# From tests to proofs I

## Why do we trust programs?

CS-214 W04 SE – 2025-10-01

Clément Pit-Claudel

# Quick announcements

**The Ed grace period**

is over; good posts only, please!

**The debugging guide**

is the perfect companion to the `calc` lab.

**The debriefs**

contain useful information (we hope!)

**Next week**

is heavily disrupted:

- videos only 😢
- + help session on Wednesday

# The plan for CS-214.SE

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

3 lectures

4 lectures

3 lectures

+ unguided lab

Callbacks

# Two weeks ago

Debugging 1: The 30’000 feet view

- Triage + Diagnose
- Observe + guess + experiment
- Apply the scientific method!

Only way to get proficient:
**practice in the labs**!

# The 2025 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

# Debugging is not just code and computers

Debugging is
Science +
Engineering

# This week

**Instrumentation**

Debugging in the small

**Testing**

From checklists to monitors

# Instrumentation

## Debugging in the small

### Tracing recursive functions

`println`

### 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

# Instrumenting recursive code

```
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
```

# Instrumenting recursive code

```
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
```

# Instrumenting recursive code

```
def length[T](l: List[T], indent: String=""): Int =
  println(f"${indent}→ ${l}")
  val res =
    l match
      case Nil ⇒ 0
      case _ :: t ⇒ 1 + length(t, indent + "  ")
  println(f"${indent}← ${res}")
  res
```

# Instrumenting recursive code

```
scala> length(List(1, 2, 4))

→ List(1, 2, 4)
  → List(2, 4)
    → List(4)
      → List()
      ← 0
    ← 1
  ← 2
← 3

val res0: Int = 3
```

# Making trustworthy software

Understand, specify, and check program behavior

**Testing**

From checklists to monitors

**Specifications**

From user stories to math (next week)

# Testing

From checklists
to `require`/`ensuring`

1. **Acceptance testing**
   Have the customer click on it

2. **System testing**
   Click on it yourself with a checklist

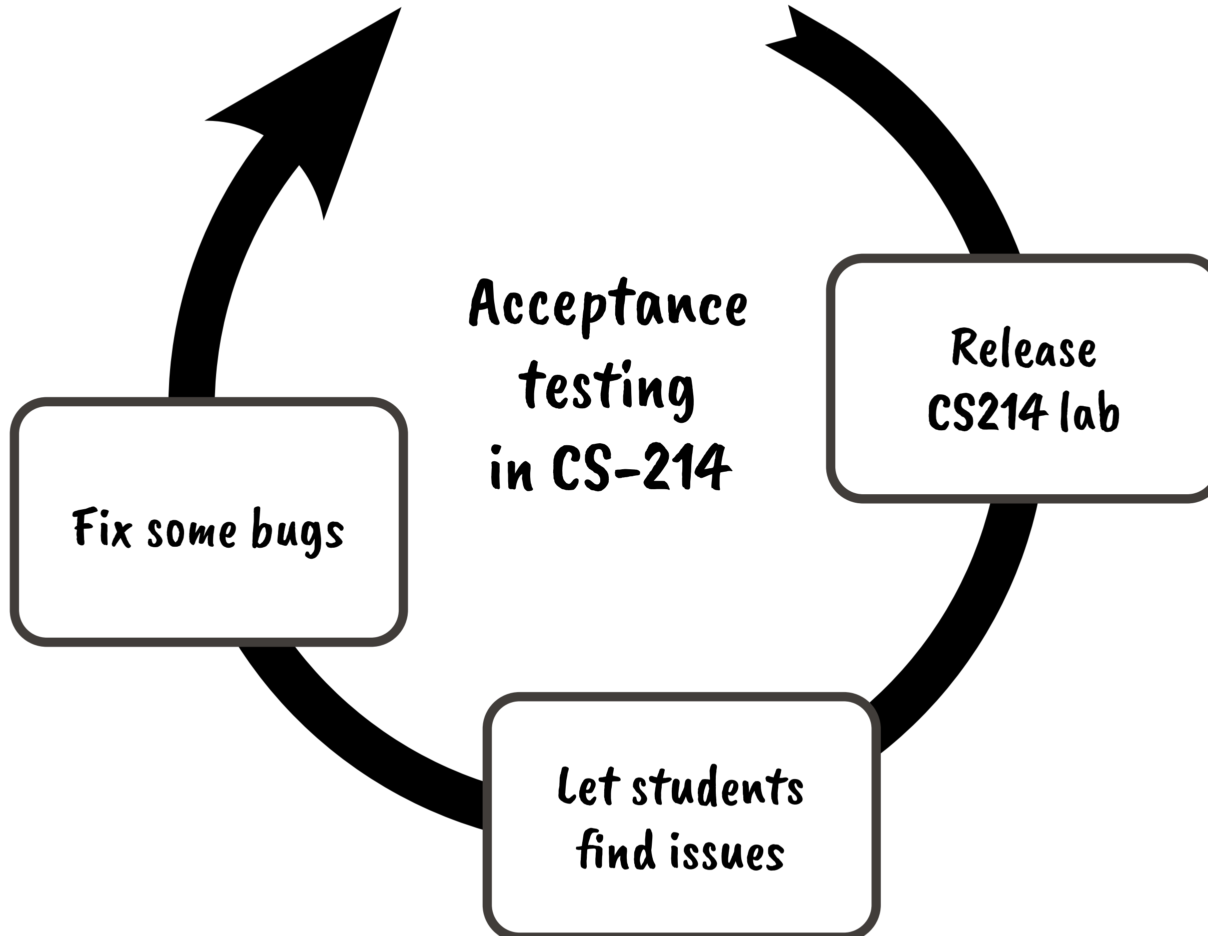
3. **Integration testing**
   Run multi-component tests

4. **Unit tests**
   Run single-component tests

5. **Pre/post conditions**
   Monitor components as they run

# Acceptance testing

## HAVE THE CUSTOMER CLICK ON IT



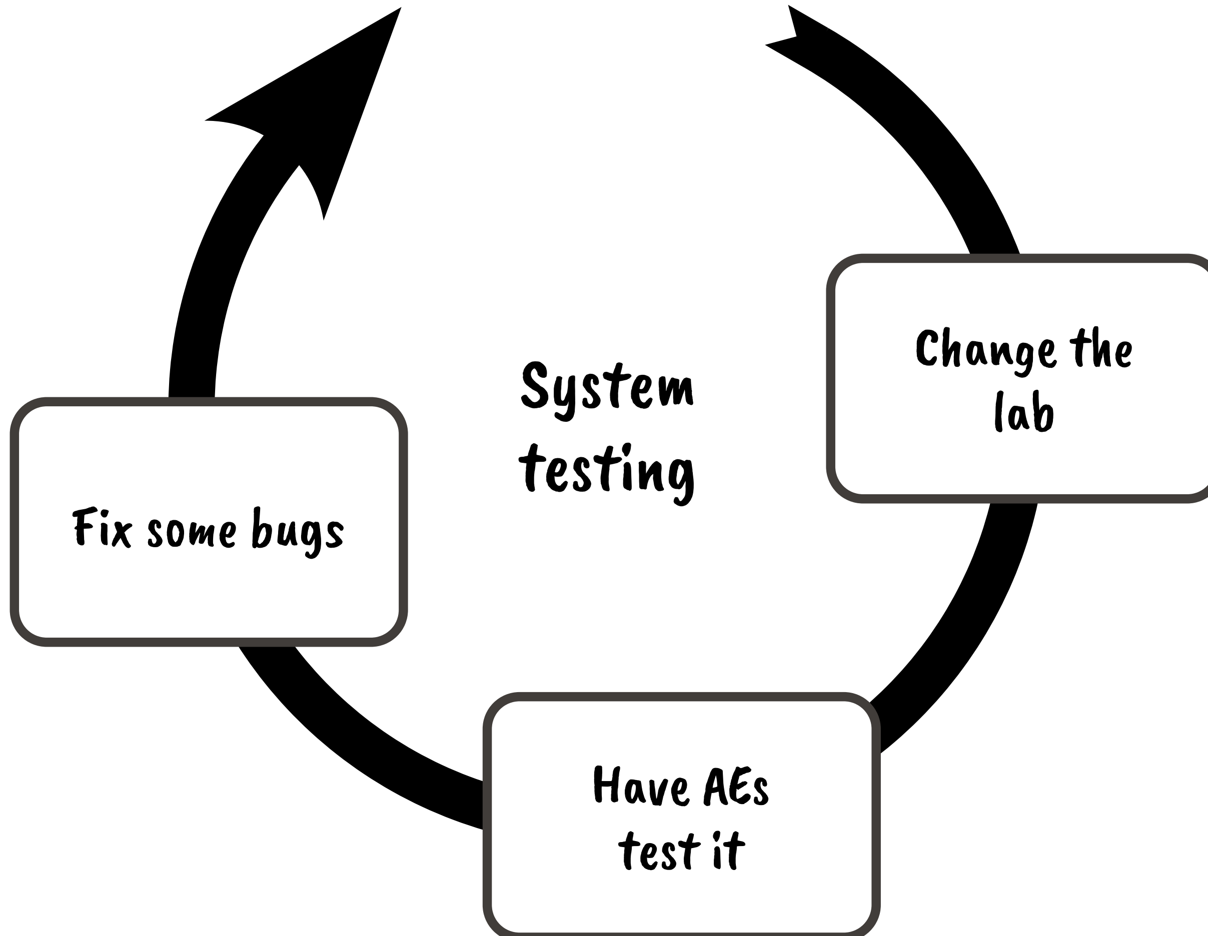
# ACCEPTANCE TESTING
## EXAMPLE

**Exercises: Recursion**

| | | Agree strongly | Agree somewhat | Neither agree nor disagree | Disagree somewhat | Disagree strongly |
|---|---|---|---|---|---|---|
| This exercise set provided adequate guidance | ◉ | ○ | ○ | ○ | ○ | ○ |
| This exercise set provided adequate opportunities to think on my own and explore | ◉ | ○ | ○ | ○ | ○ | ○ |
| Working on this exercise set helped me improve | ◉ | ○ | ○ | ○ | ○ | ○ |
| I would have liked additional easy exercises to help me get started | ◉ | ○ | ○ | ○ | ○ | ○ |
| I would have liked additional hard exercises to help me become an expert | ◉ | ○ | ○ | ○ | ○ | ○ |
| I completed all of the exercises in this set (reminder: you do not have to) | ◉ | ○ | ○ | ○ | ○ | ○ |

# System testing

## CLICK ON IT YOURSELF WITH A CHECKLIST



# System testing

## Example

### Release checklist

- ☐ Clone repo from `git@gitlab.epfl.ch:systemf/cs-214/boids-solution/`
- ☐ Check startup code:
  - `sbt compile` works
  - `sbt test` compiles but fails tests
- ☐ Check VSCode & worksheet:
  - Metal's `import build` passes
  - Worksheet is present in `src/`…
  - Worksheet runs

### Boids UI checklist

- ☐ Run `sbt run`.
- ☐ Navigate to `localhost:8080/`
- ☐ Select `12_twoBoidsAvoidanceXY`
- ☐ Press `Run` ⇒ boids move.
- ☐ Press `Pause` ⇒ boids pause.
- ☐ Press `Resume` ⇒ movement resumes from previous position.

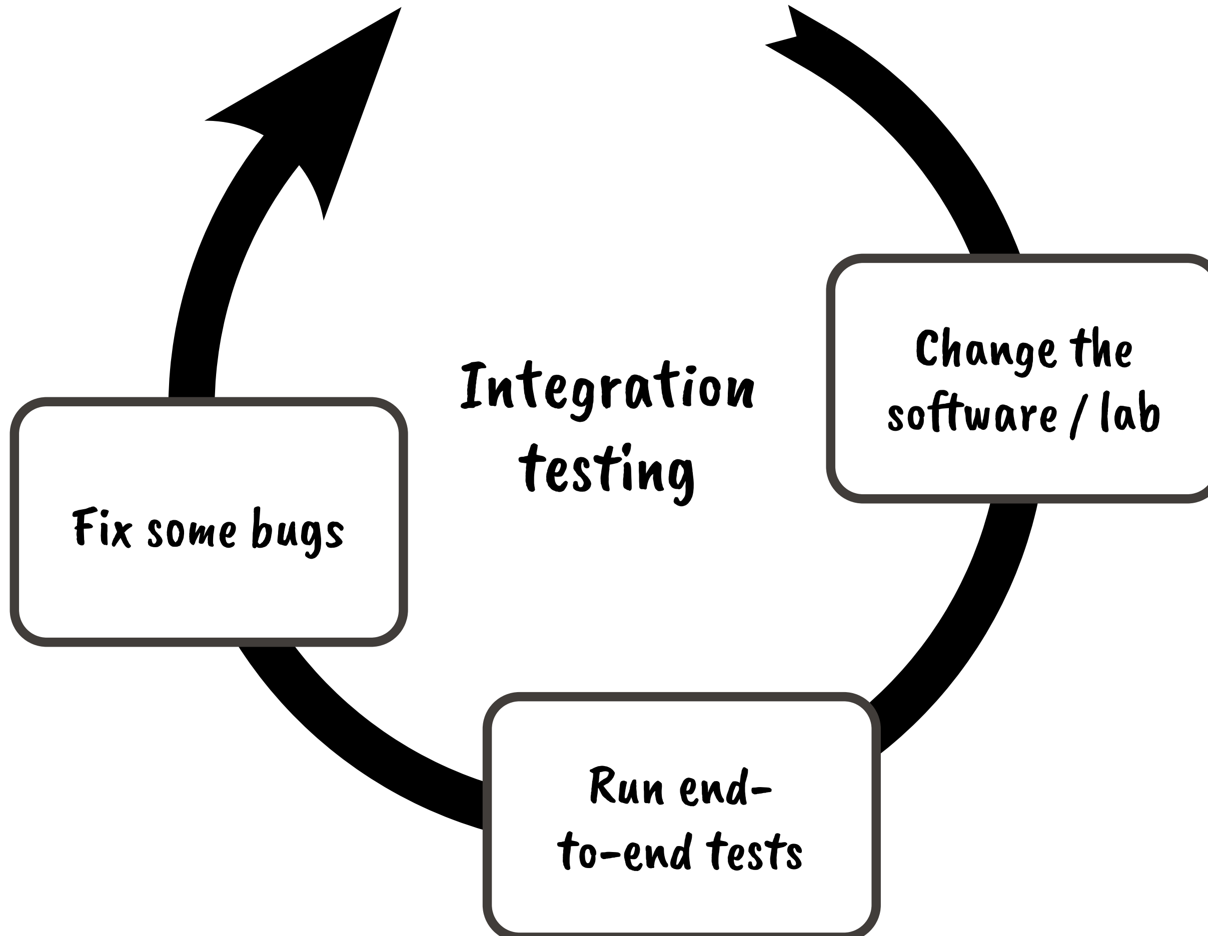
# Boids post-release issues

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

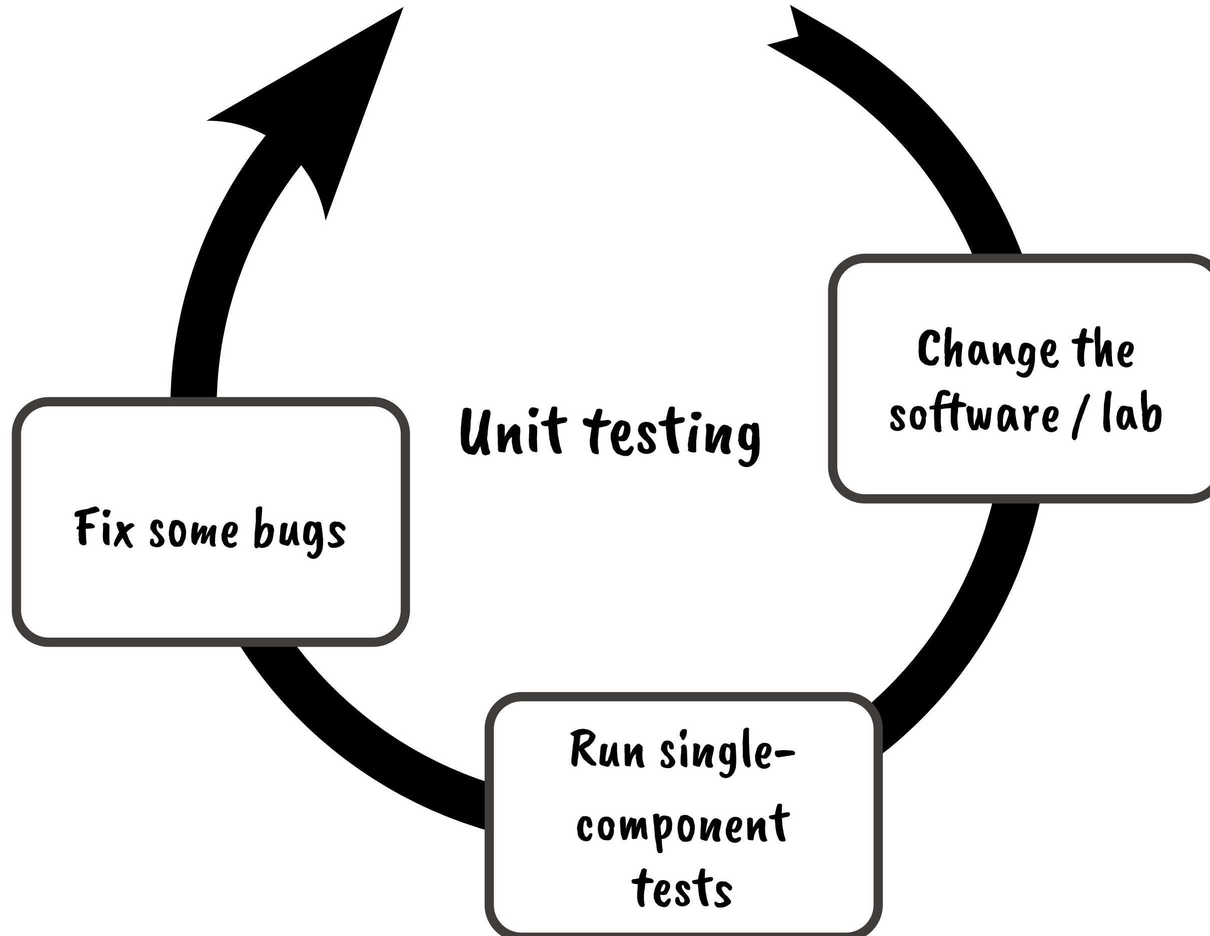
If you get any other reports from students during the lab session, please add them here before 5pm today.

# Integration testing
## TEST COMPLETE SUBSYSTEMS

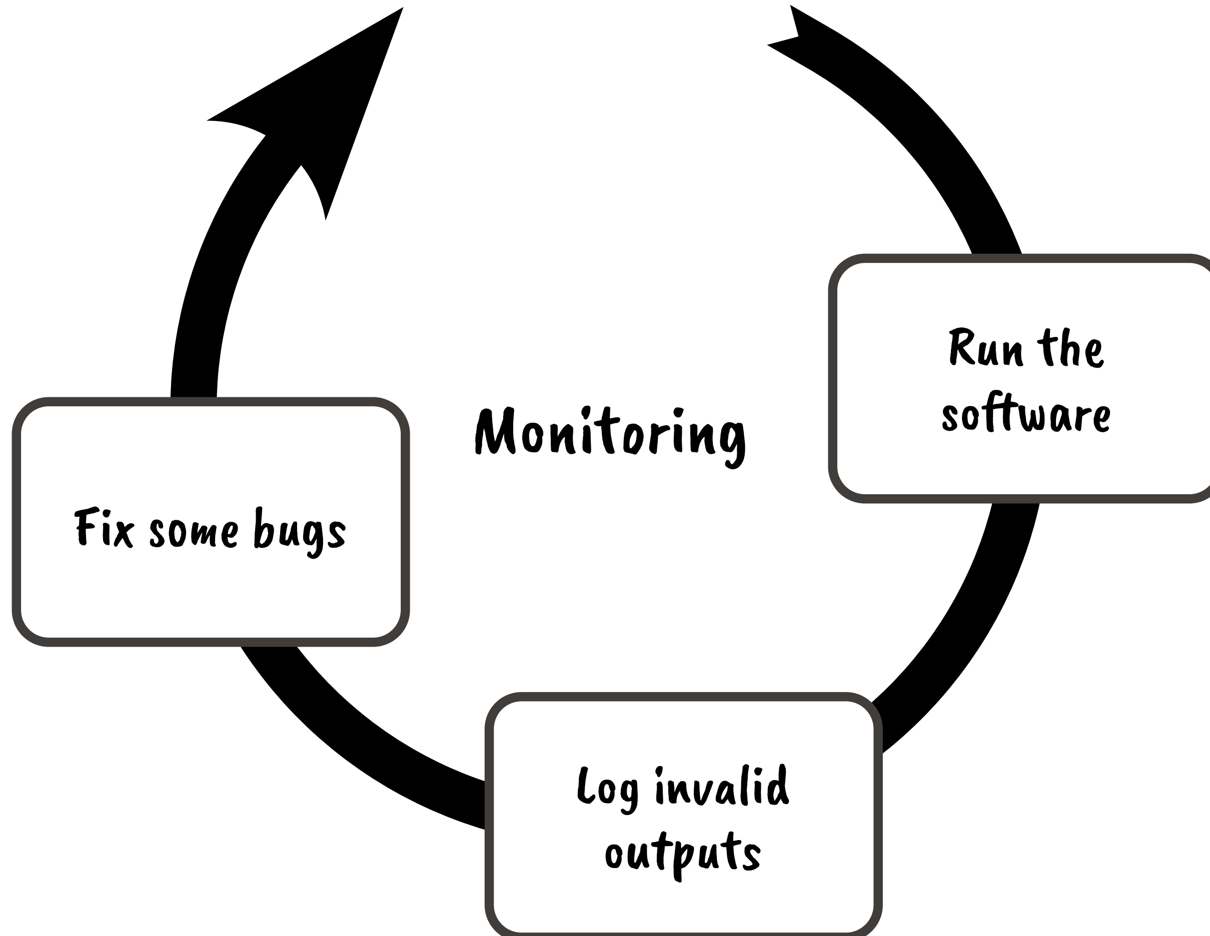


# Unit testing

## TEST INDIVIDUAL COMPONENTS



# Pre/post conditions
## MONITOR COMPONENTS AS THEY RUN



THE BIG IDEA OF MONITORING

# Use integration runs and real executions to test individual components

# ENSURING IS NOT MAGIC!

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

```
def f(in: I): O = {
            …
} ensuring Q
```

≡

```
def f(in: I): O = {
  val res = …
  assert Q(res)
  res
}
```

**Exercise:** 🔜 Is the postcondition on `sort` above complete?
Does it allow buggy implementations?

# REQUIRE AND ASSERT ARE NOT MAGIC!

```
def binarySearch(l: List[Int], x: Int): Int = {
  if !P then
    throw IllegalArgumentException(…)

  val idx = findInsertionPoint(l, idx)

  if !Q then
      throw AssertionError(…)

  idx < l.length && l(idx) == x
} ensuring (_ == l.contains(x))
```

```
def f(in: I): O =
  require(P)

  …
  assert Q

  …
```

≡

```
def f(in: I): O =
  if !P then
    throw IllegalArgumentException(…)
  …
  if !Q then
    throw AssertionError(…)
  …
```

# Monitors also provide documentation

✓ Better

```
def boidsWithinRadius(thisBoid: Boid,
                      boids: BoidSequence,
                      radius: Float) = {
  boids.filter(b ⇒
    b ≠ thisBoid && // Don't include 'thisBoid'
    b.position.distanceTo(thisBoid.position) < radius
  )
}.ensures(!_.contains(thisBoid)) // Document the exclusion
```

**Exercise:** Which postcondition would have caught the issue?

What should I add to `boidsWithinRadius`?

REVIEW

# What's the point of tests?

REVIEW

# What's 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 ?

EXERCISE

# TESTING A THEATER PLAY

Let’s assume you’re preparing a theater play. Name the tests:

- You rehearse your monologue in front of the mirror

⇒ Unit test

- You rehearse a dialogue scene in a classroom with another actor

⇒ Integration test

- You complete a full rehearsal in the actual theater

⇒ System test

- You perform at the premiere

⇒ Acceptance test

- The prompter backstage sees you struggle and feeds you lines

⇒ Monitoring