Anthony Ventresque

anthony.ventresque@ucd.ie

Practicum

COMP47360

Code Quality

School of Computer Science, UCD Scoil na Ríomheolaíochta, UCD

Designing ***Good*** Software

- Three elements can define what is a good software:
  - It does what it is supposed to do well: ***Function***
  - It is easy to adapt/modify: ***Maintenance***
  - It works well given a particular system: ***Performance***

- Different perspectives/preference (e.g., OO community focuses on maintenance).

Outline for today

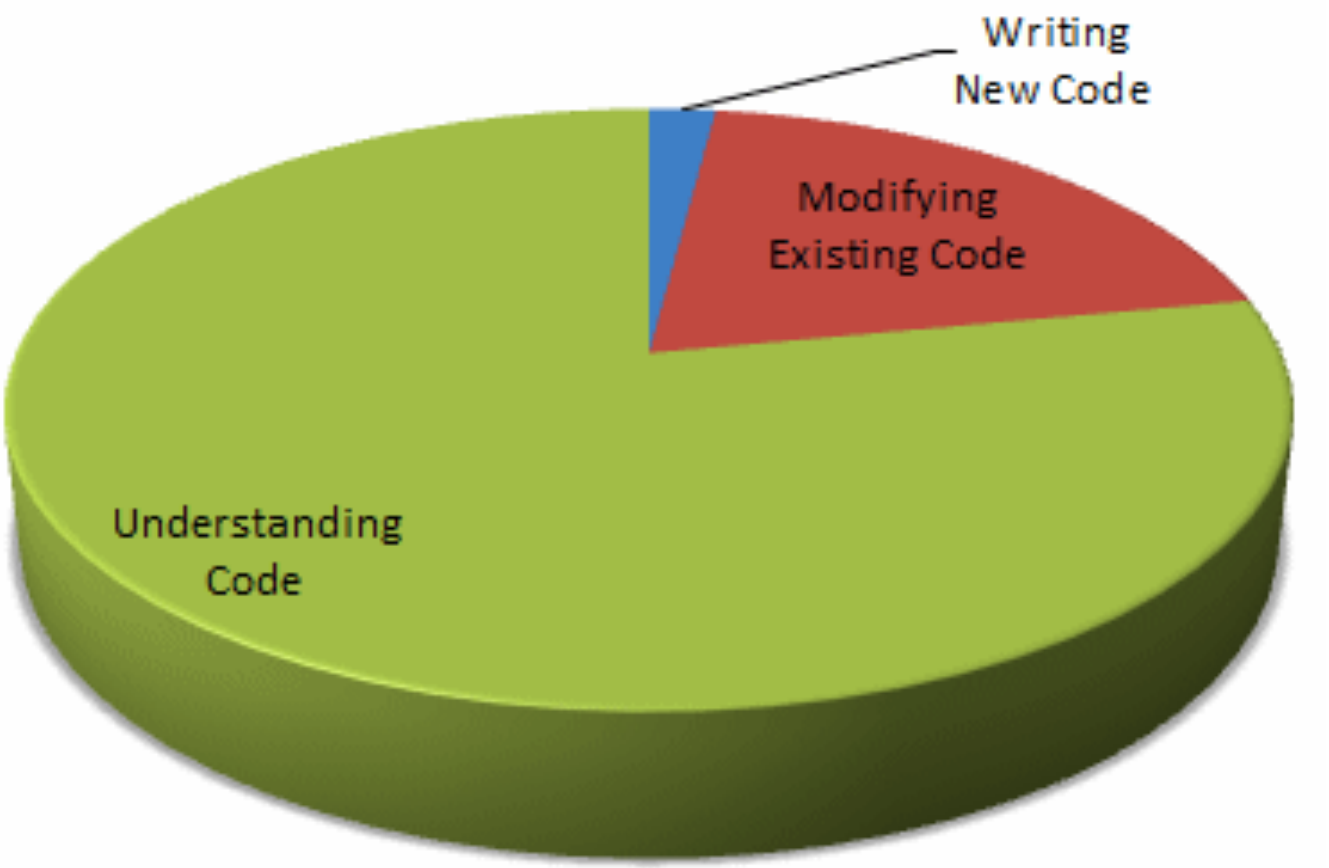
- Writing Quality Code
  - Why? Measures
  - Improve quality of written code
- Performance Profiling
  - Why? What?
  - Static instrumentation
  - Load testing
- Testing
  - Unit testing with unittest and Pytest

Take home message:

> *Designing and implementing good software artefacts is important and there are different strategies to achieve it.*

WRITING QUALITY CODE

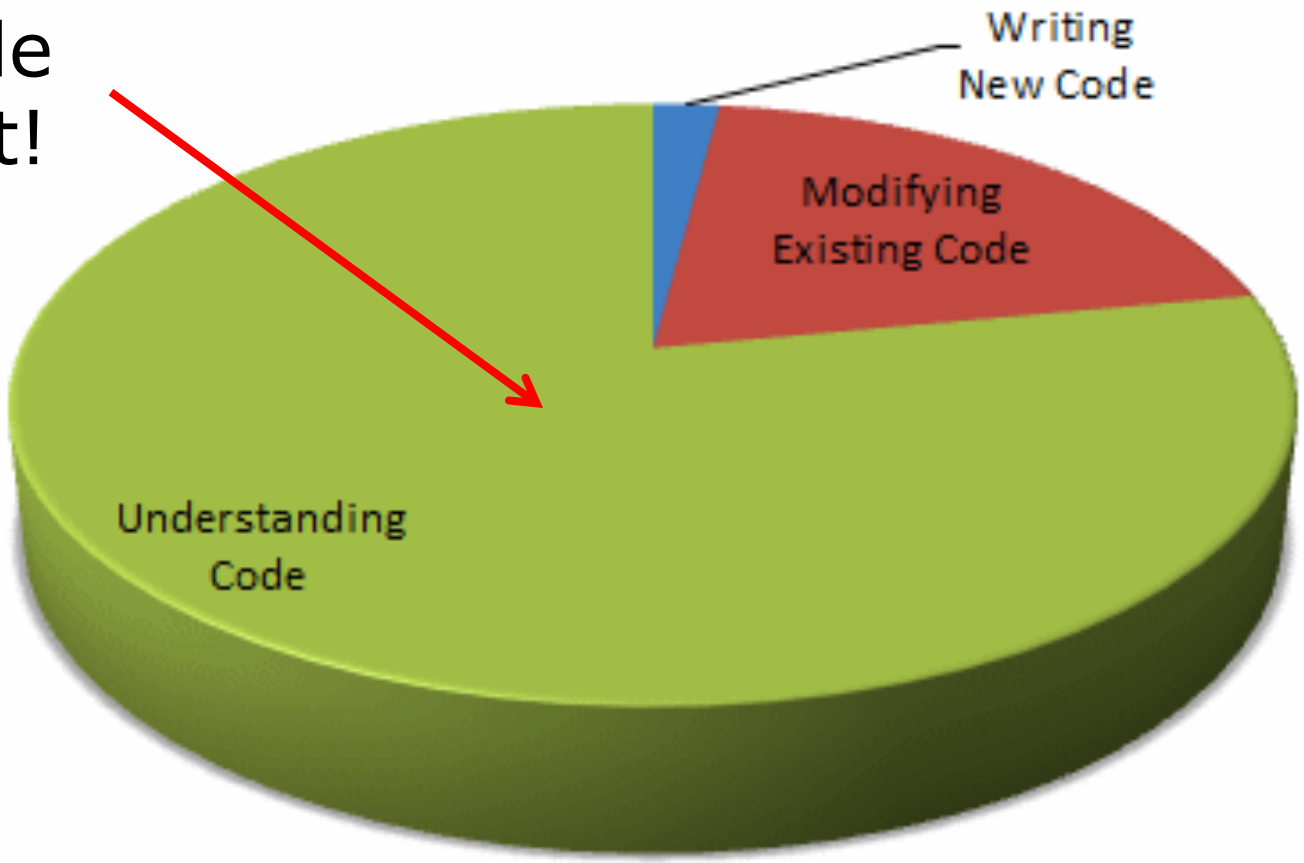
Good Code from a Developer's point of view: Understandable



Source: http://blog.codinghorror.com/when-understanding-means-rewriting/

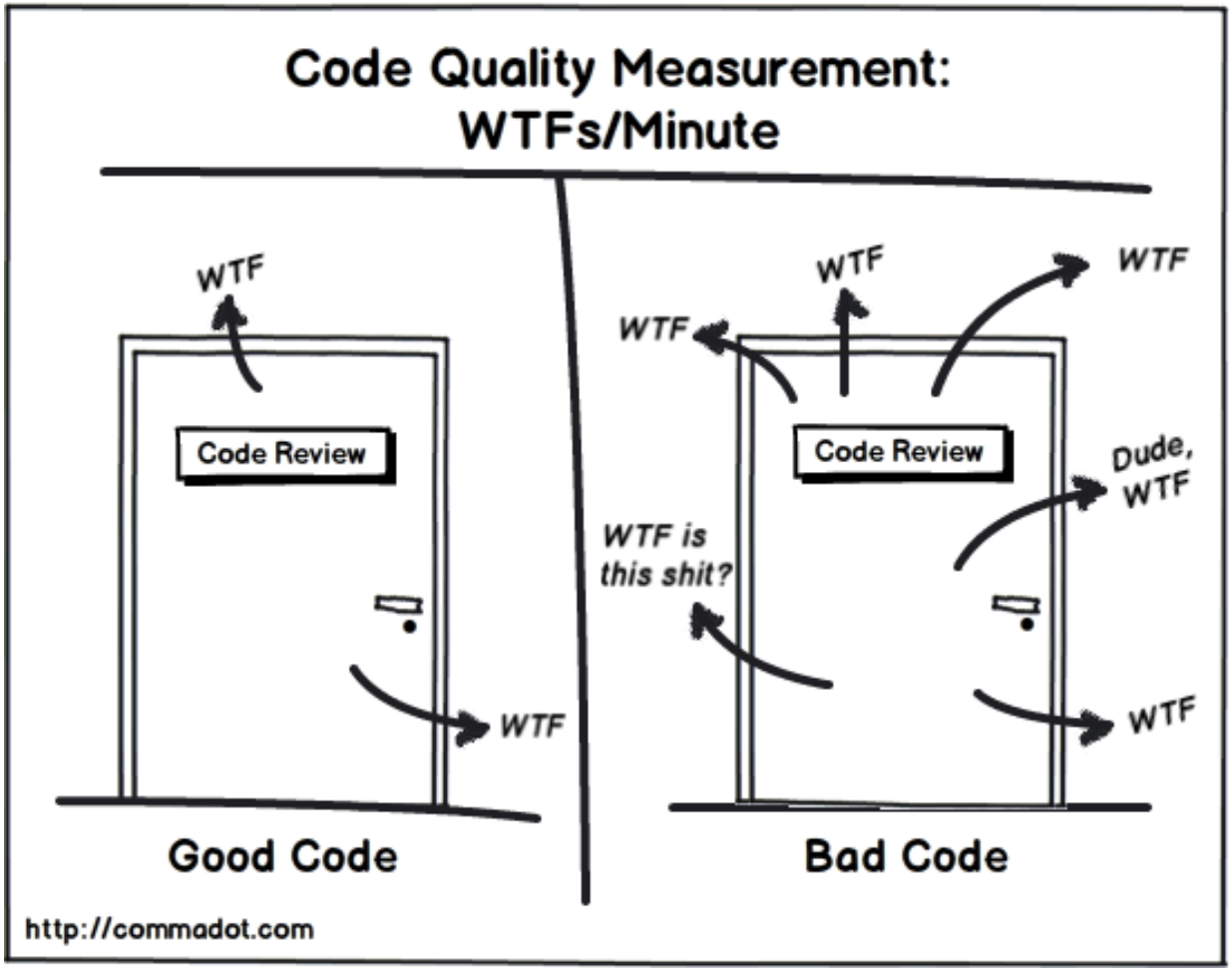
Good Code from a Developer's point of view: Understandable

bad code
will cost!



Source: http://blog.codinghorror.com/when-understanding-means-rewriting/

How to Measure the Quality of Code?



SW Quality Metrics: Cyclomatic Complexity

Construct	Effect on CC	Reasoning
if	+1	An if statement is a single decision.
else	0	The else statement does not cause a new decision. The decision is at the if.
elif	+1	The elif statement adds another decision.
for	+1	There is a decision at the start of the loop.

Source: https://radon.readthedocs.io/en/latest/intro.html

SW Quality Metrics: Cyclomatic Complexity

- CC= 1 + 1 + 1 = 3

```
x = 45
def is_it_44(x):
    if x < 44:
        return 0
    else:
        if x == 44:
            return 1
        else:
            return 2
print is_it_44(x)
```

SW Quality Metrics: Cyclomatic Complexity

```
x = 45
def is_it_44(x):
    if x < 44:
        return 0
    else:
        if x == 44:
            return 1
        else:
            return 2
print is_it_44(x)
```

anthony@hibernia:~$ sudo pip install radon

anthony@hibernia:~$ radon cc my_test.py –s

F 2:0 is_it_44 - A (3)

Function Rank CC score

SW Quality Metrics: Cyclomatic Complexity

CC score	Rank	Risk
1 - 5	A	low - simple block
6 - 10	B	low - well structured and stable block
11 - 20	C	moderate - slightly complex block
21 - 30	D	more than moderate - more complex block
31 - 40	E	high - complex block, alarming
41+	F	very high - error-prone, unstable block

SW Quality Metrics: Halstead Metrics

- η1 = the number of distinct operators
- η2 = the number of distinct operands
- N1 = the total number of operators
- N2 = the total number of operands

```
def is_it_44(x):
    if x < 44:
        return 0
    else:
        if x == 44:
            return 1
        else:
            return 2
```

η1 = 6 (is_it_44, if, else, ==, <, return)
η2 = 5 (x, 44, 0, 1, 2)
N1 = 10 (all operators)
N2 = 8

SW Quality Metrics: Halstead Metrics

- Program vocabulary: $\eta=\eta1+\eta2$
- Program length: $N=N1+N2$
- Volume: $V=N\log2\eta$
- Difficulty: $D=\eta1/2 * N2/\eta2$
- Effort: $E=D*V$
- Time required to program: $T=E/18$ seconds
- Number of delivered bugs: $B=V/3000$.

SW Quality Metrics: Raw Metrics

- LOC: The total number of lines of code
- LLOC: The number of logical lines of code (exactly one statement)
- SLOC: The number of source lines of code
- Comments: The number of comment lines
- Multi: The number of lines which represent multi-line strings.
- Blanks: The number of blank lines

SW Quality Metrics: Raw Metrics

```
x = 45
def is_it_44(x):
    if x < 44:
        return 0
    else:
        if x == 44:
            return 1
        else:
            return 2
print is_it_44(x)
```

```
anthony@hibernia:~$ radon raw my_test.py
```

my_test.py

- LOC: 10
- LLOC: 10
- SLOC: 10
- Comments: 0
- Single comments: 0
- Multi: 0
- Blank: 0

SW Quality Metrics: Maintainability Index

- $MI = 171 - 5.2 \ln V - 0.23 CC - 16.2 \ln SLOC$

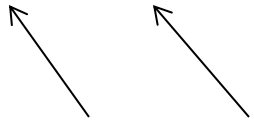
- Index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability.

SW Quality Metrics: Maintainability Index

```
x = 45
def is_it_44(x):
    if x < 44:
        return 0
    else:
        if x == 44:
            return 1
        else:
            return 2
print is_it_44(x)
```

```
anthony@hibernia:~$ radon mi
my_test.py –s

 my_test.py - A (70.23)
```



SW Quality Metrics: Maintainability Index

MI score	Rank	Maintainability
100 - 20	A	Very high
19 - 10	B	Medium
9 - 0	C	Extremely low

How to Improve Quality of Written Code? Pylint

- Static code analysis
- Coding Standard
- Error detection
- Refactoring help
- Fully customizable
- Editor/IDE integration

- https://www.pylint.org/

How to Improve Quality of Written Code? Pylint

```
def example(foo, a, blah):
  qux = 123
  if foo > a:
    return foo
  else:
    return datetime.now()
```

```
anthony@hibernia:~$ pylint my_test3.py
************* Module my_test3
W:  2, 0: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
W:  3, 0: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
W:  4, 0: Bad indentation. Found 4 spaces, expected 8 (bad-indentation)
W:  5, 0: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
W:  6, 0: Bad indentation. Found 4 spaces, expected 8 (bad-indentation)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Black listed name "foo" (blacklisted-name)
C:  1, 0: Invalid argument name "a" (invalid-name)
C:  1, 0: Missing function docstring (missing-docstring)
E:  6,11: Undefined variable 'datetime' (undefined-variable)
W:  1,20: Unused argument 'blah' (unused-argument)
W:  2, 2: Unused variable 'qux' (unused-variable)

Global evaluation
----------------
Your code has been rated at -16.67/10
```

How to Improve Quality of Written Code? Peer Code Review

- Peer code Review:
  - Decreases number of bugs
  - Enforces writting neat code
  - Speeds up learning
  - Enhances the team culture
- Rules:
  - All changesets get code reviewed
  - Automate everything you can
  - Everyone makes code reviews / everybody gets code reviewed

How to Improve Quality of Written Code? Peer Code Review

- Pull requests inline comments (Github / Bitbucket / ...)
- Gerrit
- Crucible
- Phabricator
- many more...

Conclusion

- Code needs to be good in the sense of being easy to understand
- Code Metrics help to understand how easy/complex/difficult is your code
- Peer review helps to make the code both better (function) and easier to understand

PERFORMANCE PROFILING

Our Master's Voice

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil***. Yet we should not pass up our opportunities in that critical 3%"

Donald Knuth (1974)

Performance

- Following the ***premature optimization*** argument, it is often thought that performance has to be considered last, using profiling tools to find bottlenecks.

- But certain elements are (very) difficult to refactor a posteriori:
  - ***Concurrency***: protection of shared resources, schemas and structure of data, etc.
  - ***Distribution***: partitioning, interactions, communication, etc.

Design for Performance

- Need to make some important decisions during the ***design*** process:
  - Concurrency (e.g., DBMS)
  - Distribution (e.g., communication framework)
  - General ideas for the interactions between components
  - Use of some (simple) queuing models to identify likely bottlenecks

- Performance assessment during the ***development*** process to check the decisions scale/perform well:
  - Profiling, benchmarking, load testing

Profiling

- Profiling = to identify where the code is slow
- The bottleneck could be:
  - CPU
  - Memory
  - Network
  - Disk I/O
  - DB
- Finding the solution to why a program is slow depends on many factors – no generic solution

Profiling Tools

- Manual instrumentation
  - E.g., prints, logs, all sort of timing functions
- Static instrumentation (by the compiler)
  - E.g., `gprof`, `line_profiler`, `mem_profiler`
- Dynamic instrumentation (look at the binary)
  - E.g., `callgrind`, `cachegrind`, etc. Visual VM for Java, Jconsole, Javassist
- Performance counters
  - E.g., `perf`, `time`
- Heap profiling (to figure out what's happening with the allocation)
  - E.g., massif, google-perftools, heapy

Load Testing

- Generate load to test the ability of a system to handle queries/transactions
- Load
  - Number of (simultaneous) users
  - Size of the infrastructure
  - Number of queries per seconds
  - Amount of data.
- Metrics
  - Response time,
  - RAM utilisation,
  - CPU utilisation,
  - Throughput.

Benchmarking

- A benchmark is somehow similar to load testing: generate load to test a DB system
  - Validate assumptions on the system
  - Reproduce bad behaviors
  - Measurement of
  - Identify scalability bottlenecks
  - Plans for growth/capital allocation
  - Test different hardware, software or OS configurations
  - Check configurations (e.g., for new hardware)

STATIC INSTRUMENTATION

line_profiler (Python)

- Easier than the older cProfile module
- Non-obtrusive: simple decorator *@profile*
- Only simple outputs: hits, time, time per hit, %time

- Install:

```
anthony@hibernia$ sudo easy_install line_profiler
```

- Run:

```
anthony@hibernia$ kernprof -lv slow.py
```

  - –l : line by line
  - -v : output in the terminal

Example for line_profiler (slow.py)

```
from time import sleep
@profile
def slow(x):
        sleep(2);return x
@profile
def function(x):
        a = []
        for i in xrange(x):
                a.append(i)
        b = slow(a)
def main():
        print function(100000)
```

Output of `line_profiler`

- Details of the output:
  - Function: Displays the name of the function that is profiled and its line number.
  - Line#: The line number of the code in the respective file. ☐
  - Hits: The number of times the code in the corresponding line was executed.
  - Time: Total amount of time spent in executing the line in 'Timer unit' (i.e., 1e-06s here).
  - Per hit: The average amount of time spent in executing the line once in 'Timer unit'.
  - % time: The percentage of time spent on a line with respect to the total amount of recorded time spent in the function.
  - Line content: It displays the actual source code.

memory_profiler (Python)

- Similar to `line_profiler` but for RAM instead of CPU

- Install:

```
anthony@hibernia$ sudo easy_install
memory_profiler
```

- Run:

```
anthony@hibernia$ python -m memory_profiler
copy.py
```

Example for memory_profiler (copy.py)

```
import copy

@profile
def function():
    x = range(1000000)
    y = copy.deepcopy(x)
    del x
    return y

if __name__=="__main__":
    function()
```

Output of `memory_profiler`

- Details of the output:
  - Line#: The line number of the code in the respective file. ☐
  - Mem Usage: memory usage of the Python interpreter after that line has been executed.
  - Increment: difference in memory of the current line with respect to the last one.
  - Line content: It displays the actual source code.

mprof (Python)

- memory_profiler has a built-in ability to plot the output
  - Philippe Gervais (INRIA, Google) implemented a nice feature

- Run:

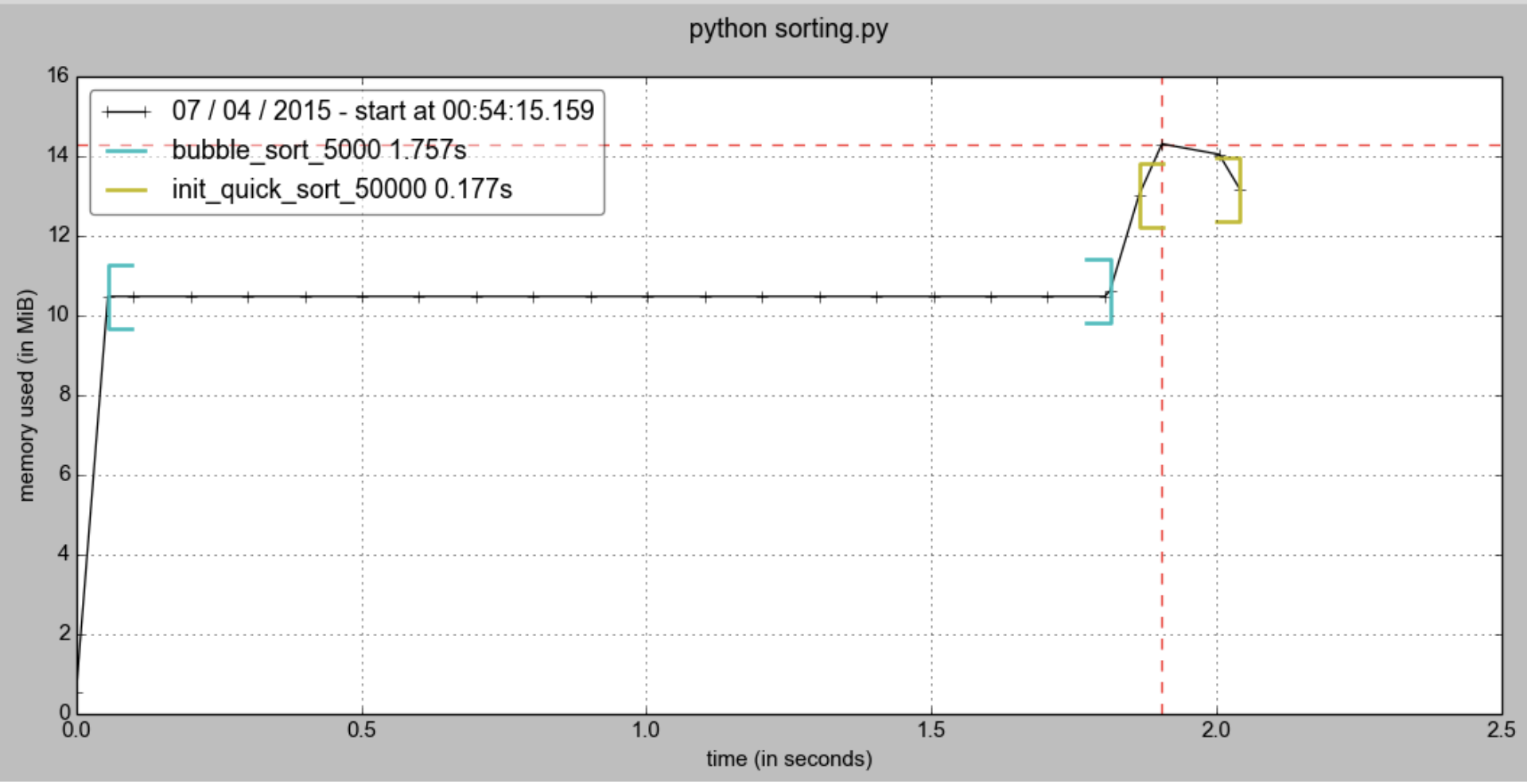
```
anthony@hibernia$ mprof run sorting.py

anthony@hibernia$ mprof plot
```

Example for mprof (sorting.py)

```
import random
import time
@profile
def bubble_sort_5000(items):
    […]
@profile
def init_quick_sort_50000(items):
    […]
def main():
        items = [random.randint(0, 10000) for c in range(5000)]
        bubble_sort_5000(items)
        items = [random.randint(0, 10000) for c in range(50000)]
        init_quick_sort_50000(items)
```

Output of `mprof`



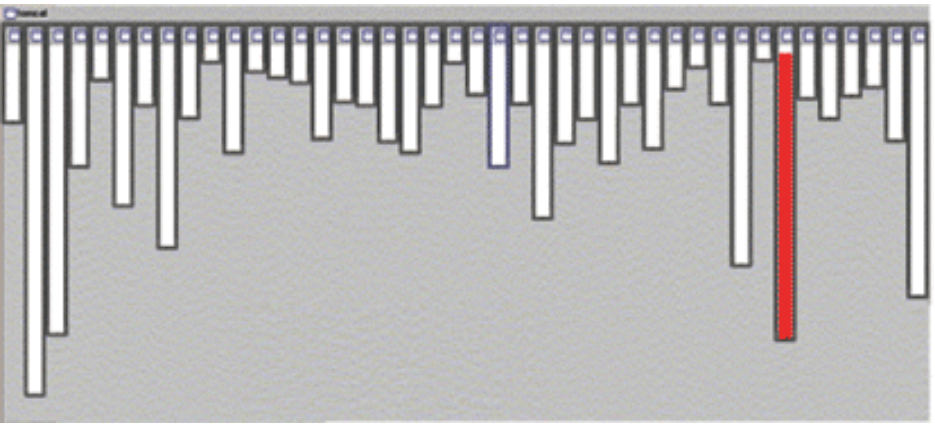
DYNAMIC INSTRUMENTATION

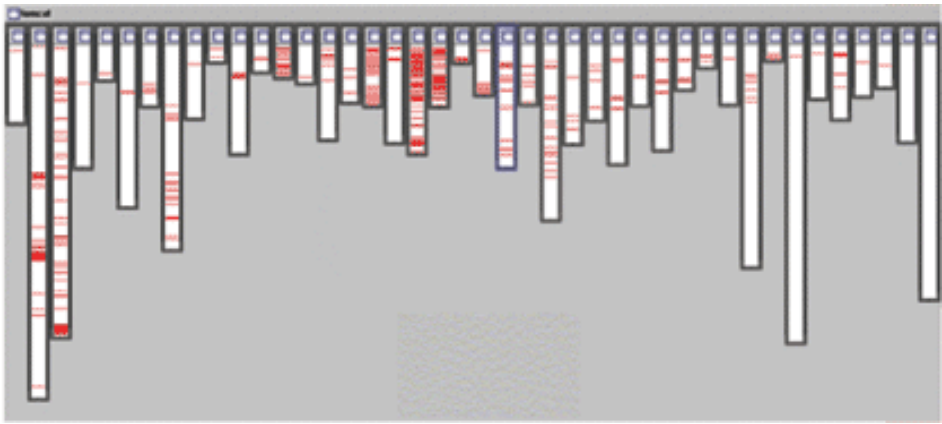
Languages

- Procedural (a.k.a., imperative)
  - Explicit list of steps to follow
- Functional
  - Everything is a function (Lisp, Haskell)
- Declarative
  - Relationships between terms (Prolog)
- Object-oriented
  - Objects and communications between them
  - Encapsulation, inheritance, polymorphism
  - Modular vs. tangled code

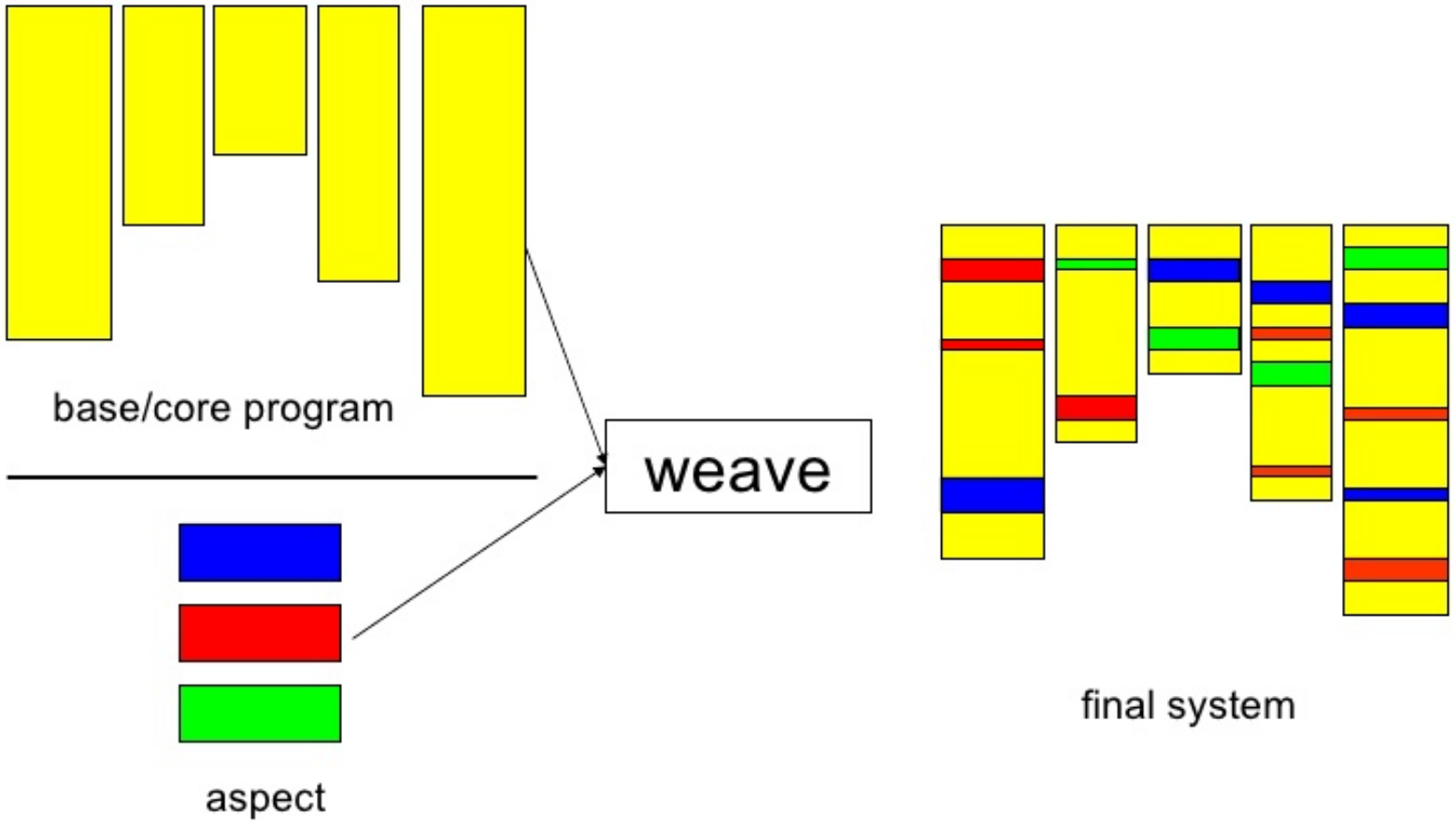
OO & Tangled code

- Apache Tomcat: XML parsing and logging





Basic Concepts of AOP



Javassist

http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

- Bytecode manipulation library

Example (HelloWorld.java)

```
public class HelloWorld {
    public void do1() {
        System.out.println("Hello World!");
    }
    public void do2(String tosay){
        System.out.println(tosay);
    }
}
```

Basic Instrumentation with Javassist (Inst1.java)

```
import javassist.*;
public class Inst1 {
    public static void main(String[] args) throws Exception {
        ClassPool pool = ClassPool.getDefault();
        CtClass hw_ctc = pool.get("HelloWorld");
        CtMethod hw_ctm = hw_ctc.getDeclaredMethod("do1");
        hw_ctm.insertBefore("System.out.println(\"HelloWorld.do1()
Start\");");
        hw_ctm.insertAfter("System.out.println(\"HelloWorld.do1() End\");");
        Class hw_class = hw_ctc.toClass();
        HelloWorld hw = (HelloWorld)hw_class.newInstance();
        hw.do1();
        hw.do2("Goodbye world");
    }
}
```

Basic Instrumentation with Javassist

- Compile:

```
javac -cp javassist-3.18.0-GA/javassist.jar:. Inst1.java HelloWorld.java
```

- Run:

```
java -cp javassist-3.18.0-GA/javassist.jar:. Inst1
```

Important elements

- ClassPool: A container of CtClass objects
- CtClass: An instance of CtClass represents a class. It is obtained from ClassPool.
- CtMethod: An instance of CtMethod represents a method.
- insertBefore(): Inserts bytecode at the beginning of the body of a method.
- insertAfter(): Inserts bytecode just before every return insturction

More Instrumentation with Javassist (Inst2.java)

```
import javassist.*;
public class Inst2 {
    public static void main(String[] args) throws Exception {
        ClassPool pool = ClassPool.getDefault();
        CtClass hw_ctc = pool.get("HelloWorld");
        CtMethod[] hw_ctms = hw_ctc.getDeclaredMethods();
        for(int i=0; i<hw_ctms.length;i++){
            hw_ctms[i].insertBefore("System.out.println(\"HelloWorld." +
hw_ctms[i].getName() + " Start\");");
            hw_ctms[i].insertAfter("System.out.println(\"HelloWorld." +
hw_ctms[i].getName() +" End\");");
        }
        Class hw_class = hw_ctc.toClass();
        HelloWorld hw = (HelloWorld)hw_class.newInstance();
        hw.do1();
        hw.do2("Goodbye world");
    }
}
```

More Instrumentation with Javassist

- Compile:

```
javac -cp javassist-3.18.0-GA/javassist.jar:.
Inst2.java HelloWorld.java
```

- Run:

```
java -cp javassist-3.18.0-GA/javassist.jar:.
Inst2
```

Example of Inefficient Code (StringBuilder.java)

```
public class StringBuilder
{
    private String buildString(int length) {
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        return result;
    }
    public static void main(String[] argv) {
        StringBuilder inst = new StringBuilder();
        for (int i = 0; i < argv.length; i++) {
            String result = inst.buildString(Integer.parseInt(argv[i]));
            System.out.println("Constructed string of length
“+result.length());
        }
    }
}
```

Example of Inefficient Code

- Example of what not to do!
  - Strings are immutable: new String created each time in the loop
- Basic, manual instrumentation (add logging or System.out.println()):

```
long start = System.currentTimeMillis();
String result = "";
[loop]
System.out.println("Call to buildString took " +
                   (System.currentTimeMillis()-start) + "
ms.");
```

Can we do that with Javassist?

- Yes… in principle but
- No…
  - added code could not reference local variables defined elsewhere in the method – and variable `start` is needed at the beginning and at the end…
- Solutions:
  - New global variable? Probably not a good option
  - Interceptor method! Change method name and add a new method (single block) calling the old method

Dynamic Instrumentation for Profiling (JavassistTiming.java)

```
public class JavassistTiming
{
    public static void main(String[] argv) {
      CtClass clas =
ClassPool.getDefault().get(argv[0]);
      // add timing interceptor to the class
      addTiming(clas, argv[1]);
      clas.writeFile();
      System.out.println("Added timing to method "
+ argv[0] + "." + argv[1]);
}
```

Dynamic Instrumentation for Profiling (cont'd)

```
private static void (CtClass clas, String mname) {
        [rename old method and create method as interceptor]
        String type = mold.getReturnType().getName();
        StringBuffer body = new StringBuffer();
        body.append("{\nlong start = System.currentTimeMillis();\n");
        body.append(type + " result = ”);
        body.append(nname + "($$);\n");
        body.append("System.out.println(\"Call to method " + mname +
            " took \" +\n (System.currentTimeMillis()-start) + " +
            "\" ms.\");\n");
        body.append("return result;\n");
        body.append("}");
        mnew.setBody(body.toString());
        clas.addMethod(mnew);
}
```

PERFORMANCE COUNTERS

Basic Idea

- The system keeps a list of what and where interesting hardware related events (cycle, branch miss, power consumption, IPC, etc.) happen
  - Keep a counter
  - When counter > threshold
    - Interrupt raised
- Tools like `perf` turn the data into interesting reports

For Linux: `perf`

- New version of other older tools: oprofile, perfmon
- Can monitor software and hardware events

```
anthony@hibernia:~$ perf list
```

- Define your own counters
- Have to install `linux-tools` (with the same version as your kernel)

```
anthony@hibernia:~$ perf help
```

For Linux: perf

- CPU counter stats for specific command:

```
anthony@hibernia:~$ perf stat [-d] cmd
```

- Various runs:

```
anthony@hibernia:~$ perf stat –r number cmd
```

- CPU counter statistics for the entire system, for 5 seconds:

```
anthony@hibernia:~$ perf stat –a sleep 5
```

- Various basic CPU statistics, system wide, for 10 seconds:

```
anthony@hibernia:~$ perf stat -e
cycles,instructions,cache-references,cache-
misses,bus-cycles -a sleep 10
```

Demo (loop interchange)

```
for k in range (0,100):
      for j in range (0,100):
            for i in range (0,5000):
                  a[i][j] = 2* a[i][j]
```

```
anthony@hibernia:~$ perf stat -e cycles -e
instructions -e L1-dcache-loads -e L1-dcache-
load-misses python loop1.py

229,745,736 L1-dcache-load-misses     #    1.55%
of all L1-dcache hits
```

Demo (loop interchange)

```
for k in range (0,100):
      for i in range (0,5000):
            for j in range (0,100):
                  a[i][j] = 2* a[i][j]
```

```
anthony@hibernia:~$ perf stat -e cycles -e
instructions -e L1-dcache-loads -e L1-dcache-
load-misses python loop2.py
```

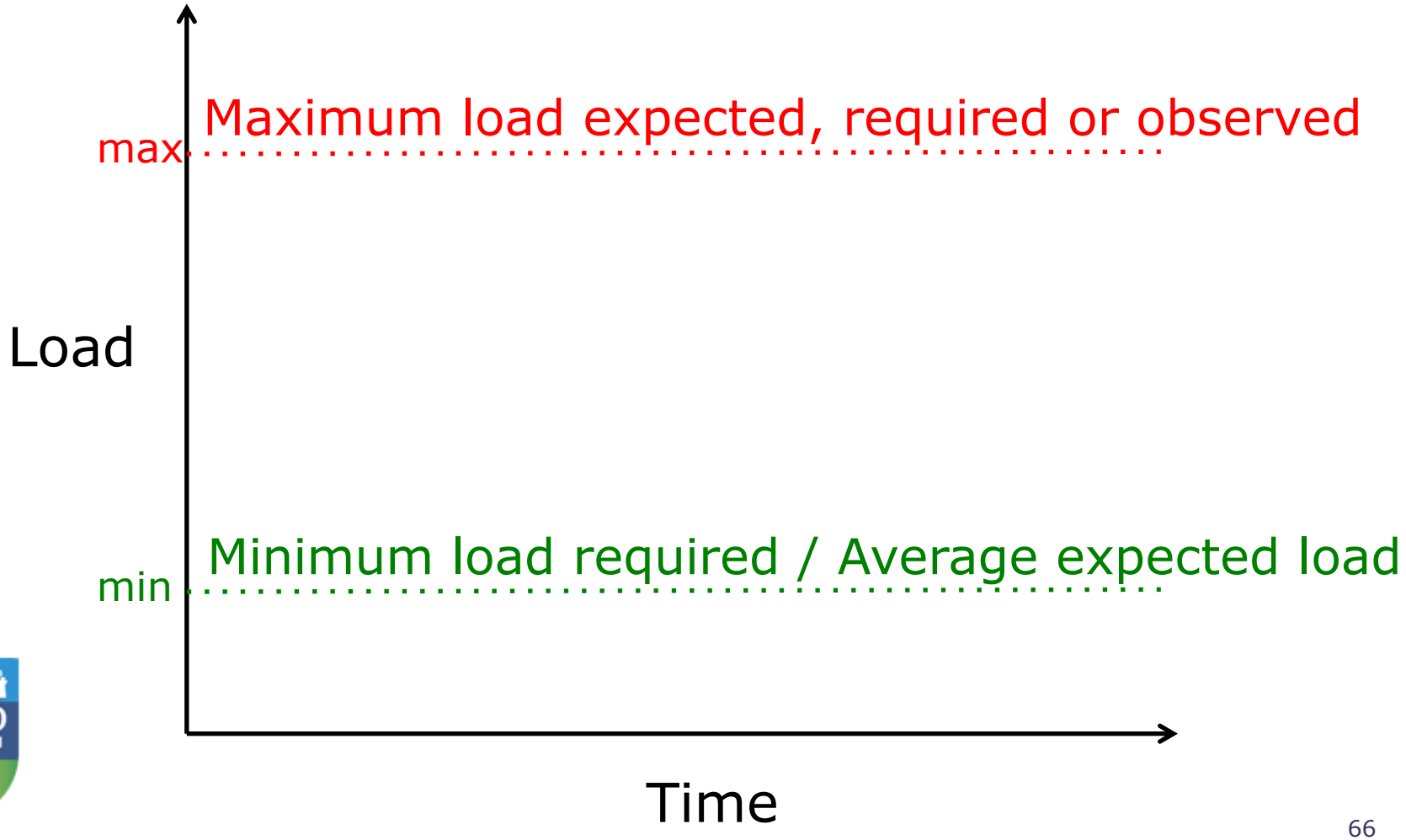
```
38,297,288 L1-dcache-load-misses     #    0.26%
of all L1-dcache hits
```

LOAD TESTING

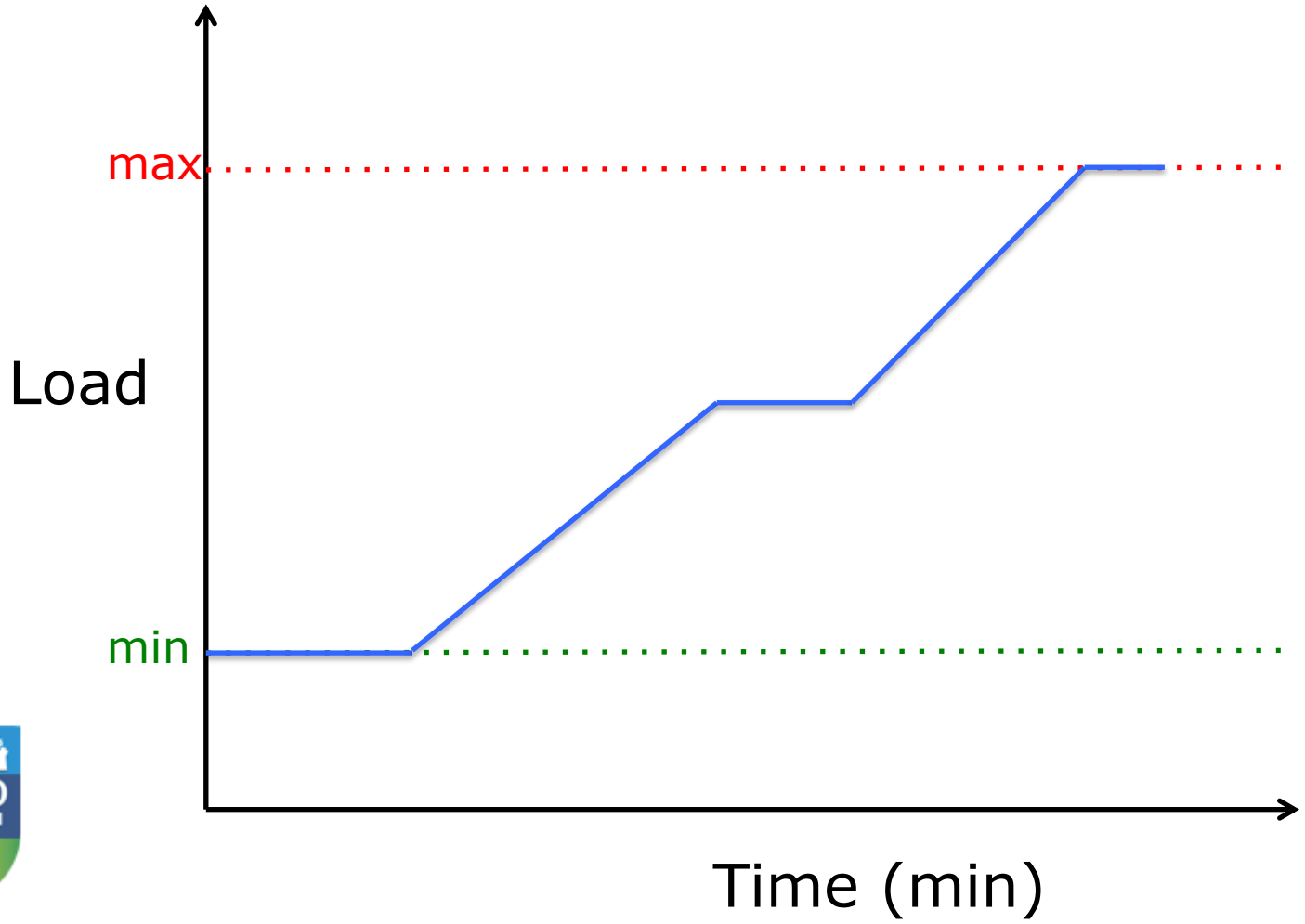
Concepts

- Load
  - Number of (simultaneous) users
  - Size of the infrastructure
  - Number of queries per seconds
  - Amount of data.
- Metrics
  - Response time,
  - RAM utilisation,
  - CPU utilisation,
  - Throughput.

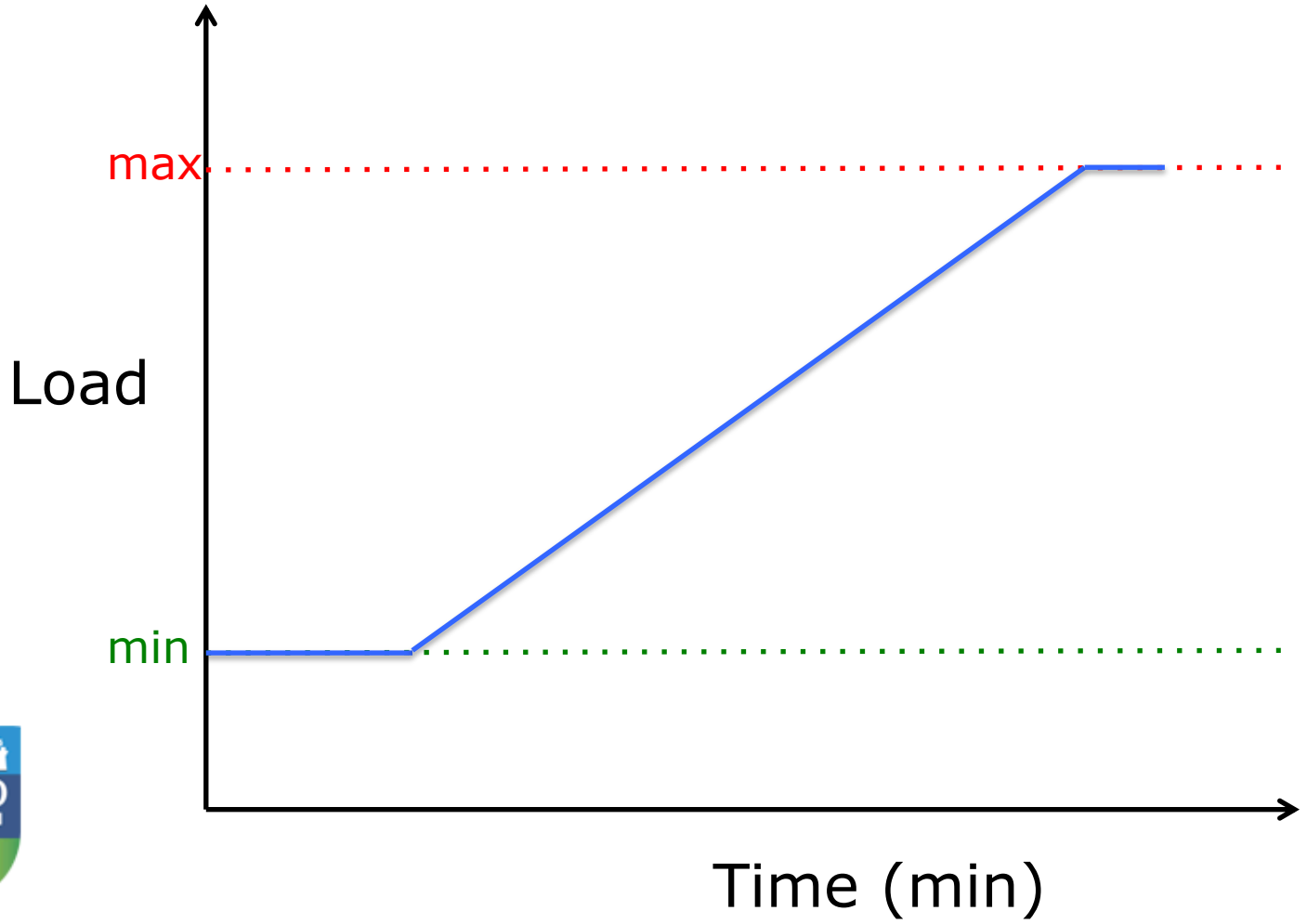
Concepts


Maximum load expected, required or observed
max
Load
Minimum load required / Average expected load
min
Time

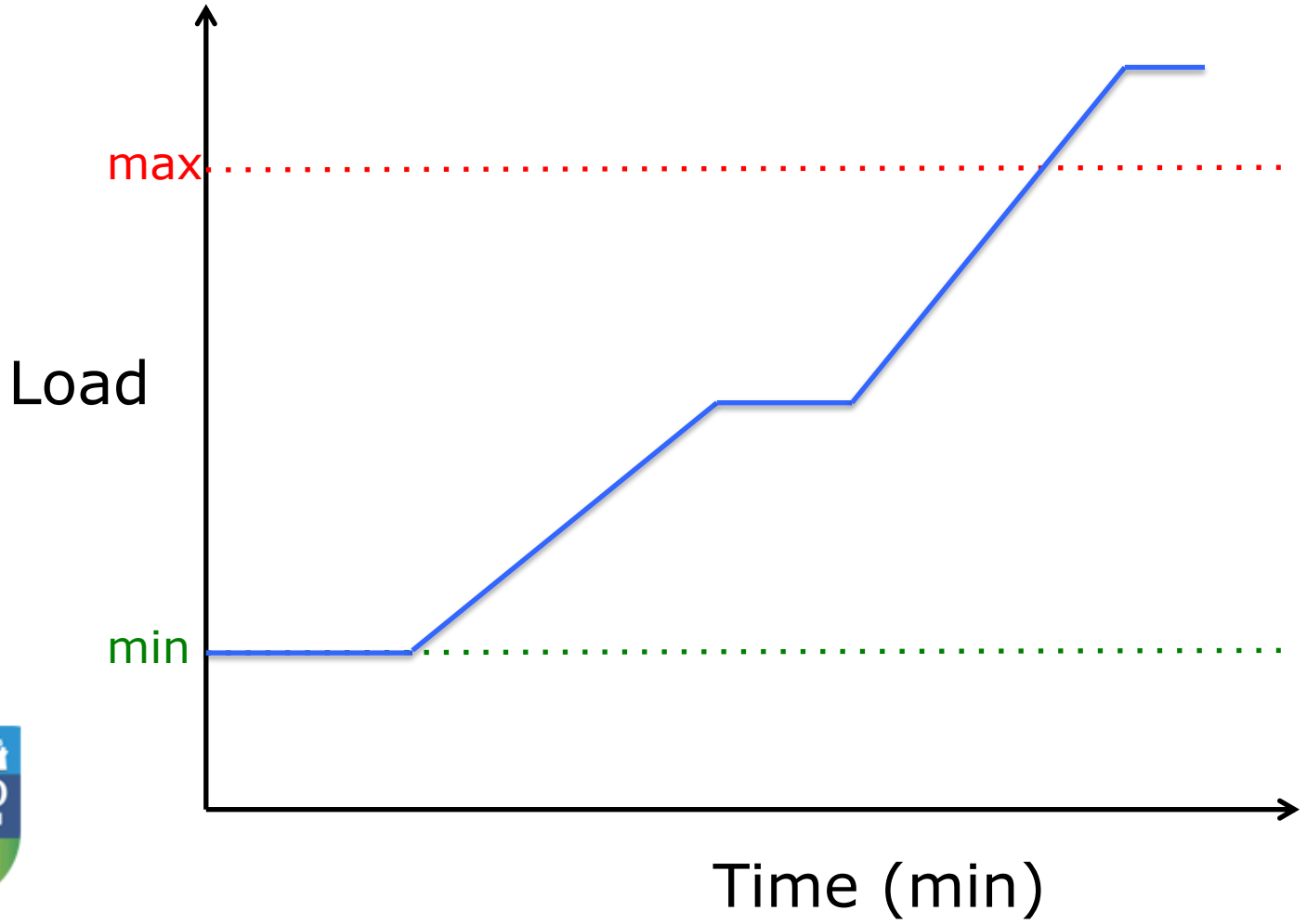
Load Testing



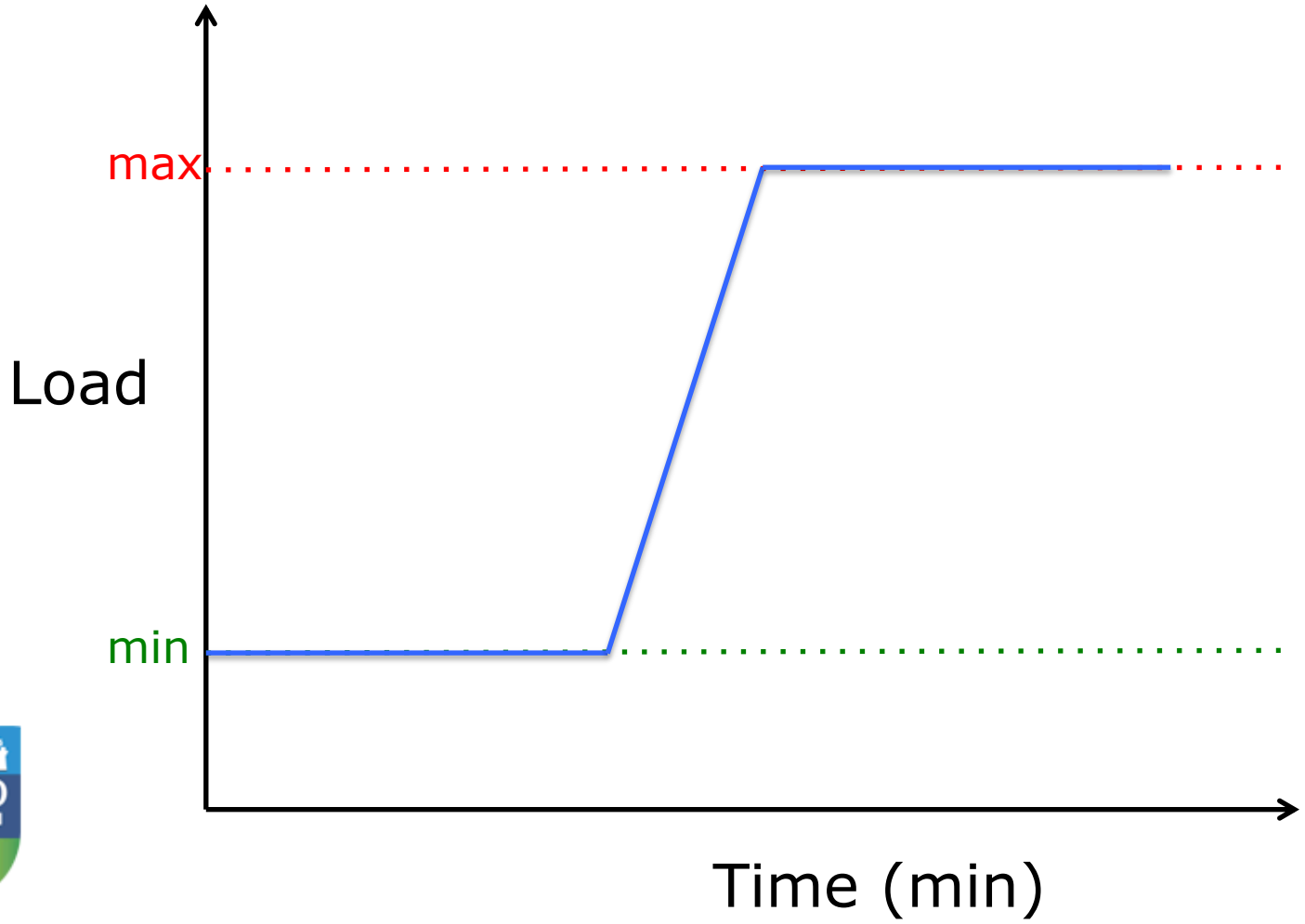
Load Testing



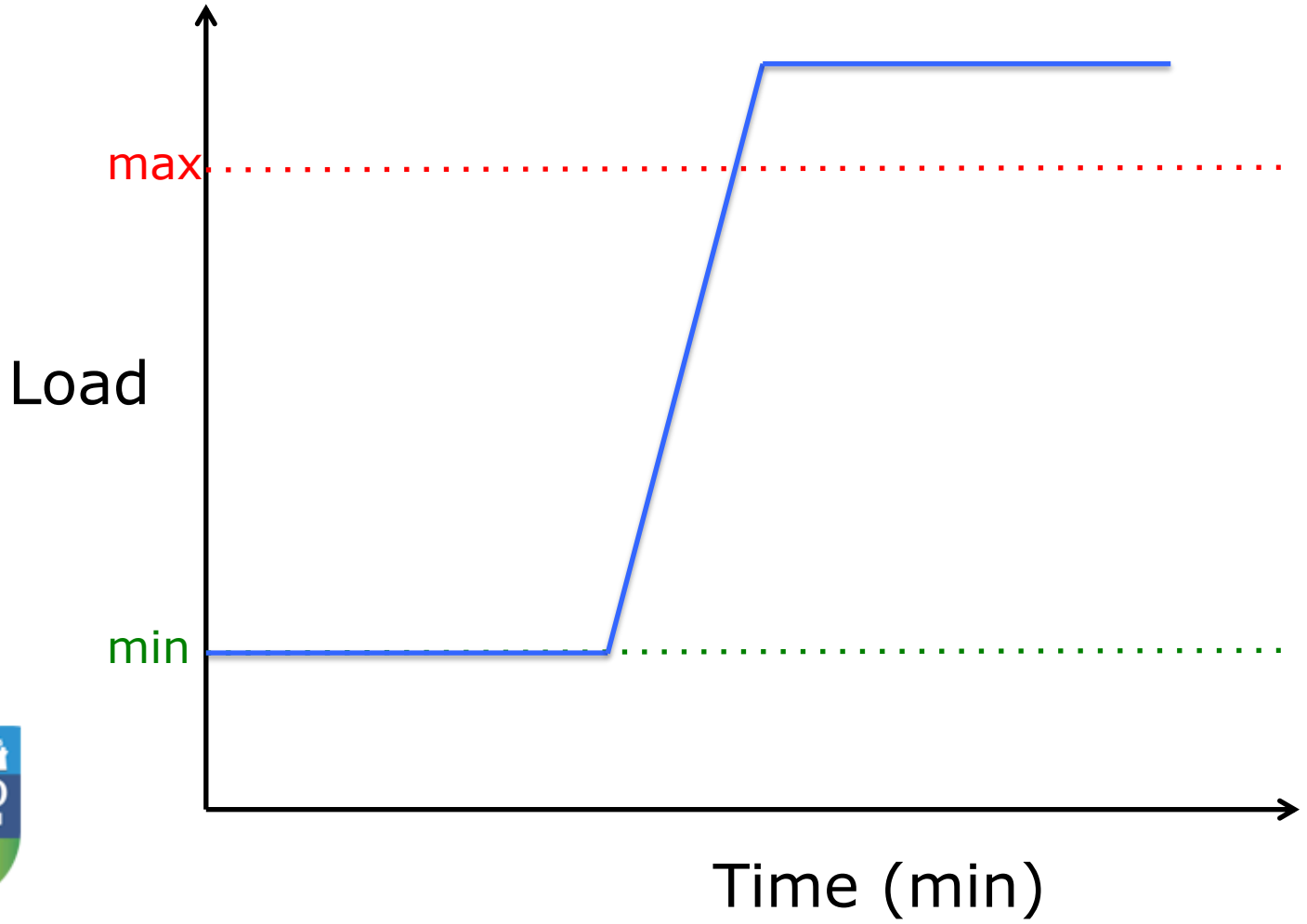
Load Testing



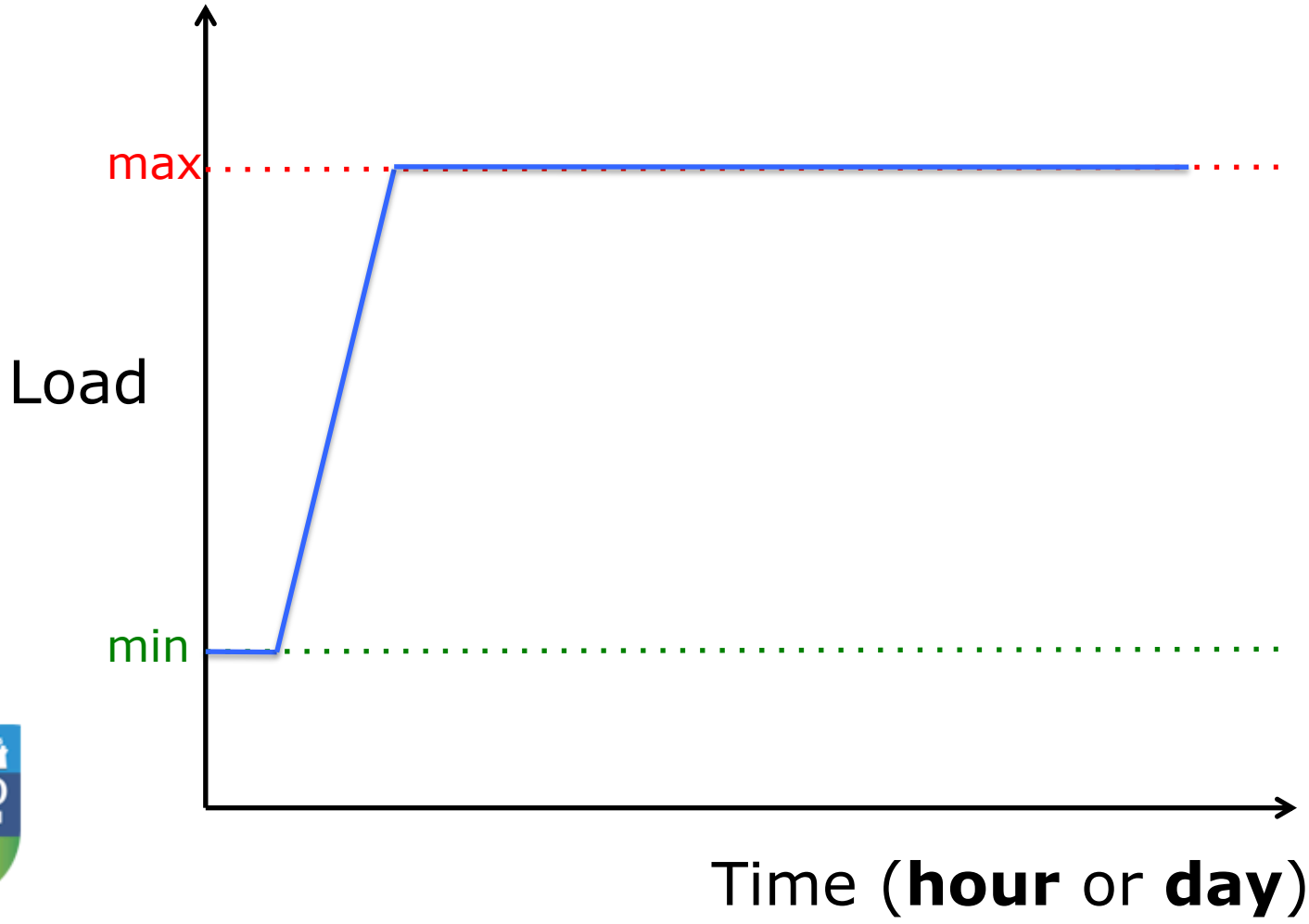
Stress Testing



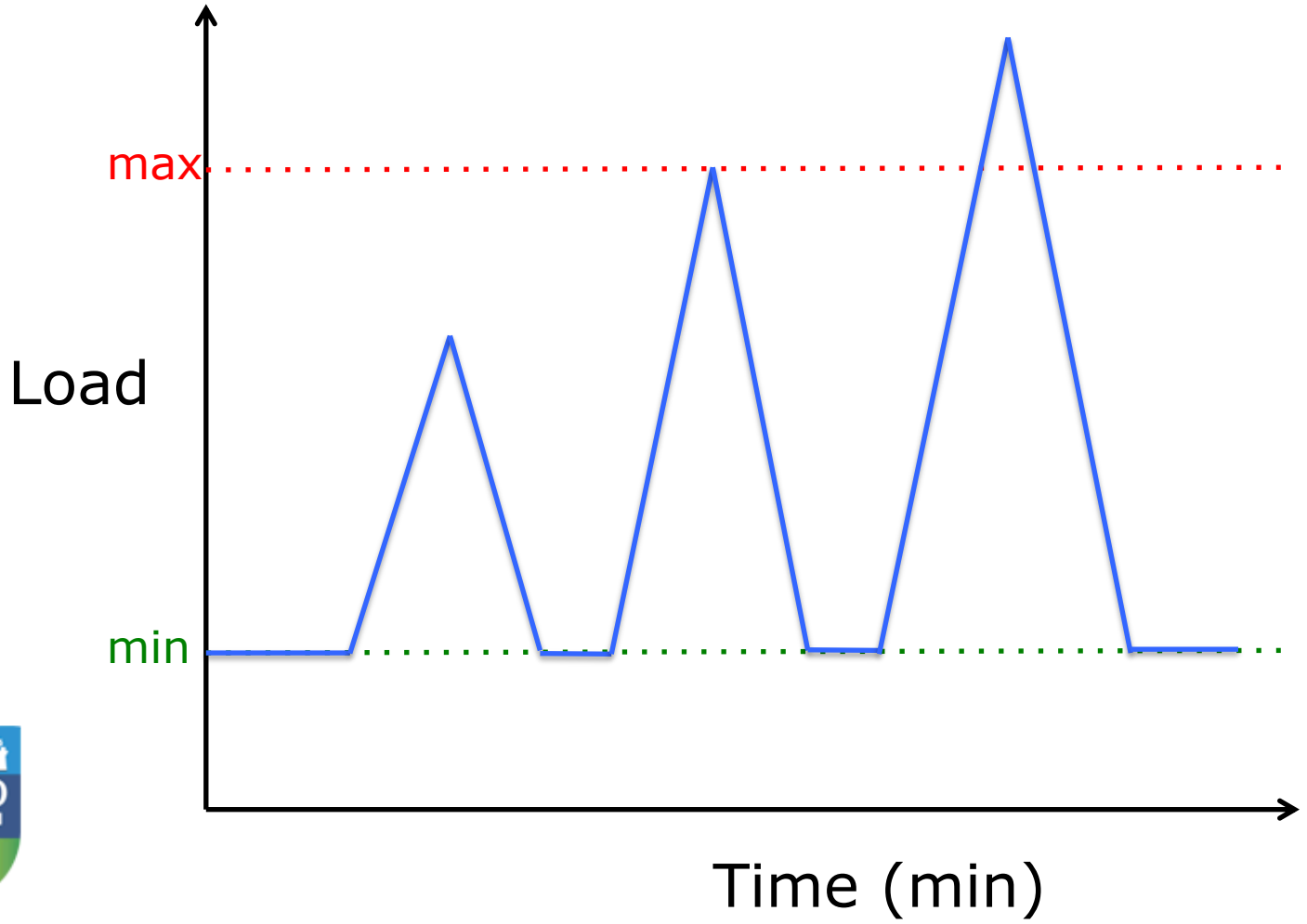
Stress Testing



Soak Testing



Spike Testing

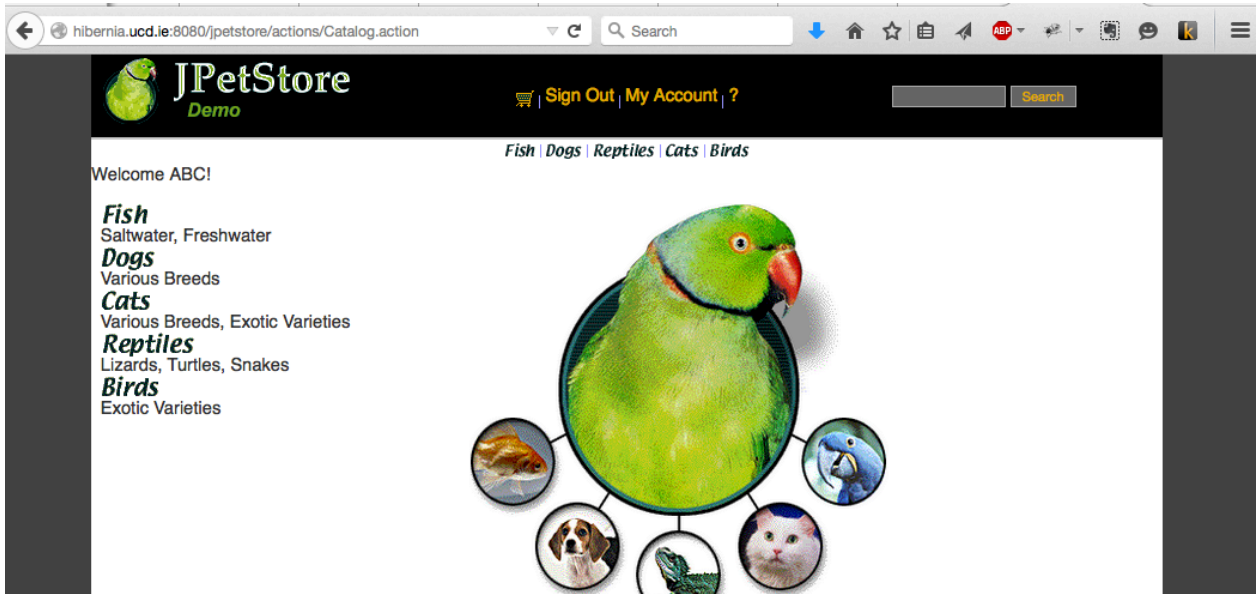


Performance Testing with Apache JMeter

- JMeter is one of numerous tools for performance testing:
  - (HP) LoadRunner, (IBM) RPT, The Grinder, etc.
- Can do distributed testing (using large scale systems for testing)
- GUI (script creation and debugging) and scripts (for proper load testing)

Test Application JPetStore

- A simple web application
  - Running on Tomcat; MySQL/HSQL DB; JSP; Java Beans
- Various transactions



Scenario

- Launch Jmeter
- Create new Web Test Plan: e.g., Google
  - Set Thread to various numbers
  - Add HTTP Requests
  - Add Listeners: Result Trees, Response Time, Results in Table
- Create a new Web Test Plan: JPetStore
  - Set threads to 100 for 10 seconds
  - Add HTTP Requests
  - Add Listeners: Result Trees, Response Time, Results in Table
  - Observe the server's load
  - Change threads to 1000
  - Observe the server's load

Conclusion

- Profiling is key to optimisation
- It allows to support intuitions
- Practice is very important (in SE in general but especially in performance engineering)

- Large-scale testing techniques (load, stress, soak) allow to
  - Detect performance defects
  - Increase the code coverage

CODE TESTING

What is Unit Test?

- Unit Tests are software programs written to exercise other software programs (called Code Under Test) with specific preconditions and **verify the expected behaviours** of the CUT.
- Unit tests are usually written in the same programming language as their code under test.
- Each unit test should be small and test only **limited piece of code functionality**. Test cases are often grouped into Test Groups or Test Suites. There are many open source unit test frameworks. The popular ones usually follows an xUnit pattern invented by Kent Beck E.g. JUnit for Java, CppUTest for C/C++.
- Unit tests should also run very fast. Usually we expect to run hundreds of unit test cases within a few seconds.

Why Should We Test?

- Testing allows one to **ensure** that **changes** to the code **did not break existing functionality**. Providing a higher level of confidence in the future when refactoring and generally making maintenance less scary.
- Testing forces one to think about the code under unusual conditions, possibly revealing logical errors
- Good testing requires modular, decoupled code, which is a hallmark of good system design
- Testing helps you to realise when to stop coding. Your tests give you **confidence** that you've done enough for now and can stop tweaking and move on to the next thing.

Who Writes the Tests?

- Depends on what sort of tests:
  - Unit (individual functions or classes) -> Software Engineers [We focus on this here]
  - Integration (modules or logical subsystems) -> Software Engineers
  - Functional (checking a program against design specifications) ->QA Engineers
  - System (checking a program against system requirements) -> QA Engineers

Unittest

- The unittest framework comes with Python.
- Creating a Python unit test is easy.
- import the unittest module, or TestCase from the unittest module.
- Inherit from the unittest.TestCase class. The derived class is a “test group”
- Inside the test, there are potentially several assertion method calls that are used to check the expected result, e.g. assertEqual, assertTrue, assertFalse.

Example: unittest

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello',
'world'])

unittest.main()
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Pytest

- Pytest is easier to use and more “Python-ic”
- Install:
  - pip install -U pytest # or
  - easy_install -U pytest
- Content of test_1.py

```
def test_the_answer():
    assert 19 + 23 == 42
```

Pytest: Features

- Simple assertions
- Dependency injection
- Parametrized tests
- Test discovery
- Test runner
- Distributed testing
- Plugin support (e.g. PEP8, coverage or pytest-django)
- Conditonal skipping
- Easy to get started
- ...

Pytest: Test Discovery

- Good practice: Place tests in a tests/ directory
- Naming
  - Module naming: test_*.py or *_test.py
  - Function naming: test_* or *_test
  - Class naming: Test* or *Test
  - Customizable!
- Run py.test command

No News is Good News

- If the test passes, it should just print OK (and perhaps some dots to show the progress). No other information.
- Rule of thumb:
  - No human intervention should be needed, either to get ready for the test, running the test cases, or checking the result.

Golden Rule of a Unit Test

- In general, this is a good rule for each unit test case:
  - Each unit test case should be very limited in scope.
- So that:
  - When the test fails, no debugging is needed to locate the problem.
  - Tests are stable because dependencies are simple.
  - Less duplication, easier to maintain.

Simple Example

```
import roman

def test_sanity():
    for i in range(1, 4000):
        assert roman.fromRoman(roman.toRoman(i))
                    == i
```

Example

- Let's assume you want to build a prime numbers generator.
- You could start with test, which would make the engineering more… systematic and less adhoc ☺

```
from prime import PrimeGenerator

def test_first_primes():
    pg = PrimeGenerator()
    assert pg.first_primes(6) == [2, 3, 5, 7, 11, 13]
```

- py.test test_primgen.py

Example

```
from prime import PrimeGenerator

def test_first_primes():
    pg = PrimeGenerator()

    assert pg.first_primes(6) == [2, 3, 5, 7, 11, 13]

def test_primality():
    pg = PrimeGenerator()

    known_primes = (17, 19, 23, 29, 31)
    known_nonprimes = (21, 27, 33, 49)

    for p in known_primes:
        assert pg.isprime(p)
    for np in known_nonprimes:
        assert not pg.isprime(np)
```

Conclusion

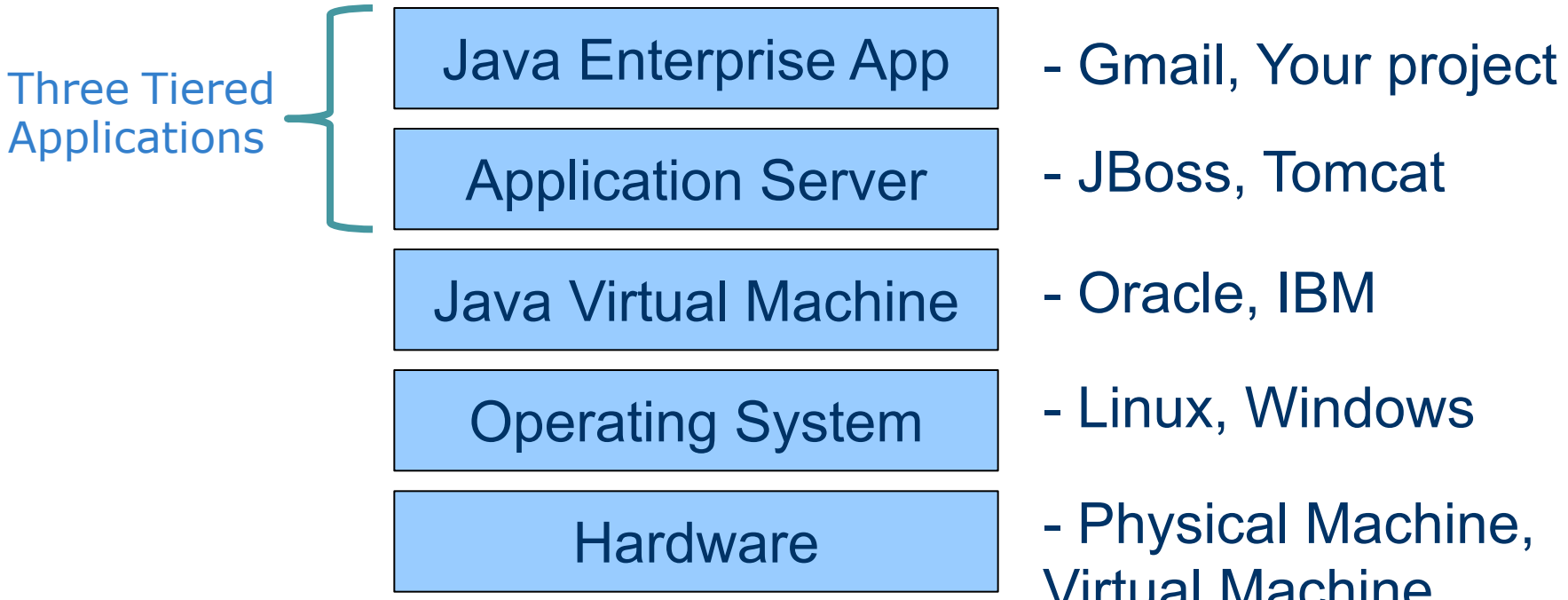
- Testing is the way of checking the code of functionally correct
- Unit tests allow are supposed to run fast and often – brings peace of mind to the developers

Designing ***Good*** Software

- Three elements can define what is a good software:
  - It does what it is supposed to do well: ***Function => Tests***
  - It is easy to adapt/modify: ***Maintenance => code metrics and peer review***
  - It works well given a particular system: ***Performance => Instrumentation and Load Testing***

WEB SYSTEM ARCHITECTURE

Java Enterprise Application Stack



Multi-tiered Applications