From tests to proofs I

Why do we trust programs?

CS-214 - 2 Oct 2024

Clément Pit-Claudel

Quick announcements

Our first poll
is out, please help!

Our syllabus
Has all deadlines.

Proofs exercises
Are out.

The debugging guide
is the perfect companion to the calc lab.

The debriefs
contain useful information
(we hope!)

Last week

Debugging I: the 30’000 feet view

- CS214 step-by-step guide
  - Triage + diagnose
  - Observe + guess + experiment
- Debugging
  - goes beyond code
  - is a scientific endeavor

Only way to get proficient:
practice in the labs!

The 2023 CS214 guide to debugging: on one slide

Process

Triage phase

1. Check that there is a problem
2. Reproduce the issue
3. Decide whether it's your problem
4. Write it up

Diagnosis phase

1. Learn about the system
2. Simplify, minimize, and isolate
3. Observe the defect
4. Guess and verify
5. Fix and confirm the fix
6. Prevent regressions

Techniques

- Keep notes
- Change one thing at a time
- Apply the scientific method
- Instrument
- Divide and conquer
- Ask for help

Pitfalls

- Random mutation
- Staring aimlessly
- Wasting time
- Assuming a bug went away
- Fixing effects, not causes
- Losing data

Computer Science

Software Engineering

Making trustworthy software

Learning objectives:

1. Write unit tests, integration tests, and assertions to check that software behaves well

2. Next week: Formulate software specifications

- Testing: from checklists to require/ensuring
- **Next week:** Specifications: from user stories to math

Last week This week Next week

The 2023 CS214 guide to debugging: on one slide

Process

Triage phase

1. Check that there is a problem
2. Reproduce the issue
3. Decide whether it's your problem
4. Write it up

Diagnosis phase

1. Learn about the system
2. Simplify, minimize, and isolate
3. Observe the defect
4. Guess and verify
5. Fix and confirm the fix
6. Prevent regressions

Techniques

- Keep notes
- Change one thing at a time
- Apply the scientific method
- Instrument
- Divide and conquer
- Ask for help

Pitfalls : *On your own!*

- Random mutation
- Staring aimlessly
- Wasting time
- Assuming a bug went away
- Fixing effects, not causes
- Losing data

Part I

Instrumentation: debugging in the small

1. **Tracing recursive functions**
   `println`
2. **Understanding stack usage**
   The dreaded stack overflow

Warning: Running code is great, but **you should also be able to run code by hand / in your head.**

Demo

Instrumenting recursive code

Original implementation

```
def eval(e: Expr): Int =
 e match
   case Num(n) ⇒ n
   case Plus(e1, e2) ⇒
     val res1 = eval(e1)
     val res2 = eval(e2)
     res1 + res2
```

Traced implementation

```
def eval(e: Expr, indent: String=""): Int =
  println(f"${indent}→ ${e}")
  val res =
   e match
      case Num(n) ⇒ n
      case Plus(e1, e2) ⇒
        val res1 = eval(e1, indent + " ")
        println(f"${indent}~ ${res1}")
        val res2 = eval(e2, indent + " ")
        res1 + res2
 println(f"${indent}← ${res}")
 res
```

Tracing output

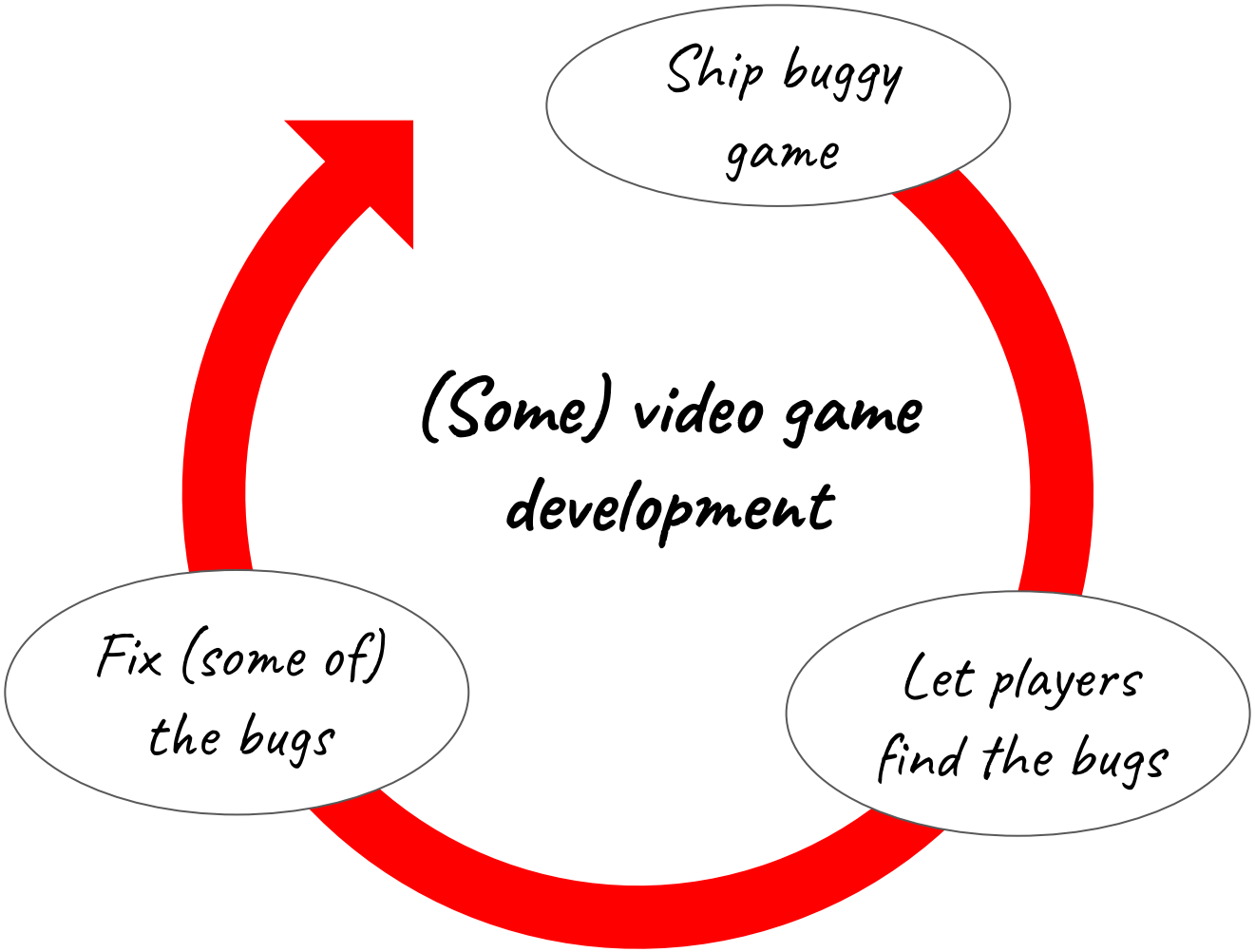
```
scala> eval(Plus(Plus(Num(3), Num(4)), Num(2)))

→ Plus(Plus(Num(3),Num(4)),Num(2))
  → Plus(Num(3),Num(4))
    → Num(3)
    ← 3
  ~ 3
    → Num(4)
    ← 4
  ← 7
~ 7
  → Num(2)
  ← 2
← 9

val res0: Int = 9
```

Part II

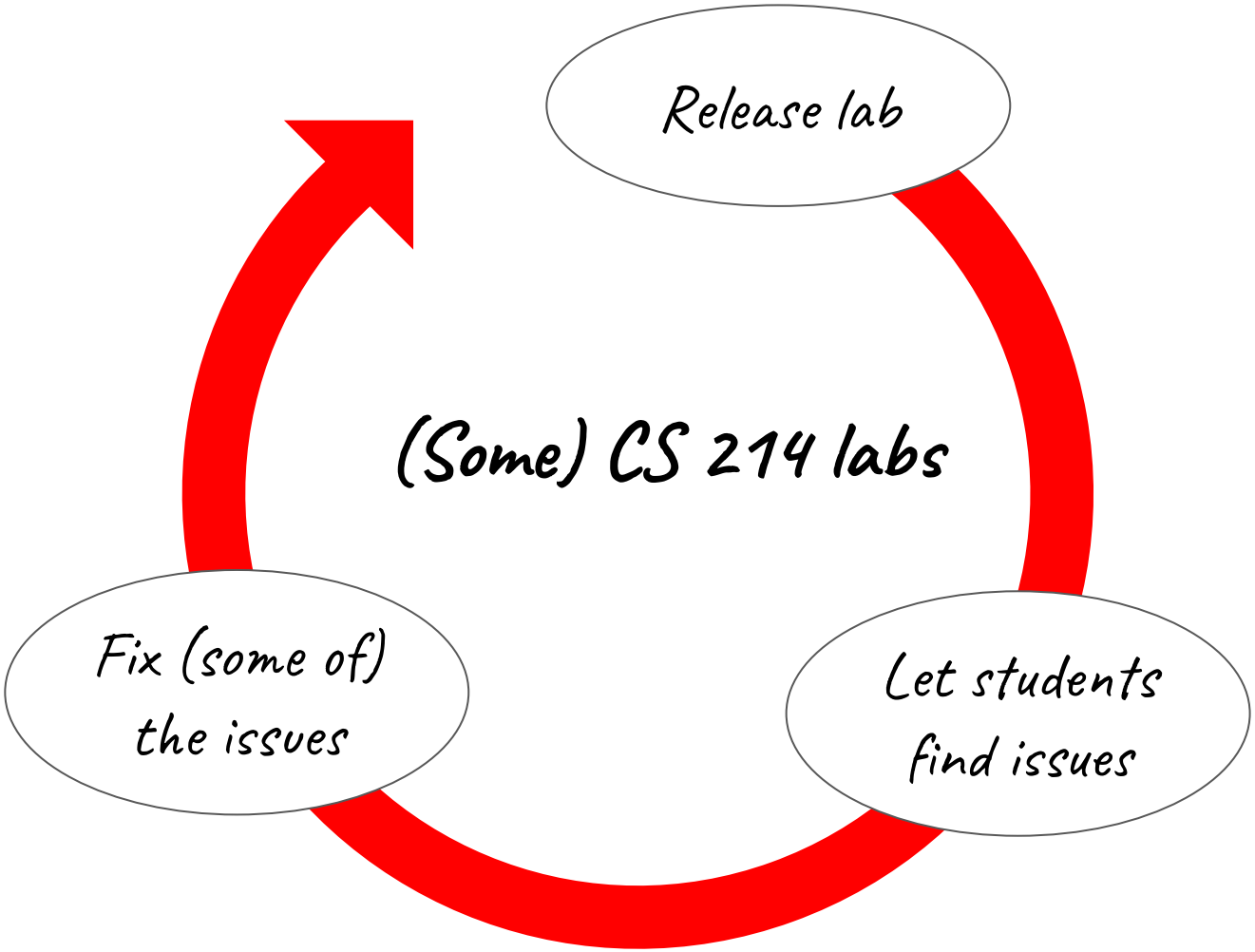
Testing: from manual checks to automated monitoring

1. **Acceptance testing**
   Have the customer click on it
2. **System testing**
   Click on it yourself with a checklist
3. **Integration testing**
   Run automated tests end to end
4. **Unit tests**
   Run per-component tests
5. **Pre/post conditions**
   Monitor components as they run

Acceptance testing: have the users click on it



Acceptance testing: have the users click on it



Boids post-release issues

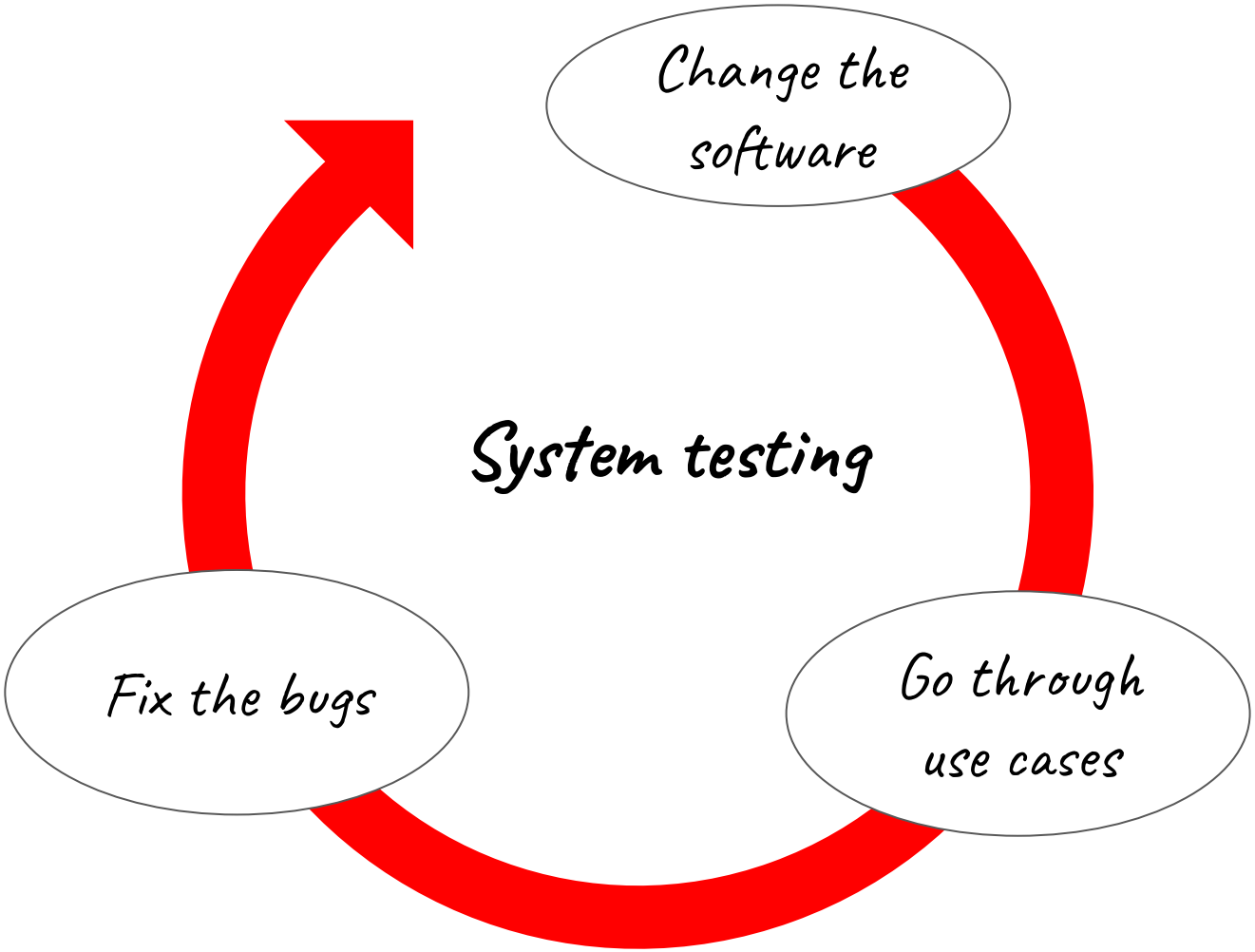
Closed Issue created 1 week ago by

- Server silently drops exceptions thrown by student code
- "Time spent on lab" poll is missing
- UI should have a legend for reference vs student implementation
- "Step" button doesn't work in UI

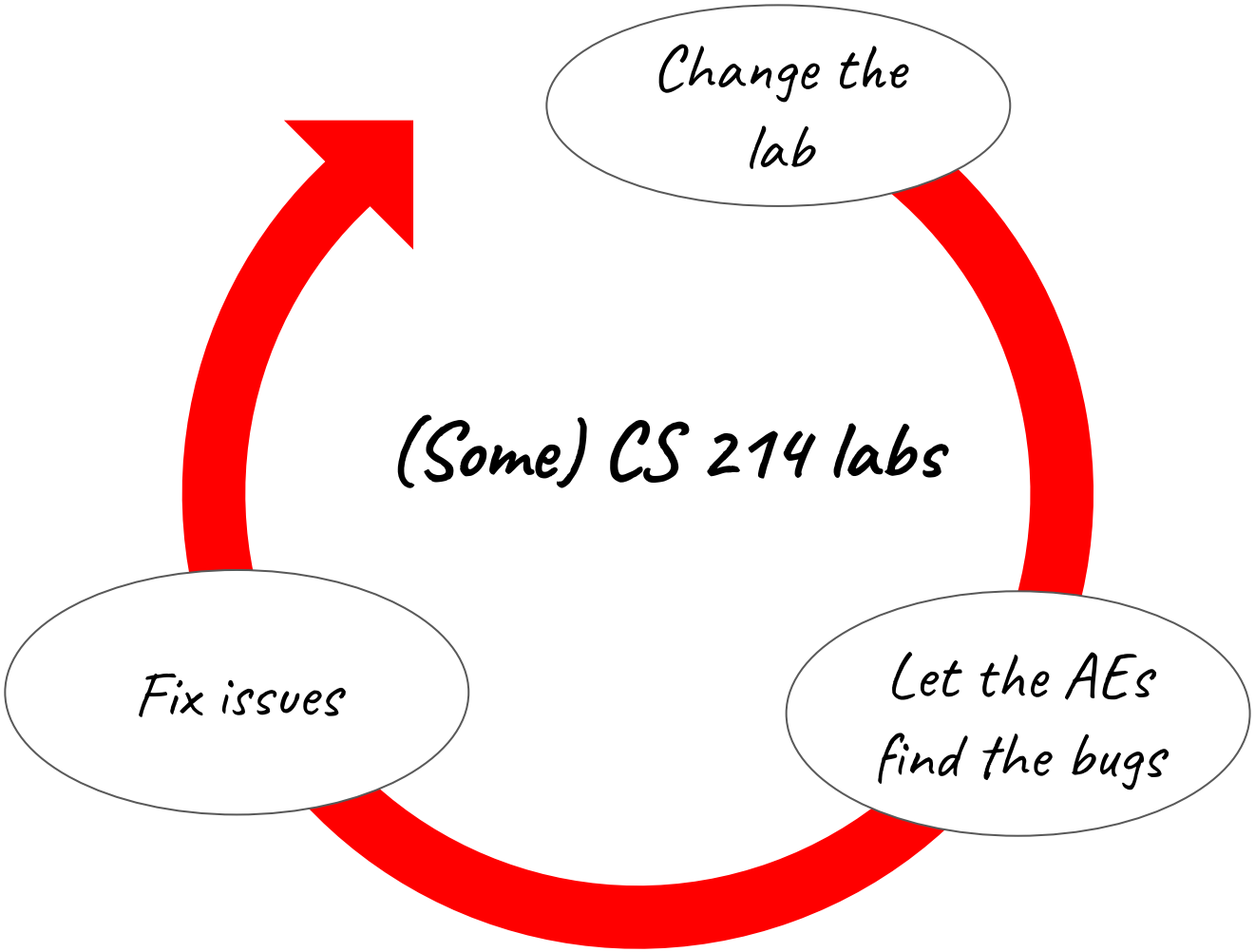
If you get any other reports from students during the lab session, please add them here before 5pm today.

Exercise: What's one danger of acceptance tests?

System testing: click on it yourself



System testing: click on it yourself



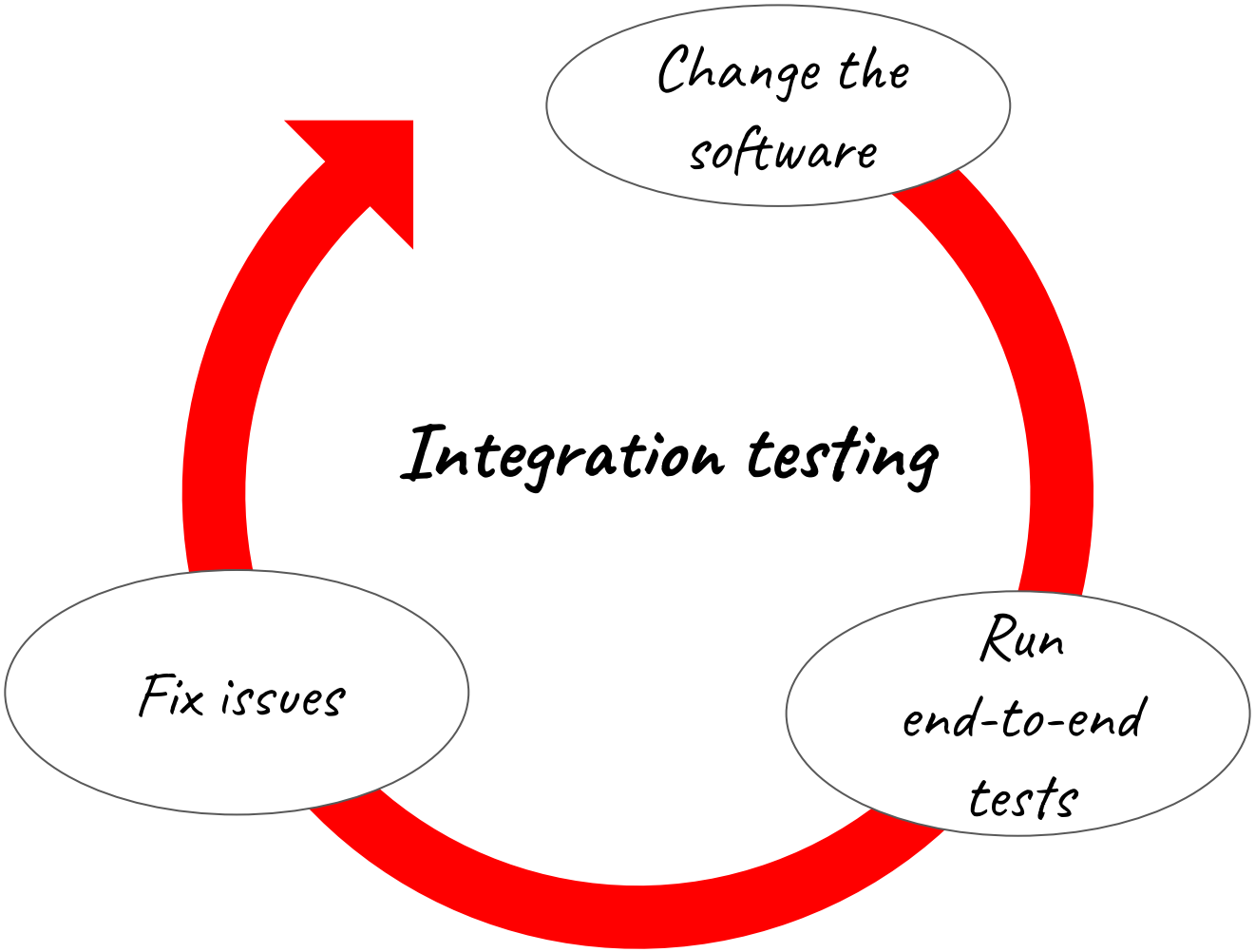
Boids

Merged requested to merge `boids` into `main` 1 month ago

Overview 117 Commits 19 Pipelines 10 Changes 18

Exercise: What's one downside of system testing?

Integration testing: run automated tests end to end



```
test("`find simple -empty` returns empty directories and files (2pts)"):
  testFind("""
    $ find simple -empty
    simple/.gitignore
    simple/target/streams/.agignore
    simple/target/streams/compile/externalDependencyClasspath/_global/streams/out
    simple/target/streams/compile/managedClasspath
    simple/target/streams/compile/scalacOptions
    simple/target/streams/compile/unmanagedClasspath
    simple/target/streams/runtime
    simple/target/streams/test/.agignore
   """)
```

Exercise: In which way was this not a fully faithful integration test?
How does this differ from the operation of the real system?

```
test("world with single boid, no forces"):
  runTestCase("00_singleBoidNoForces")

test("world with three boids, no forces"):
  runTestCase("01_threeBoidsNoForces")

test("avoidance doesn't affect lone boid"):
  runTestCase("10_singleBoidAvoidance")

test("avoidance between two boids face-to-face"):
  runTestCase("11_twoBoidsAvoidanceX")

test("same as above, with orthogonal velocity component"):
  runTestCase("12_twoBoidsAvoidanceXY")

test("no avoidance between far boids"):
  runTestCase("13_twoBoidsAvoidanceFar")

test("avoidance among mixed boids"):
  runTestCase("14_mixedAvoidance")
```

Exercise: What kind of issues do integration tests not catch?

Integration tests don’t test problems in isolation

```
def findBySizeGeAndPrint(entry: cs214.Entry, minSize: Long): Boolean =
  val thisFound =
    !entry.isDirectory()
      && entry.size() ≥ minSize
      && { println(entry.path()); true }

  val childrenFound =
    entry.isDirectory()
      && entry.hasChildren()
      && findBySizeGeAndPrint(entry.firstChild(), minSize)

  val nextSiblingsFound =
    entry.hasNextSibling()
      && findBySizeGeAndPrint(entry.nextSibling(), minSize)

  thisFound || childrenFound || nextSiblingsFound
```

```
def sizeGe(e: entry):
    e.size() ≥ minSize

def findBySizeGeAndPrint(entry: cs214.Entry, minSize: Long): Boolean =
  findAndPrint(entry, e ⇒ !e.isDirectory() && sizeGe(e, minSize))
```

Integration tests don’t tell you what’s wrong!

```
random test: simplify 1.0 * (1.0 *
(-4.719951559702111 * y1 + -8.927959938264125 *
y0 - (0.0 + (-4.719951559702111 * y1 +
-8.927959938264125 * y0)) - 0.0 + 0.0 + 0.0 +
(0.0 - 0.0)) + 1.0 * 0.0 * 1.0) / 1.0 + (0.0 -
0.0) should be 0.0, but it is actually -0.0
=> Obtained
Neg(
  e = Number(
    value = 0.0
  )
)
=> Diff (- obtained, + expected)
-Neg(
-  e = Number(
-    value = 0.0
-  )
+Number(
+  value = 0.0
 )
```

Pitfalls

- Random mutation
- Staring aimlessly
- Wasting time

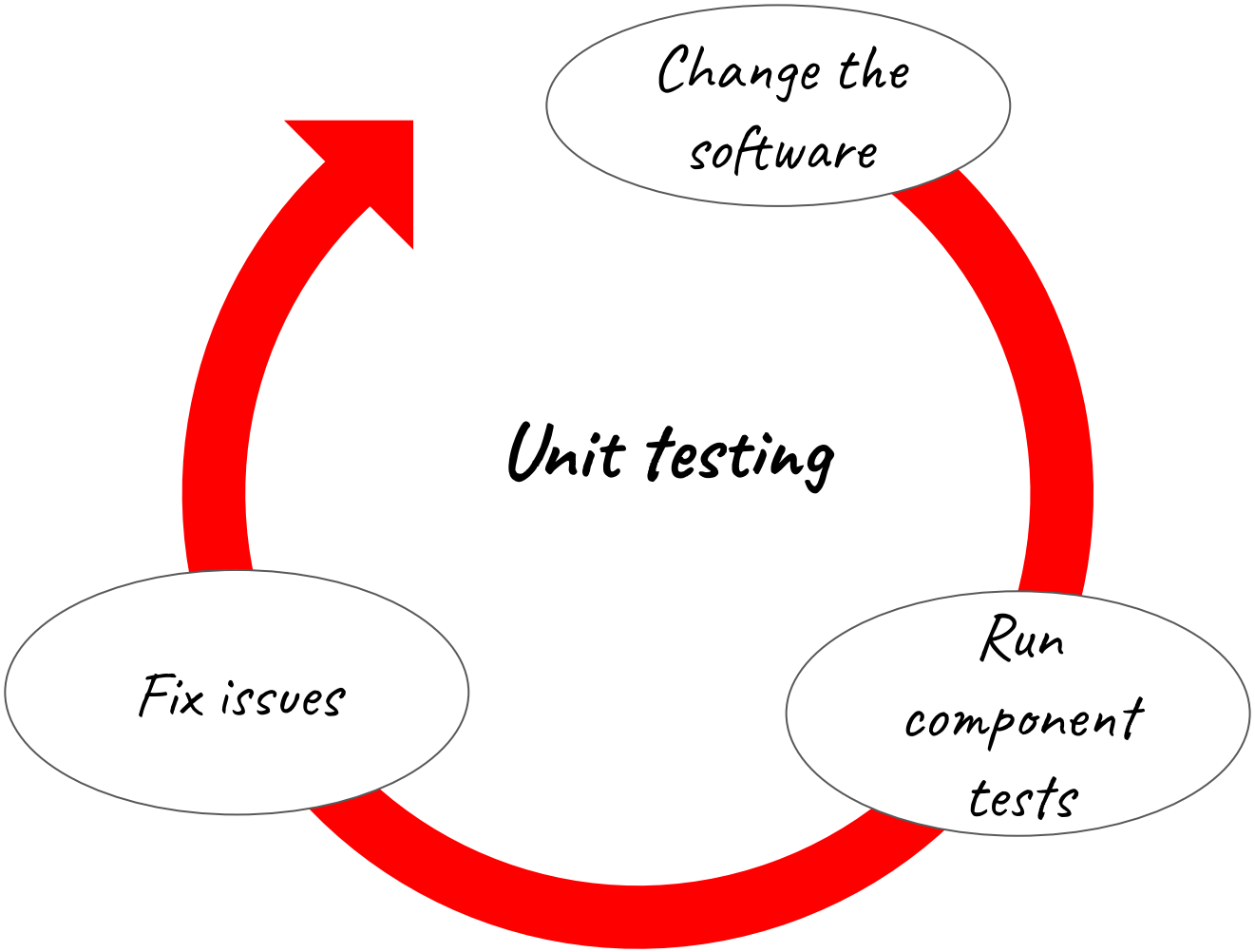
Diagnosis phase

- Simplify, minimize, and isolate

Techniques

- Divide and conquer

Unit testing: run automated single-component test



```
test("discardWord: empty string"):
  assertEquals(discardWord(""), "")

test("discardWord: 1 word with one space before"):
  assertEquals(discardWord(" hello"), " hello")

test("discardWord: 1 word with two spaces before"):
  assertEquals(discardWord("  hello"), "  hello")

test("discardWord: 1 word with one space after"):
  assertEquals(discardWord("hello "), " ")

test("discardWord: 1 word with two spaces after"):
  assertEquals(discardWord("hello  "), "  ")

test("discardWord: 1 word with one space before and after"):
  assertEquals(discardWord(" hello "), " hello ")

test("discardWord: 2 words with one space between"):
  assertEquals(discardWord("hello world"), " world")
```

Exercise: Will the tests above catch all bugs?

➡️ SOON Can unit tests *ever* catch all bugs?

A bad `discardWords`

```
def discardWord(s: String): String = {
  if !s.isEmpty && 'a' ⩽ s.head && s.head ⩽ 'z' then
    discardWord(s.tail)
  else s // Blank space found, exit
}
```

Exercise: How can I prevent that?

Some properties of `discardWord`

"The output of `discardWord` must always be empty or start with a blank"

```
ensuring(result ⇒
  result.isEmpty ||
  result.head.isBlank)
```

"The output of `discardWord` must be a suffix of its input"

```
ensuring(s.endsWith(_))
```

These properties are executable!

I can make Scala check them *as I run my code, on every input I give it*.

A bad `discardWords`

```
def discardWord(s: String): String = {
  if !s.isEmpty && 'a' ⩽ s.head && s.head ⩽ 'z' then
    discardWord(s.tail)
  else s // Blank space found, exit
}
```

Exercise: How can I prevent that?

Checking that we found a blank space

```
def discardWord(s: String): String = {
  if !s.isEmpty && 'a' ⩽ s.head && s.head ⩽ 'z' then
    discardWord(s.tail)
  else s // Blank space found, exit
} ensuring(suffix ⇒
  suffix.isEmpty || suffix.head.isWhitespace)


discardWord("ABC def")
// java.lang.AssertionError: assertion failed
// at scala.Predef$.assert(Predef.scala:264)
// at scala.Predef$Ensuring$.ensuring$extension(Predef.scala:359)
// at repl.MdocSession$MdocApp.discardWord(debug.worksheet.sc:78)
// at repl.MdocSession$MdocApp.<init>(debug.worksheet.sc:83)
// at repl.MdocSession$.app(debug.worksheet.sc:3)
```

Big idea of monitoring:

Use integration runs and real executions to test individual components

1 unit test = 1 input/output pair

1 monitor = infinitely many tests,
over the whole life of the application

Check yourself: boids lab

Good:

```
boids.filter(b ⇒
  b != thisBoid &&
  b.position.distanceTo(thisBoid.position) < radius
)
```

Bad:

```
boids.filter(b ⇒
  b.position.distanceTo(thisBoid.position) < radius
)
```

Exercise: Which postcondition would have caught the issue?
What should I add to `boidsWithinRadius`?

`requires` and `ensuring` are not magic!

```
def sort(l: List[Int]): List[Int] = {
  …
} ensuring (res ⇒ (0 to res.length - 2).forall(idx ⇒
    res(idx) ⩽ res(idx + 1)))
```

```
def sort(l: List[Int]): List[Int] = {
  val res = …
  assert (0 to res.length - 2).forall(idx ⇒
    res(idx) ⩽ res(idx + 1))
  res
}
```

Exercise: 🔜 Are these postconditions complete?

🔜 Do they allow buggy implementations?

🔜 Would it be enough to check `(0 to l.length - 2)`?

`assert` is not magic!

```
def maxMagic(l: List[Int]): Int = {
  requires(!l.isEmpty)
  …
} ensuring (res ⇒ l.forall(x ⇒ x ⩽ res))

def maxAssert(l: List[Int]): Int =
  assert(!l.isEmpty)
  val res = …
  assert (res ⇒ l.forall(x ⇒ x ⩽ res))
  res

def maxByHand(l: List[Int]): Int =
  if l.isEmpty then throw AssertionError()
  val res = l.reduce(Math.max)
  if !l.forall(x ⇒ x ⩽ res) then throw AssertionError()
  res
```

Exercise: What is the point of tests?

- Tests **document** what your program is supposed to do
- Tests **protect** you from regressions (bugs reappearing)
- Tests **pinpoint** the source of issues
- Tests **confirm** that your software is fit for release
- Tests **detect** changes in components you don’t control
- Tests **facilitate** interaction between components
- … more?

Recap: Testing a theater play

Let's assume you're preparing a theater play.

1. You rehearse your monologue in front of the mirror

2. You rehearse a scene in a classroom with the other actor

3. You complete a full rehearsal in the actual theater

4. You perform at the premiere

5. The prompter backstage sees you struggle and feeds you line

Recap: Testing a theater play

Let's assume you're preparing a theater play.

1. You rehearse your monologue in front of the mirror

   ⇒ Unit test
2. You rehearse a scene in a classroom with the other actor

   ⇒ Integration test
3. You complete a full rehearsal in the actual theater

   ⇒ System test
4. You perform at the premiere

   ⇒ Acceptance test
5. The prompter backstage sees you struggle and feeds you line

   ⇒ Monitoring

Recap exercise

What the test?!

Your clothes washer blinks “F09” / “E01”

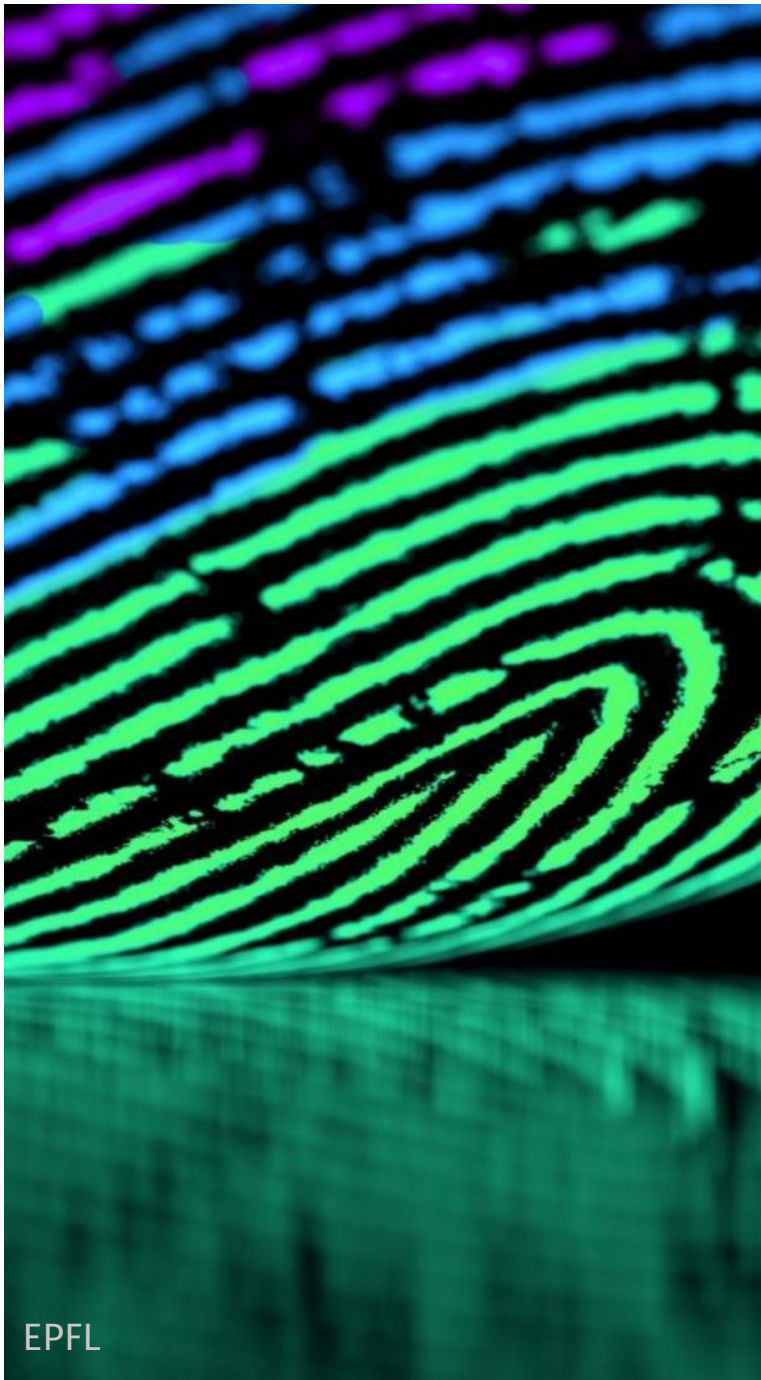
a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring

Your landlord walks through the apartment before returning your deposit

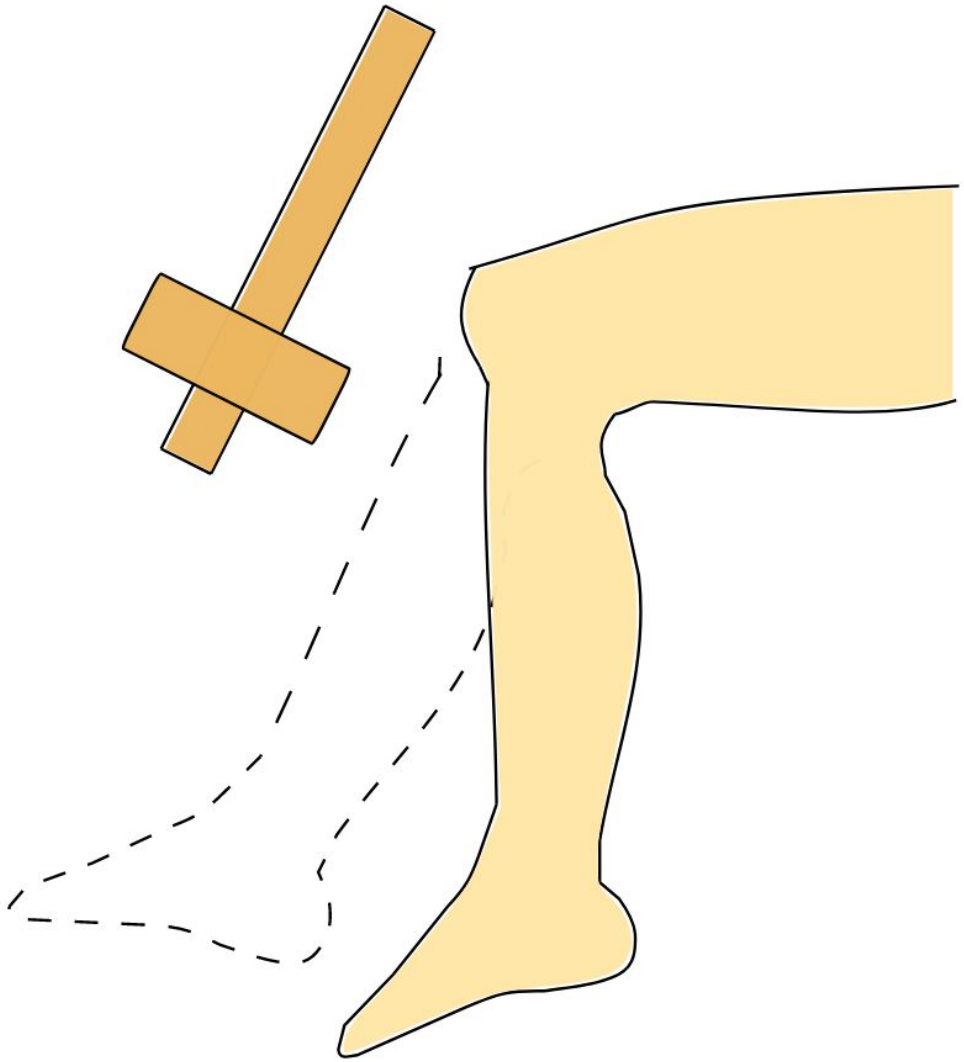
a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring

EPFL staff runs rehearses the *Magistrale* ceremony one day before the even

a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring



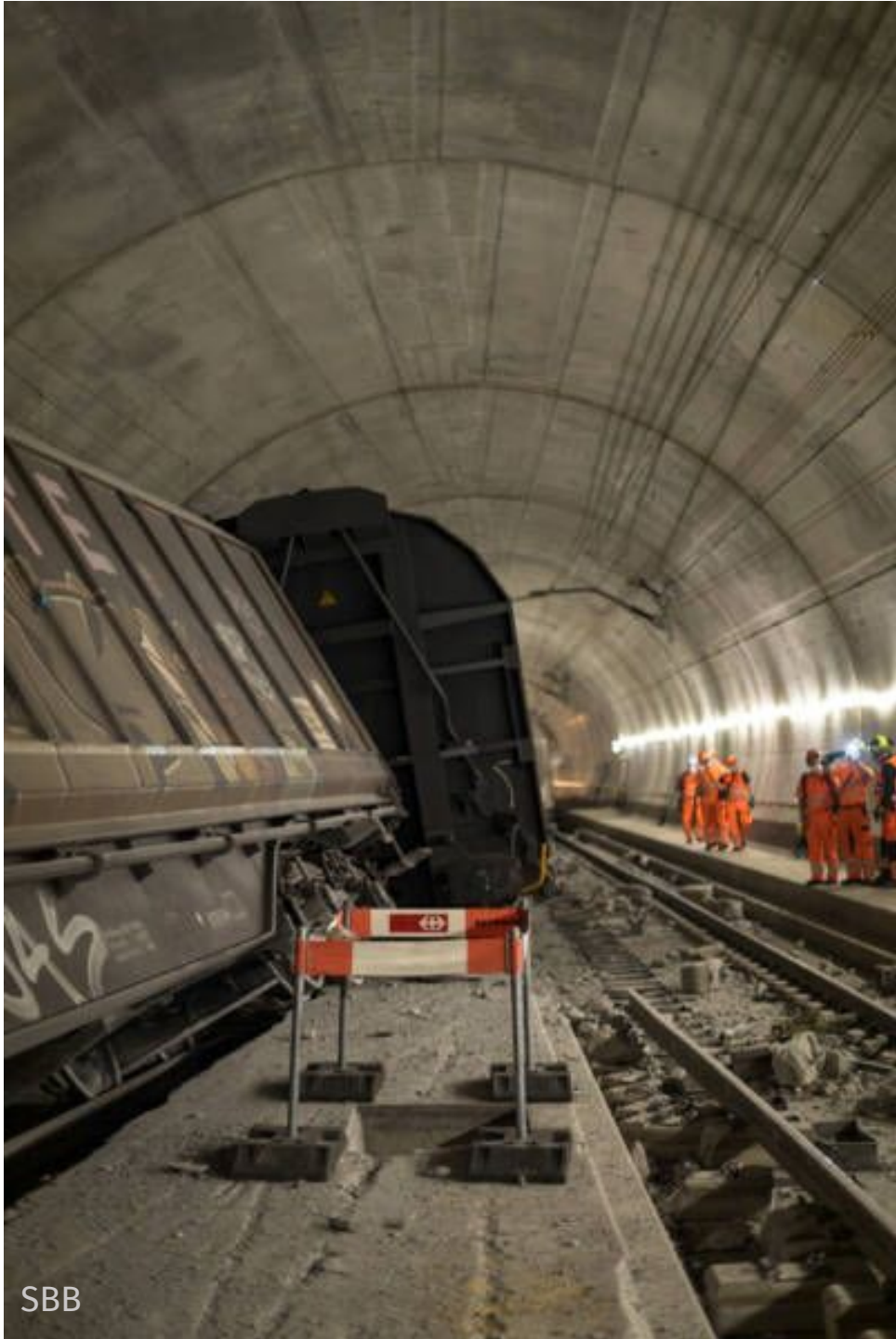
The doctor hits your knee with a hammer to check your reflexes



a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring

A **freight company** runs an empty train in the Gotthard tunnel after SBB completes repairs

a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring


SBB

You smell the milk to check if it's still good before making a *renversé*

a. Acceptance test
b. System test
c. Integration test
d. Unit test
e. Monitoring

Reminder

Fill the poll!

Syntax recap

Unit tests: syntax

```
class XYZTests extends munit.FunSuite:

  test("Addition works on 1 + 1"):
    assertEquals(1+1, 2)

  test("Less-than handles (1, 3)"):
    assert(1 < 3)


  test("Two is not 5"):
    if 2 == 5 then
      fail("Uh oh, 2 should not be 5!")
```

Full reference: https://scalameta.org/munit/docs/tests.html
Includes info on testing for exceptions, mocking, etc. → check it out!

Pre/post conditions + assertions: syntax

```
def sqrt(x: Double): Double = {
  require(x ≥ 0)
  …
} ensuring (res ⇒ Math.abs(x * x - res) < 0.000001f)


def lookup(bst: BinarySearchTree, k: K): V = {
  require bst.IsBST()
  …
  val left = bst.leftChild
  assert(left == Leaf || left.value < bst.value)
  …
}
```

Next week!

Specs: From English to Math

1. **User stories**
   Capture needs and goals
2. **Requirements**
   Say what the user wants
3. **Specifications**
   Say what the program does
4. **Formal specifications**
   Tie it all together with code & math