Software Engineering

Lecture 2

Software Testing

David Baelde

M1 MPRI @ ENS Paris-Saclay

September 25, 2020

Introduction

Program testing can be used to show the presence of bugs, but never to show their absence! – Dijkstra

Introduction

Program testing can be used to show the presence of bugs, but never to show their absence! – Dijkstra

Python development follows a practice that all semantic changes and additions to the language and stdlib are accompanied by appropriate unit tests. Unfortunately Python was in existence for a long time before the practice came into effect. This has left chunks of the stdlib untested which is not a desirable situation to be in. – Python Developer's Guide

Testing: why?

No matter your tools, debugging is hard!

Testing: why?

No matter your tools, debugging is hard!

We must test software in order to:

- Detect problems earlier.
- Facilitate identification of root cause.
- Prevent regressions.

Testing: what & how?

What?

- Explicit spec and/or “good behavior”.

Testing: what & how?

What?

- Explicit spec and/or "good behavior".

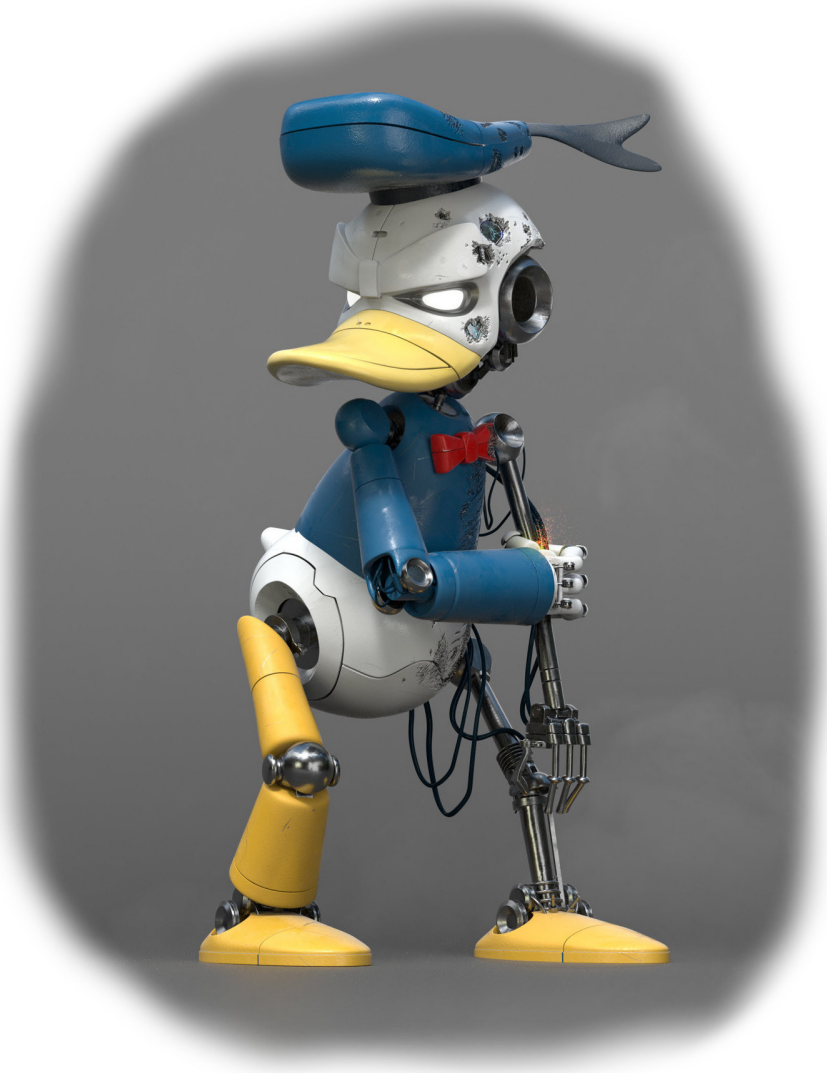
How?

- Unit testing on . . . basic units.
- Integration testing, complete system testing.
- Use tools that make it easy and systematic!

Automated testing

From 

Automated testing

From  to  ...

White box

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must probe "enough" behaviors.

Citeria: lines,

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must probe "enough" behaviors.

Citeria: lines, control flow, conditions,

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must probe "enough" behaviors.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must probe "enough" behaviors.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

Selecting test values,
based on code and spec: equivalence classes, boundaries...
manually (demo: `triangle.ml`)...

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must probe "enough" behaviors.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

Selecting test values,
based on code and spec: equivalence classes, boundaries...
manually (demo: `triangle.ml`)...or automatically.

Pex 1 (C#)

Generate "interesting" test values, by symbolic execution and constraint solving. Demo: ~~http://www.pexforfun.com~~

```
public class Point  {
  public readonly int X, Y;
  public Point(int x, int y) { X = x; Y = y; }
}
public class Program  {
  public static void Puzzle(Point p)
  {
    if (p.X * p.Y == 42)
      throw new Exception("Bug!");
  }
}
```

Pex 1 (C#)

Generate "interesting" test values, by symbolic execution and constraint solving. Demo: ~~http://www.pexforfun.com~~

```
public class Point  {
  public readonly int X, Y;
  public Point(int x, int y) { X = x; Y = y; }
}
public class Program  {
  public static void Puzzle(Point p)
  {
    if (p.X * p.Y == 42)
      throw new Exception("Bug!");
  }
}
```

Propose 3 inputs: `null`, `(0,0)` and `(3,14)`.

Pex 2 (C# + contracts)

```
public class Program  {
  public static string Puzzle(string value) {
    Contract.Requires(value != null);
    Contract.Ensures(Contract.Result<string>() != null);
    Contract.Ensures(
      char.IsUpper(Contract.Result<string>()[0]));
    return char.ToLower(value[0]) + value.Substring(1);
  }
}
```

Find inputs that trigger bugs...

Pex 2 (C# + contracts) fixed

```
public class Program  {
  public static string Puzzle(string value) {
    Contract.Requires(value != null);
    Contract.Requires(value=="" ||
                      char.IsLower(value[0]));
    Contract.Ensures(Contract.Result<string>() != null);
    Contract.Ensures(
      Contract.Result<string>()=="" ||
      char.IsUpper(Contract.Result<string>()[0]));
    if (value=="") return value;
    return char.ToUpper(value[0]) + value.Substring(1);
  }
}
```

Pex 3 (C# + contracts)

```
using System;

public class Program {
  static int Fib(int x) {
    return x == 0 ? 0 : x == 1 ? 1 :
           Fib(x - 1) + Fib(x - 2);
  }
  public static void Puzzle(int x, int y)
  {
    if (Fib(x + 27277) + Fib(y - 27277) == 42)
      Console.WriteLine("puzzle solved");
  }
}
```

Black box

Black box

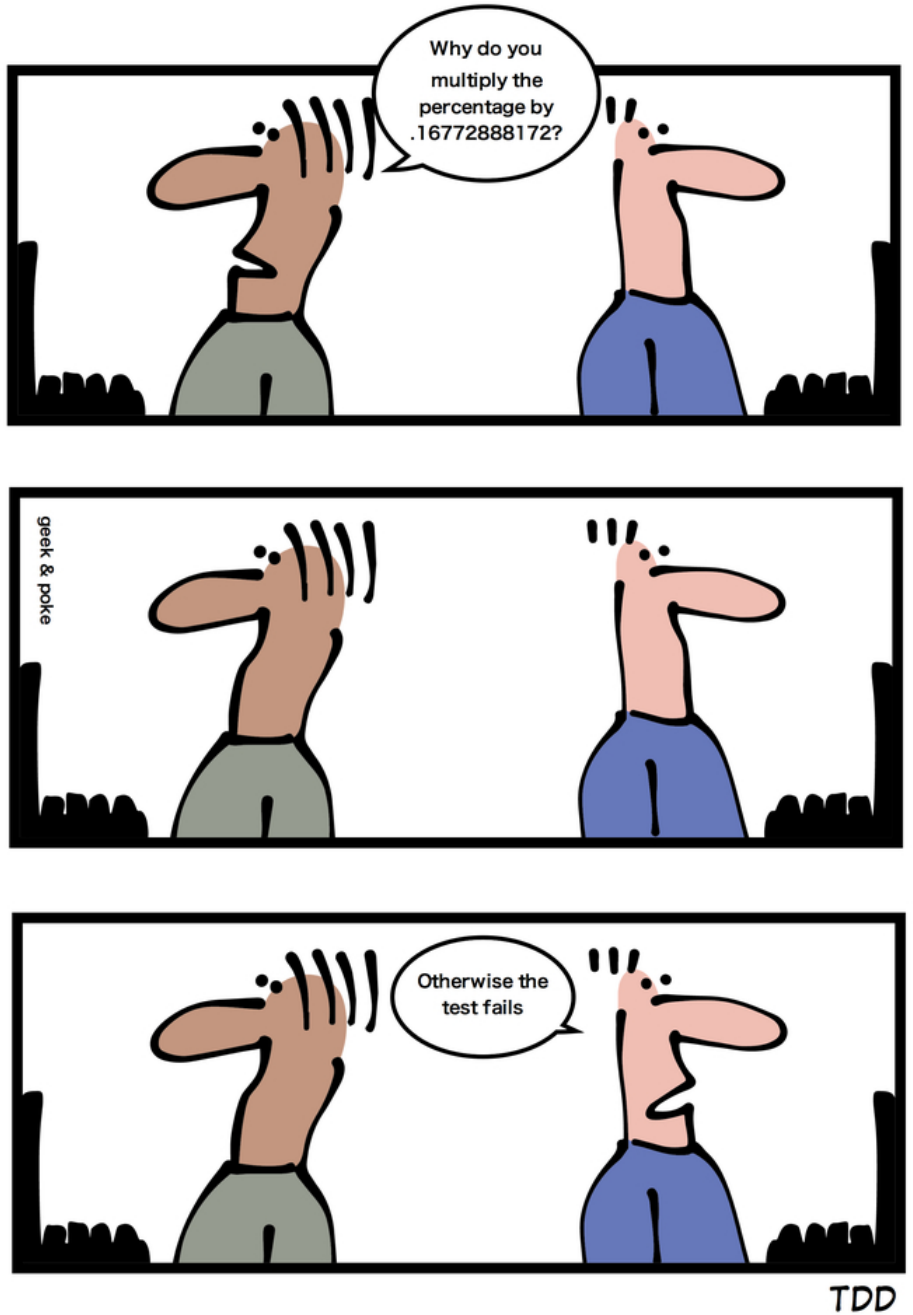
What if we cannot / don't
want to rely on the code?

Black box: TDD

Test driven development:
write tests first, then code
that passes them.

Black box: TDD

Test driven development: write tests first, then code that passes them.



Black box: test & spec

Tests cannot replace specs, but allow to exploit it more.

Generate tests from specs:

spec coverage, *e.g.*, cause/consequence, clauses

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  $\rightsquigarrow$ no need for a spec (phew!)

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  $\rightsquigarrow$ no need for a spec (phew!)

Stress

- Flood a server with requests
- Execution with constrained resources (memory, disk)
- Create latency (network)

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  $\rightsquigarrow$ no need for a spec (phew!)

Stress

- Flood a server with requests
- Execution with constrained resources (memory, disk)
- Create latency (network)

Fuzz testing

- Mainly for file formats and protocols
- Test on (partly) randomly generated/modified data
- `zzuf` (demo), `LibFuzzer`, `afl-fuzz`, ...

Chaos engineering

Today's large distributed systems bring problems for which testing is insufficient. New slogans:

design for failure and experiment in production.

Chaos engineering

Today's large distributed systems bring problems for which testing is insufficient. New slogans:

design for failure and experiment in production.

Example (Netflix)

- The "Chaos Monkey" tool randomly disables machines/services
- `http://principlesofchaos.org`

In practice

Tooling

Librairies to write tests more easily:

`xUnit`, `Scalacheck`, `Scalatest`, etc.

Environments and tools to use them effectively:

`pytest`, `sbt`, hooks & CI, etc.

Demo

Objection 1

Writing tests = wasting time ?

Objection 1

Writing tests = wasting time ?

When coding, you're already writing tests:

maybe in an interpreter,

often in temporary `printf` checks, visual verification,

etc.

The goal is to preserve such tests, so as to fully exploit them.

Objection 1

Writing tests = wasting time ?

When coding, you're already writing tests:
maybe in an interpreter,
often in temporary `printf` checks, visual verification,
etc.

The goal is to preserve such tests, so as to fully exploit them.

Regression test

Good practice integrating testing and debugging:
before debugging, turn minimized bug into a test;
the test will validate the fix and prevent future regressions.

Objection 2

"That's easy for a sorting function,
but another story for a server. . ."

Often, hard to test = poorly designed !

Examples

- Interaction with the filesystem, a database, etc.: sandboxing
- Graphical interface: possibility to script or capture (`xnee`)
  beware: testing the interface or the underlying logic?
- Non-functional aspects (time, space): profiling

Conclusion

Summary

- Test your code systematically.
- Design for unit tests.

What's next

- Exercises:
  - Code FIND with `pytest` and `hypothesis`
  - Debug `bheap.py` with the same tools
- Project: each goal must be tested for validation