Google

All Your Tests are Terrible

Tales from the Trenches

Goofus and Gallant

By Garry Cleveland Myers

Pictures by Marion Hull Hammel

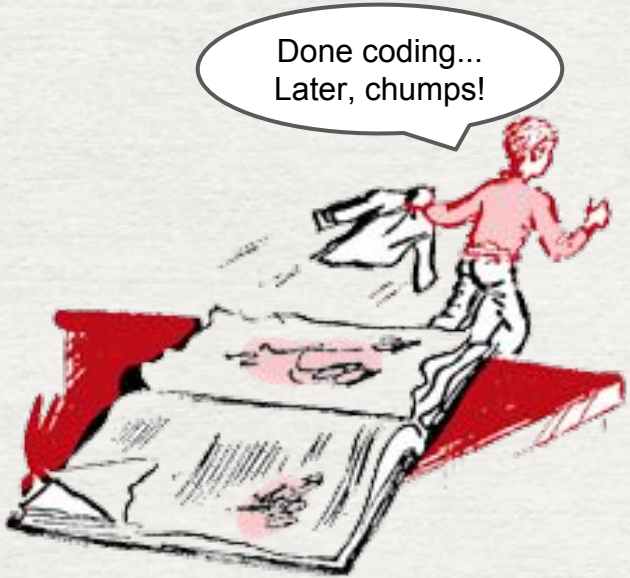
When Goofus falls and skins his knees, he gets up angry.

When Gallant falls and skins his knees, he gets up smiling.

5 Properties of Good Tests

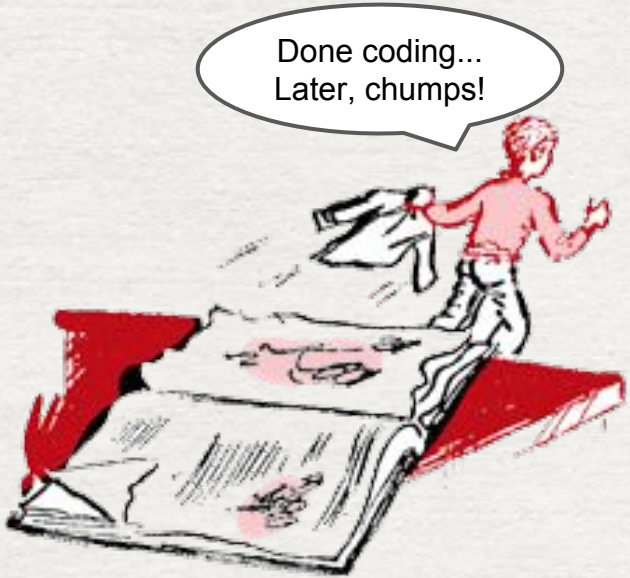
- Correctness
- Readability
- Completeness
- Demonstrability
- Resilience

Goofus and Gallant



Goofus doesn't write tests.

Goofus and Gallant

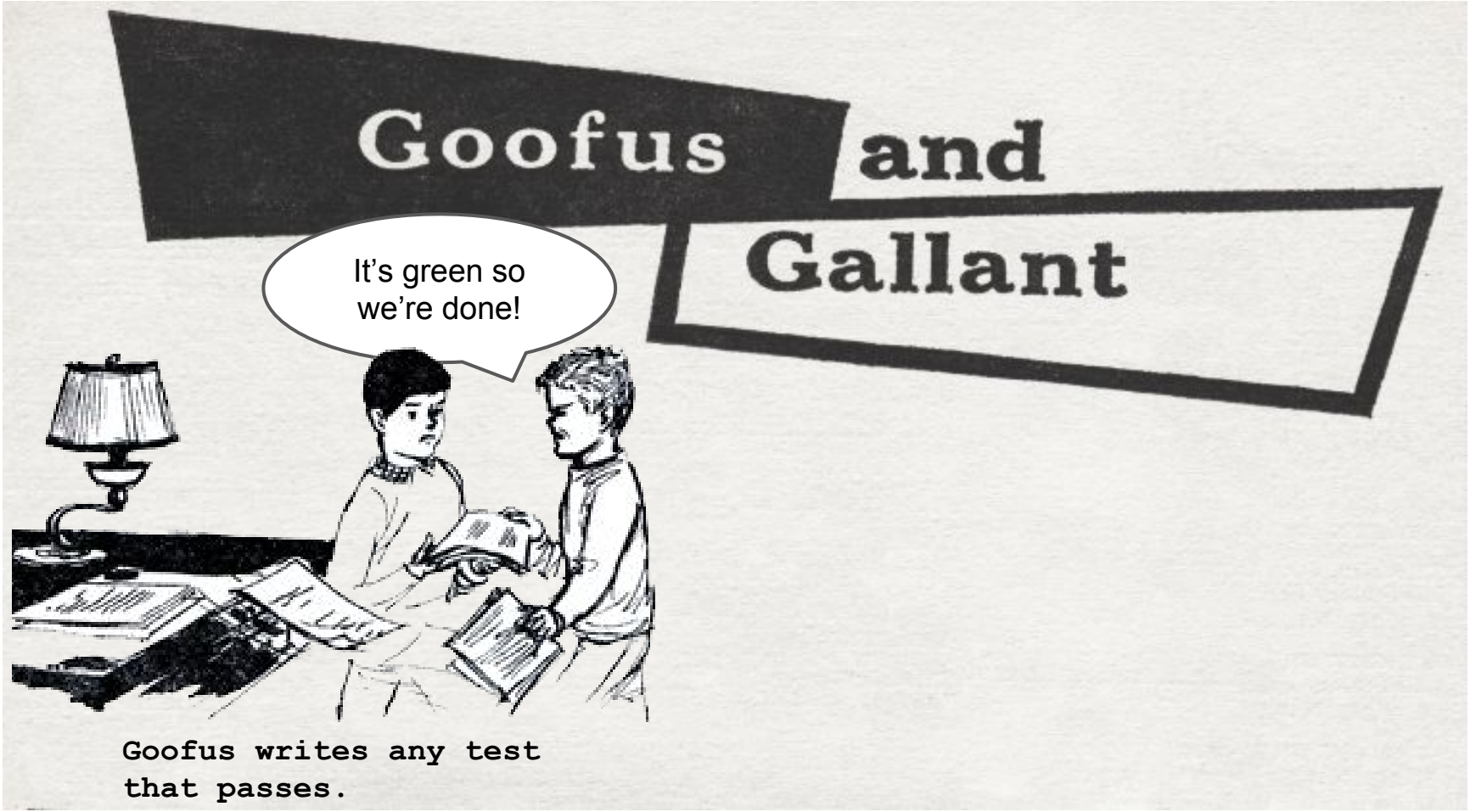


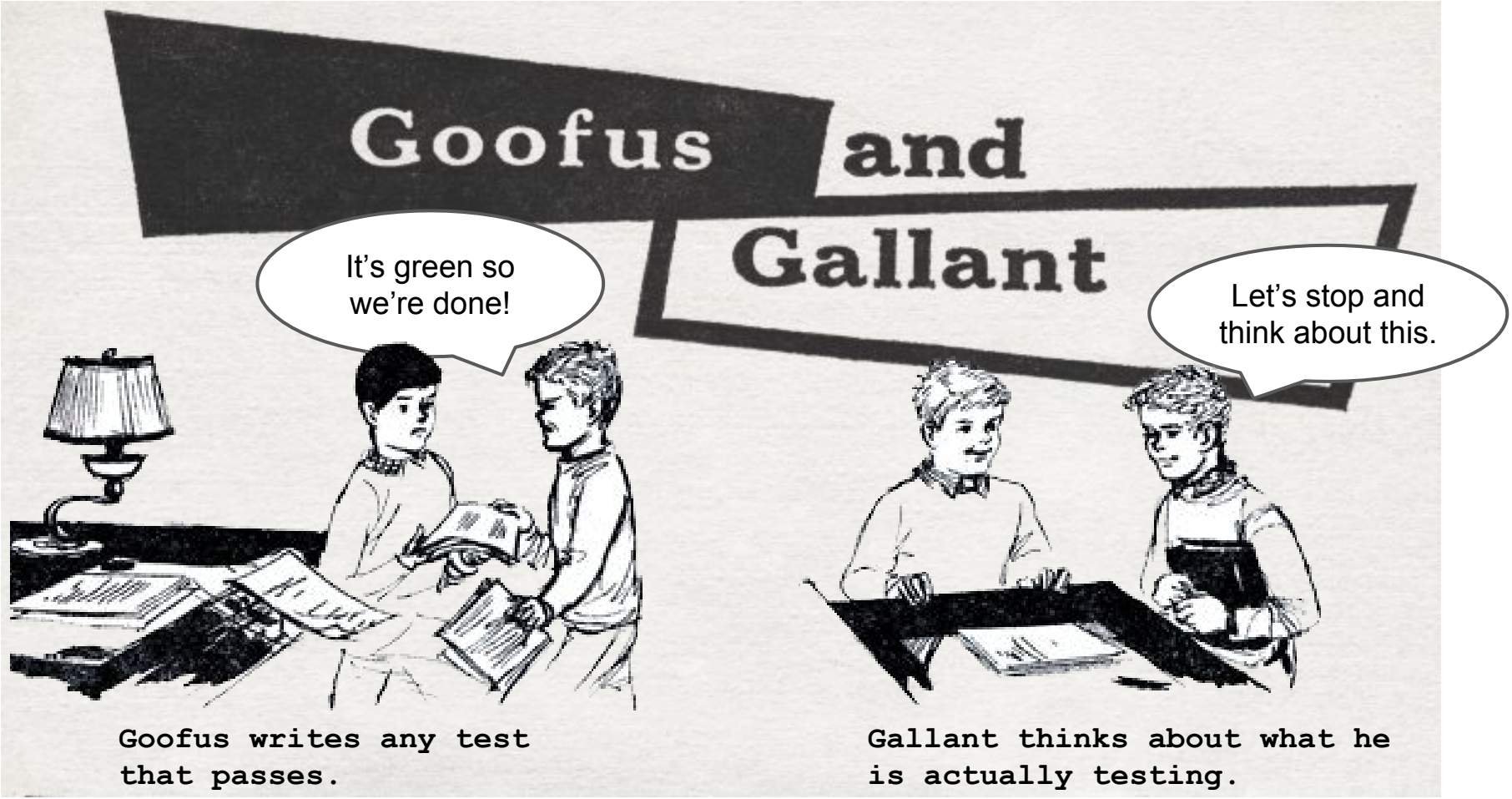
Goofus doesn't write tests.

Gallant WRITES TESTS.

Step 0

Write Tests!


Goofus and Gallant
It's green so we're done!
Goofus writes any test that passes.


Goofus and Gallant
It's green so we're done!
Let's stop and think about this.
Goofus writes any test that passes.
Gallant thinks about what he is actually testing.

Correctness

Tests must verify the requirements of the system are met.

Instead, Goofus writes:

- Tests that depend upon known bugs

Correctness

Goofus' tests depend upon known bugs.

```
int square(int x) {
  // TODO(goofus): Implement
  return 0;
}

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

Correctness

Goofus' tests depend upon known bugs.

```
int square(int x) {
  // TODO(goofus): Implement
  return 0;
}

TEST(SquareTest, MathTests) {
  EXPECT_EQ(4,  square(2));
  EXPECT_EQ(9,  square(3));
  EXPECT_EQ(49, square(7));
}
```

Correctness

Tests must verify the requirements of the system are met.

Instead, Goofus writes:

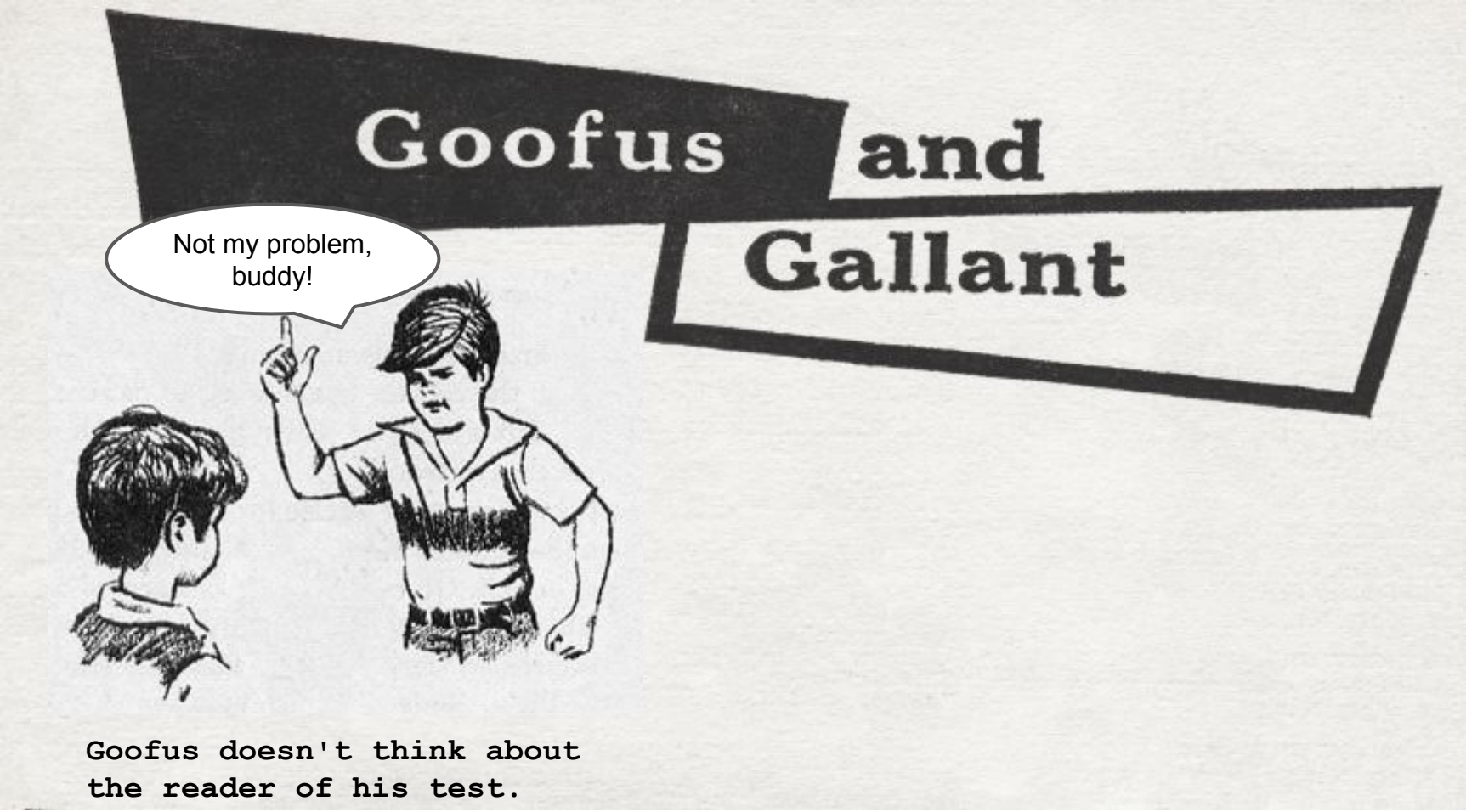
- Tests that depend upon known bugs
- Tests that don’t actually execute real scenarios

Correctness

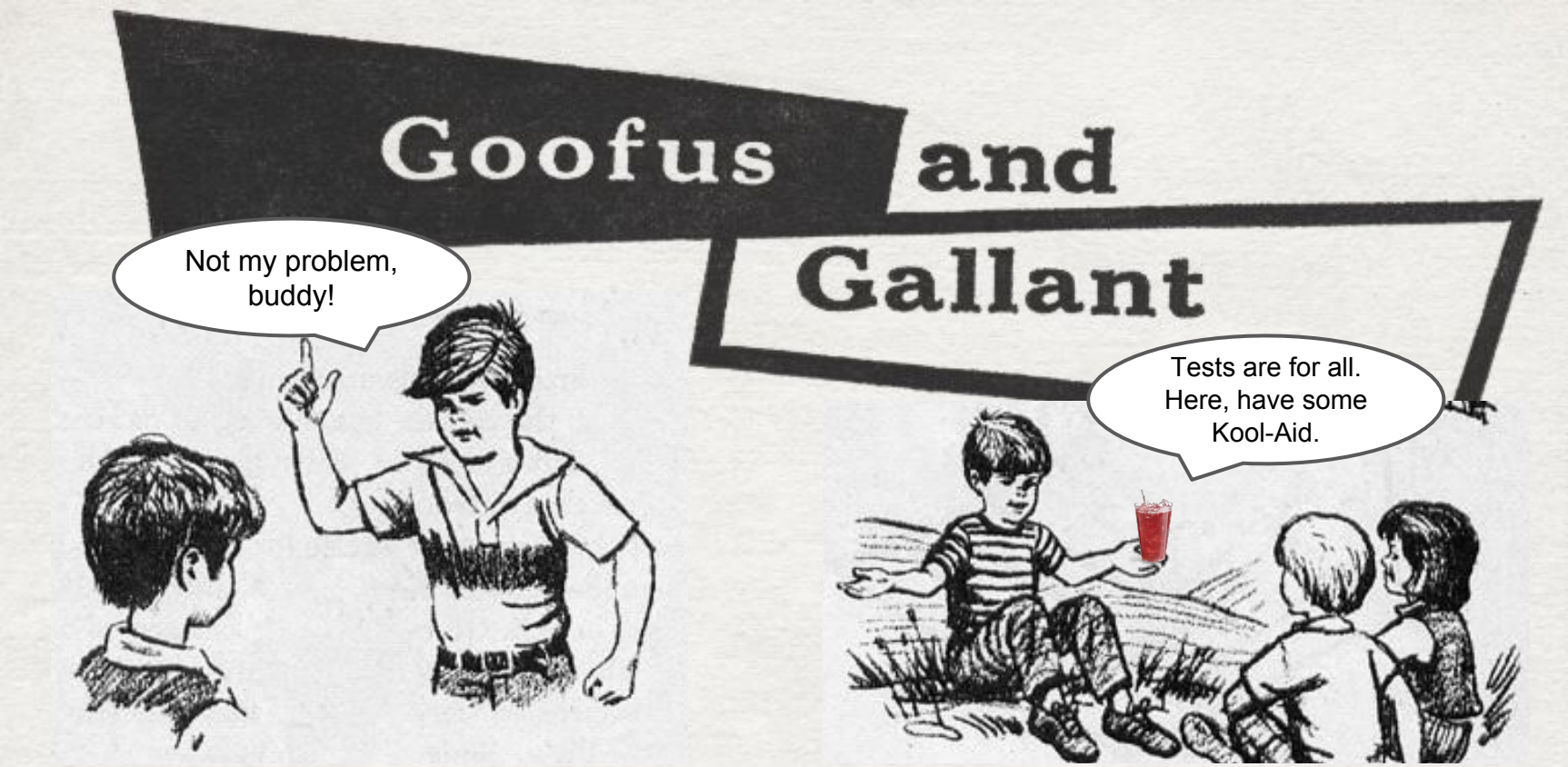
Goofus' tests are not executing real scenarios

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

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



Goofus doesn't think about the reader of his test.



Goofus doesn't think about the reader of his test.

Gallant writes tests that are easy to understand, and clearly correct.

Readability

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

Goofus writes tests that have:

- Too much boilerplate and other distraction

Readability

Avoid too much boilerplate or 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()));
}
```

Meaningless setup.

Actual test

Readability

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

Goofus writes tests that have:

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

Readability

Keep enough context for the reader

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

Readability

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

Goofus writes tests that have:

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

Readability

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());
}
```

Readability

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());
}
```

Readability

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());
}
```

Readability

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

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

Readability

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

Goofus writes tests that have:

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

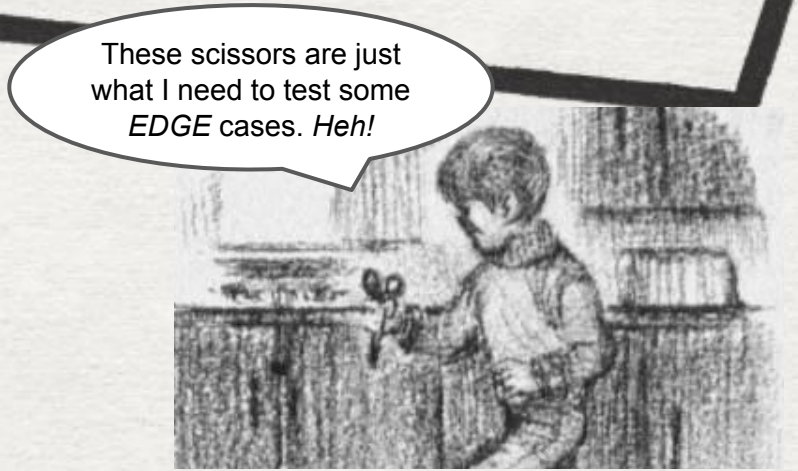
A test should be like a novel: setup, action, conclusion, and it should all make sense.

Goofus and Gallant

Goofus writes tests for two simple cases and then runs with scissors.

Goofus and Gallant

Goofus writes tests for two simple cases and then runs with scissors.



Gallant tests edge cases and demonstrates that his code is correct.

Completeness

Goofus writes tests only for the easy cases.

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

Completeness

Goofus writes 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(goofus): figure this out.
}
```

Completeness

Goofus writes tests only for the easy cases.

Gallant 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));
}
```

Completeness

Goofus writes tests for APIs that aren't his (see resilience).

```
TEST(FilterTest, WithVector) {
  vector<int> v;     // Make sure that vector is working.
  v.push_back(1);
  EXPECT_EQ(1, v.size());
  v.clear();
  EXPECT_EQ(0, v.size());
  EXPECT_TRUE(v.empty());

  // Now test our filter.
  v = Filter({1, 2, 3, 4, 5}, [](int x) { return x % 2 == 0; });
  EXPECT_THAT(v, ElementsAre(2, 4));
}
```

Completeness

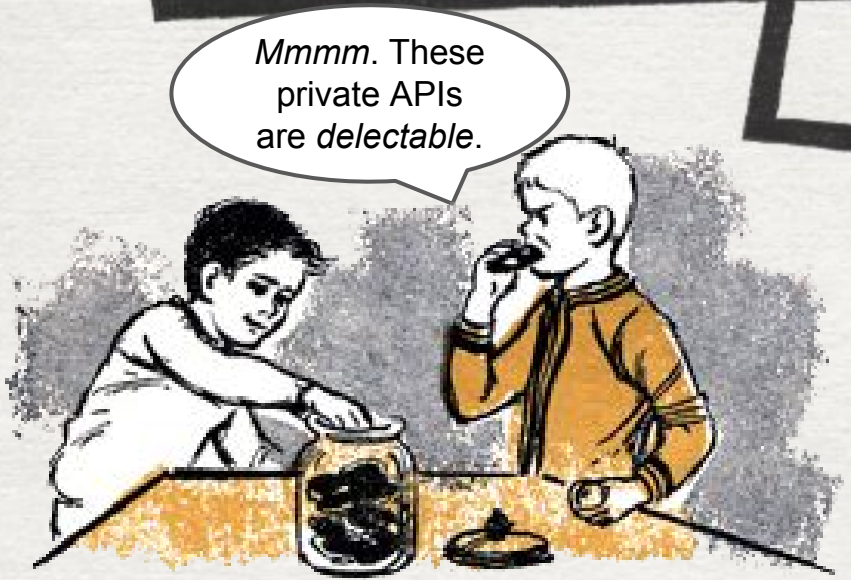
Goofus writes tests for APIs that aren't his (see resilience).

Gallant only tests that his API behaves properly while it uses that API.

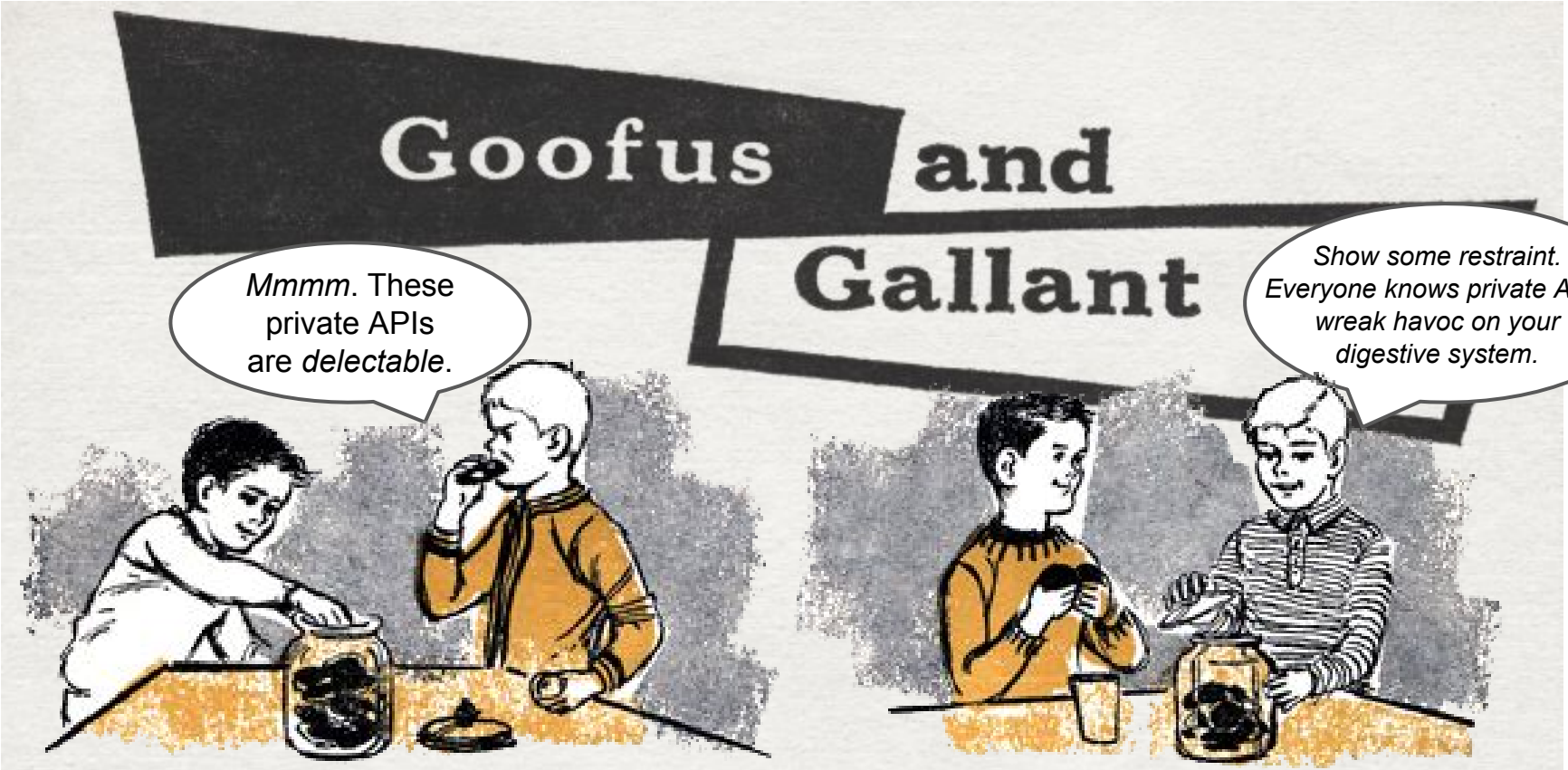
```
TEST(FilterTest, WithVector) {
  vector<int> v;     // Make sure that vector is working.
  v.push_back(1);
  EXPECT_EQ(1, v.size());
  v.clear();
  EXPECT_EQ(0, v.size());
  EXPECT_TRUE(v.empty());

  // Now test our filter.
  v = Filter({1, 2, 3, 4, 5}, [](int x) { return x % 2 == 0; });
  EXPECT_THAT(v, ElementsAre(2, 4));
}
```

Goofus and Gallant



Goofus uses private APIs in tests in ways that users couldn't.



Goofus uses private APIs in tests in ways that users couldn't.

Gallant uses tests to show how the API should be used.

Demonstrability

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

Goofus writes tests with

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

Demonstrability

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

 private:
  bool ShortcutSetupForTesting();
};

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

Demonstrability

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

 private:
  bool ShortcutSetupForTesting();
};

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

Demonstrability

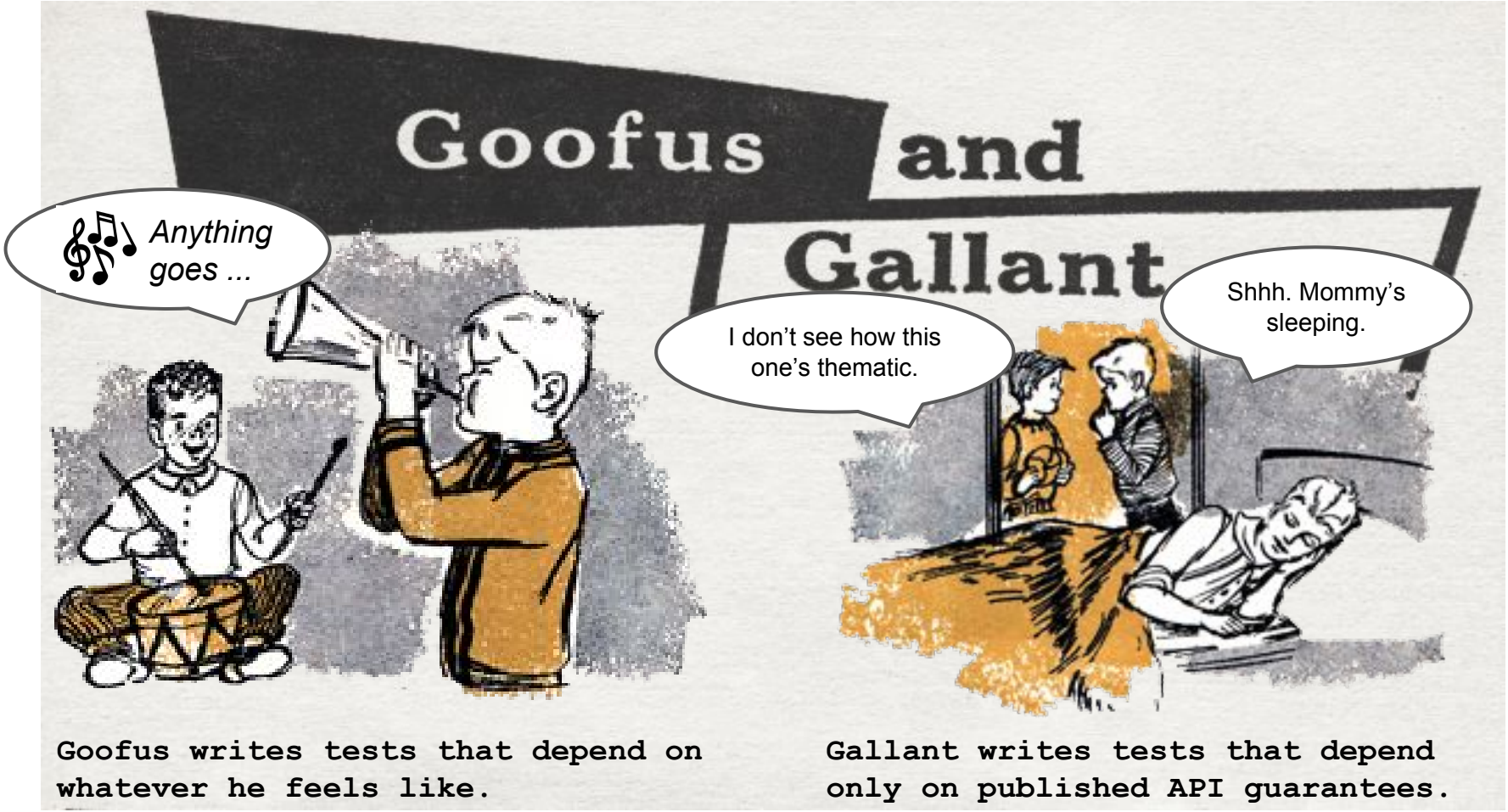
```
class Foo {
  friend FooTest;
 public:
  bool Setup();

 private:
  bool ShortcutSetupForTesting();
};

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

Goofus and Gallant

Goofus writes tests that depend on whatever he feels like.



Goofus writes tests that depend on whatever he feels like.

Gallant writes tests that depend only on published API guarantees.

Resilience

Goofus **loves** 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

Resilience

Goofus writes 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());
}
```

Resilience

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

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

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

Resilience

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

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

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

Resilience

Goofus writes 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"));
}
```

Resilience

Resilience

Resilience - Ordering

Goofus writes 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;
}
```

Resilience - Nonhermeticity

Goofus 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"));
}
```

Resilience - Deep Dependence

Goofus writes 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.

Don’t be Goofus.

Goofus and Gallant

Goofus ignores his audience.

Goofus and Gallant

Goofus ignores his audience.

Gallant answers questions.