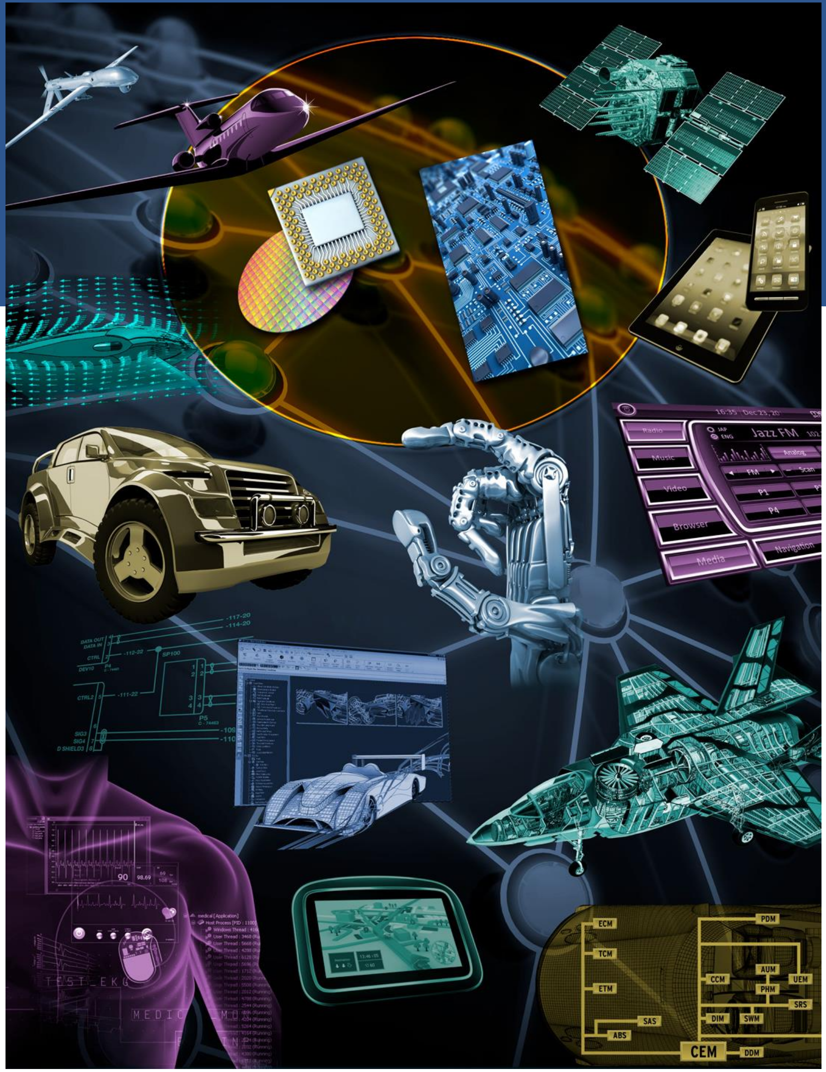


Test-Driven Development in C++
(back to basics)

Fedor G Pikus

Chief Scientist,

Design2Silicon Division

CPPCon 2019

Mentor®
A Siemens Business

Test-Driven Development in a Nutshell

- Test-Driven Development: write tests before the code
  — Seems backwards, why test when there is nothing to test?
- You’re probably doing it already, in part
- This talk is not “only” about TDD
- Much of it is just about good testing practices

TEST-DRIVEN DEVELOPMENT

Test-Driven Development in a Nutshell

- Test-Driven Development: write tests before the code
  - Seems backwards, why test when there is nothing to test?
- You're probably doing it already, in part
- This talk is not "only" about TDD
- Much of it is just about good testing practices
- Test-driven development is a software engineering process
  - It helps the programmers to organize their thoughts and keep discipline
  - It offers a method and a system for interactions within teams
  - It helps to set and track goals
  - It helps to approach complex problems and partition them
  - It's mostly in your head

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…
  - Why?!

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features — Before ANY code is written?
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Test-Driven Development in a Nutshell

- Very short development cycle:
  - — Write tests for the new features
  - — Run tests, expect them to fail
  - — Write new code to make tests pass
  - — Run tests, confirm that they pass
  - — Refactor and improve the code
  - — Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Before ANY code is written?

Run?! They won't even compile!

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features — Should we declare interfaces first?
  - Run tests, expect them to fail — Should we write dummy implementation?
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Just to make test pass, nothing more?

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass
- Many interesting questions can be asked here…

Was it enough tests?

Test-Driven Development in a Nutshell

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass

Until when?

- Many interesting questions can be asked here…

Test-Driven Development in a Nutshell

- Very short development cycle:
  - — Write tests for the new features
  - — Run tests, expect them to fail
  - — Write new code to make tests pass
  - — Run tests, confirm that they pass
  - — Refactor and improve the code
  - — Run tests, confirm that they still pass
- Many interesting questions can be asked here…

What kind of tests?

Are these still the same tests?

TESTS, TESTS, AND MORE TESTS

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- System tests: tests for the entire software system

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- System tests: tests for the entire software system

Confusion alert!
"System testing" sometimes used instead of "integration testing"
"End-to-end testing" used instead of "system testing"

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- End-to-end tests: tests for the entire software system
- Point tests: “unit” tests using the whole system as a test driver
  - Technically end-to-end tests but the scope is narrowly targeted

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- End-to-end tests: tests for the entire software system
- Point tests: “unit” tests using the whole system as a test driver
- Acceptance tests: tests for passing certain requirements
  — Usually lighter suit of tests

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- End-to-end tests: tests for the entire software system
- Point tests: “unit” tests using the whole system as a test driver
- Acceptance tests: tests for passing certain requirements
- Regression tests: tests done to detect unexpected changes
  - Can be any of the above type

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- End-to-end tests: tests for the entire software system
- Point tests: “unit” tests using the whole system as a test driver
- Acceptance tests: tests for passing certain requirements
- Regression tests: tests done to detect unexpected changes
- Performance tests: tests for meeting performance targets
  - Can be any of the above type

What kinds of tests are there?

- Unit tests: tests for a unit of code in isolation
- Integration tests: tests for interaction of several components
- End-to-end tests: tests for the entire software system
- Point tests: “unit” tests using the whole system as a test driver
- Acceptance tests: tests for passing certain requirements
- Regression tests: tests done to detect unexpected changes
- Performance tests: tests for meeting performance targets
  - Can be any of the above type
- One of the most common reasons for abandoning a test suite is trying to use the wrong kind of tests

Unit Testing

- Most commonly used with test-driven development
- The kind of tests developers run more than any other
- Most misunderstood

Unit Testing

- Most commonly used with test-driven development
- The kind of tests developers run more than any other
- Most misunderstood

Unit Testing

- Testing a unit of code outside of the rest of the program
- What is a “unit”?
- How to test a “unit” by itself?

Unit Testing – What is a Unit?

- Unit is a section of code that has a specific function
- In object-oriented programming, classes are (almost always) units
  - But not all units are classes
- Functions, procedures, modules, class methods can be units
- Units are often small
  - We always start testing with the smallest units
- Units may be very large
  - A database could be a unit
  - Large units are built from smaller units that were tested earlier

Unit Testing – Without the Program?

- Unit testing does not use the rest of the program
- Where do we get main()? – it's written specifically for each test
  - Doing so by hand is hard
  - There are unit testing frameworks to simplify and automate unit testing
- Where do we get the test data?
  - More generally, how do we recreate the internal state of the unit to be exactly what we need for the test we want to run?
  - Sometimes it's trivial
  - It could also be very difficult (this is mainly why programmers switch to point tests)
  - There are many techniques to manage the test data problem

Unit Testing – Example – Sudoku Puzzle Solver

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Unit Testing – Example – Sudoku Puzzle Solver

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Unit Testing – Example – Sudoku Puzzle Solver

- Unit – Sudoku class (represents the puzzle)

```
class Sudoku {
    public:
    Sudoku();
    unsigned char get(size_t i, size_t j) const;
    void set(size_t i, size_t j, unsigned char value);
};
```

Unit Testing – Example – Sudoku Puzzle Solver

- Unit tests using GoogleTest

```
#include "sudoku.h“                    // Code to test
#include <gtest/gtest.h>               // Unit test framework
TEST(Sudoku, Construct) {
    Sudoku S;                          // Should be empty
    EXPECT_EQ(0, S.get(0, 0));         // Make sure it is
    EXPECT_EQ(0, S.get(8, 8));
}
```

- No main() – provided by the framework

Unit Testing – Example – Sudoku Puzzle Solver

- Unit tests using GoogleTest

```
TEST(Sudoku, Set) {
    Sudoku S;                          // Already tested
    S.set(0, 0, 5);                    // Should fill a cell
    EXPECT_EQ(5, S.get(0, 0));         // Did it?
    EXPECT_EQ(0, S.get(0, 2));         // Did it mess anything up?
}
```

Unit Testing – Example – Sudoku Puzzle Solver

- All aspects of the documented behavior should be tested

```
TEST(Sudoku, Set) {
    Sudoku S;
    S.set(0, 0, 5);
    EXPECT_THROW(S.get(9, 0), std::logic_error);
    EXPECT_THROW(S.set(2, 1, 10), std::logic_error);
    EXPECT_THROW(S.set(0, 0, 6), std::logic_error);}
}
```

Test-Driven Development Often Uses Unit Tests

- Test-driven development emphasizes incremental development
- We need tests for just the code we have added in the last step
- Testing is faster if we don't have to build the whole program
- It is often hard to fully test a unit (component) using the rest of the system as the test driver (hard to create certain inputs)
- The rest of the system may not exist yet!
  - We code in small steps, the first "complete" program does not happen until after many steps
- There is no rule that TDD must use only unit tests!

THE PROCESS

Test-Driven Development Process

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass

Test-Driven Development – Many Questions

- Very short development cycle:
  - Write tests for the new features

    Before ANY code is written?
  - Run tests, expect them to fail

    Should they compile? Link with dummy implementation?
  - Write new code to make tests pass

    Just to make test pass, nothing more?
  - Run tests, confirm that they pass

    Was it enough tests?
  - Refactor and improve the code
  - Run tests, confirm that they still pass

    Until when?

Test-Driven Development – Many Questions

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass

Before ANY code is written?

Should they compile? Link with dummy implementation?

Just to make test pass, nothing more?

Was it enough tests?

Until when?

When to Write Tests

1. Before any code is written
   – They won't compile
   + They describe how the client code should look like
   + They specify the requirements on the interface in C++ instead of English

When to Write Tests

1. Before any code is written
   - − They won't compile
   - + They describe how the client code should look like
   - + They specify the requirements on the interface in C++ instead of English

Note to self:
Can tests be used instead of a spec?

When to Write Tests

1. Before any code is written
   - − They won't compile
   - + They describe/mockup how the client code would look like
   - + They specify the requirements on the interface in C++ instead of English
2. After the interfaces are declared
   - −+ They will compile but won't link
3. After writing a dummy implementation
   - + The minimum needed to compile and run the tests
   - + We expect the tests to fail
     - But some won't!
     - We know whether our tests are sensitive enough to detect that we at least tried to write some real code
   - −+ Interfaces must be written to the spec (no client code mockup)

When to Write Tests

1. Before any code is written
   - − They won't compile
   - + They describe/mockup how the client code would look like
   - + They specify the requirements on the interface in C++ instead of English
2. After the interfaces are declared
   - −+ They will compile but won't link
3. After writing a dummy implementation
   - + The minimum needed to compile and run the tests
   - + We expect the tests to fail
     - But some won't!
     - We know whether our tests are sensitive enough to detect that we at least tried to write some real code
   - −+ Interfaces must be written to the spec (no client code mockup)

Wait, what?!

Testing Pitfalls: Even a Broken Clock is Right Twice a Day

- Function to test:

```
double add(double x, double y);
```

- Dummy implementation:

```
int add(int x, int y) {
    return 42;          // Why not - has to return something
}
```

- The unluckiest test in the world:

```
EXPECT_EQ(42, add(39,3));
```

When to Write Tests

1. Before any code is written

Example-driven design

   − They won't compile
   + They describe/mockup how the client code would look like
   + They specify the requirements on the interface in C++ instead of English

2. After the interfaces are declared
   −+ They will compile but won't link

Most common

3. After writing a dummy implementation
   + The minimum needed to compile and run the tests
   + We expect the tests to fail
     – But some won't!
     – We know whether our tests are sensitive enough to detect that we at least tried to write some real code
   −+ Interfaces must be written to the spec (no client code mockup)

Test-Driven Development – Many Questions

- Very short development cycle:
  - Write tests for the new features

    Before ANY code is written?
  - Run tests, expect them to fail

    Should they compile? Link with dummy implementation?
  - Write new code to make tests pass

    Just to make test pass, nothing more?
  - Run tests, confirm that they pass

    Was it enough tests?
  - Refactor and improve the code
  - Run tests, confirm that they still pass

    Until when?

Test-Driven Development: How Much Implementation is Enough?

- Function to test:

```
double add(double x1, double x2, double x3, double x4);
```

- Tests:

```
EXPECT_EQ(10, add(1,2,3,4));     // Can’t test all inputs anyway…
```

- Implementation to pass tests:

```
double add(double x1, double x2, double x3, double x4) { return 10; }
```

- That’s not what they mean when they say “write just enough code to pass tests”

Test-Driven Development: How Much Implementation is Enough?

```
double add(double x1, double x2, double x3, double x4) {
    if (CPUID() & 0x1F25…73) {      // Has AVX!
        inline asm { vpadd … };
    } else if (*magicptr == 3 && x1 >= 16384) {
        return x2-(-x1)+…
        // On FunkyRISK Rev 3, subtraction is faster for large numbers
    } else { … }
}
```

- That’s what they mean when they say “write just enough code to pass tests”

Test-Driven Development – Many Questions

- Very short development cycle:
  - Write tests for the new features — Before ANY code is written?
  - Run tests, expect them to fail — Should they compile? Link with dummy implementation?
  - Write new code to make tests pass — Just to make test pass, nothing more?
  - Run tests, confirm that they pass — Was it enough tests?
  - Refactor and improve the code
  - Run tests, confirm that they still pass

Until when?

How Many Tests is Enough?

- We will never have full test coverage:

```
EXPECT_DOUBLE_EQ(2.0,10.0/5.0);
EXPECT_EQ(5.5,94.05/17.1);
```

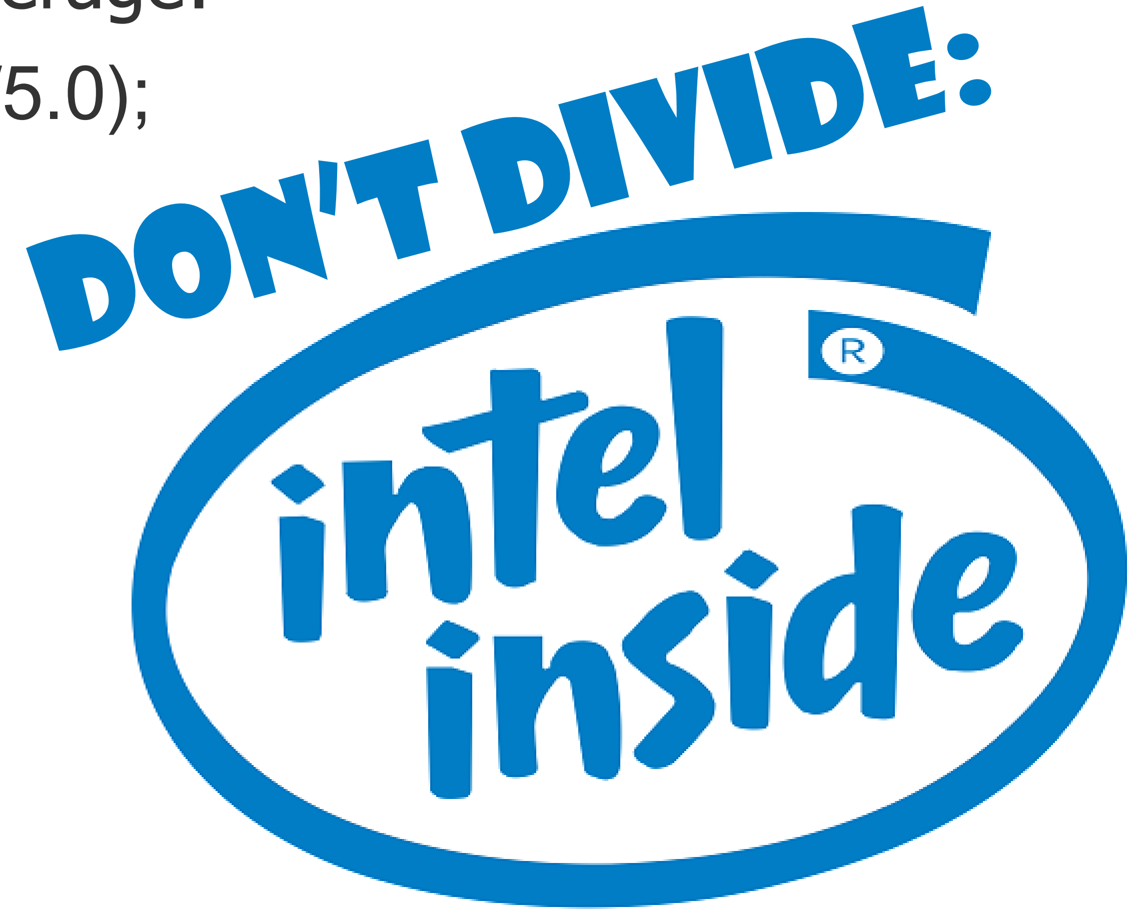
… and many more …

How Many Tests is Enough?

- We will never have full test coverage:

```
EXPECT_DOUBLE_EQ(2.0,10.0/5.0);
EXPECT_EQ(5.5,94.05/17.1);
```

… and many more …



How Many Tests is Enough?

- We will never have full test coverage
- We have to trust that some things “just work”
  - They should have been tested earlier, but that’s no guarantee
- One or few tests for each “normal case”
- Tests for all “special cases”, “corner cases”, etc.

How Many Tests is Enough?

- One or few tests for the “normal case”

```
class Sudoku {};
```

- Is this enough:

```
Sudoku S;
S.set(0, 0, 5);                  // One cell is just like another
EXPECT_EQ(5, S.get(0, 0));       // No value is special
```

- Do we need to test every cell? Every value?
- Test automation is very important for repetitive tests
  - All good testing framework have it

How Many Tests is Enough?

- One or few tests for the "normal case"
- We have to be practical about this
- The tests are written by the developer – "white box" testing
  - You know which two cases are essentially the same and which ones are handled very differently
- One of the most common reason for declining testing discipline is that tests take too long to run!

How Many Tests is Enough?

- One or few tests for the "normal case"
- A test for each "special case"
- Be mindful of splitting development into small steps:

```
EXPECT_EQ(5, S.get(2, 3));          // General case
EXPECT_EQ(7, S.get(8, 8));          // Corner cell, could be "special"
EXPECT_THROW(S.get(9, 0), std::logic_error);
```

- One of these is not like the others!

How Many Tests is Enough?

- One or few tests for the "normal case"
- A test for each "special case"
- Be mindful of splitting development into small steps:

```
EXPECT_EQ(5, S.get(2, 3));          // General case
EXPECT_EQ(7, S.get(8, 8));          // Corner cell, could be "special"
```

- Error handling is a logically separate function and should be tested in a separate step (could be before or after the normal case)

```
EXPECT_THROW(S.get(9, 0), std::logic_error);
```

Test-Driven Development – Many Questions

- Very short development cycle:
  - Write tests for the new features
  - Run tests, expect them to fail
  - Write new code to make tests pass
  - Run tests, confirm that they pass
  - Refactor and improve the code
  - Run tests, confirm that they still pass

Before ANY code is written?

Should they compile? Link with dummy implementation?

Just to make test pass, nothing more?

Was it enough tests?

Until when?

When Good is Good Enough?

- Code refactoring is often done for clarity and maintainability
  — "Good enough" is up to the programmer (reviewers, coding guidelines)

When Good is Good Enough?

- Code refactoring is often done for clarity and maintainability
  — “Good enough” is up to the programmer (reviewers, coding guidelines)
- Code improvement can be optimization
  — Needs performance goals and tests to verify the results
- Code reorganization can be required for the next coding step
  — New interfaces are needed, additional options and parameters, etc.
  — Do not mix changes and new development!
- Code improvements may themselves be guided by tests
  — More tests are added during this stage

TESTING IS OBSERVING

What Can (or Should) Be Tested?

- We tested user-observable behavior (public API)
- In class hierarchies, we can test restricted API (protected)
- Is there anything else?

What Can (or Should) Be Tested?

```
template <class Cntr> void sort(Cntr& C, size_t from, size_t to) {
    for (size_t i = from + 1; i <= to; ++i) {
        if (C[i] < C[i - 1]) {
            auto tmp = C[i];
            size_t j = i;
            do { C[j] = C[j - 1]; } while ((--j > from) && (tmp < C[j - 1]));
            C[j] = tmp;
        }
    }
}
```

- Insertion sort… should be easy to test

What Can (or Should) Be Tested?

- General case:

```
int C[] {2, 8, 1, 4}; sort(C, 0, std::size(C)); … test results …
```

- Special cases:

```
int C[] {1, 2, 3, 4};
int C[] {4, 3, 2, 1};
int C[] {5, 2, 7, 2};
sort(C, 2, 5);
sort(C, 0, 0);
```

- All observable behavior is correct

What Can (or Should) Be Tested?

- The “general” case wasn’t very general (used only one type of C):

```
vector<int> C {2, 8, 1, 4}; sort(C, 0, std::size(C)); … test results …
```

- Works fine
- Probably

What Can (or Should) Be Tested?

- The “general” case wasn’t very general:

vector<int> C {2, 8, 1, 4}; sort(C, 0, std::size(C)); … test results …

- Works fine
- Probably
- Until it doesn’t:

```
/usr/include/c++/7/debug/vector:417:
In function:
../run_t: line 4: 22383 Segmentation fault      (core dumped) ./$FILE ${@:2}
```

What Can (or Should) Be Tested?

- The “general” case wasn’t very general:

vector<int> C {2, 8, 1, 4}; sort(C, 0, std::size(C)); … test results …

- Works fine
- Probably
- Until it doesn’t:

```
/usr/include/c++/7/debug/vector:417:
In function:
../run_t: line 4: 22383 Segmentation fault      (core dumped) ./$FILE ${@:2}
```

  — But only if compiled with certain options (-D_GLIBCXX_DEBUG on GCC)
  — Detects out-of-bound access

What Can (or Should) Be Tested?

```
class Sudoku {
  public:
  Sudoku();
  unsigned char get(size_t i, size_t j) const;
  void set(size_t i, size_t j, unsigned char value);
};
```

- We tested user-observable behavior (public API)
- In class hierarchies, we can test restricted API (protected)
- Is there anything else?

What Can (or Should) Be Tested?

```
class Sudoku {
    public:
    Sudoku();
    unsigned char get(size_t i, size_t j) const;
    void set(size_t i, size_t j, unsigned char value);
    private:
    unsigned char cells_[9][9];
};
```

What Can (or Should) Be Tested?

```
class Sudoku {
  public:
  Sudoku();
  unsigned char get(size_t i, size_t j) const;
  void set(size_t i, size_t j, unsigned char value);
  private:
  unsigned char cells_[9][9];
};
```

- Can cells_ be accessed out of bounds?

Should Implementation Be Tested?

- Yes: many errors cannot be observed through public APIs
- No: implementation can change, tests will be fragile
- One of the most common reasons for abandoning a test suite is the need to keep updating the tests

Should Implementation Be Tested?

- Yes: many errors cannot be observed through public APIs
- No: implementation can change, tests will be fragile
- Sometimes, but only as a last resort, when everything else fails
  — "Everything else" is a lot of tools and techniques!
- What other options are there?

Does Testing Implementation Really Test Implementation?

- The problem: out of bound access of the container
  - Or a class data member array
  - Or a method call on a data member with invalid parameters
- The root of the problem is still an incorrect API use
- We were testing the interface of the unit
- We should be testing the interfaces the unit uses
  - Subtle difference vs testing implementation: we’re not testing exactly what the implementation does, only that it doesn’t violate “internal” contracts as well as “external” contracts

How to Test Internal Contracts?

- External contracts are easy to test: we use special testing main()
  - More generally, a special client program just for testing
- Internal code is whatever is compiled in, it's not made for testing
- For testing, we need substitutes for lower-level components
- These substitutes are called “test doubles”
  - They are “doubles” of the real thing but with additional capabilities (think “body double” in the movies)

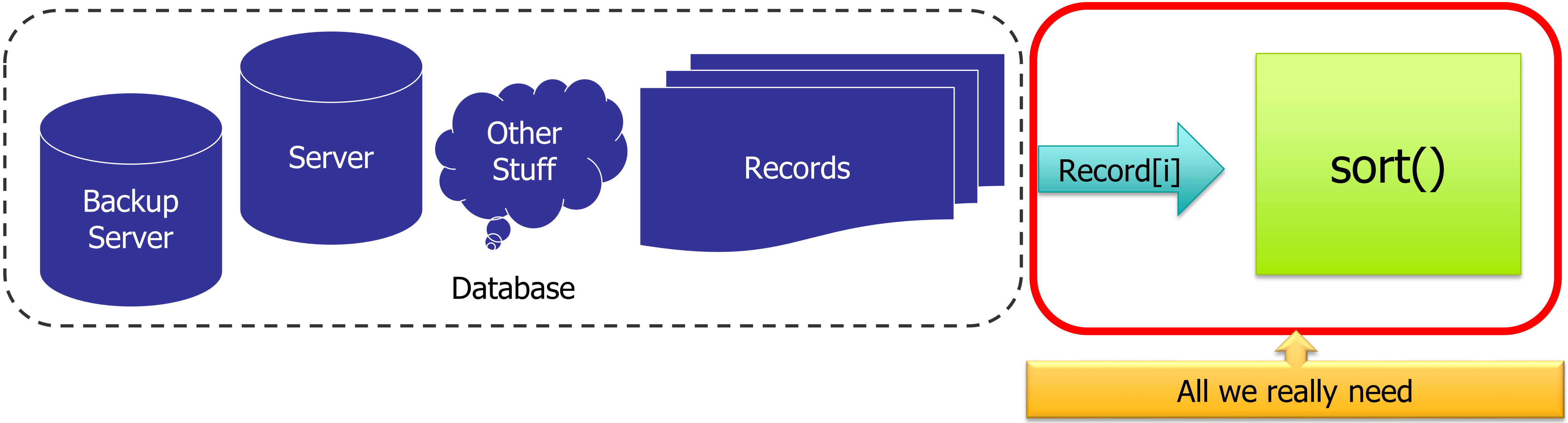
TEST DOUBLES

What Can Test Doubles Do?

- Test the code “from the inside”
  - Verify that all the calls the tested code makes to other units are valid
  - Verify that all the calls the tested code makes to other units are expected
- Allow testing before the real implementation of other units exists
- Simplify creating the desired state for testing
  - Sorting a container is easy, how about sorting records in a database?

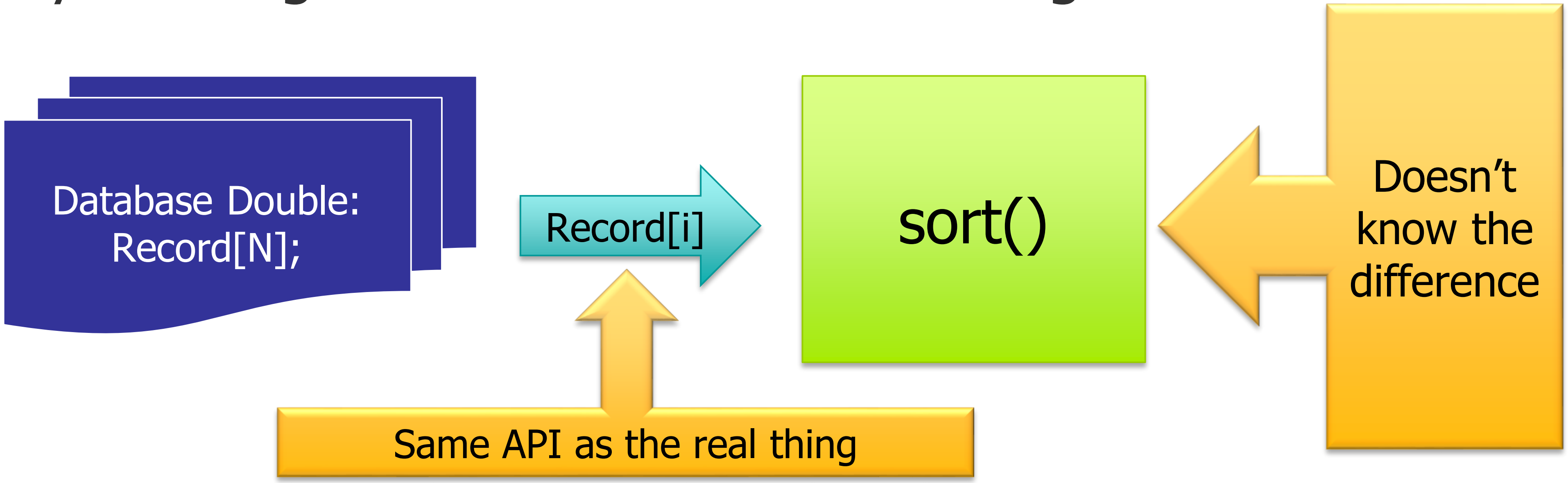
What Can Test Doubles Do?

- Test the code "from the inside"
- Allow testing before the real implementation of other units exists
- Simplify creating the desired state for testing



What Can Test Doubles Do?

- Test the code “from the inside”
- Allow testing before the real implementation of other units exists
- Simplify creating the desired state for testing



Types of Test Doubles

- Stubs: “dummy” implementations, make the code compile, API calls return predefined values
- Fakes: “real-ish” implementations, usually simplified
- Mocks: Test-only implementations that validate or track or otherwise analyze calls to internal APIs

Confusion alert!
These terms are not used consistently (e.g., “fake” is anything other than the real implementation, stubs return values, mocks also analyze calls)

Types of Test Doubles

- Stubs: “dummy” implementations, make the code compile, API calls return predefined values
- Fakes: “real-ish” implementations, usually simplified
  — The line between “stubs” and “fakes” is not always clear
- Mocks: Test-only implementations that validate or track or otherwise analyze calls to internal APIs
  — Mocks do more analysis on their arguments than is needed to produce results
- Some test doubles can do several things at once

```
int randstub() { ++count; return 1; }          // Stub? Mock? Yes
```

Types of Test Doubles

- Stubs: “dummy” implementations, make the code compile, API calls return predefined values

```
template <class R> result_t analyze_distribution(R rand_gen);
double rand_stub() {
    static double x = 0;
    return ++x;  // Uniform distribution
}
```

Types of Test Doubles

- Stubs: “dummy” implementations, make the code compile, API calls return configurable predefined values (fakes or stubs)?

```
class rand_stub {
    const double max_;
    public:
    rand_stub(double m) : max_(m) {}
    double operator() {
        static double x = 0;
        if (++x > max_) x = 0; return x;
    }
}
```

Types of Test Doubles

- Fakes: simplified implementations

```
class Database {                    // Class to test
    Backend backend_;               // Redundant, distributed, concurrent
    void store(const Record& r) { backend_.store(r); }
    …
};
class FakeBackend {                 // Good enough for testing
    std::set<Record> records_;
    void store(const Record& r) { records_.insert(r); }
    …
};
```

Types of Test Doubles

- Mocks: Test-only implementations that validate or track or otherwise analyze calls to internal APIs

```
void sort(Container& c, size_t to, size_t from) { … c[i] < c[j] … }
class MockSortContainer {
    std::vector<int> data_;
    std::vector<bool> seen_;              // Was every element inspected?
    int& operator[](size_t i) {
        if (i >= data_.size()) throw …;  // Range check
        seen_[i] = 1; return data_[i];
    }
};
```

How Do Test Doubles Work?

- How to get our code to use a double instead of the real thing?
- Sometimes it's easy:

```
void sort(Container& c, size_t to, size_t from);
                        class MockSortContainer { … };
```

- and sometimes not:

```
class Database {
  Backend backend_;
};
                  ?
                        class FakeBackend { … };
```

How to Use Test Doubles

- Problem: the code we're testing is written to use some types
  - May be even compiled with these types
- We have other types we want to use instead

1. Compiler magic
2. Dependency injection (C++ magic)

Test Doubles by [Compiler] Magic

- The tested code and all its dependencies must be recompiled
- During compilation, test doubles are substituted for some of the dependencies
- We have already seen this at work

Test Doubles by [Compiler] Magic

- The tested code and all its dependencies must be recompiled
- During compilation, test doubles are substituted for some of the dependencies
- We have already seen this at work: Debug STL
  - STL headers compiled with -D_GLIBCXX_DEBUG
  - Compiler uses different source code for STL classes and functions
  - Debug STL is a testing version of production STL
- Compiler flags, usually activating #define, used to enable testing functionality
- Test-only features are usually (effectively) mocks

Test Doubles by [Build] Magic

- The tested code and all its dependencies must be recompiled
  — Possibly only relinked
- During compilation and linking, test doubles are substituted for some of the dependencies
- Include paths and link paths are altered to use testing versions of dependencies (sources and libraries)
- Example: memory-checking malloc() enabled by linking alternative libc
  — usually not the whole libc, just a shim

DEPENDENCY INJECTION

Test Doubles by Dependency Injection

- Test doubles are substitutes for the code used by our tested code
  - Not the calling code but the called code, the dependencies
- Tested code is written to use certain dependencies
- The goal is to trick it to use something else that looks similar – test doubles
- The technique is known as dependency injection
  - Dependency injection (DI) is a form of inversion of control: normally the client controls the dependencies, with DI the external injector code does it outside of the client's control
  - It has applications outside of testing

Dependency Injection in C++

- Problem #1: the dependency in our code has a fixed type

```
class Database {
   Backend backend_;
};
```

- Options:

1. Keep the type as written, change what it means ← build magic
2. Make the type polymorphic
3. It's not a type, it's a template parameter

← Dependency Injection

Dependency Injection in C++

- Problem #1: the dependency in our code has a fixed type

```
class Database {
   Backend backend_;
};
```

- Options:

1. Keep the type as written, change what it means
2. Make the type polymorphic
3. It's not a type, it's a template parameter

Code must allow DI

- Dependency Injection doesn't just happen, code must be written to be testable (that's good)

Dependency Injection in C++

- Problem #2: a specific instance of the test double must be supplied
- This is not always necessary, especially for template injection
  - Tested code may construct a test double just like it does the real class and it may be sufficient (often true for mocks, e.g. range checking, tracking)
- If the tested code already accepts the dependency object (or function), then this is just a subset of problem #1
  - Example: sorting function, accepts mock container like any other container
- If the tested code constructs the dependency object (or gets it from another dependency), we have to inject an object
  - Always an issue for polymorphic test doubles

Dependency Injection in C++

1. Constructor injection: the test double is passed to the constructor
2. Interface injection: the test double is given through the interface
3. Set injection: subset of interface injection, uses setter methods

Constructor Injection

- Dependency objects are created outside of the main class
- Class does not have full knowledge of its own dependencies
  - Just the interfaces and other behavior contracts
- Can be used with polymorphic classes:

```
class Database {
    Database(BackendInterface& backend) : backend_(backend) { … }
    BackendInterface& backend_;
};
class TestMockBackend : public BackendInterface { … } tmb;
Database db(tmb);
```

Constructor Injection

- Dependency objects are created outside of the main class
- Class does not have full knowledge of its own dependencies
  - Just the interfaces and other behavior contracts
- Can be used with template types:

```
template <typename B> class Database {
    Database(B& backend) : backend_(backend) { … }
    B& backend_;
};
class TestMockBackend { … } tmb;
Database db(tmb);
```

Constructor Injection

- Dependency objects are created outside of the main class
- Class does not have full knowledge of its own dependencies
- Can be used with polymorphic classes or template types
- Testing needs force a particular design style
  — Implications go beyond testability
  — We may choose this design pattern for reasons other than testing
  — Even if we do it for testing, there are other advantages
- Injecting all dependencies for a large class adds many parameters
  — Group dependencies into larger components
  — Again design for testability – testing needs force design decision

Constructor Injection

- Dependency objects are created outside of the main class
- Class does not have full knowledge of its own dependencies
- Can be used with polymorphic classes or template types
- Testing needs force a particular design style
- Injecting all dependencies for a large class adds many parameters unless design anticipates this problem
- TDD → Design for testability → Better modularity
- Software has to be designed with testing in mind
  - Grafting testability onto a design as an afterthought is hard
- Code not organized for testing usually does not get tested

Interface Injection

- Dependency objects may be created outside of the main class

```
class Database {
    Database() {};
    void set_backend(Backend* backend) { backend_ = backend; }
    Backend* backend_;          // Non-owning pointer
                                // Or unique_ptr, or optional with move…
};
```

Interface Injection

- Default dependency objects may be created by the main class, substitution is possible

```
class Database {
    Database() : backend() {};
    void set_backend(Backend* backend) {
        backend_.swap(backend);
    }
    Backend* backend_;          // Non-owning pointer
                                // Or unique_ptr, or optional with move…
};
```

Interface Injection

- Dependency objects may be created outside of the main class or substituted later
- Injection methods are added to the interface as needed
  — If injection is used only for testing, this interface is test-only

Dependency Injection in C++

- Substitute types – templates or polymorphism
- Polymorphism: run-time overhead, methods made virtual just for testing, heap allocations (no std::optional etc)
  - Does not work if dependencies are created by the main class
- Templates: larger code, larger headers
  - Policy-based design is often used
- Substitute objects – constructors or interface
  - Not needed if the main class creates dependencies (templates/policies)
- In all cases, inversion of control affects the overall design
  - May have benefits other than testability

A TEST IS MORE THAN A TEST

When to Write Tests

1. Before any code is written
   - ➖ They won't compile
   - ➕ They describe how the client code should look like
   - ➕ They specify the requirements on the interface in C++ instead of English

Note to self:
Can tests be used instead of a spec?

Tests as Specification

- We write the tests before the code…
- Tests describe what the code should do…
- Can tests replace specification?

Tests as Specification

- Can tests replace specification?
- Tests do specify exactly what the code should do:

```
EXPECT_EQ(3, f(1));
```

- It's easy to check whether the code complies with the spec:

```
Expected equality of these values:
  3
  f(1)
    Which is: 2
```

- Tests are very unambiguous, can't misinterpret "3"

Tests as Specification

- Can tests replace specification?
- Tests do specify exactly what the code should do
- It’s easy to check whether the code complies with the spec
- Tests are very unambiguous, can’t misinterpret “3”
- Tests are never complete – require inference

```
EXPECT_EQ(3, f(1));
EXPECT_EQ(5, f(2));
EXPECT_EQ(7, f(3));
```

Tests as Specification

- Can tests replace specification?
- Tests do specify exactly what the code should do
- It's easy to check whether the code complies with the spec
- ~~Tests are very unambiguous, can't misinterpret "3"~~
- Tests are never complete – require inference

```
EXPECT_EQ(3, f(1));
EXPECT_EQ(5, f(2));
EXPECT_EQ(7, f(3));
EXPECT_EQ(9, f(4)); or EXPECT_EQ(11, f(4)); ?
```

- Inference is often ambiguous

Tests as Specification

- Can tests replace specification?
- This is actually two questions in one:

1. Can tests be used to fully specify the behavior?
2. Can tests be used as a specification for the API?

Can tests be used to fully specify the behavior?

- Not really – complete behavior description must be general
  - We can test for properties instead of values:
    EXPECT_TRUE(is_odd(f(1)));
  - This is still not a complete description, and the set of input values we can test for is finite
- Tests complement the general description very well
- Tests make a good specification of special cases (what's f(0)?)
- Tests illustrate the general case
  - But the need for inference should be removed by an explicit specification

Can tests be used as a specification for the API?

- If we write the tests before the code, the code won't compile if it does not match the expectations set in the test
- Tests give a [somewhat] general description of the API
  — Removing ambiguities requires a lot of attention to details

```
EXPECT_EQ(3, f(1));
```

- Nothing special about "1" (if this compiles, f(2) will compile too)

Can tests be used as a specification for the API?

- If we write the tests before the code, the code won't compile if it does not match the expectations set in the test
- Tests give a [somewhat] general description of the API
  - Removing ambiguities requires a lot of attention to details

```
EXPECT_EQ(3, f(1));
```

- Nothing special about "1" (if this compiles, f(2) will compile too)
- What is the parameter type? int? long? size_t?
- What is the return type? – the tests do not say
- Even more ambiguous for template functions and classes

Can tests be used as a specification for the API?

- If we write the tests before the code, the code won't compile if it does not match the expectations set in the test
- Tests give a [somewhat] general description of the API
  - Removing ambiguities requires a lot of attention to details
- Tests complement the general description, not replace it
- Tests illustrate the specification
- Tests make good specification for special cases
  - Some specific types must or must not compile
  - Which exceptions may be thrown

Tests as a Documentation

- Can tests be used to document what the program does?
  - Not exactly the same as tests as specification
  - Shares many advantages and disadvantages
- Tests don’t document why the code does something

```
double z = x + y;          // Add x and y
```

Tests as a Documentation

- Can tests be used to document what the program does?
  - Not exactly the same as tests as specification
  - Shares many advantages and disadvantages
- Tests don't document why the code does something

```
double z = x + y;          // x = 2; y = 3; EXPECT_EQ(5, z);
```

Tests as a Documentation

- Can tests be used to document what the program does?
  - Not exactly the same as tests as specification
  - Shares many advantages and disadvantages
- Tests don't document why the code does something

```
double z = x + y;          // Total time spent resting (x) and traveling (y)
```

Tests as a Documentation

- Can tests be used to document what the program does?
  - Not exactly the same as tests as specification
  - Shares many advantages and disadvantages
- Tests don’t document why the code does something
- Tests document behavior for specific inputs
  - General behavior needs inference
- Tests support documentation and make good examples
- Tests make good documentation for special cases

```
EXPECT_THROW(f(0), … );          // 0 is not valid
```

CONCURRENCY IS JUST ONE BUG BEFORE, AFTER, OR TOGETHER WITH ANOTHER

Testing Multi-Threaded and Concurrent Programs

- Test-driven development of multi-threaded programs:
  - Develop and test each single-threaded component
  - Develop and test synchronization and communication components
  - Implement and test concurrency (in small increments)
- This does not mean that the design should be single-threaded at first and concurrency added later!
- Concurrency should be designed into the program from the start
  - If implementing concurrency requires changes, again separate making changes to the old code from adding new code

Testing Multi-Threaded and Concurrent Programs

- Test-driven development of multi-threaded programs:
  - Develop and test each single-threaded component
  - Develop and test synchronization and communication components
  - Implement and test concurrency (<u>in small increments</u>)

This is hard!

- What is “a little bit of concurrency?”
  - One multi-threaded loop at a time? – could be
  - Start with just two threads? – sometimes
- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
  - Each component should be tested, including concurrency

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components

```
class Database {
    std::queue<Transaction> tqueue_;
    std::mutex queue_mutex_;
    enum { INSERT, ERASE, … };
    void add(const Record& r) {
        std::lock_guard g(mutex_);
        tqueue.push(Transaction(Record, INSERT);
   }
};
```

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components

```
class Database {
  std::queue<Transaction> tqueue_;
  std::mutex queue_mutex_;
  enum { INSERT, ERASE, … };
  void add(const Record& r) {
    std::lock_guard g(mutex_);
    tqueue.push(Transaction(Record, INSERT);
  }
};
```

a.k.a.
“big mess”

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components

```
class Database {
  TransactionQueue tqueue_;      // Thread-safe
  enum { INSERT, ERASE, … };
  void add(const Record& r) {
    tqueue.push(Transaction(Record, INSERT);
  }
};
```

- More complex than combining classes with mutexes
  — Look for data structures that provide atomic transactions

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
- Look for data structures that provide atomic transactions
- Concurrency has to be a design goal from the start

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
    void push(const Transaction& t) { … lock, push … }
    … forward all calls to queue_ …
};
```

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
- Look for data structures that provide atomic transactions
- Concurrency has to be a design goal from the start

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
    void pop();
    const Transaction& front() const;
    bool empty() const;
};
```

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
- Look for data structures that provide atomic transactions
- Concurrency has to be a design goal from the start

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
    void pop();
    const Transaction& front() const;
    bool empty() const;
};
```

Now what?

Testing Multi-Threaded and Concurrent Programs

- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
- Look for data structures that provide atomic transactions
- Concurrency has to be a design goal from the start

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
    std::optional<Transaction> pop(); // Return nothing if empty
};
```

- Concurrent data structures must present transactional API

Testing Multi-Threaded and Concurrent Programs

- Test-driven development of multi-threaded programs:
  - Develop and test each single-threaded component
  - Develop and test synchronization and communication components
  - Implement and test concurrency (in small increments)
- Concurrent programs should be built from concurrent (thread-safe, distributed, etc) components
  - Each component should be tested, including concurrency
- How do we test concurrent programs?

Testing Concurrent Programs

- Integration testing is frequently used for concurrent programs
  - Concurrent programs are made from single-threaded components
  - Each component is tested (unit testing)
  - Concurrency is interaction of code running on several threads
  - Testing how multiple components interact is integration testing
- “Live” testing of concurrent programs is probabilistic

Testing Concurrent Programs

- Integration testing is frequently used for concurrent programs
  - Concurrent programs are made from single-threaded components
  - Each component is tested (unit testing)
  - Concurrency is interaction of code running on several threads
  - Testing how multiple components interact is integration testing
- "Live" testing of concurrent programs ~~is probabilistic~~ relies on luck
  - Race conditions may or may not happen
  - Even if something bad happens, it may not be detectable
  - Results depend on scaling, number of CPUs, network speed, etc
- Flaky tests eventually get ignored or disabled
- Unit testing can be used to verify certain invariants, especially for testing thread-safe components

Testing Concurrent Programs

- Integration testing is frequently used for concurrent programs
  - Concurrent programs are made from single-threaded components
  - Each component is tested (unit testing)
  - Concurrency is interaction of code running on several threads
  - Testing how multiple components interact is integration testing
- Unit testing can be used to verify certain invariants, especially for testing thread-safe components

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
};
```

mutex_ must be held whenever queue_ is accessed

Unit Testing Concurrent Programs

- Unit testing can be used to verify certain invariants, especially for testing thread-safe components

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
};
```

mutex_ must be held whenever queue_ is accessed

- This invariant is valid for a single thread
  - The thread that accesses the queue must already hold the mutex
  - Other threads are blocked on the mutex so can’t access the queue
  - Unless a thread accesses the queue and does not hold the mutex!
- Testing for single-threaded invariants not race conditions

Unit Testing Concurrent Programs

- Unit testing can be used to verify certain invariants, especially for testing thread-safe components

```
class TransactionQueue {
    std::mutex mutex_;
    std::queue<Transaction> queue_;
};
```

mutex_ must be held whenever queue_ is accessed

- Testing invariants requires test doubles (mocks) for all involved dependencies (including mutexes)
- Heavy use of dependency injection
  - Unless the program is designed for testability in this manner, making all necessary dependencies injectable is probably not practical

DEPENDENCY INJECTION VS MAGIC

Practical Dependency Injection

1. Design the system with inverted control from the start
2. Take advantage of the interfaces that exist for other reasons
3. Add few interfaces to inject some dependencies (easy ones)
4. Rewrite classes to be fully injectable when it benefits testing

Practical Dependency Injection

1. Design the system with inverted control from the start
2. Take advantage of the interfaces that exist for other reasons
3. Add few interfaces to inject some dependencies (easy ones)
4. ~~Rewrite classes to be fully injectable when it benefits testing~~

Yeah, right

Practical Dependency Injection (ish)

1. Design the system with inverted control from the start
2. Take advantage of the interfaces that exist for other reasons
3. Add few interfaces to inject some dependencies (easy ones)
4. ~~Rewrite classes to be fully injectable when it benefits testing~~
5. Use compiler/build magic to inject debug code into everything everywhere all at once
   - Debug builds, asserts, etc
6. Sanitizers!

What are Sanitizers?

- Sanitizers are tools that instrument the code and/or the runtime environment and detect certain errors
  - Built into compilers
  - Replace system libraries or intercept system calls
  - External tools that run and observe programs
- ASAN (Address Sanitizer) – detects memory access errors
- TSAN (Thread Sanitizer) – detects race conditions
- UBSAN (Undefined Behavior Sanitizer) – detects ill-defined C++

Testing and Sanitizers – Powerful Combination

- Sanitizers instrument the entire code to detect certain errors
  - A kind of blanket dependency injection: replace all locks with monitored locks or all arrays with guarded arrays
  - Only some predefined "sanitized doubles" are available
  - All sanitized code is converted to use them, without rewriting anything
- Sanitizers are more reliable than failure detection
  - Sanitizers can detect potential failures, e.g. a concurrent read before write without synchronization that may happen (not only when it happens)
- Tests of units and components make failures easier to analyze
- Tests exercise code paths and states that are difficult to recreate otherwise

Mandatory C++ subject

EFFICIENCY

Testing For Performance

- Testing for performance is not exactly the same as benchmarking
  — The difference is mostly the intent, often the same tools are used
- Performance can be one of specified targets, needs verification
- Performance of individual units can be tested
  — Microbenchmark frameworks are the equivalent of unit testing frameworks
- Performance of large systems can be tested
  — End-to-end testing on real data is often used
- Continuous performance monitoring may be used to collect performance data from production runs
- Testing results are necessarily noisy and may change in time
  — Testing criteria must be thought out carefully

LESSONS LEARNED

Reasons For Not Testing (or Testing Don't's)

- Flaky tests or flaky testing environment
- Tests take too long to run
- The need to keep updating the tests
- Tests are hard to maintain because the wrong kind of test was used (e.g. using unit tests for system integration)
- The code organization makes it difficult to test
- The values that needs to be tested are impossible to access
- Code was not designed with testing in mind

Good Practices (or Testing Do's)

- Design your code to be testable
  - Starting development from scratch is rare
  - New components can be designed testable even in old code base
- Learn essential elements of the test-driven development process
  - E.g. TDD can be used without unit tests
- Recognize problems that lower testing discipline and fix them early
  - Fix or disable flaky tests, separate slow tests, redesign fragile tests
- Take full advantage of the testing practices you do follow
  - Got regression tests? Make a test suit for refactoring
- Be practical: perfect is the enemy of good
  - Incremental benefits of better testing practices are better than trying and failing to follow the "pure" process

Mentor®

A Siemens Business

www.mentor.com