Automatic testing

Leveraging monitors and executable specifications for automatic bug finding

CS-214 - 3 Dec 2025

Clément Pit-Claudel

Quick announcements

Unguided lab checkoffs

Dec 15th, 16th, 19th.

Early check-offs on Dec 12th.

Debrief for week 11

Out now, with midterm tips

This week:

Automated testing

Learning objectives:

1. Leverage property-based testing to find bugs

2. Describe extensions and alternatives to PBT

- Testing + Specs recap
- Property-based testing
  - Formulating specs
  - Writing generators
- Beyond properties
  - Differential testing
  - Mutational fuzzing
  - Crash fuzzing
- Beyond generators
  - Black-box fuzzing
  - Grey-box fuzzing
  - White-box fuzzing

Automated testing with unit and integration tests

Tests

- Requirements
  - Acceptance tests
  - System tests
- **Functionality**
  - **Unit/Integration tests**

    Typically 1 input

Expectations

- Model based expectation

```
List(1,2,1).distinctWithHashMap
  == List(1,2)
List(1,3,2).quickSort
  == List(1,2,3)
```

- Axiomatic expectation

```
noDuplicates:
  List(1,2,1).distinctWithHashMap
isSorted:
  List(1,3,2).sort
```

Automated test = System under test + Input + Expectation

Exercise: Limitations of unit/integration tests?

- Writing tests is tedious and time consuming
- Basic tests crowd out interesting tests
- Tests are often incomplete: need to think of the right inputs!

Regression tests are easy.

Comprehensive tests are hard.

Automated testing with monitors

Tests

- Requirements
  - Acceptance tests
  - System tests
- **Functionality**
  - Unit/Integration tests

    Typically 1 input
  - **Monitors**

    Arbitrarily many inputs

Specs

- Model based spec

```
ls.distinctWithHashMap.ensuring: r⇒
  r == ls.distinct
ls.quickSort.ensuring: r⇒
  r == ls.sorted
```

- Axiomatic spec

```
ls.distinctWithHashMap.ensuring: r⇒
 noDuplicates(r)
ls.quickSort.ensuring: r⇒
  isSorted(r)
```

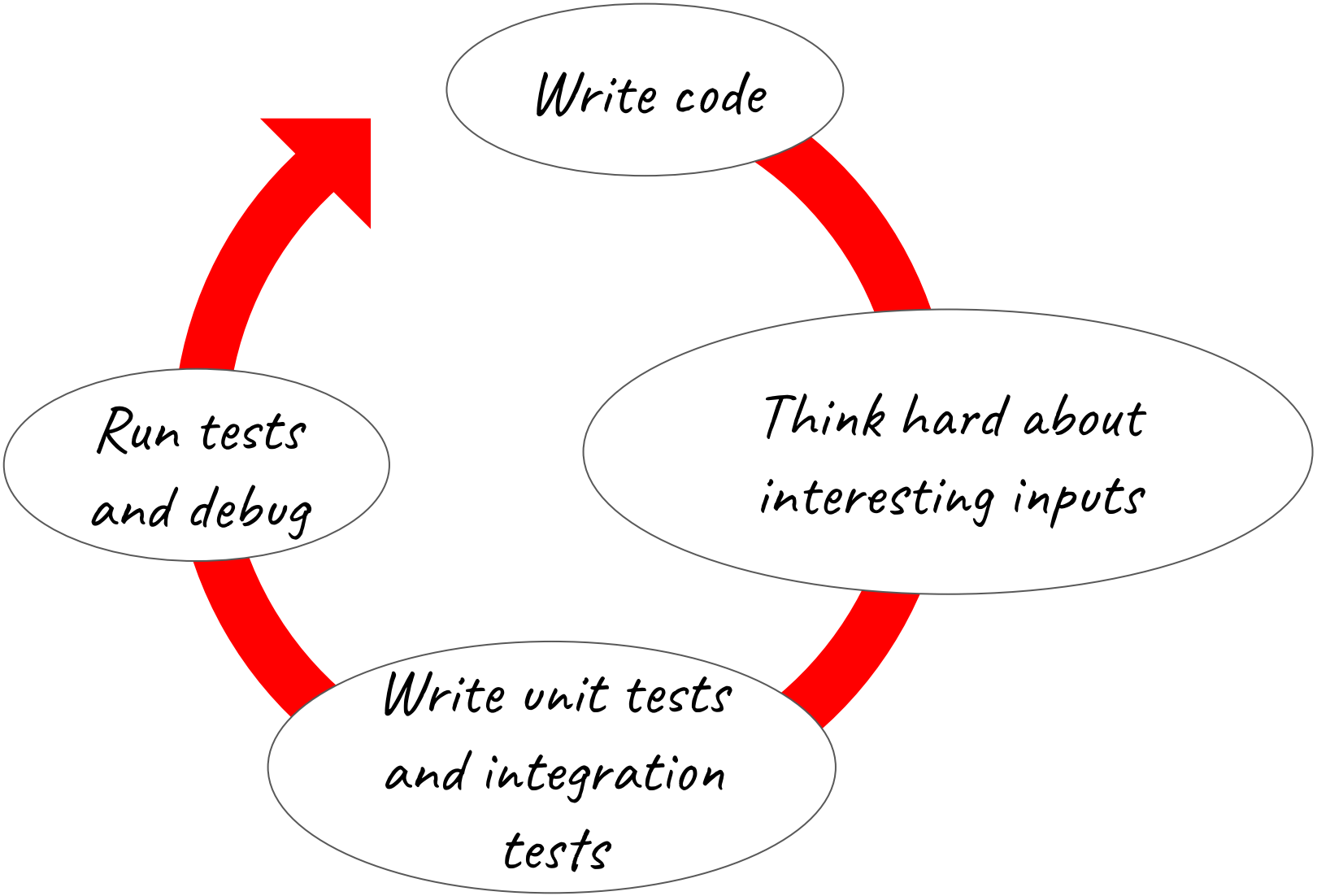
Big idea of monitoring:

Use integration runs and real executions to test individual components

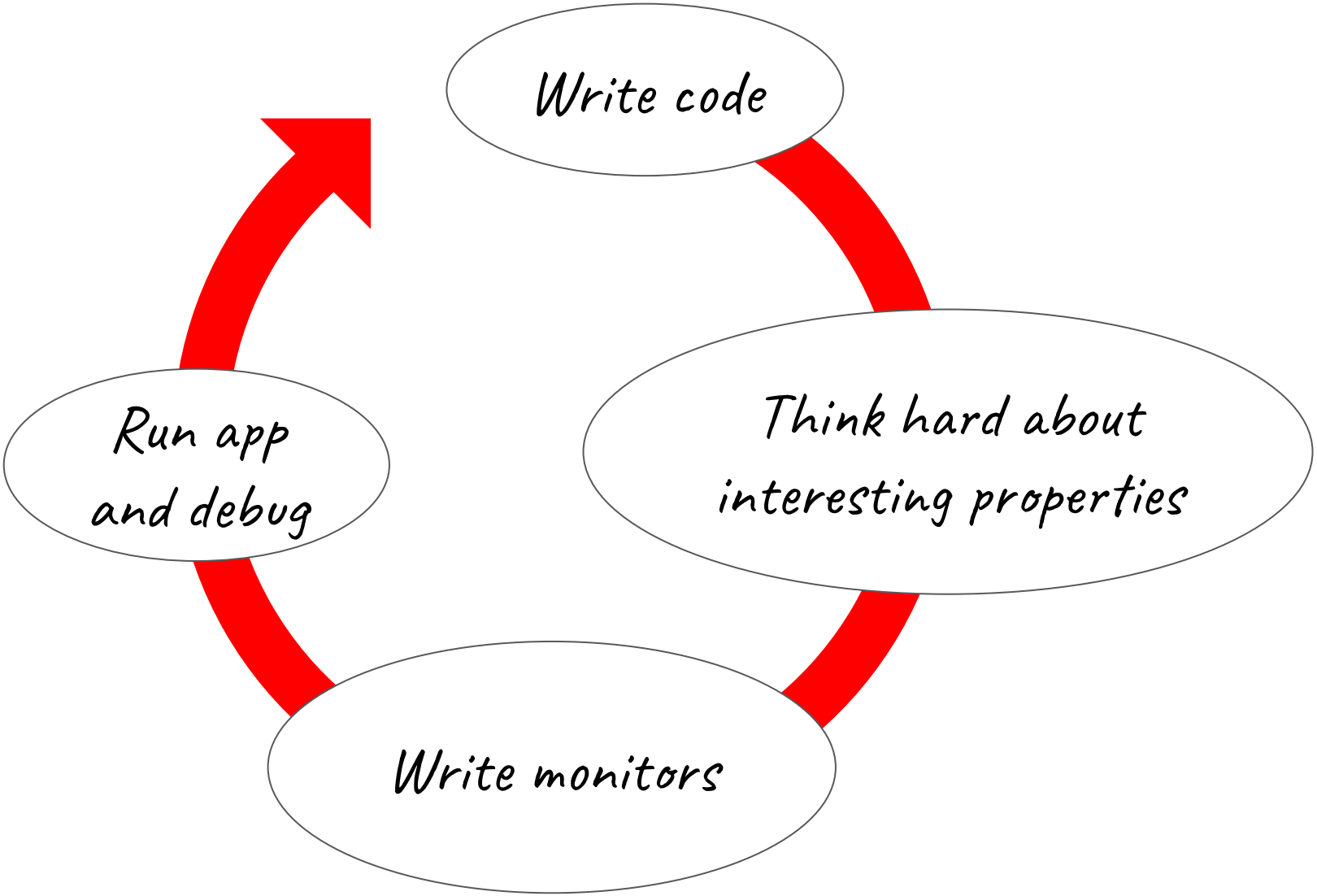
1 unit test = 1 input/output pair

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

Programming with unit tests and integration tests



Programming with monitors

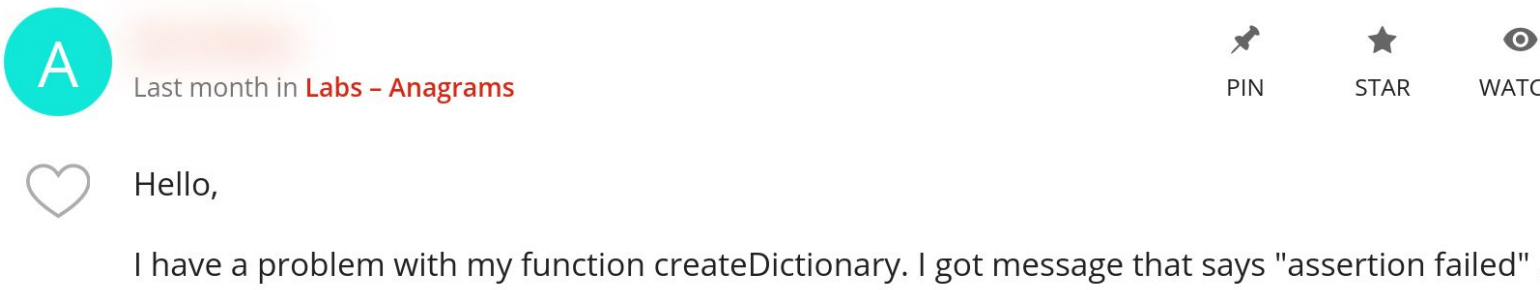


Monitoring detects errors… in the wild!

1: Write nice, monitored code

```scala
/** Removes diacritics and all non-alphabetic characters from `s`. */
  def normalizeString(str: String): String = {
    Normalizer.normalize(str, Normalizer.Form.NFD)
              .replaceAll("\p{InCombiningDiacriticalMarks}+", "")
              .replaceAll("[^a-zA-Z]+", "")
              .toLowerCase
  } ensuring (_.forall(c ⇒ 'a' ≤ c && c ≤ 'z'))
```

2: Woops, Assertion Failed for createDictionary #682



Two problems in this function

1. It's not pure

```scala
/** Removes diacritics and all non-alphabetic characters from `s`. */
  def normalizeString(str: String)(using locale: Locale): String = {
    Normalizer.normalize(str, Normalizer.Form.NFD)
              .replaceAll("\p{InCombiningDiacriticalMarks}+", "")
              .replaceAll("[^a-zA-Z]+", "")
              .toLowerCase(locale)
  } ensuring (_.forall(c ⇒ 'a' ≤ c && c ≤ 'z'))
```

2. It's not properly tested!

   Needs all locales and many strings

```scala
import java.util.Locale
Locale.setDefault(Locale.forLanguageTag("tr"));
"I".toLowerCase // : String = ı
"i".toUpperCase // : String = İ
```

Big idea of property-based testing:

Generate synthetic inputs to validate specifications

Demo

Scalacheck

```
forAll((x: Int) => x + 1 - 1 == x).check()

forAll { (l: List[Int]) =>
     l.reverse == l.foldLeft(Nil)((acc, x) => x :: acc)
}.check()

forAll { (l: List[Int]) =>
     l.reverse == l.foldRight(Nil)((x, acc) => x :: acc)
}.check()

forAll { (l: List[Int]) =>
     l.head :: l.tail == l
}.check()

forAll { (l: List[Int]) =>
     (l != Nil) ==> (l.head :: l.tail == l)
}.check()

forAll { (x: Int) =>
  (x != Int.MaxValue) ==> (x + 1 > x)
}.check()
```

Exercise: PBT for stateful code

State machines are pure!

- Input: Sequence of events
- Specs:
  - Model based (function of all events)
  - Axiomatic (property of the state)

Exercise: Test your state machines using ScalaCheck

(It has custom support for it!)

Beyond PBT: Getting rid of specs

- **Differential testing:** Use two SUTs (systems under test)

  Like model-based testing, but the model my be wrong:

  ```
  ls.quickSort == ls.mergeSort
  ```

- **Mutational testing:** Change inputs without changing output

  ```
  eval(e) == eval(Plus(e, 0)) == eval(Times(e, 1))
  ```

- **Crash testing:** Use “does not crash” as spec

  ```
  try { eval(e) } catch _ ⇒ “Test failed!”
  ```

Beyond PBT: Getting rid of input generators

- **Basic (black-box) fuzzing:** Explore bit patterns

  `main()` work with bytes, so no need for custom generators!

- **Instrumentation-guided (grey-box) fuzzing:** Maximize coverage

  Record program execution to find interesting inputs

- **Concolic (white-box) fuzzing:** Use symbolic execution

  Use logic solver to reverse-engineer interesting inputs