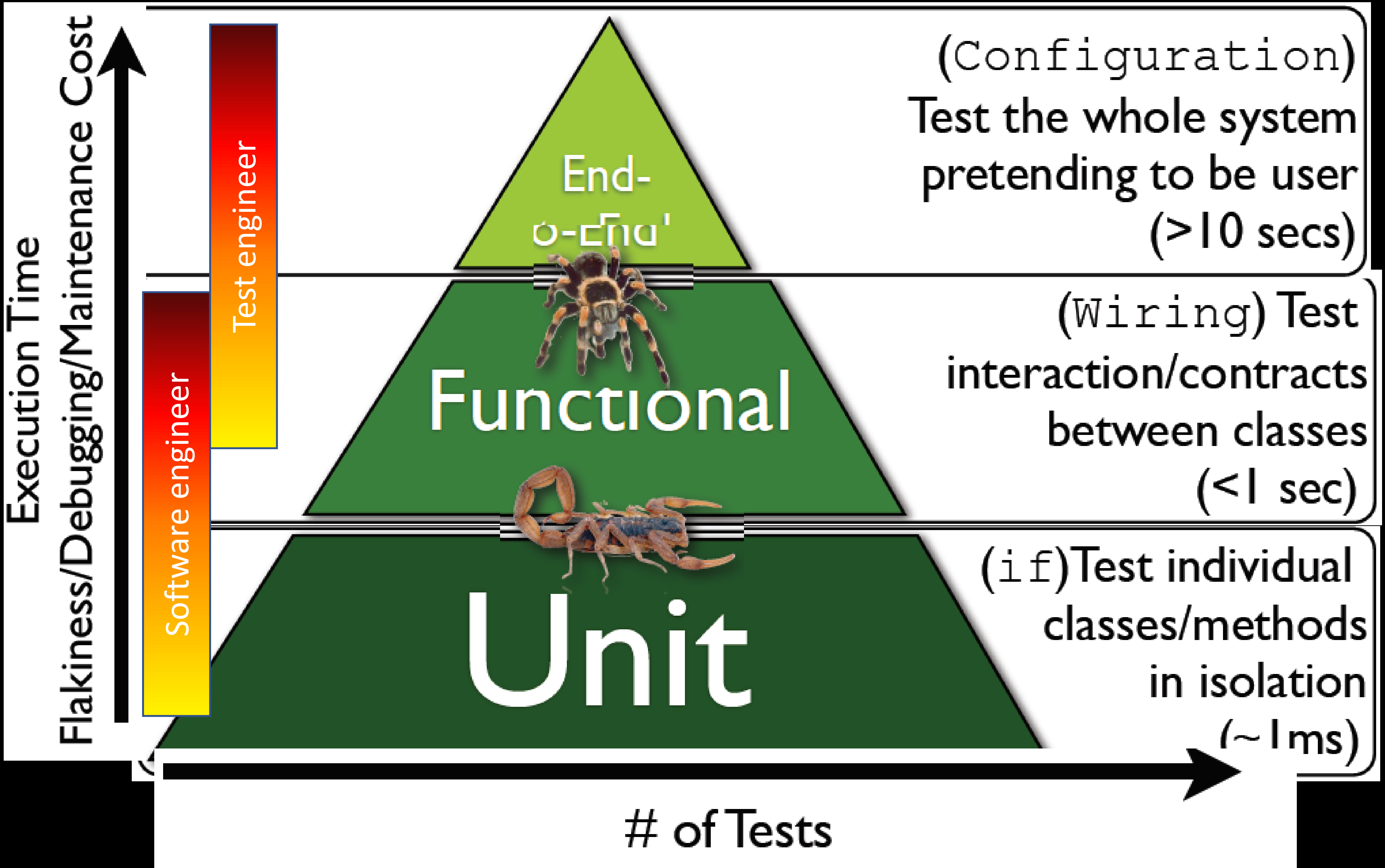
Why xUnit Testing

The state of art
software engineering practice
nowadays

品質從 *bottom up*
But HOW?

Different Kind of Testing


Flakiness/Debugging/Maintenance Cost
Execution Time
Software engineer
Test engineer
End-to-End
Functional
Unit
(Configuration) Test the whole system pretending to be user (>10 secs)
(Wiring) Test interaction/contracts between classes (<1 sec)
(if)Test individual classes/methods in isolation (~1ms)
# of Tests

The Main spirit of **AGILE** Development

- Agile focus on release a workable, runnable program iteratively and frequently
- Quality should be built from bottom, instead of waiting until the last moment. It is often too late when the system quality is a crap
- Test Driven Development
- Say NO!! to “我把我的 classes(code) 都寫好了，我們來把程式整合起來”

Module Testing

An old fashion way

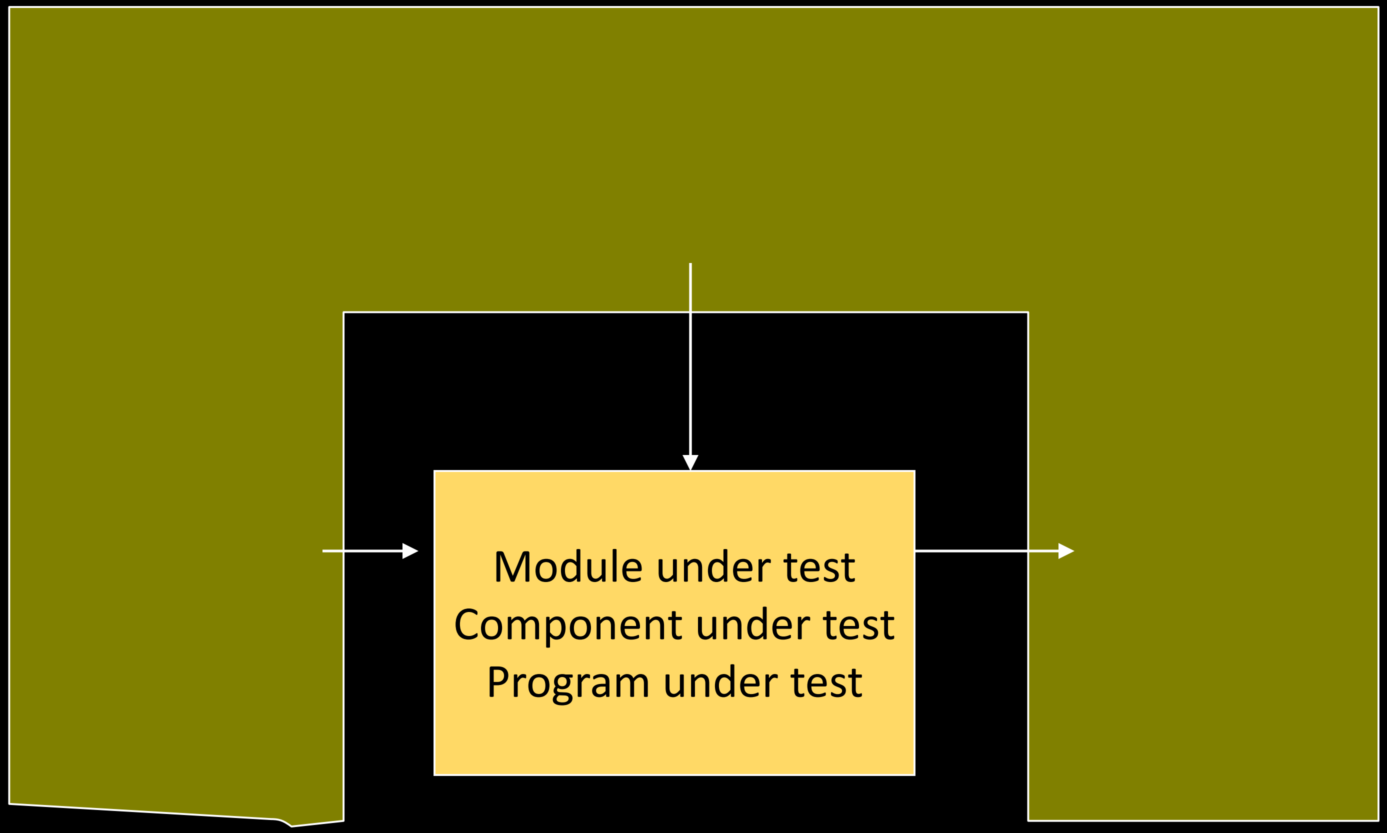
以前可能只有非常成熟開發大型軟體產品的公司會這麼做

教科書沒有教或是說得很隱諱

Module Testing

- You should not and not allowed to add your module into project if it is not fully tested.
- However, a module is an incomplete collections of code, how it can be tested?
- 有句俗諺，當一個 programmer 告訴你他寫了500行程式碼，準備整合了，他告訴你進度已經到達90%，實際上他的進度可能只有 20%
- 你應該這麼問他: “你這500行程式碼測了沒?”，測試案例有多少

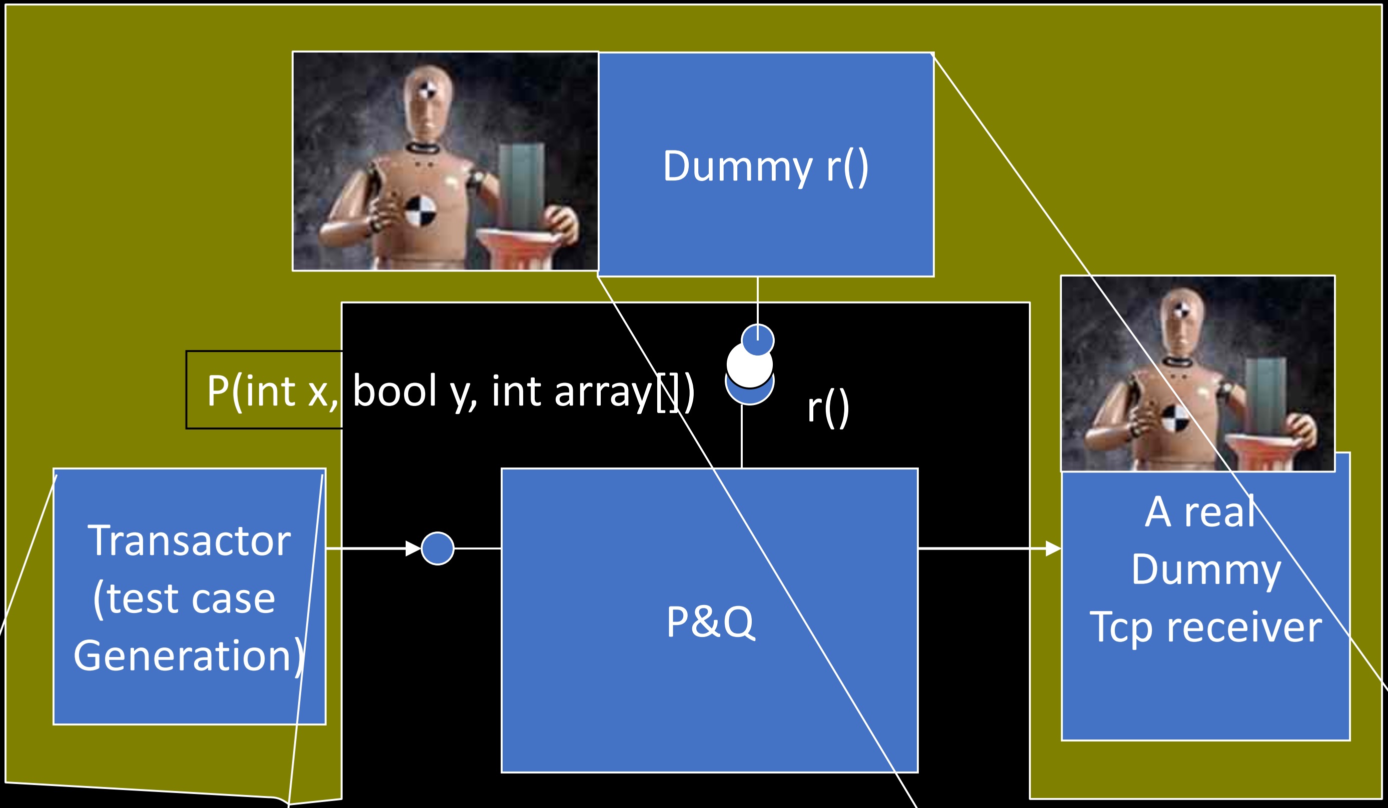
Test Bench



A first simple module (procedure-oriented)

- Consider the following module composed by two procedures p() and q()

```
void p(int x, bool y, int array[]) {
     …..                                  // do something
     q(x-10, array);
     …..                                  // do something
}
void q(int a, int array[]) {
     …..
     r(array, &msg) ;  // well I need r() to do something for me
                                  // however r() is not written by me.
     …..
     sendTCPmsg(msg) ; // I will send a message
}
```


Dummy r()
P(int x, bool y, int array[])
r()
Transactor (test case Generation)
P&Q
A real Dummy Tcp receiver

```
Main() {
    prepare a good array
    p(100, true, array) ;
    prepare a good array
    p(50, false , array);
     …..
    prepare a bad array
    p(-1, true, array) ;
}
```

Test case 1

Test case 2

Test case n

```
void r(int array, string &msg) {
    if array content is test case 1
    return msg1 ;
    if array content is test case 2
     return msg2 ;
     …….
}
```

How good your test is?

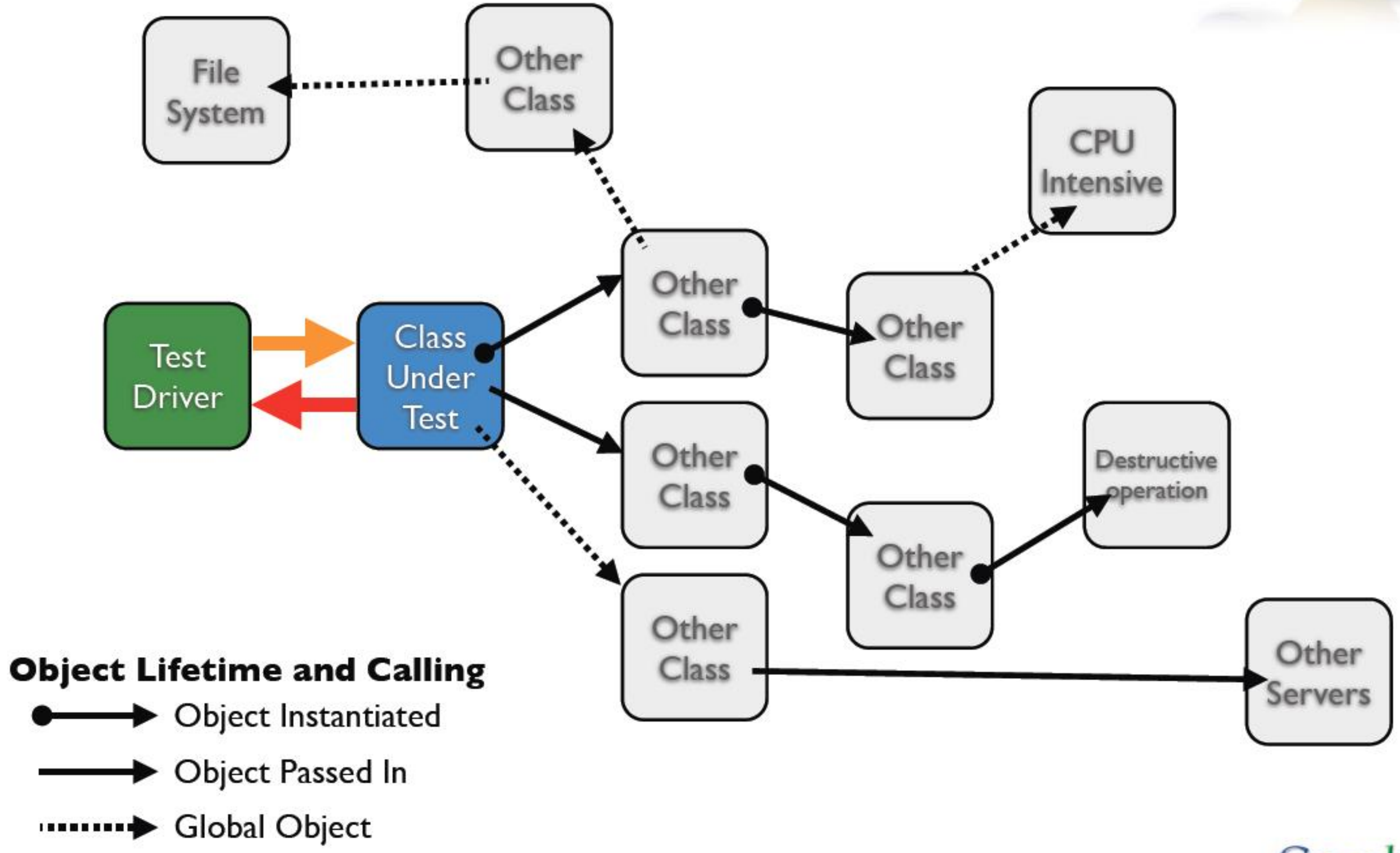
- The theory of testing will be explained in later chapters
- Here are some general rules
    - Each statement of your program should be visited at least by one test case
    - If your code contains complex algorithms to deal with many combinations of test input, each combination should be tested as much as possible.
    - You need to try bad input to show your code can defend malicious input from your teammate.

Problem of old fashion module testing

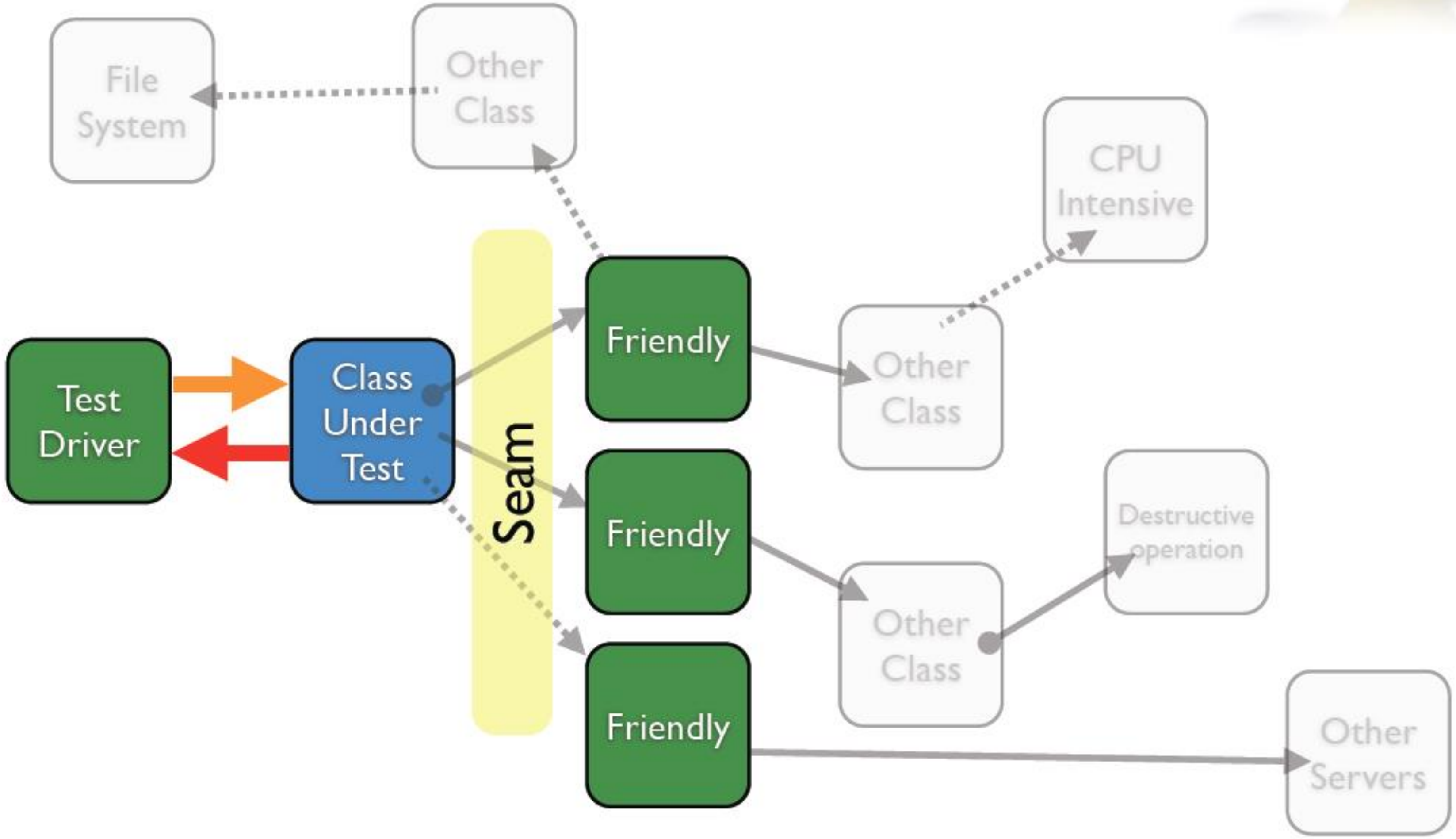
- No standard
- A lot of work must be done by programmers to automate the module testing and programmers do not like it.
- Most software development team choose not to.

The basic idea –
Testing in isolation

Unit Testing a Class



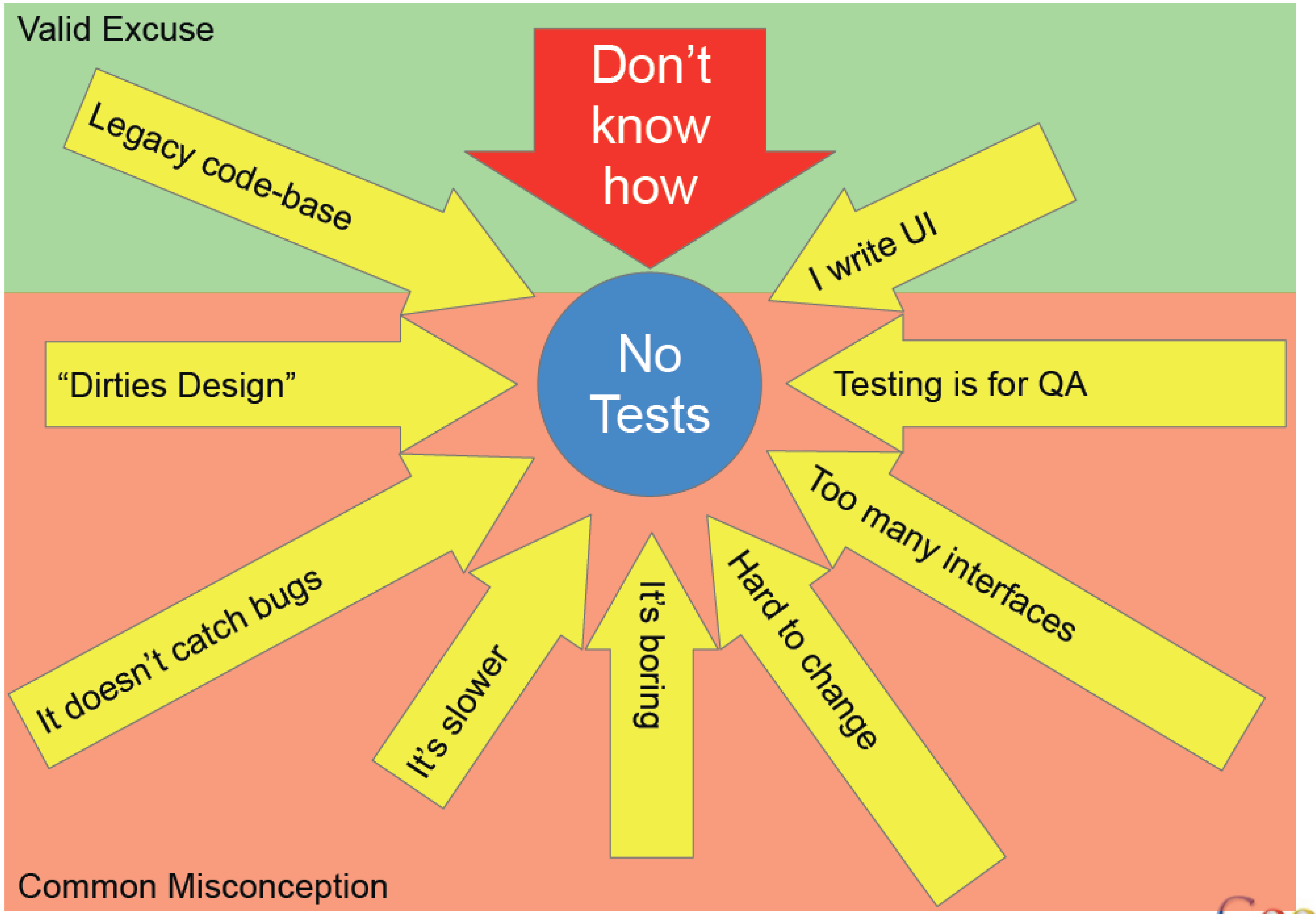
Unit Testing a Class



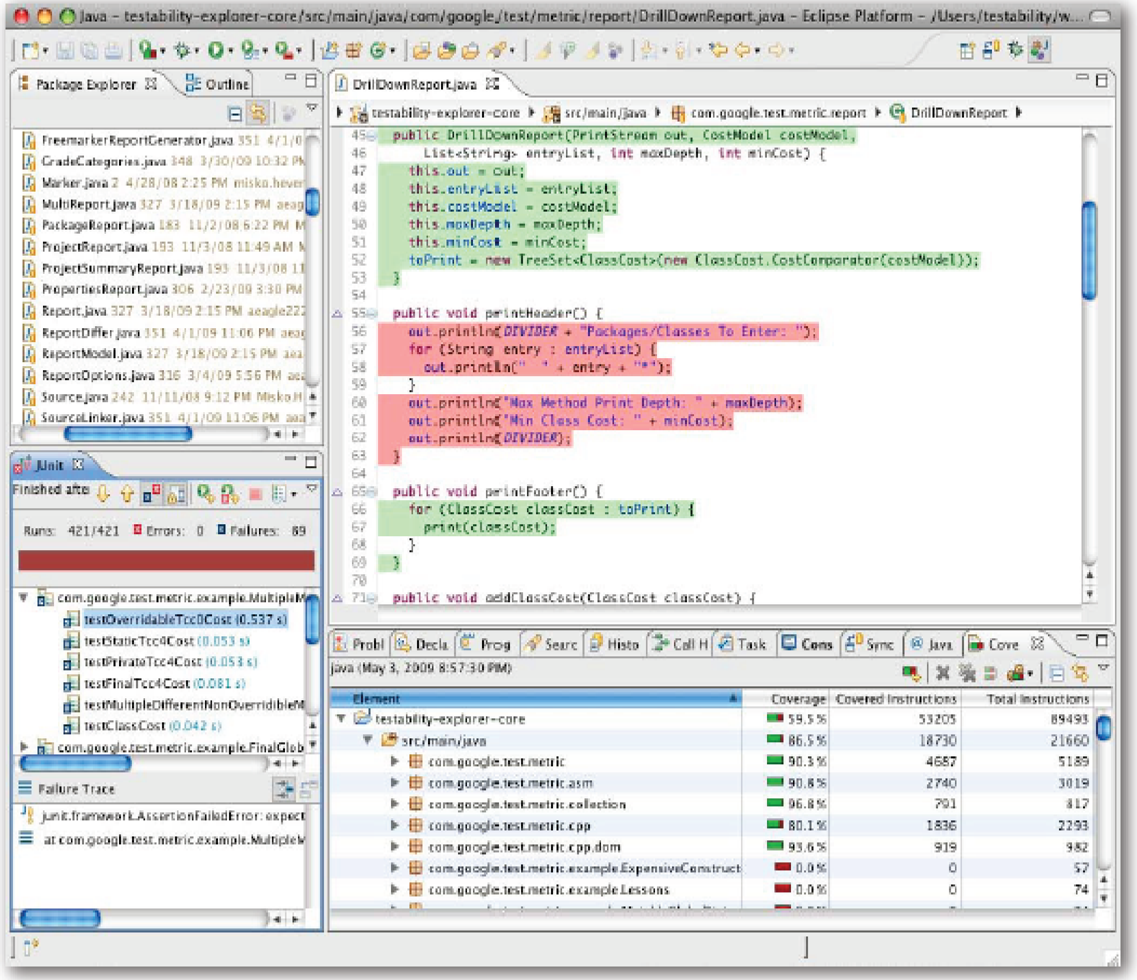
Google

Paradigm Shift (典範轉移)
is never easy
Lets' see how Google
achieved it

Excuses


Valid Excuse
Don't know how
Legacy code-base
I write UI
No Tests
"Dirties Design"
Testing is for QA
It doesn't catch bugs
It's slower
It's boring
Hard to change
Too many interfaces
Common Misconception

Tools: Coverage



Java – testability-explorer-core/src/main/java/com/google/test/metric/report/DrillDownReport.java – Eclipse Platform – /Users/testability/w...

Package Explorer

```
FreemarkerReportGenerator.java 351 4/1/0
GradeCategories.java 348 3/30/09 10:32 PM
Marker.java 2 4/28/08 2:25 PM misko.hever
MultiReport.java 327 3/18/09 2:15 PM aeag
PackageReport.java 183 11/2/08 6:22 PM M
ProjectReport.java 193 11/3/08 11:49 AM M
ProjectSummaryReport.java 193 11/3/08 11
PropertiesReport.java 306 2/23/09 3:30 PM
Report.java 327 3/18/09 2:15 PM aeagle222
ReportDiffer.java 351 4/1/09 11:06 PM aeag
ReportModel.java 327 3/18/09 2:15 PM aea
ReportOptions.java 316 3/4/09 5:56 PM aea
Source.java 242 11/11/08 9:12 PM Misko.H
SourceLinker.java 351 4/1/09 11:06 PM aea
```

JUnit

```
Finished afte
Runs: 421/421   Errors: 0   Failures: 89

com.google.test.metric.example.MultipleM
    testOverridableTcc0Cost (0.537 s)
    testStaticTcc4Cost (0.053 s)
    testPrivateTcc4Cost (0.053 s)
    testFinalTcc4Cost (0.081 s)
    testMultipleDifferentNonOverridibleM
    testClassCost (0.042 s)
com.google.test.metric.example.FinalGlob

Failure Trace
junit.framework.AssertionFailedError: expect
at com.google.test.metric.example.MultipleM
```

DrillDownReport.java

testability-explorer-core ▸ src/main/java ▸ com.google.test.metric.report ▸ DrillDownReport ▸

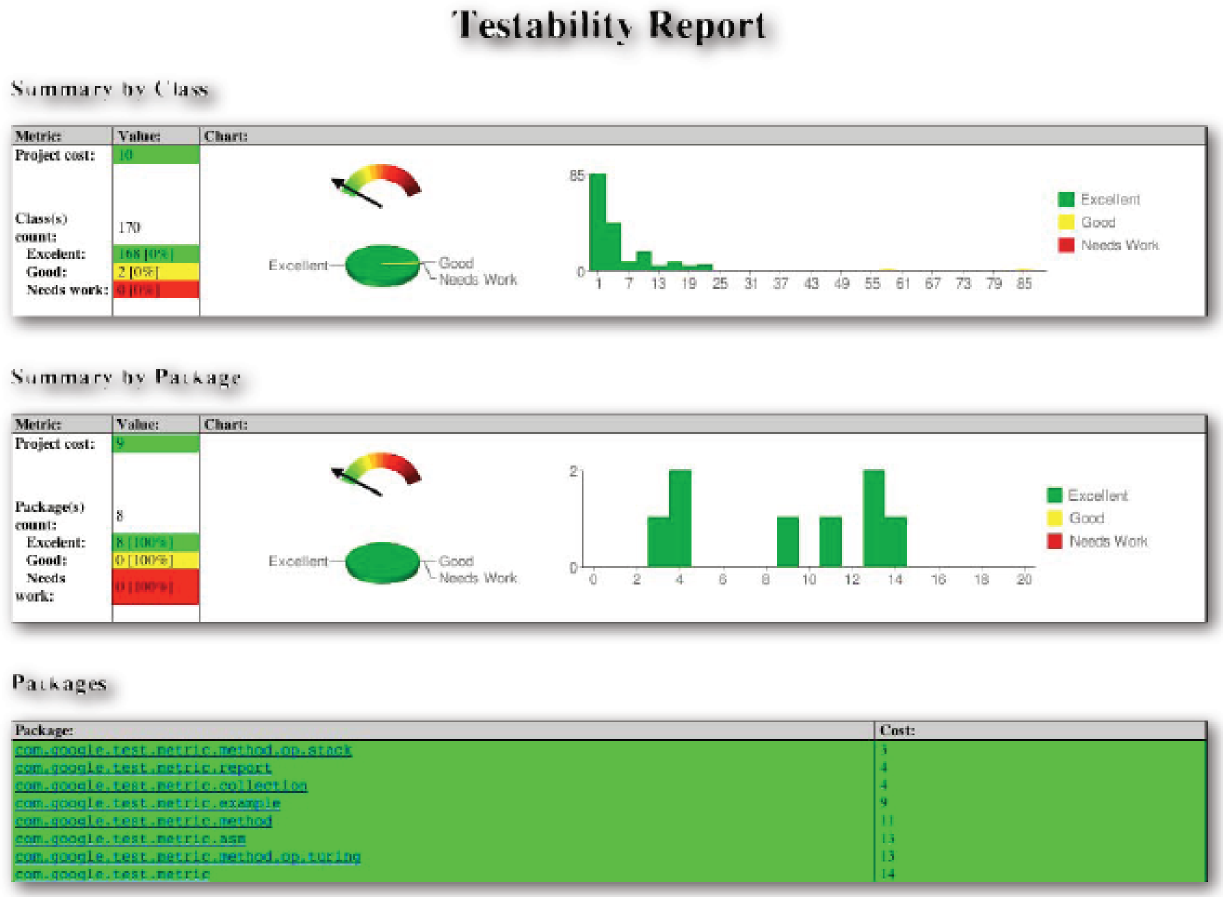
```java
  public DrillDownReport(PrintStream out, CostModel costModel,
      List<String> entryList, int maxDepth, int minCost) {
    this.out = out;
    this.entryList = entryList;
    this.costModel = costModel;
    this.maxDepth = maxDepth;
    this.minCost = minCost;
    toPrint = new TreeSet<ClassCost>(new ClassCost.CostComparator(costModel));
  }

  public void printHeader() {
    out.println(DIVIDER + "Packages/Classes To Enter: ");
    for (String entry : entryList) {
      out.println("  " + entry + "*");
    }
    out.println("Max Method Print Depth: " + maxDepth);
    out.println("Min Class Cost: " + minCost);
    out.println(DIVIDER);
  }

  public void printFooter() {
    for (ClassCost classCost : toPrint) {
      print(classCost);
    }
  }

  public void addClassCost(ClassCost classCost) {
```

java (May 3, 2009 8:57:30 PM)

Element	Coverage	Covered Instructions	Total Instructions
testability-explorer-core	59.5 %	53205	89493
src/main/java	86.5 %	18730	21660
com.google.test.metric	90.3 %	4687	5189
com.google.test.metric.asm	90.8 %	2740	3019
com.google.test.metric.collection	96.8 %	791	817
com.google.test.metric.cpp	80.1 %	1836	2293
com.google.test.metric.cpp.dom	93.6 %	919	982
com.google.test.metric.example.ExpensiveConstruct	0.0 %	0	57
com.google.test.metric.example.Lessons	0.0 %	0	74

- Simple but effective
- Just having it installed makes people thing about improving coverage
- Squeaky wheel gets the grease

Tools: Testability Report



Testability Report

Summary by Class

Metric:	Value:	Chart:
Project cost:	10	
Class(s) count:	170	
Excelent:	168 [0%]	
Good:	2 [0%]	
Needs work:	0 [0%]	

Summary by Package

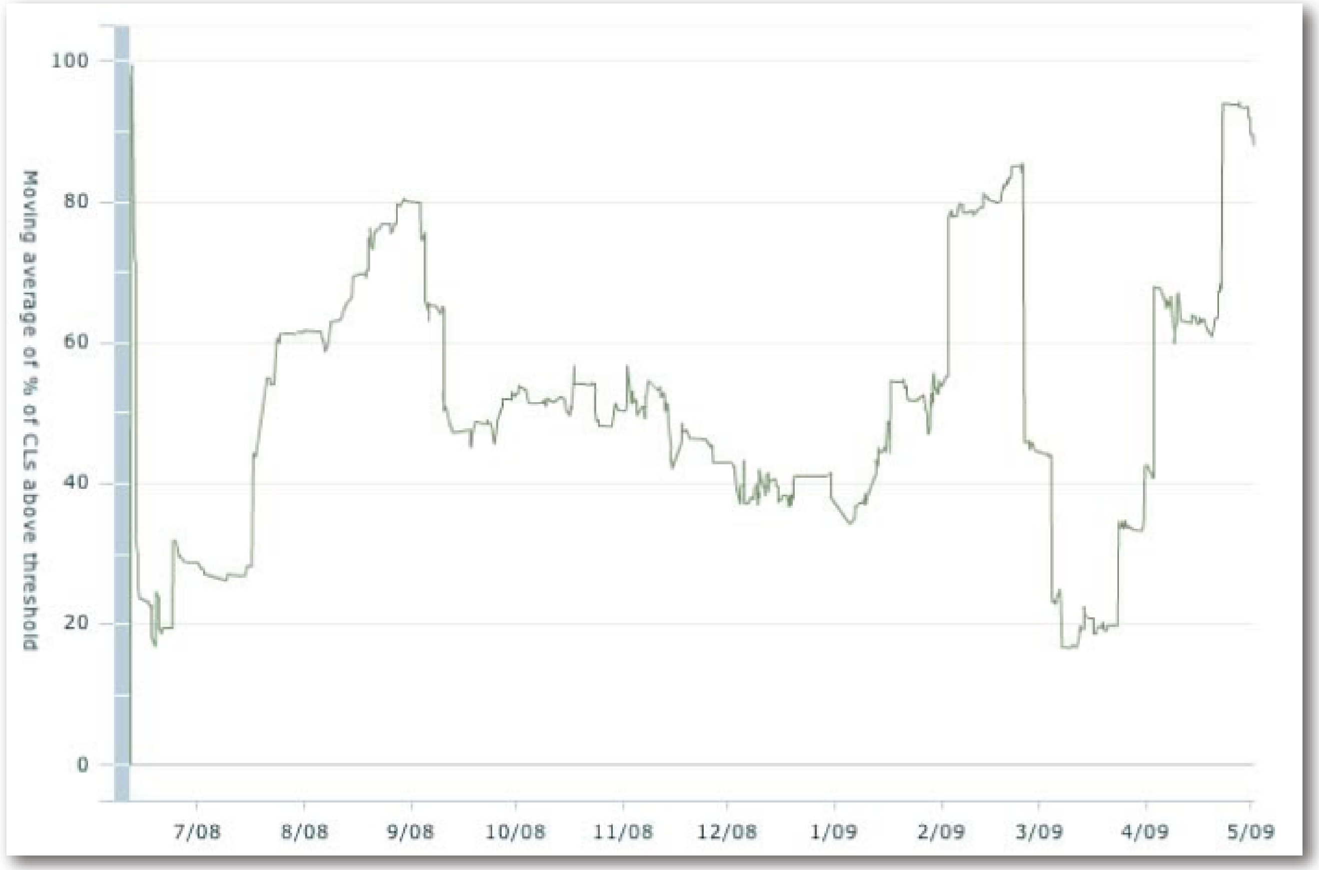
Metric:	Value:	Chart:
Project cost:	9	
Package(s) count:	8	
Excelent:	8 [100%]	
Good:	0 [100%]	
Needs work:	0 [100%]	

Packages

Package:	Cost:
com.google.test.metric.method.op.stack	3
com.google.test.metric.report	4
com.google.test.metric.collection	4
com.google.test.metric.example	9
com.google.test.metric.method	11
com.google.test.metric.asm	13
com.google.test.metric.method.op.turing	13
com.google.test.metric	14

- Code coverage for testability
- Get high level view about how testable the code is

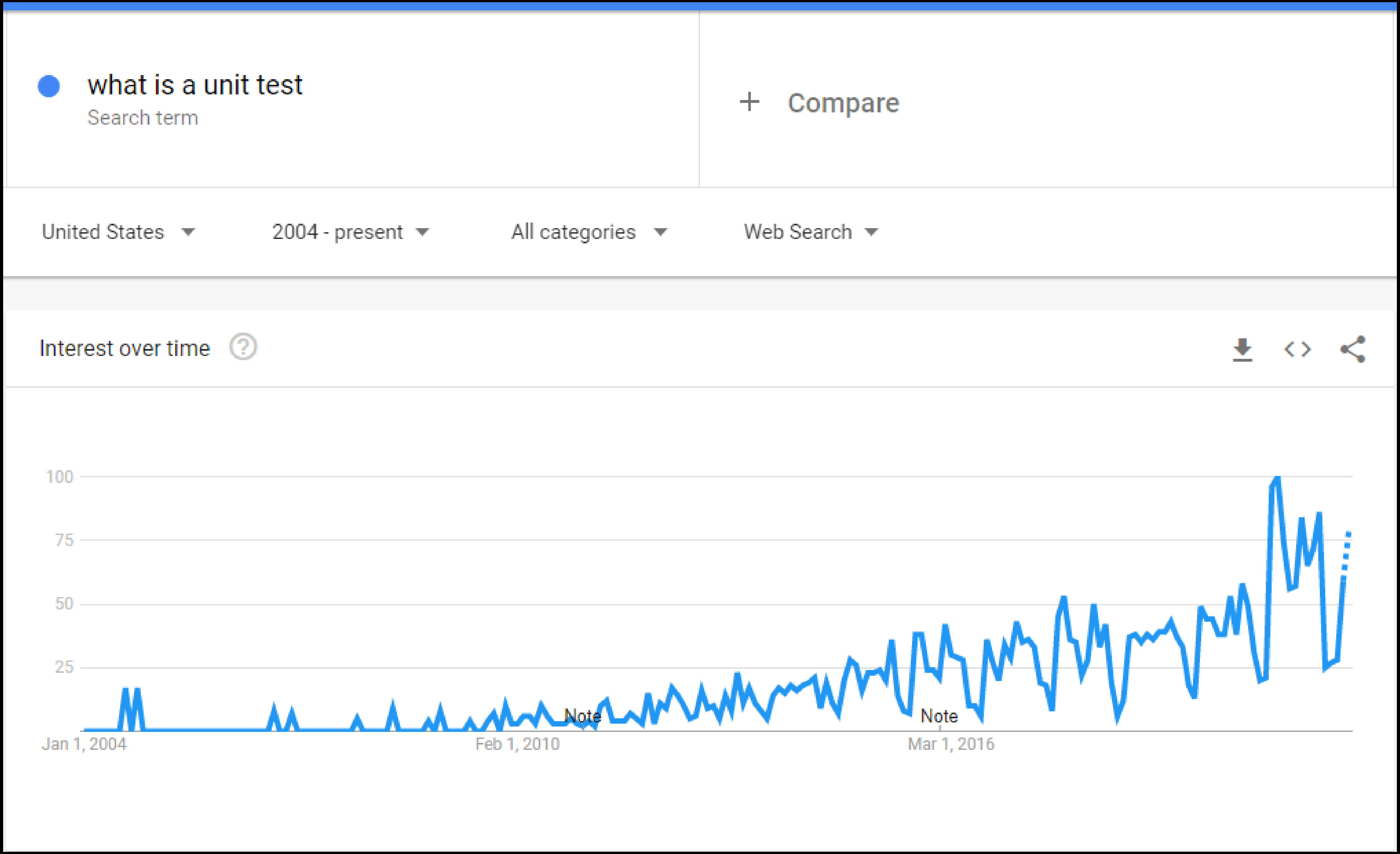
Visibility: Test Trend



- Test focus over time

Coding Paradigm Shift

- Most test efforts (quality assurance responsibility) are shifted to programmers
- YES !! Programmer loading is increased !! Fortunately, your salary is increased !!
- Kicked his ass when a programmer says “I am not here to do testing !!”
- Google spend a lot of effort to make writing unit tests as a must
- You guys are comparatively, learn from scratch



what is a unit test
Search term

Compare

United States
2004 - present
All categories
Web Search

Interest over time

Benefits of Unit Testing 1

- **Unit testing provides documentation**
  Unit tests are a kind of living documentation of the product. To learn what functionality is provided by one module or another, developers can refer to unit tests to get a basic picture of the logic of the module and the system as a whole. Unit test cases represent indicators that carry information about appropriate or inappropriate use of a software component. Thus, these cases provide the perfect documentation for these indicators.

Benefits of Unit Testing 2

- **R2: Reusable and Reliable**
- Within unit tests environments, the individual modules of a product become isolated from one another and have their own area of responsibility. That means that the code is more reliable—it's been tested in a contained environment—and therefore, reusable. Reusable code is a win-win for all: It's clean, efficient, and consistent. All of this is accelerated with unit testing.

Benefits of Unit Testing 3

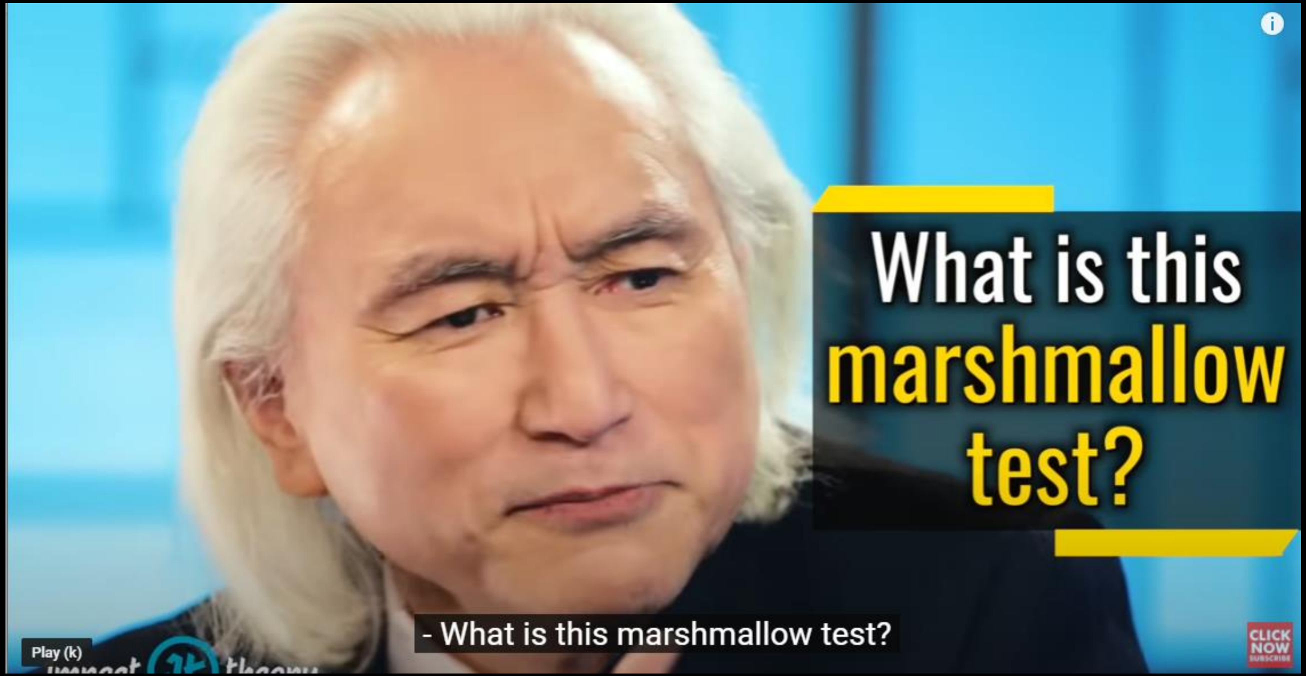
- **Unit testing helps gauge performance**
- Wouldn’t it be amazing if you could find out the possible breaking points of your software before it goes into production and users spot them on their own? Unit tests can provide such an opportunity, and can prevent the unnecessary effort of searching for solutions to non-existent issues. For instance, if you work on a hashed list, you may face a need to check how it will perform if the list grows. The rate of growth is unknown. What you will likely do from here is apply unit tests to try out scenarios of varying levels of probability — from highly probable to absurd. If you are already sure that the number of items in the hashed list won’t surpass 10,000 under any circumstances, at 100,000 you are all set and can stop at this point. You have proven your software capacity is good enough, and there’s no need to spend more time testing.

Benefits of Unit Testing 4

- **Unit testing improves code coverage**
- Okay, so software unit testing is great, but how much test coverage is necessary? American software engineer Robert C. Martin, also known as Uncle Bob, argues that the goal of test coverage should be that 100 percent of the code is covered. Opinions among developers on this issue differ: some support a complete code coverage policy, while others consider this practice redundant — a topic too lengthily for the purposes of this article. In any case, when writing unit tests, you can use many tools that determine the total percentage of the coverage of a project, separate module, or function. These tools are also able to graphically display the code sections covered by tests and indicate the sections in the code for which it makes most sense to write unit tests.
- It's very useful to know at the active code writing stage if a particular line will ever be executed or you if can painlessly remove it. If you have valid unit tests, you can have coverage metrics on hand right away, and decide whether a code line is ever relevant. If it's not, consider expanding your code coverage with one more test. If your test suite already accounts for all possible scenarios, eliminate the unnecessary code. However, the need for more tests is a sign of increasing cyclomatic complexity.

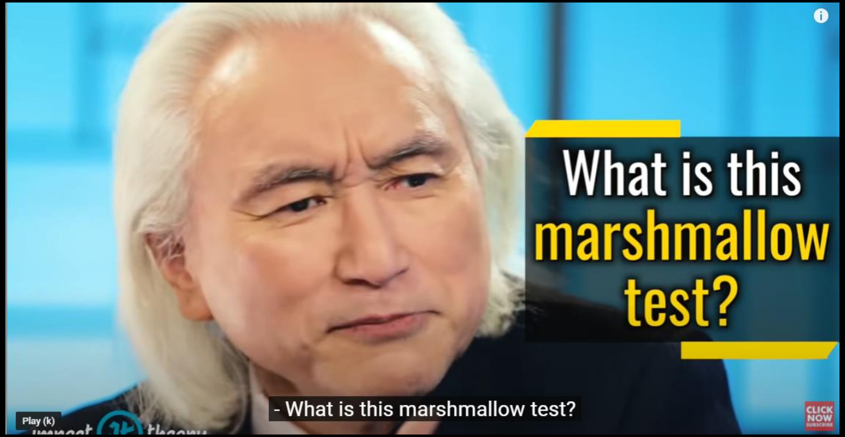
To do or not to do
(that is the question)

Marshmallow Test


What is this marshmallow test?
- What is this marshmallow test?
Play (k)
CLICK NOW

- https://www.youtube.com/watch?v=47DmUM7MW7s

Bottom Up vs Top Down


What is this marshmallow test?
- What is this marshmallow test?
Play (k)

- Project vs Product
- Short-term fix product vs long-term maintained project?
- Marshmallow test told you that long term and teamwork is a key factor
- Should you build xUnit tests when you are building a start-up product?
- Old trial code are not always unit testable
- There is a critical timing that you need to refactor/build unit tests/build e2e test automation. If it is a concrete product, unit tests should be done from the very beginning
- 迷思: 我寫 code 又要寫單元測試，只會拖累我的速度。
  - 這個迷思可能是真的，如果你的系統大概就是1000-2000 LOCs
  - 這個迷思大部分不是真的，如果你的系統偏大，而且是多人團隊合作
  - 跟棉花糖測試一樣，你選擇 quick kill，但是你卻不去計算花在團隊整合以及整合測試除錯的時間。通常你只是計算寫完程式碼的時間

軟體工程的作為常常讓你覺得是烏龜賽跑的烏龜，但是時間越長卻讓你跑得越快。(正確的說法是長遠你會變得更快速)

實踐是唯一的真理，當然如果你一直實踐在程式碼不大，不太長遠維護，變動擴充很少的專案或產品，我認為你也會寫測試寫到懷疑人生。

對井裡的蛙不可與它談論關於海的事情，是由於它的眼界受著狹小居處的局限；對夏天生死的蟲子不可與它談論關於冰雪的事情，是由於它的眼界受著時令的制約。