Google

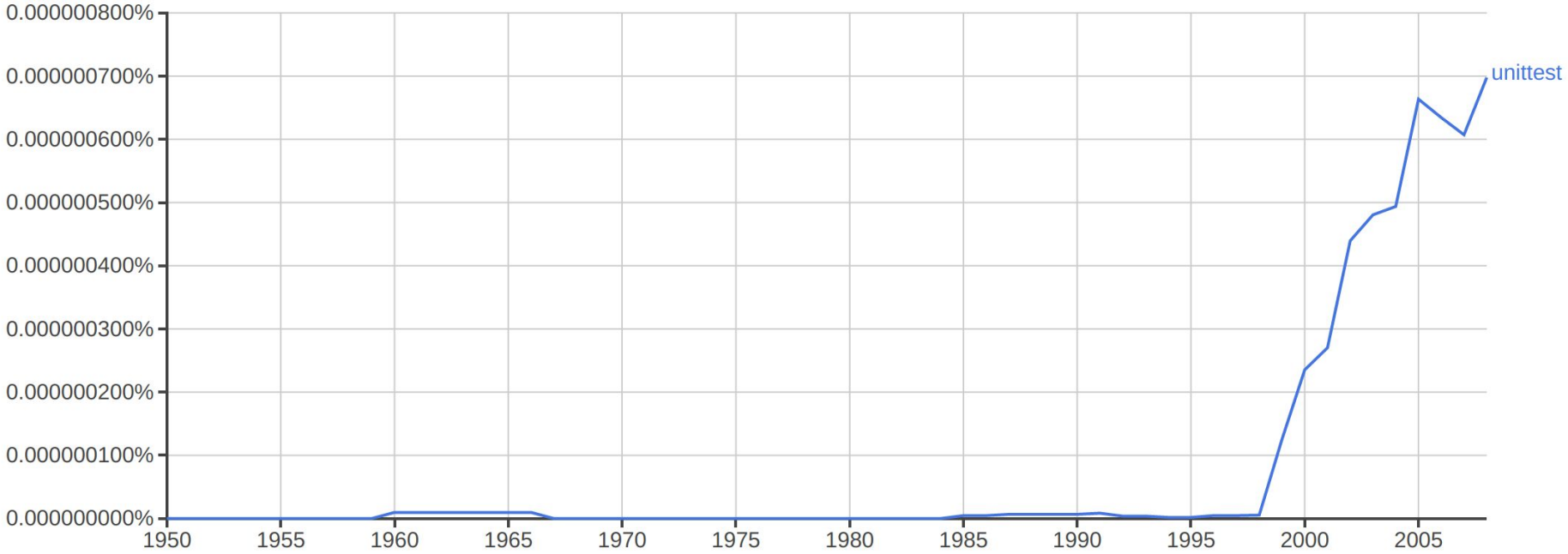
Testing in Software Engineering

~2010-2030

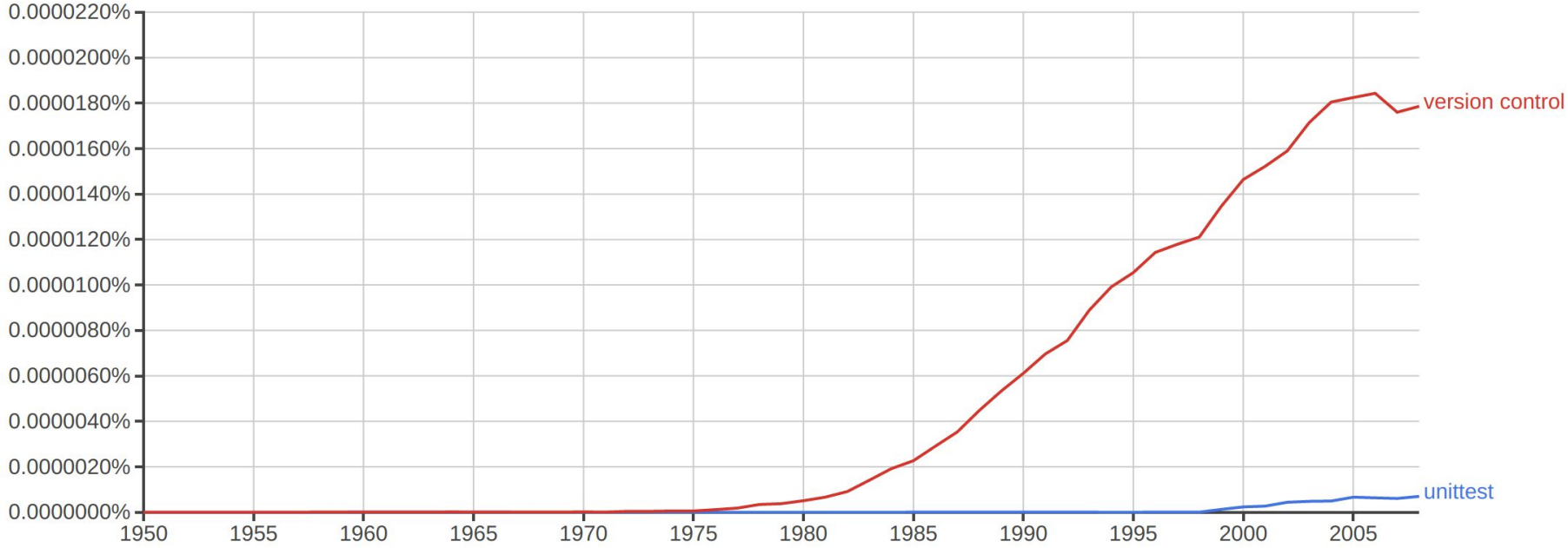
Titus Winters (titus@google.com)

Testing - History

Testing - History



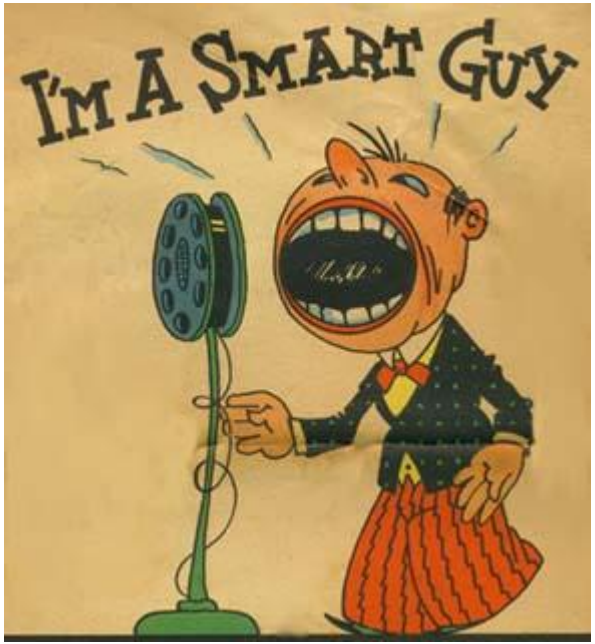
Testing - History



Testing - Present

Resistance to testing: most common flavors

1. I’m a good and smart programmer.



Testing - Present

Resistance to testing: most common flavors

1. I’m a good programmer.
2. Tests slow down development.

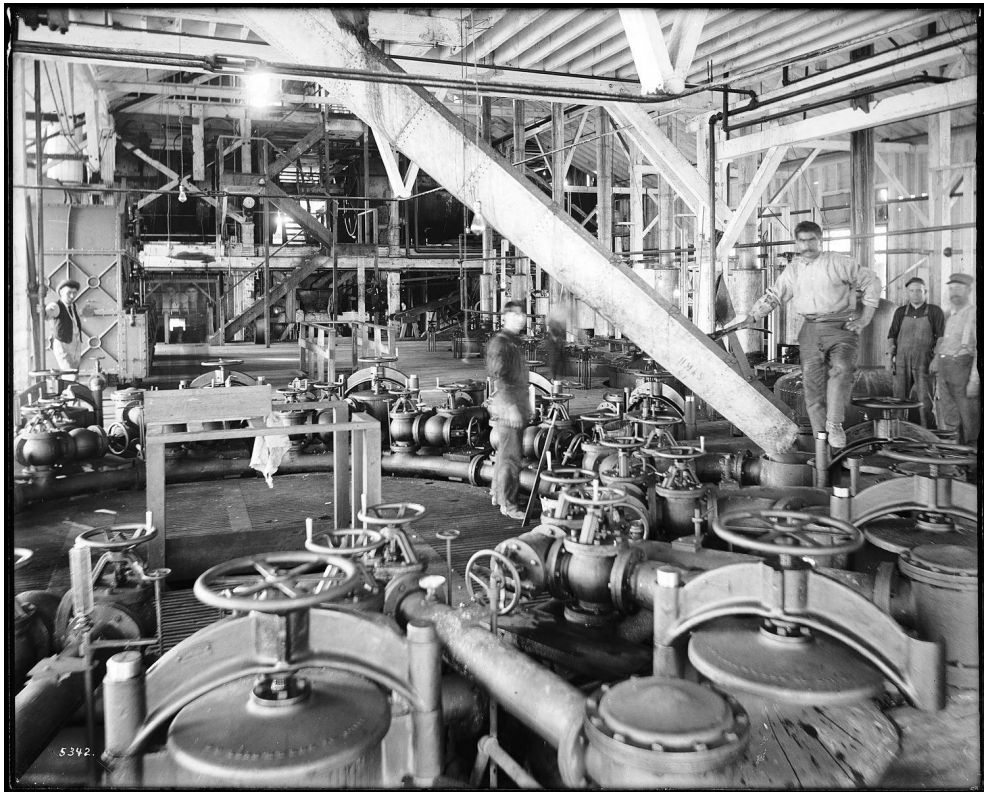
Context matters!

Testing - Styles

Unittests vs. Integration tests

Testing - Unittests

Testing - Integration



Testing - Properties of Good Tests

- Correct
- Readable
- Complete
- Explanatory
- Resilient

Correct

Tests must verify the requirements of the system are met.

Please don’t write:

- Tests that depend upon known bugs

Correct

Tests shouldn’t depend upon known bugs.

```
// NOTE: Unimplemented
int square(int x) {
    return 0;
}

TEST(SquareTest, MathTests) {
  EXPECT_EQ(0, square(2));
  EXPECT_EQ(0, square(3));
  EXPECT_EQ(0, square(7));
}
```

Correct

Tests must verify the requirements of the system are met.

Please don’t write:

- Tests that depend upon known bugs
- Tests that don’t actually execute real scenarios

Correct

Bad: tests that are not executing real scenarios

```
class FakeWorld : public World {
  // For simplicity, we assume the world is flat
  bool IsFlat() override { return true; }
};

TEST(Flat, WorldTests) {
  FakeWorld world;
  EXPECT_TRUE(world.Populate());
  EXPECT_TRUE(world.IsFlat());
}
```

Correct

Bad: tests that are not executing real scenarios

```
class StubWorld : public World {
  MOCK_METHOD1(IsFlat, bool());
};

TEST(Flat, WorldTests) {
  StubWorld world;
  ON_CALL(world, IsFlat()).WillByDefault(Return(true));

  EXPECT_TRUE(world.Populate());
  EXPECT_TRUE(world.IsFlat());
}
```

Correct

Bad: tests that are not executing real scenarios

```
class MockWorld : public World {
  MOCK_METHOD1(IsFlat, bool());
};

TEST(Flat, WorldTests) {
  MockWorld world;
  EXPECT_CALL(world, IsFlat()).WillOnce(Return(true));

  EXPECT_TRUE(world.Populate());
  EXPECT_TRUE(world.IsFlat());
}
```

Readable

Tests should be obvious to the future reader (including yourself!)

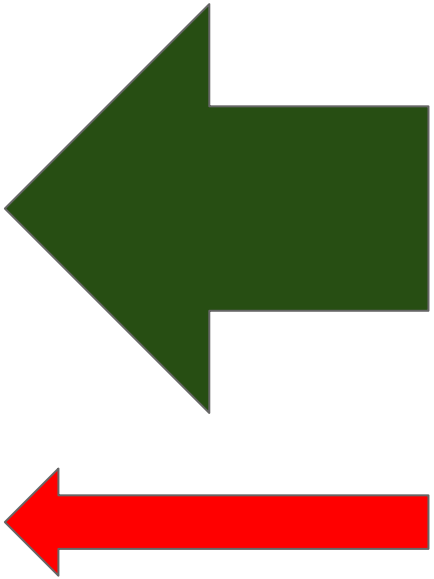
Don’t write tests that have:

- Too much boilerplate and other distraction

Readable

Avoid boilerplate and distraction in tests

```
TEST(BigSystemTest, CallIsUnimplemented) {
  TestStorageSystem storage;
  auto test_data = GetTestFileMap();
  storage.MapFilesystem(test_data);
  BigSystem system;
  ASSERT_OK(system.Initialize(5));
  ThreadPool pool(10);
  pool.StartThreads();

  storage.SetThreads(pool);
  system.SetStorage(storage);

  ASSERT_TRUE(system.IsRunning());

  EXPECT_TRUE(IsUnimplemented(system.Status()));
}
```



Readable

Tests should be obvious to the future reader (including yourself!)

Don’t write tests that have:

- Too much boilerplate and other distraction
- Not enough context in the test

Readable

Keep enough context for the reader

```
TEST(BigSystemTest, ReadMagicBytes) {
  BigSystem system = InitializeTestSystemAndTestData();
  EXPECT_EQ(42, system.PrivateKey());
}
```

Readable

Tests should be obvious to the future reader (including yourself!)

Don’t write tests that have:

- Too much boilerplate and other distraction
- Not enough context in the test
- Gratuitous use of advanced test framework features

Readable

Don’t use advanced test framework features when it isn’t necessary.

```
class BigSystemTest : public ::testing::Test {
 public:
  BigSystemTest() : filename_("/foo/bar/baz") { }

  void SetUp() {
    ASSERT_OK(file::WriteData(filename_, "Hello World!\n"));
  }

 protected:
  BigSystem system_;
  string filename_;
};

TEST_F(BigSystemTest, BasicTest) {
  EXPECT_TRUE(system_.Initialize());
}
```

Readable

Don’t use advanced test framework features when it isn’t necessary.

```
class BigSystemTest : public ::testing::Test {
 public:
  BigSystemTest() : filename_("/foo/bar/baz") { }

  void SetUp() {
    ASSERT_OK(file::WriteData(filename_, "Hello World!\n"));
  }

 protected:
  BigSystem system_;
  string filename_;
};

TEST_F(BigSystemTest, BasicTest) {
  EXPECT_TRUE(system_.Initialize());
}
```

Readable

Don’t use advanced test framework features when it isn’t necessary.

```
class BigSystemTest : public ::testing::Test {
 public:
  BigSystemTest() : filename_("/foo/bar/baz") { }

  void SetUp() {
    ASSERT_OK(file::WriteData(filename_, "Hello World!\n"));
  }

 protected:
  BigSystem system_;
  string filename_;
};

TEST_F(BigSystemTest, BasicTest) {
  EXPECT_TRUE(system_.Initialize());
}
```

Readable

Don’t use advanced test framework features when it isn’t necessary.

```
TEST(BigSystemTest, BasicTest) {
  BigSystem system;
  EXPECT_TRUE(system.Initialize());
}
```

Readable

Tests should be obvious to the future reader (including yourself!)

Don’t write tests that have:

- Too much boilerplate and other distraction
- Not enough context in the test
- Gratuitous use of advanced test framework features

A test should be like a novel: setup, action, conclusion, and it should all make sense.

Complete

Don’t write tests only for the easy cases.

```
TEST(FactorialTest, BasicTests) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(120, Factorial(5));
}
```

Complete

Don’t write tests only for the easy cases.

```
TEST(FactorialTest, BasicTests) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(120, Factorial(5));
}

int Factorial(int n) {
  if (n == 1) return 1;
  if (n == 5) return 120;
  return -1;  // TODO: figure this out.
}
```

Complete

Don’t write tests only for the easy cases.

Do write tests for common inputs, corner cases, outlandish cases

```
TEST(FactorialTest, BasicTests) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(120, Factorial(5));
  EXPECT_EQ(1, Factorial(0));
  EXPECT_EQ(479001600, Factorial(12));
  EXPECT_EQ(std::numeric_limits::max<int>() , Factorial(13));
  EXPECT_EQ(1, Factorial(0));
  EXPECT_EQ(std::numeric_limits::max<int>() , Factorial(-10));
}
```

Demonstrative

Tests should serve as a demonstration of how the API works.

Don’t write tests with

- Reliance on private methods + friend / TestOnly methods.
- Bad usage in unit tests, suggesting a bad API

Demonstrative

```
class Foo {
  friend FooTest;
 public:
  bool Setup();

 private:
  bool ShortcutSetupForTesting();
};

TEST(FooTest, Setup) {
  EXPECT_TRUE(ShortcutSetupForTesting());
}
```

Demonstrative

```
class Foo {
  friend FooTest;
 public:
  bool Setup();

 private:
  bool ShortcutSetupForTesting();
};

TEST(FooTest, Setup) {
  EXPECT_TRUE(Setup());
}
```

Demonstrative

```
class Foo {
  friend FooTest;
 public:
  bool Setup();

 private:
  bool ShortcutSetupForTesting();
};

TEST(FooTest, Setup) {
  EXPECT_TRUE(Setup());
}
```

Resilient

Engineers **love** to write tests that fail in all sorts of surprising ways.

- Flaky tests
- Brittle tests
- Tests that depend on execution ordering
- Mocks with deep dependence upon underlying APIs
- Non-hermetic tests

Resilient

Avoid flaky tests: Tests that can be re-run with the same build in the same state and flip from passing to failing (or timing out).

```
TEST(UpdaterTest, RunsFast) {
  Updater updater;
  updater.UpdateAsync();
  SleepFor(Seconds(.5));  // Half a second should be *plenty*.
  EXPECT_TRUE(updater.Updated());
}
```

Resilient

Avoid brittle tests: Tests that can fail for changes unrelated to the code under test.

```
TEST(Tags, ContentsAreCorrect) {
  TagSet tags = {5, 8, 10};

  // TODO: Figure out why these are ordered funny.
  EXPECT_THAT(tags, ElementsAre(8, 5, 10));
}
```

Resilient

Avoid brittle tests: Tests that can fail for changes unrelated to the code under test.

```
TEST(Tags, ContentsAreCorrect) {
  TagSet tags = {5, 8, 10};

  // TODO: Give a talk about hash iteration ordering.
  EXPECT_THAT(tags, UnorderedElementsAre(5, 8, 10));
}
```

Resilient

Avoid brittle tests: Tests that can fail for changes unrelated to the code under test.

```
TEST(MyTest, LogWasCalled) {
  StartLogCapture();
  EXPECT_TRUE(Frobber::Start());
  EXPECT_THAT(Logs(), Contains("file.cc:421: Opened file frobber.config" ));
}
```

Resilient

Resilient

Resilient - Ordering

Avoid tests that fail if they aren't run all together or in a particular order.

```
static int i = 0;

TEST(Foo, First) {
  ASSERT_EQ(0, i);
  ++i;
}

TEST(Foo, Second) {
  ASSERT_EQ(1, i);
  ++i;
}
```

Resilient - Hermetic

Avoid writes tests that fail if anyone else in the company runs the same test at the same time.

```
TEST(Foo, StorageTest) {
  StorageServer* server = GetStorageServerHandle();
  auto my_val = rand();
  server->Store("testkey", my_val);
  EXPECT_EQ(my_val, server->Load("testkey"));
}
```

Resilient - Deep Dependence

Avoid mock tests that fail if anyone refactors those classes.

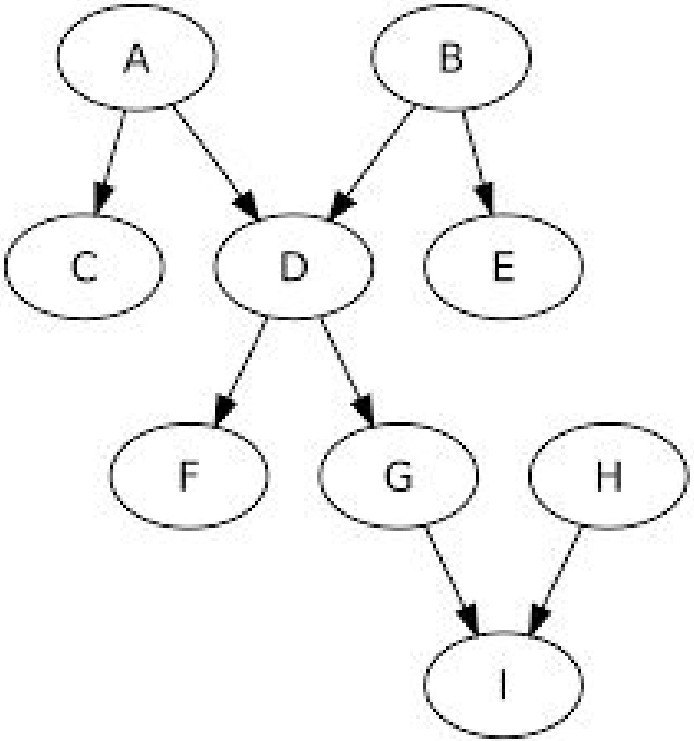
```
class File {
  public:
    ...
    virtual bool Stat(Stat* stat);
    virtual bool StatWithOptions(Stat* stat, StatOptions options) {
        return Stat(stat);  // Ignore options by default
    }
};

TEST(MyTest, FSUsage) {
  ...
  EXPECT_CALL(file, Stat(_)).Times(1);
  Frobber::Start();
}
```

Recap: What's the Goal?

0. Write tests.
1. Write tests that test what you wanted to test.
2. Write readable tests: correct by inspection.
3. Write complete tests: test all the edge cases.
4. Write demonstrative tests: show how to use the API.
5. Write resilient tests: hermetic, only breaks when there is an unacceptable behavior change.

What’s Next?

Inter-repo dependencies



Semantic Indexing

Perfect ability to answer any question about how your code is being used.

- Where is this API called?
- Where is this variable referenced?
- What types are used when instantiating this template?

Compiler assisted code transformation

Compiler assisted code transformation

For instance, we can look at changing:

```
LOG(INFO) << StringPrintf("Size on disk is %lld", size);
```

Into:

```
LOG(INFO) << StrCat("Size on disk is ", size);
```

Compiler assisted code transformation

For instance, we can look at changing:

```
LOG(INFO) << StringPrintf("Size on disk is %lld", size);
```

Into:

```
LOG(INFO) << "Size on disk is " << size;
```

What's Next?

- Unittests
  - Consistent interface for execution
  - Know when assumptions are broken
- Inter-repo dependencies
  - Build from source
  - Hosted repos not tarballs
- Semantic indexing
  - Maintainers know exactly how things are being used
- Compiler-assisted code transformation
  - Low/zero false positives

What's Next?

A new life cycle for enterprise code.

What’s Next

Sustainability