Evelyn Haslinger  
eh@symflower.com

Markus Zimmermann  
mz@symflower.com
Fantastic Bugs

... and Where to Find Them

Evelyn Haslinger  eh@symflower.com
Markus Zimmermann mz@symflower.com
Some Fantastic Bugs(/Problems)

- **Ariane 5 Flight 501** ($370.0M)
  - Reuse of code led to overflow 64bit->16bit

- **Mars Climate Orbiter** ($327.6M)
  - Expected different unit for metrics in one component

- **Heartbleed Bug** ($>500.0M)
  - Wrong bound-checking in kind of unused feature

- **Year 2000 problem** ($>300.0B < B as in billion!)
  - Huge part: software testing
The Problem

- Software has errors
- Testing is time-consuming
- Humans are inaccurate

Over 30% of software development time is consumed by quality assurance.

https://symflower.com
Even More Problems

- Worst-case: customer finds problems first
- Customers are usually really bad reporters
- Reproducing external problems is very hard
- Reproducing system problems is hard
- Fixing production problems is “tricky”
- Fixing problems often creates more problems
Solutions

1) Early detection
   - Find problems before they go into production
   - Find problems before they even go into staging

2) Automate the complete testing process
   - Machines scale, humans do not

3) (Very very) thorough testing
   - Only feasible on the unit level (?)
Implementation of Solution 1&2

- CI/CD (Continuous Integration/Deployment)
  - Build, static analysis, automated tests, ... per change
  - Automated deployments do not make mistakes
- “Testing” deployments
  - Find component/system problems on live environment
  - Copy “real” data to find outliers
  - Migrate deployment first to get ~real scenario
- CODE REVIEWS!
Problems with 2&3

- Am I testing the “right” things?
  - Specification-based testing?
(Java Code)

```java
static int compare(int a, int b) {
    int c = a - b;

    if (c < 0) {
        return -1;
    } else if (c > 0) {
        return 1;
    } else {
        return 0;
    }
}
```
(Java Code)

```java
static int compare(int a, int b) {
    int c = a - b;  // Overflow, e.g. with a=0, b=-2147483648 -> c=-2147483648

    if (c < 0) {
        return -1;
    } else if (c > 0) {
        return 1;
    } else {
        return 0;
    }
}
```
Problems with 2&3

- Am I testing the “right” things?
  - Specification-based testing?
    (not enough, e.g. implementation diverges)
Problems with 2&3

• Am I testing the “right” things?
  • Specification-based testing?
    (not enough, e.g. implementation diverges)

• What is “thorough testing” anyway?
  • **MC/DC** coverage?
(Java Code)

```java
public static int division(int x, int y) {
    return x / y;
}
```
public static int division(int x, int y) {
    return x / y;  // y=0 leads to "division by zero"
}
public static int division(int x, int y) {
    return x / y;  // y=0 leads to "division by zero"
}

... and there is an overflow:

x=-2147483648, y=-1 -> -2147483648
(Java Code)

```java
public static boolean isSorted(int[] a) {
    int i = 0;
    while (i < a.length - 1 && a[i] <= a[i + 1]) {
        i = i + 1;
    }

    return i == a.length - 1;
}
```
public static boolean isSorted(int[] a) {
    int i = 0;
    while (i < a.length - 1 && a[i] <= a[i + 1]) {
        i = i + 1;
    }
    return i == a.length - 1;
}
Problems with 2&3

- Am I testing the “right” things?
  - Specification-based testing?
    (not enough, e.g. implementation diverges)

- What is “thorough testing” anyway?
  - **MC/DC** coverage?
    (not enough, e.g. runtime errors)
Problems with 2&3

- Am I testing the “right” things?
  - Specification-based testing?
    (not enough, e.g. implementation diverges)
- What is “thorough testing” anyway?
  - **MC/DC** coverage?
    (not enough, e.g. runtime errors)
  - When do I know that I have tested enough?
Problems with 2&3

- Am I testing the “right” things?
  - Specification-based testing?
    (not enough, e.g. implementation diverges)

- What is “thorough testing” anyway?
  - MC/DC coverage?
    (not enough, e.g. runtime errors)
  - When do I know that I have tested enough?

- How to automate not just the test execution?
Suggestions to Approximate 2&3

• “Mutation Testing” for existing tests
  • Check if the whole implementation is really covered
  • E.g. https://github.com/zimmski/go-mutesting

• Mold implementation into test cases
  • One test case for every “interesting” path
  • Specification can then be checked with all cases

• Find test cases
  • E.g. Fuzzing+MBT or better: Symflower

https://symflower.com
We Are Hiring!

• We offer
  • challenging algorithmic tasks to work on
  • state of the art development processes and tools
  • a work environment that you can shape with us

• Talk to me, or Evelyn after the lecture
• Drop us an email with your CV at you@symflower.com