rule can hinder compiler optimizations.

© 2024 Peter Sommerlad

TEST macro relies on dynamic initialization of static lifetime pointer

```
   static ::testing::TestInfo* const    \
     test_info_;                                                          \
};                                                                        \
                                                                          \
::testing::TestInfo* const HelloTest_sayHelloSaysHello_Test::test_info_ =   \
    ::testing::internal::MakeAndRegisterTestInfo(                         \
        "HelloTest", "sayHelloSaysHello", nullptr, nullptr,                   \
        ::testing::internal::CodeLocation("hellogtest.cpp", 5),
      (::testing::internal::GetTestTypeId()),  \
        ::testing::internal::SuiteApiResolver<                            \
            ::testing::Test>::GetSetUpCaseOrSuite("hellogtest.cpp", 5),       \
        ::testing::internal::SuiteApiResolver<                            \
            ::testing::Test>::GetTearDownCaseOrSuite("hellogtest.cpp", 5),    \
        new ::testing::internal::TestFactoryImpl<HelloTest_sayHelloSaysHello_Test>);
      \
```


Speaker notes

The sequence of dynamic initialization is not defined across translation units. Automatic test registration relies on dynamic initialization of objects with static storage duration.

While it is possible to make tricks that enforce that the registry object is initialized before all registrations happen, it is not generally advisable and might not work, if production code also relies on a specific static initialization happens before some other.

When code is put into shared libraries/DLLs such initialization dependencies can break down.

In addition we see the use of `operator new` for heap allocation instead of `std::make_unique()`

Final thoughts on GoogleTest

- It requires linking
- When not using the provided main() function, writing own main can become complicated
- The EXPECT macros foster tests with more than one reason to fail
- Too much magic is hidden in complex to use macros, when compilation fails in expanded code it is very hard to diagnose

Speaker notes

© 2024 Peter Sommerlad