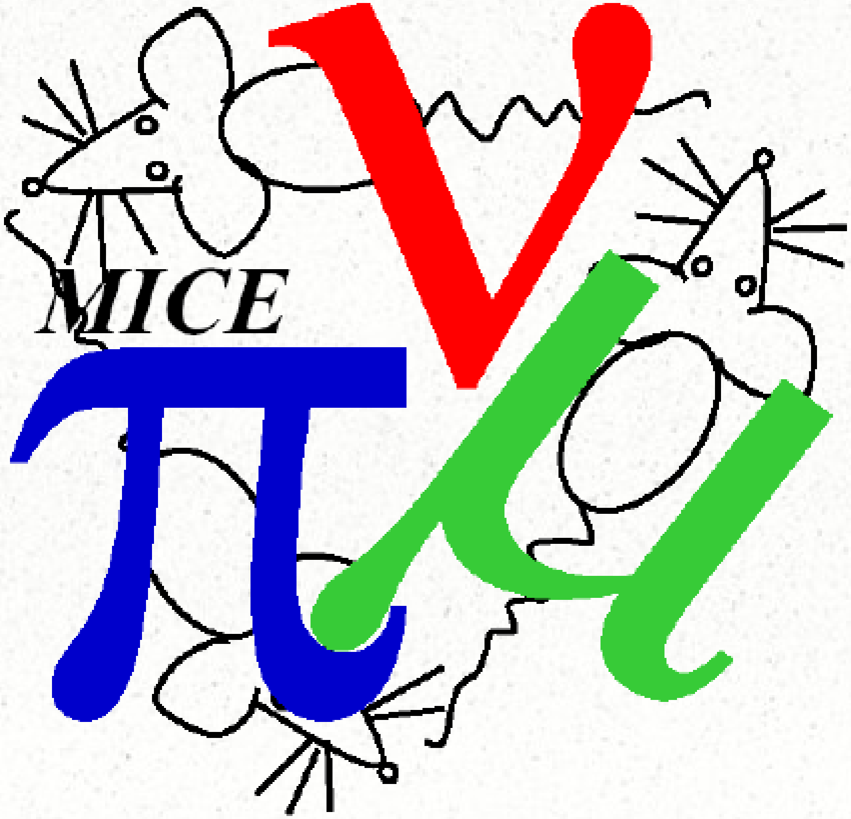
Continuous Integration



Continuous Integration (CI)

- CI is widely accepted as the best way to create large-scale software in a collaborative environment
- In its most basic form you use a service which monitors the version control system, then builds your code and runs specified tests
- The CI server watches for changes in the source code repository and builds whenever something changes
- The CI server continuously reports on the build status

Continuous Integration (CI)

- Owning an exercise bicycle doesn’t make you fit (I wish)
- Likewise having a CI server doesn’t mean we’re doing CI – CI is a discipline!
- To do CI we have to check-in our code almost as often as we hit the save button
- You will probably check-in your incremental changes daily but more frequently is good
- Hence integration goes from being “a big deal” to “a non-event”

Continuous Integration (CI)

- A failed build is one that doesn’t compile, lacks a dependency, has a failed unit test
- The CI server can be configured to fail a build when there are compiler warnings and it is proposed to do this.
- We can also configure the CI server to run additional scripts such as test coverage scripts, style-guide scripts, etc.

CI and unit tests

- The CI server will run your unit tests and display the messages accordingly
- A failed unit test means it’s found something that would have bitten either you or somebody else at some time in the future
- Which is a **good thing**. ☺
- So we should not be embarrassed about tests which fail: it’s good to see you use them and it’s good they’re finding stuff out for you!

Essential rules of CI

- When the build has failed (compiler error, dependency error, unit test failure) it’s you who must fix it or you must revert
- No-one should check-in on top of a broken build – compounds the problem
- You should run all commit tests locally
- Make sure your commit tests all pass before starting new work
- Revert if necessary to keep the main-line clean

Bazaar

- Starting with MAUS, we are using a distributed version control system (DVCS) called Bazaar
- Bazaar is written in Python
- With a DVCS you don't need to be on-line to commit (work on the train, in a cave)
- Theoretically, no need for a backup (ideally everyone has the master copy)
- You can also run a local copy of the currently preferred CI server, “Jenkins”, and integrate Bazaar

Jenkins (was Hudson)

- As of MAUS, we are using Jenkins as the CI server. Chris Tunnell has already set this up here: http://christesting.streiff.net/
- Jenkins monitors anything checked in using Bazaar
- Jenkins is easy to install: java -jar jenkins.war
- In the “configure” link, you can choose the Bazaar, TestLink (automates tests, e.g gtests) and many other plugins
- Thus it’s relatively easy to create you own local environment to run local commit tests
- We are still trying Jenkins out but it looks good so far

Google Testing Framework (gtest)

- gtest is based on xUnit, which is a port from JUnit
- TEST(testCaseName, individualTestName)
- Can group a set of tests by testCase
- Can reuse tests using fixtures
- For more information see: http://code.google.com/p/googletest/wiki/Documentation
- See also Chris Rogers' examples on G4MICE

Google C++ Testing Framework (aka Google Test)

- What it is
  - A library for writing C++ tests
  - Open-source with new BSD license
  - Based on xUnit architecture
  - Supports Linux, Windows, Mac OS, and other OSes
  - Can generate JUnit-style XML, parsable by Hudson [Jenkins]

Simple tests

- Simple things are easy:
- TEST() remembers the tests defined, so you don’t have to enumerate them later.
- A rich set of assertion macros

```
// TEST(TestCaseName, TestName)
TEST(NumberParserTest, CanParseBinaryNumber) {
  // read: a NumberParser can parse a binary number.

  NumberParser p(2); // radix = 2

  // Verifies the result of the function to be tested.
  EXPECT_EQ(0, p.Parse("0"));
  EXPECT_EQ(5, p.Parse("101"));
}
```

Reusing the same data configuration

- Define the set-up and tear-down logic in a test fixture class – you don’t need to repeat it in every test.

```
class FooTest : public ::testing::Test {
  protected:
   virtual void SetUp() { a = ...; b = ...; }
   virtual void TearDown() { ... }
   ...
};
TEST_F(FooTest, Bar) { EXPECT_TRUE(a.Contains(b)); }
TEST_F(FooTest, Baz) { EXPECT_EQ(a.Baz(), b.Baz()); }
```

- Google Test creates a fresh object for each test – tests won’t affect each other!

What to test for: good and bad input

- Good input leads to expected output:
  - Ordinary cases
    - EXPECT_TRUE(IsSubStringOf("oo", "Google"))
  - Edge cases
    - EXPECT_TRUE(IsSubStringOf("", ""))
- Bad input leads to:
  - Expected error code – easy
  - Process crash
    - Yes, you should test this!
      - Continuing in erroneous state is *bad*.
    - But how?

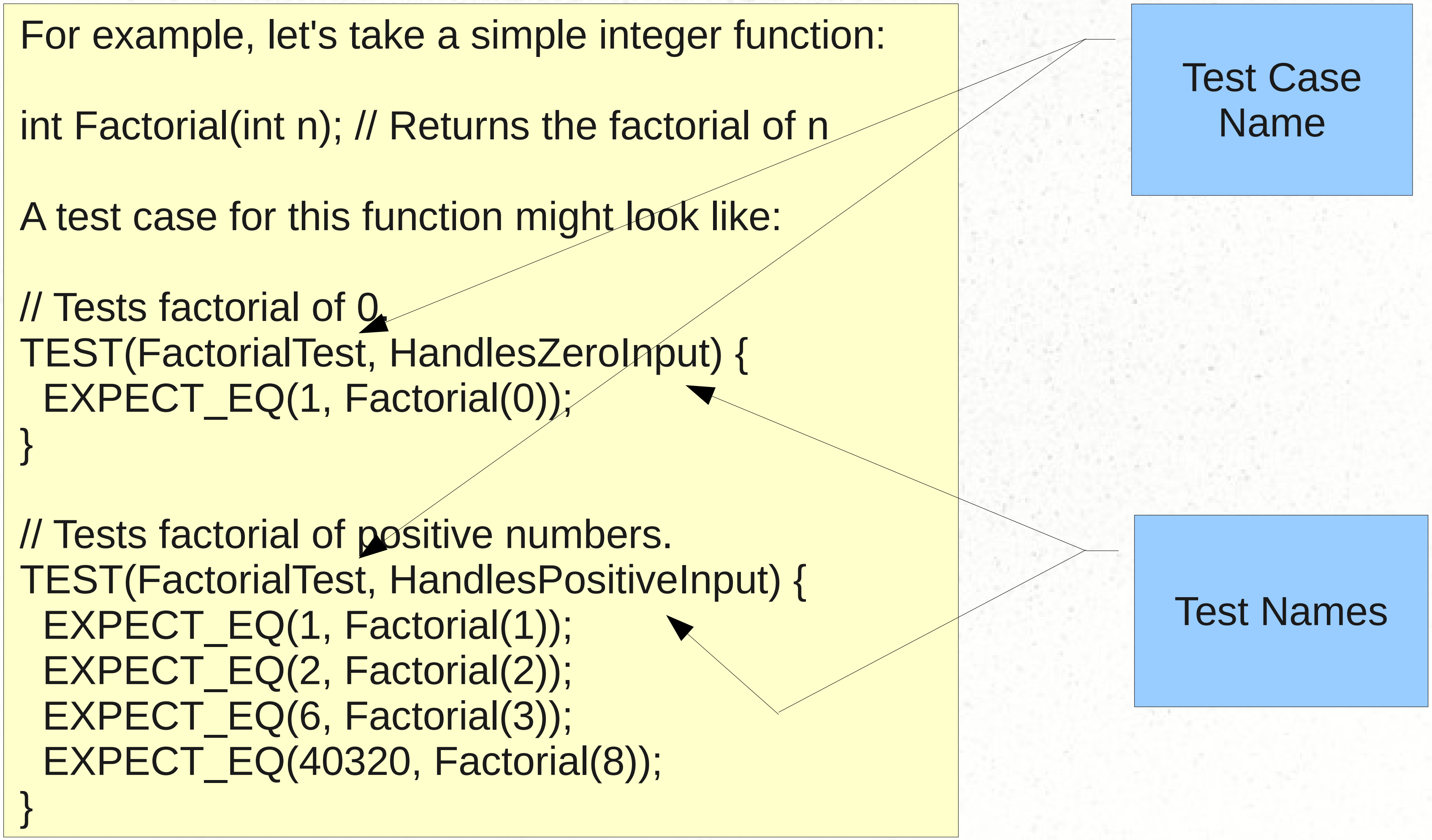
Death Tests

```
TEST(FooDeathTest, SendMessageDiesOnInvalidPort) {
  Foo a;
  a.Init();
  EXPECT_DEATH(a.SendMessage(56, "test"),
               "Invalid port number");
}
```

- How it works
  - The statement runs in a *forked* sub-process.
  - Very fast on Linux
  - Caveat: side effects are in the sub-process too!

gtest example

from http://code.google.com/p/googletest/wiki/Primer



For example, let's take a simple integer function:

```
int Factorial(int n); // Returns the factorial of n
```

A test case for this function might look like:

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Test Case Name

Test Names

gtest example

from http://code.google.com/p/googletest/wiki/Primer

You can easily provide informative messages:

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {
  EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

What not to test

- It’s easy to get over-zealous.
- Do not test:
  - A test itself
  - Things that cannot possibly break (or that you can do nothing about)
    - System calls
    - Hardware failures
  - Things your code depends on
    - Standard libraries, modules written by others, compilers
    - They should be tested, but not when testing your module – keep tests focused.
  - Exhaustively
    - Are we getting diminishing returns?
    - Tests should be fast to write and run, obviously correct, and easy to maintain.

What makes good tests?

- Good tests should:
  - Be independent
    - Don’t need to read other tests to know what a test does.
    - When a test fails, you can quickly find out the cause.
    - Focus on different aspects: one bug • one failure.
  - Be repeatable
  - Run fast
    - Use mocks.
  - Localize bugs
    - Small tests
- Next, suggestions on writing better tests

Favor small test functions

- Don’t test too much in a single TEST.
    - Easy to localize failure
        - In a large TEST, you need to worry about parts affecting each other.
    - Focus on one small aspect
    - Obviously correct

Make the messages informative

- Ideally, the test log alone is enough to reveal the cause.
- Bad: “foo.OpenFile(path) failed.”
- Good: “Failed to open file /tmp/abc/xyz.txt.”
- Append more information to assertions using <<.
- *Predicate assertions* can help, too:
  - Instead of: EXPECT_TRUE(IsSubStringOf(needle, hay_stack))
  - Write: EXPECT_PRED2(IsSubStringOf, needle, hay_stack)

EXPECT vs ASSERT

- Two sets of assertions with same interface
  - EXPECT (continue-after-failure) vs ASSERT (fail-fast)
- Prefer EXPECT:
  - Reveals more failures.
  - Allows more to be fixed in a single edit-compile-run cycle.
- Use ASSERT when it doesn’t make sense to continue (seg fault, trash results). Example:

```
TEST(DataFileTest, HasRightContent) {
  ASSERT_TRUE(fp = fopen(path, "r"))
     << "Failed to open the data file.";

  ASSERT_EQ(10, fread(buffer, 1, 10, fp))
     << "The data file is smaller than expected.";

  EXPECT_STREQ("123456789", buffer)
    << "The data file is corrupted.";
  ...
}
```

Getting back on track

- Your project suffers from the low-test-coverage syndrome. What should you do?
    - Every change must be accompanied with tests that verify the change.
        - Not just any tests – must cover the change
        - No test, no check-in.
        - Test only the delta.
        - Resist the temptation for exceptions.
    - Over time, bring more code under test.
        - When adding to module Foo, might as well add tests for other parts of Foo.
    - Refactor the code along the way.
- It will not happen over night, but you can do it.

Google Tests

- Key points to take home:
  - Keep tests small and obvious.
  - Test a module in isolation.
  - Break dependencies in production code.
  - Test everything that can possibly break, but no more
  - No test, no check-in

Mocking framework

- When an object is unavailable for testing (e.g. it's not yours, not written yet) a mock-up can be used
- Also useful if the real object is slowing down your tests by (say) carrying out operations you don't need to test (e.g. a library)
- It is, of course, possible to roll your own (these are usually called "fakes" depending on how you do it)
- A mocking framework allows *expected behaviour* to be defined, and a run-time choice of functions
- Many unit test frameworks provide mocking; Google is no exception: http://code.google.com/p/googlemock/

Python – PyUnit

- Python has it’s own unit test framework
- Very similar to gtest because, like gtest, it’s derived from JUnit
- Usage: **`import unittest`**
- Like gtest, it supports test fixtures, test cases, etc.
- You should write PyUnit tests just as you write gtests

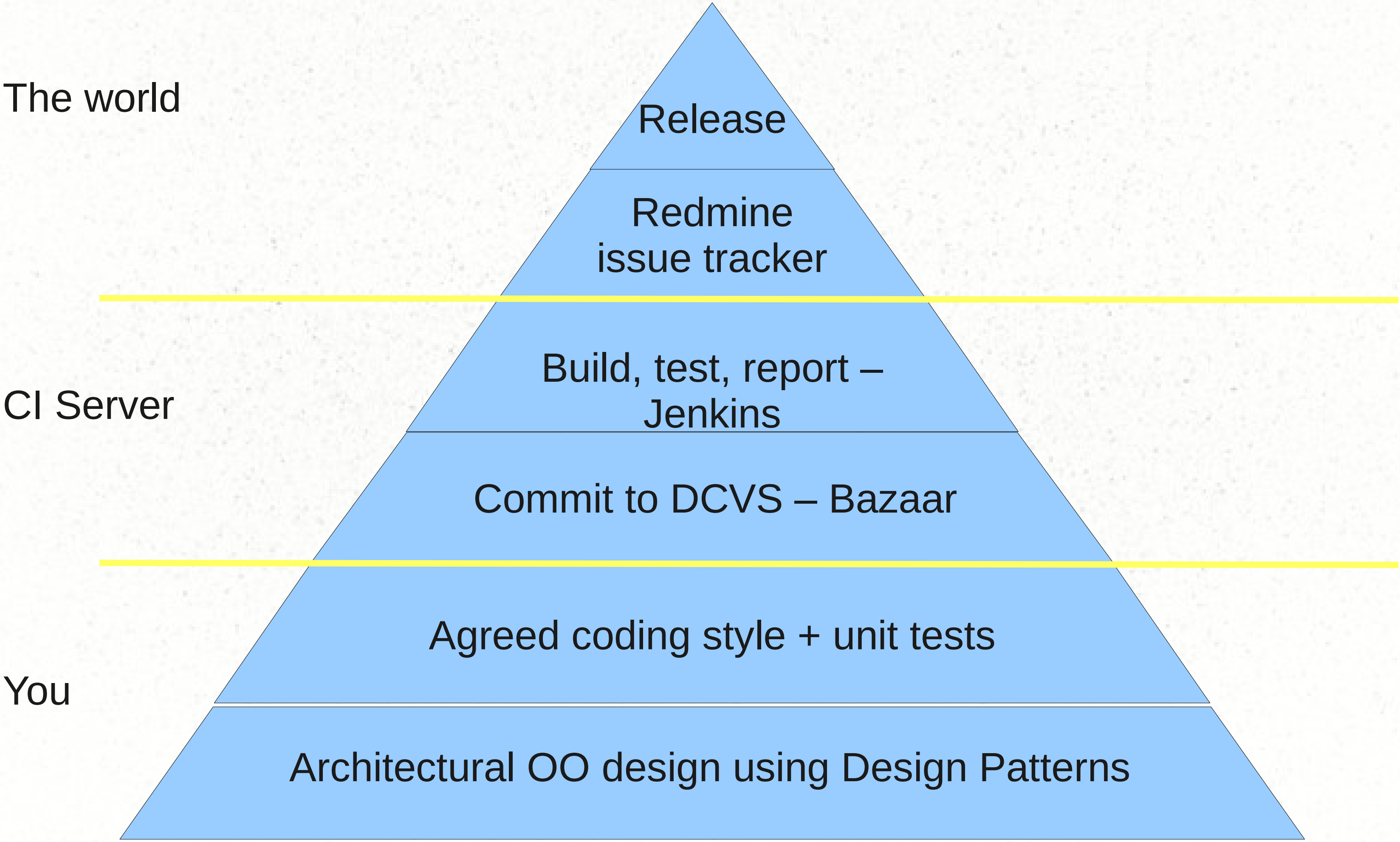
Google's C++ style guide?

- Google is the most comprehensive and detailed
- Is readable (pros and cons to justify decisions)
- Is practical (allows reasonable variations)
- Exceptions: Google state they would probably have recommended native exception handling if they were starting from scratch!
- Exceptions are obviously a minefield: Test carefully to understand exception handling
- We could use cpplint.py, cppclean.py or similar to automatically check style

Documentation

- Use Google Style Guide on comments
- Remember, others may want to understand, reuse or just read your code
- Use dOxygen to produce Javadoc-like HTML pages that describe your classes
- Check for broken links!

Continuous Integration Stack



The world

Release

Redmine
issue tracker

CI Server

Build, test, report –
Jenkins

Commit to DCVS – Bazaar

You

Agreed coding style + unit tests

Architectural OO design using Design Patterns