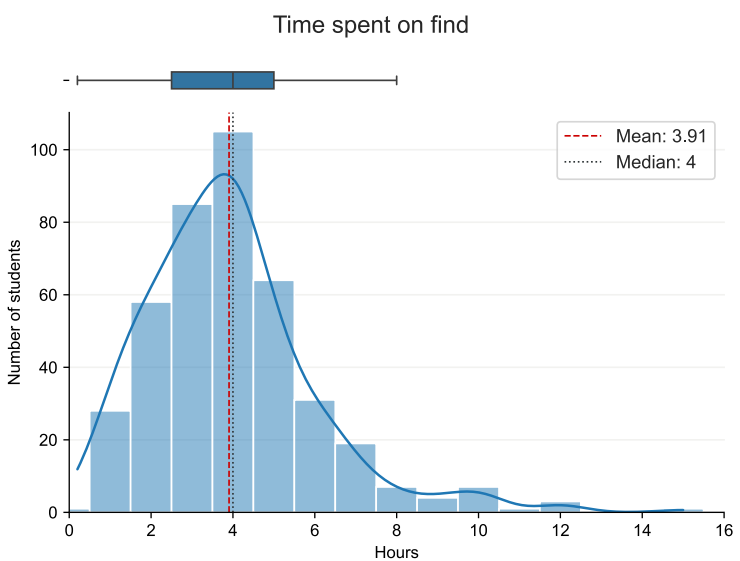
How to diagnose (almost) anything

CS-214 – 2024-09-25

Clément Pit-Claudel

Quick announcements



Good job on `find`!

Don't `var`

(won't work on the midterm)

This week's lab

has a callback!

New help sessions

Tuesdays 4–7 PM in INF [123]

(INF 1 from 5 PM next week)

The plan for CS-214.SE

- Make you (better) software engineers
- Show you how to figure things out

Versioning and collaboration

3 lectures

Writing correct code

4 lectures

Building apps from scratch

3 lectures

+ unguided lab & callback

Refactoring & code evolution

Labs + exercises

Exercise

Think of a bug that frustrated you in a recent lab.

Share with your neighbor.

Today

Debugging:
Principles + case studies

- Identify and describe **unwanted behaviors**
- Produce useful and complete **defect reports**
- Pinpoint the **root cause** of an issue in a software system

The 2024 CS-214 guide to debugging: on one slide

Process

Triage

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

Diagnose and fix

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

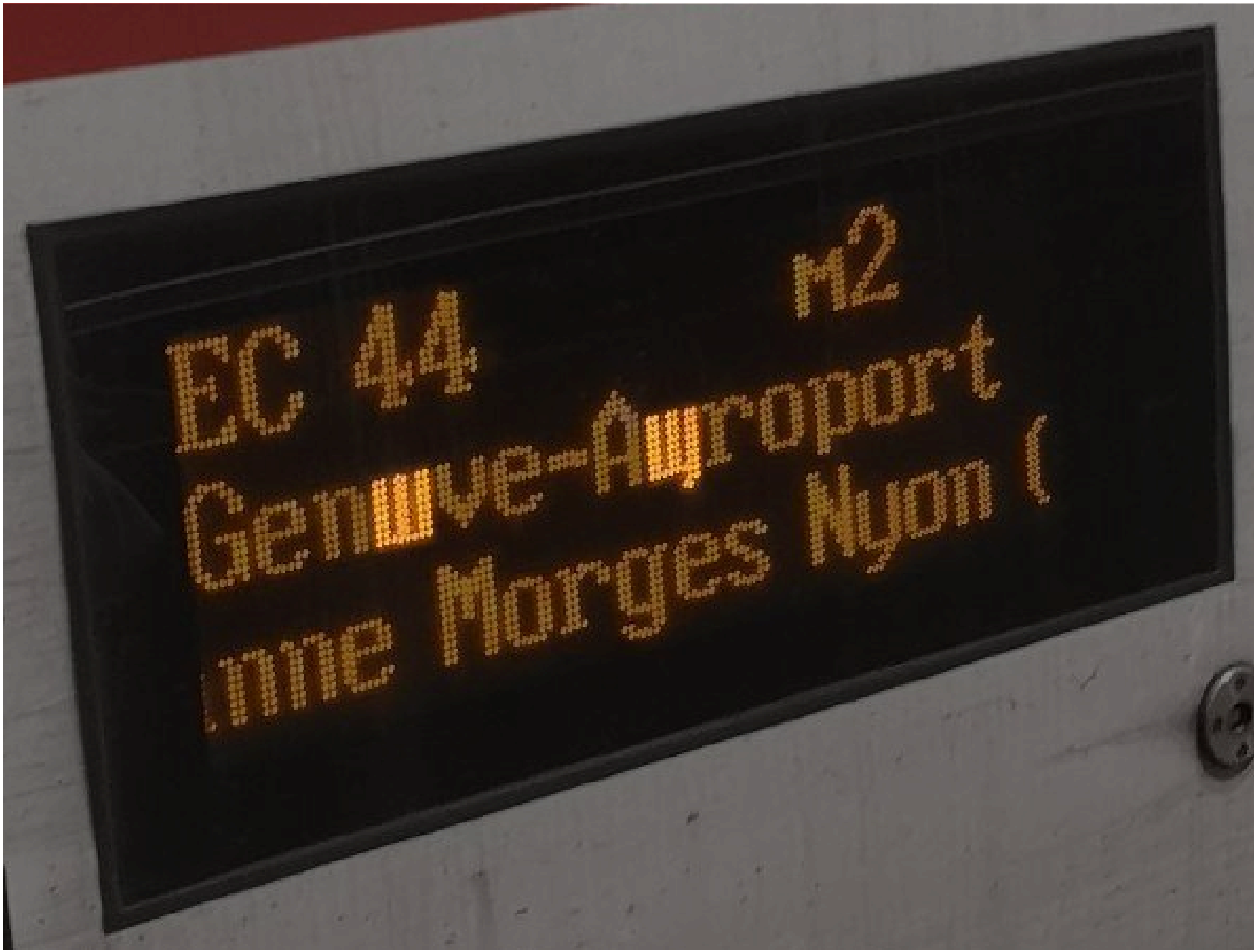
- Keep notes
- Change one thing at a time
- Apply the scientific method
- Instrument
- Divide and conquer
- Ask for help

Pitfalls

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

A CASE STUDY


2
EC 44 m2
Genuve-Auroport
nme Morges Nyon (
500m


EC 44
H2
Geneve-Aeroport
Lme Morges Nyon (

Step 1 (of 2)

Triage

Check

that there is a problem

Reproduce

the issue

Decide

whether it’s your problem

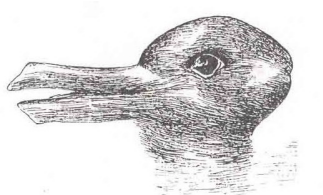
Write up

a detailed bug report

Triage 1&2: Confirm and reproduce

1. Confirm that there is a problem
Know what you're looking for

- State what the issue is.
  (May be refined later)
  → *"Coursier exits silently without installing Scala"*
- Compare to the spec, the documentation, the requirements.
  → *"Pressing 'step' must advance the simulation"*
- Is it obvious why the bad case is different from a good case?
  → *"System will identify animal in picture"*



2. Reproduce the issue
Gather information

- Does it happen every time?
  → *"Only on Tuesdays"*
- Does it happen for every input?
  → *"Only in test 3"*
  → *"Only when clicking repeatedly"*
- Does it depend on system config?
  → *"Only with SBT version XYZ"*
- Are there any diagnostics?
  → Error messages, logs, dialogs
- Did it work at one point?
  → Previous versions, commit history

Triage 3&4: Decide and write up

3. Decide whether it's your pb
Not all bugs are worth fixing

- Do / need to fix it?
  - → Closed-source SW, lack of expertise/interest
- Is it worth fixing?
  - → Workaround may be sufficient
- Does it need to be fixed *now*?
  - → Lab must be released to students at 4PM
- Do I know where to complain?
  - → May require diagnosis

4. Write it up
Don't waste your work

- Check previous reports
  - → Browse bugs DB
  - → Check known issues
  - → **Browse previous questions!**
- Find where to report
  - → Email, bug tracker, contact form
- Check reporting guidelines
  - → Is there a security policy?
- Write clearly and completely
  - → Pick a good title
  - → Include steps taken, results observed, expectations
  - → Include system details, any customization

Exercise

Apply this to the SBB example!

```
Number: CS006551005
Date: 25.09.2024
Description:
```

How to ask a CS214 Ed question

What does 'A pure expression does nothing in statement position' mean?

PIN STAR WATCH VIEWS

Anonymous

5 hours ago in Lab 2

Hi,

When I replace the contents of `playground.worksheet.sc` with the following code, VSCode underlines the first 0 in red. The corresponding error message is "*A pure expression does nothing in statement position; you may be omitting necessary parentheses*"

```
def f(): (Int) =
  if true then 0
  else if true
    val r = f()
    (r)
  else 0
```

What does the error message mean, and how can I fix my code?

Comment Edit Delete Endorse •••

- Helpful title
- Good punctuation
- Clear problem
- Has expected behavior
- Has reproduction steps
- Properly formatted code
- Relevant info present

Ed post checklist

☑ Is the title clear?

- ✗ *“Lab 2”* → ✓ *“Broken link to lab 2 on Moodle”*
- ✗ *“Git”* → ✓ *“Git error: ‘corrupt loose object’”*
- ✗ *“infinite loop”* → ✓ *“Infinite loop in minMax”*

☑ Is the description precise?

- ✗ *“It doesn’t work”* → ✓ *“When given input x, it produces value y […]”*
- ✗ *“It’s weird”* → ✓ *“[…] it produces value y, which is unexpected because […]”*
- ✗ *“In the lecture […]”* → ✓ *“On slide n of the lecture on x, […]”*

☑ Is the description usable?

- ✗ Code screenshot → ✓ Code block
- ✗ Code snippet → ✓ Self-contained example

☑ Are attempts at a solution documented?

- ✗ *“I tried everything”* → ✓ *“Here are the things I tried: […]”*

Step 2 (of 2)

Diagnose & fix

Learn

about the system

Observe

the defect

Simplify

and minimize

Guess

and verify

Fix

and confirm the fix

Prevent

regressions

Diagnosis 1&2: Learn and observe

1. Learn about the system
Know what to expect

- Consult the docs/manual
  - → Read the lab write-up in detail
- Search for relevant resources
  - → Known issues, error guides
  - → Documentation on error messages
- Know what you don't know
  - → *"What happens if %PATH% contains special characters?"*
  - → *"Does `1 until n` include n?"*
  - → *"I'm not sure how VSCode starts the JVM to run Metals"*
- Know the relevant tools
  - → JTAG, perf, strace, objdump, a multimeter, …
- Skim through the code
  - → Find entry point, seemingly relevant functions

2. Observe the issue
Learn more about the problem

- Exercise different angles
  - → Vary the inputs
  - → Look upstream and downstream (consequences)
- Read error messages
- Add logging / tracing Print relevant variables and rerun
  - → Turn on errors and warnings
- Use the right tools
  - → But don't get distracted
- Use the VCS history
  - → Especially for regressions
- Read the code

Diagnosis 3&4: Simplify and guess

3. Simplify and minimize the issue
Find the relevant subsystem

- Simplify the inputs
  - → Shorten the test case
  - → Find a short and simple input
- Simplify the system
  - → Remove unnecessary code
  - → Reproduce the problem in isolation
  - → Check a different machine/config
- Slow things down
  - → Introduce delays or pauses
- Determinize the failure
  - → Set random seeds
  - → Stop other processes
- Automate the failure
  - → Have tests not just a user story
  - → Use simplest possible tests (unit tests if possible)
  - → Run tests on CLI, not in UI with clicks

4. Guess and verify
Be a scientist!

- Formulate a hypothesis
  - → *"I forgot to clamp the speed"*
  - → *"The acceleration may be applied twice"*
  - → *"Hard drive delays are causing our latency issues"*
  - → *"Values get corrupted in step 5"*
- Design an experiment
  - → Transform or observe the system to test your hypothesis.
  - → Add logging, breakpoints, assertions, prints,
- Narrow down issue
  - → Divide bug into smaller bugs using informed guesses and hypothesis testing
  - → Move failing tests up until they pass
  - → Look for the root cause

Diagnosis 5&6: Fix and prevent

5. Fix and confirm the fix
Change relevant system

- Decide whether to fix the problem
  - → Is the problem easily fixable?
  - → Is a workaround preferable?
  - → Revisit *"Do* I *have to fix it"*?
- Apply the changes
- Revert other changes
  - → Use your VCS to undo unrelated changes
  - → Confirm the fix on a clean system
- Confirm the fix
  - → Re-run all tests
  - → Confirm that other behavior isn't affected

6. Prevent regressions
Protect future developers

- Document the resolution
  - → Write a detailed commit message
  - → Update the report to document the root cause, the fix or workaround, and any required follow-ups
- Look for similar instances
  - → Fix all bugs in that family, not just one
- Add missing tests
  - → Prevent future regressions

Triage & diagnosis techniques (do this…)

✓ Keep notes

- → Write every change down:
  "Ran `cs_setup.exe` from `cmd` in `C:\`"
- → Write your observations
  "The checksum is correct after the download"
- → Work on a Git branch
- → Keep decision trees, guesses, mind maps

✓ Change one thing at a time

- → Needed to establish cause / effect relationships
  Avoid: *"I added a lock and a cooldown period and an automated retry and a generic try-catch block and the error is gone"*

✓ Apply the scientific method

- → Write down a conjecture (potential cause)
  "The function is not exploring all directories"
- → Design an experiment
  Add `println` to every call to `find`
- → Reject or accept
  "All directories are printed" ⇒ reject

✓ Instrument

- → Add `println` calls
- → Write to logs

✓ Divide and conquer

- → Design tests to pinpoint the source
  "Bug happens in UI but not on CLI"
- → Inspect intermediate states
  "System is stable until cohesion force kicks in"
- → Test components in isolation
  Move from integration tests to unit tests

✓ Ask for help

- → Find a system expert
- → Write up problem online
- → Look for troubleshooting guides
- → Post on Ed

Pitfalls (… don't do this)

✗ Random mutation

→ Don’t change things “until it works”

✗ Staring aimlessly

→ *“I spent 8 hours looking at this code”*

✗ Guessing without verifying

→ Fixing a bug takes time. Don’t fix correct code!
→ Guessing well takes practice.

✗ Wasting time

→ Experiments are not all equally easy
→ Look out for signs of going down a rabbit hole
→ Sometimes a quick check can eliminate many possible hypotheses
→ Reinstalling the OS is a last resort

✗ Assuming bug went away

→ Bugs don’t disappear
→ … and they reappear at the worst time

✗ Focusing on the wrong thing

→ *“I just added cohesion and it fails, so the error must be there”*

✗ Fixing effects, not causes

→ *“I will just multiply all speeds by two”*
→ *“I just need to reinstall SBT every time I log in”*
→ *“I rewrote all the code and now it works”*

✗ Losing data

→ Take a disk image
→ Save the buggy code somewhere
→ Keep the logs / core dumps / …

Conclusion: Use the CS214 debugging guide!

After fixing a bug:

- Go through this guide
- Reflect on what went right and wrong

When tackling a new bug:

- Pull up this guide
- Follow the steps in order

When asking a question on Ed:

- Pick a helpful title
- Describe which steps you took and where you got stuck, in detail

Suggested reading

- How do I ask a good question? - Help Center - Stack Overflow
- Bug Reporting Guide | Contributing to Scala's OSS Ecosystem
- Understanding Bug Reporting (GNU Emacs Manual)
- Checklist for bug reports (GNU Emacs Manual)
- Submitting Bugs and Suggestions · microsoft/vscode Wiki · GitHub

- *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems (David J. Agans)*
- How to Debug – Embedded in Academia

Number: CS006551005
Date: 04.10.2023 15:50:10 CEST
Description: Incorrect display on train to Geneva in Lausanne:
"Genève-Aéroport" as "Genшve-Ащroport"

Hi,

My name is Clément, I work at EPFL.
On September 3rd, in Lausanne, at 11:12 PM, I noticed an incorrect display on my train door.

It appears that “Genève” and “Aéroport” were not displayed correctly:
"Genève-Aéroport" was displayed as "Genшve-Ащroport".
I have attached a picture.

Some speculation: it could be that the text "Genève-Aéroport" was encoded using Latin-1, and decoded using ISO-8859-5 or one of its variants.

Thank you for your time,
Clément Pit-Claudel.

From: SBB Customer Service <customerservice@sbb.ch>
To: clement.pit-claudel@epfl.ch
Subject: AW: CS006551005 - SBB - Customer information on the display
Date: Mon, 09 Oct 2023 05:58:03 -0700

Sehr geehrte Damen und Herren

* Der Fehler lag hier vermutlich im «KIS» System, dem Software System welche die Ansage und die Beschriftung koordiniert.

mit freundlichen Grüssen