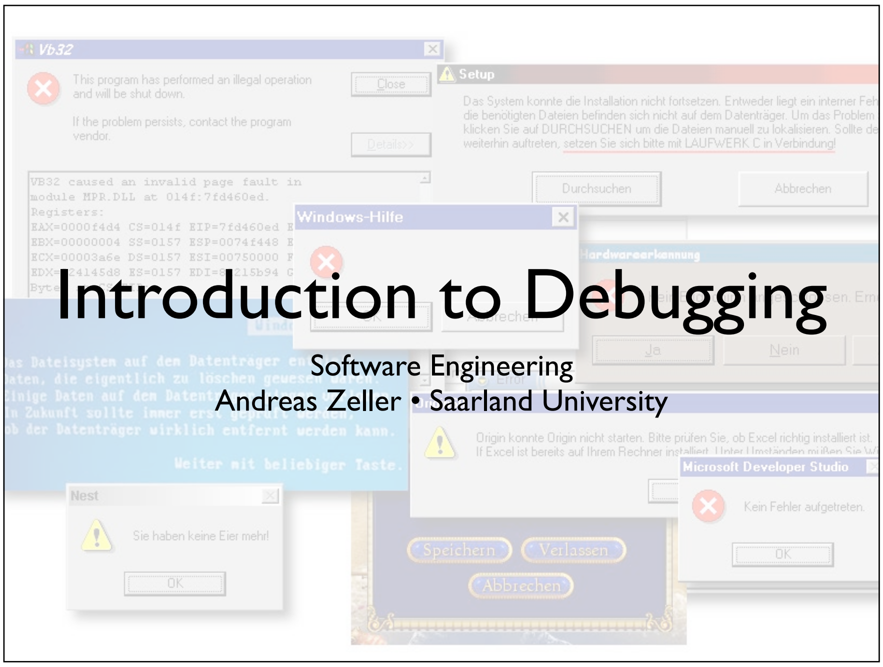
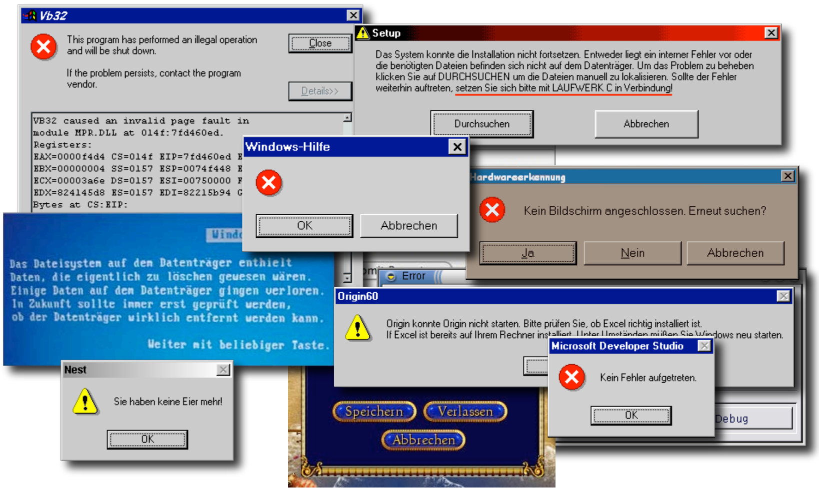


Introduction to Debugging

Software Engineering

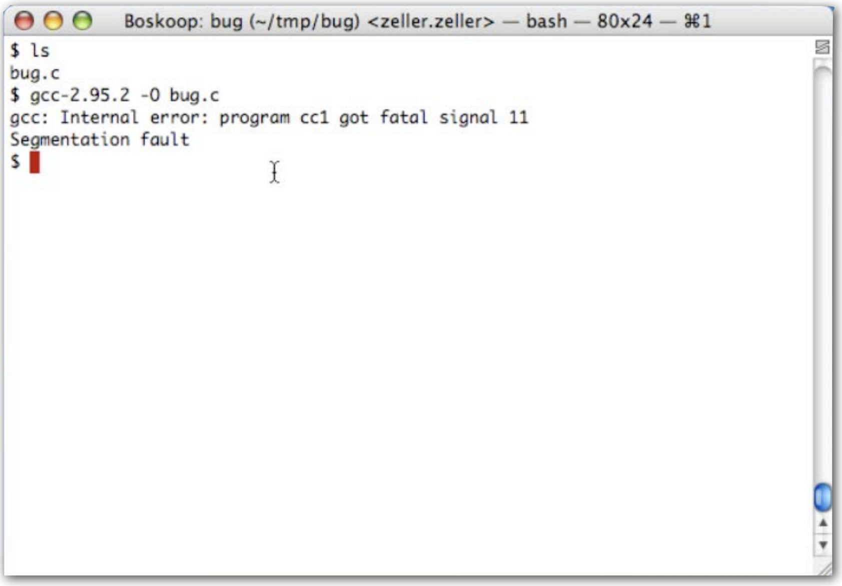
Andreas Zeller • Saarland University

The Problem



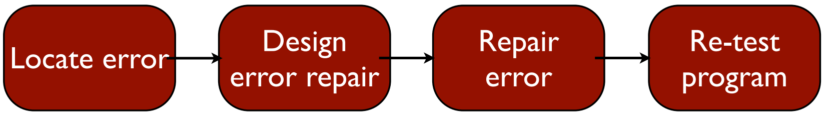
Facts on Debugging

- Software bugs cost ~60 bln US$/yr in US
- Improvements could reduce cost by 30%
- Validation (including debugging) can easily take up to 50-75% of the development time
- When debugging, some people are *three times* as efficient than others



How to Debug

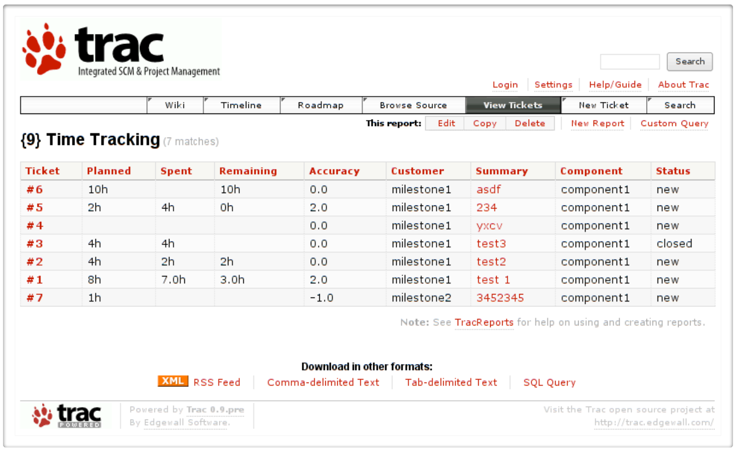
(Sommerville 2004)



The Process

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect

Tracking Problems

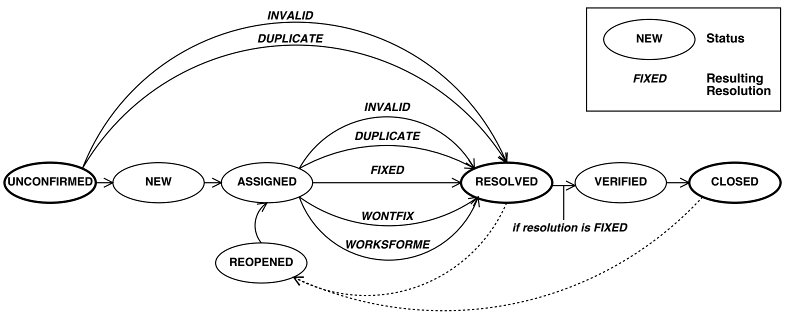


Ticket	Planned	Spent	Remaining	Accuracy	Customer	Summary	Component	Status
#6	10h		10h	0.0	milestone1	asdf	component1	new
#5	2h	4h	0h	2.0	milestone1	234	component1	new
#4				0.0	milestone1	yxcv	component1	new
#3	4h	4h		0.0	milestone1	test3	component1	closed
#2	4h	2h	2h	0.0	milestone1	test2	component1	new
#1	8h	7.0h	3.0h	2.0	milestone1	test 1	component1	new
#7	1h			-1.0	milestone2	3452345	component1	new

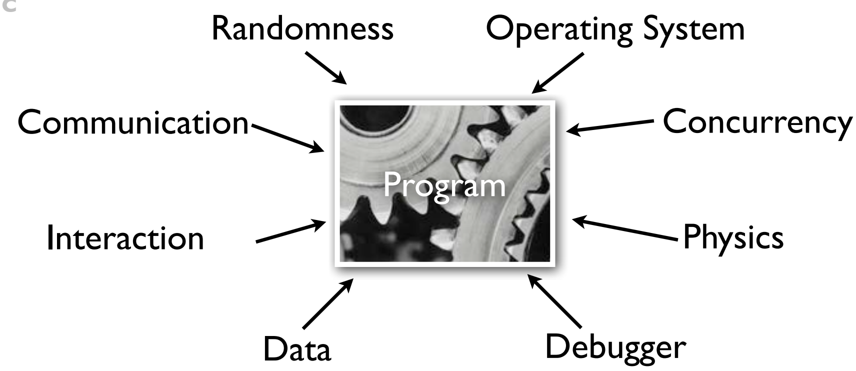
Tracking Problems

- Every problem gets entered into a *problem database*
- The *priority* determines which problem is handled next
- The product is ready when all problems are resolved

Problem Life Cycle



Reproduce



Automate

```
// Test for host
public void testHost() {
  int noPort = -1;
  assertEquals(askigor_url.getHost(), "www.askigor.org");
  assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
  assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
  assertEquals(askigor_url.getQuery(), "id=sample");
}
```

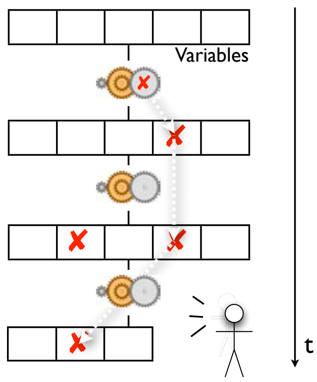

Automate

- Every problem should be *reproducible automatically*
- Achieved via appropriate (unit) tests
- After each change, we re-run the tests

Finding Origins

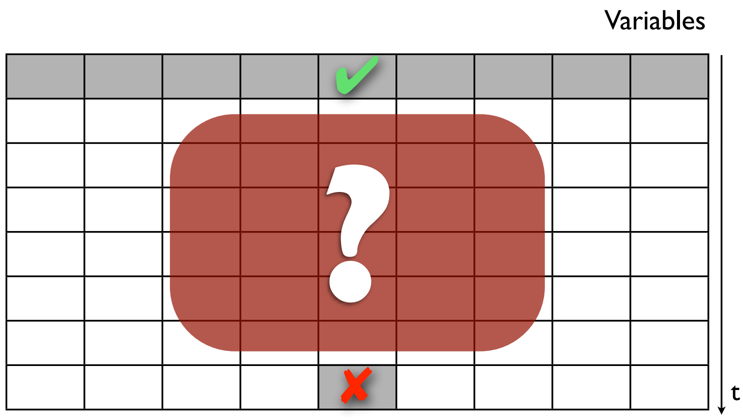
1. The programmer creates a *defect* in the code.
2. When executed, the defect creates an *infection*.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

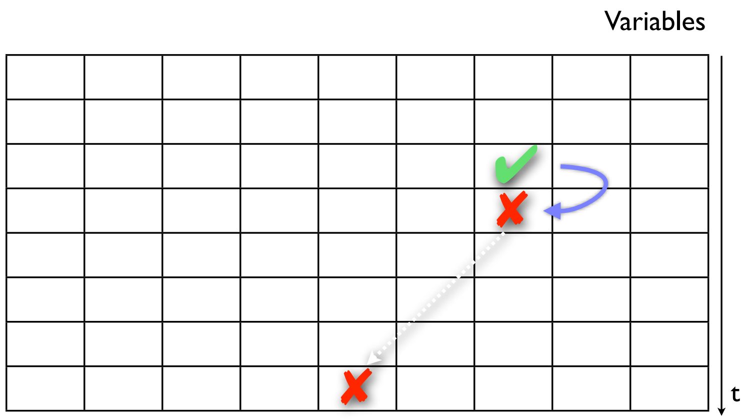


Not every defect creates an infection – not every infection results in a failure

Finding Origins

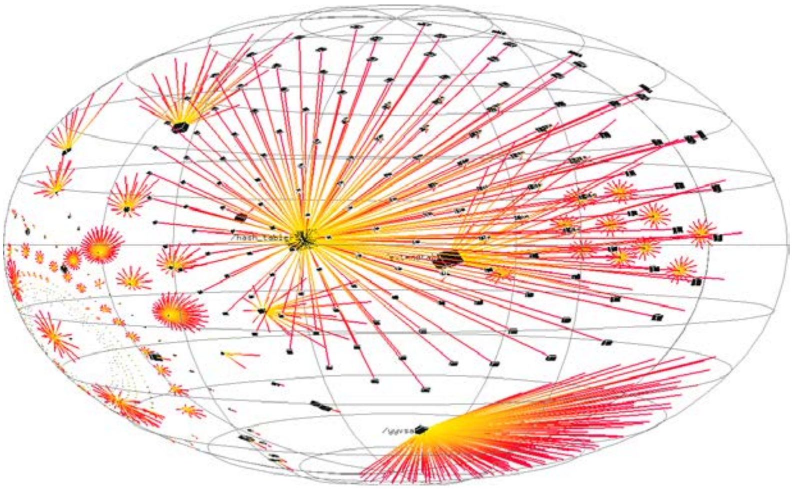


The Defect



T
R
A
F
F
I
C

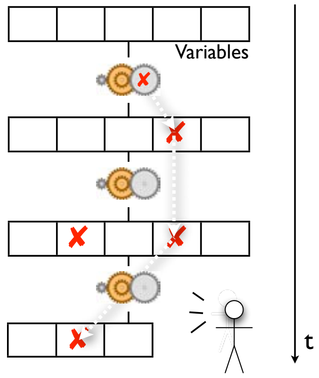
A Program State



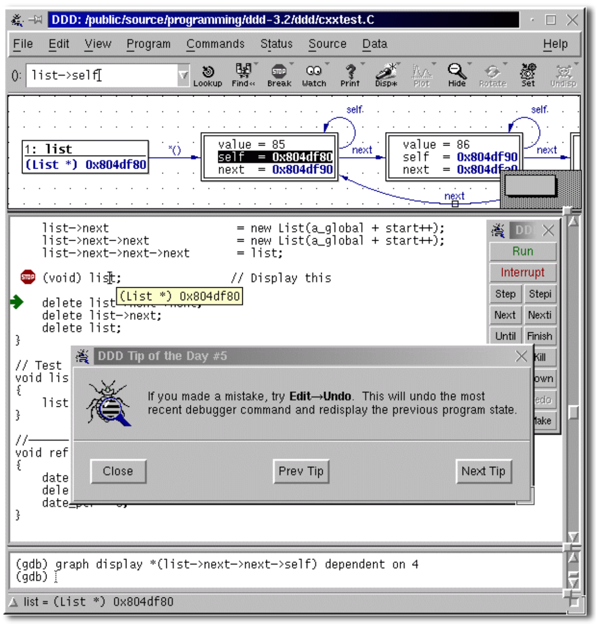
T
R
A
F
F
I
C

Finding Origins

1. We start with a *known infection* (say, at the failure)
2. We search the infection in the *previous state*

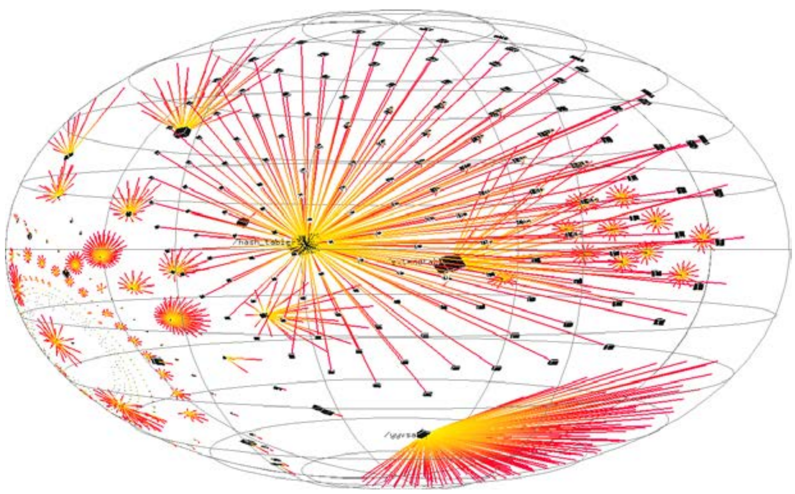


TRAFFIC

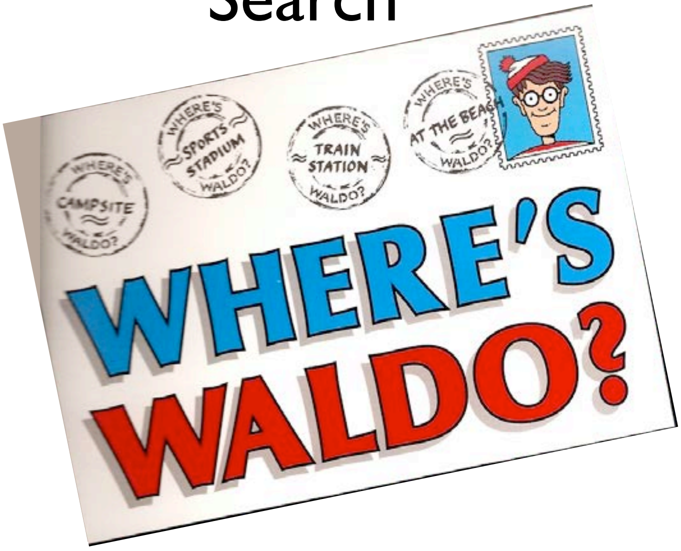


TRAFFIC

A Program State



Search



Focus

During our search for infection, we focus upon locations that

- *are possibly wrong*
  (e.g., because they were buggy before)
- *are explicitly wrong*
  (e.g., because they violate an *assertion*)

Assertions are the best way to find infections!

T R A F F I C

Finding Infections

```
class Time {
public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (incl. leap seconds)

    void set_hour(int h);
    …
}
```

Every time between 00:00:00 and 23:59:60 is valid

T R A F F I C

Finding Origins

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());  // Precondition
    …
    assert (sane());  // Postcondition
}
```

T R A F F I C

Finding Origins

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

sane() is the *invariant* of a Time object:

- valid *before* every public method
- valid *after* every public method

Finding Origins

- Precondition fails = Infection *before* method
- Postcondition fails = Infection *after* method
- All assertions pass = no infection

```
void Time::set_hour(int h)
{
    assert (sane());  // Precondition
    …
    assert (sane());  // Postcondition
}
```

T R A F F I C

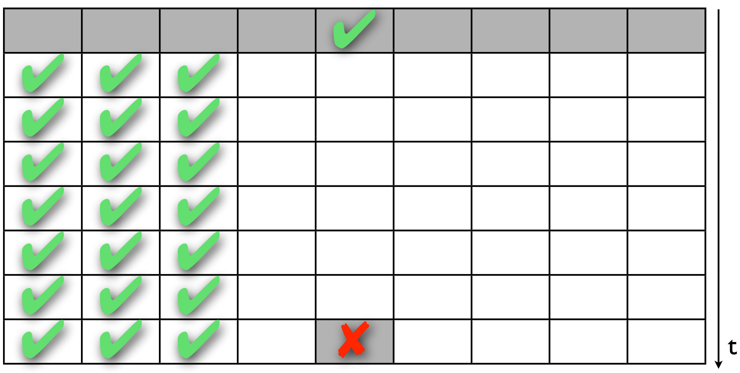
Complex Invariants

```
class RedBlackTree {
  …
  boolean sane() {
      assert (rootHasNoParent());
      assert (rootIsBlack());
      assert (redNodesHaveOnlyBlackChildren());
      assert (equalNumberOfBlackNodesOnSubtrees());
      assert (treeIsAcyclic());
      assert (parentsAreConsistent());

      return true;
  }
}
```

T R A F F I C

Assertions



Focusing

- All possible influences must be checked
- Focusing on most likely candidates
- Assertions help in finding infections fast

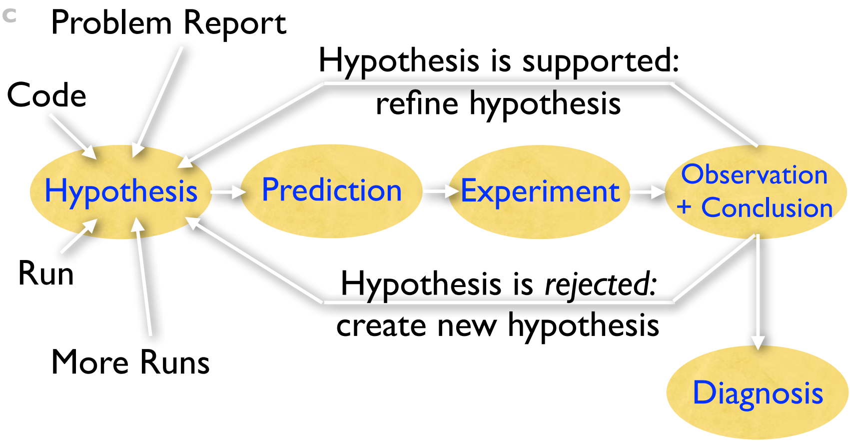
Isolation

- Failure causes should be *narrowed down systematically*
- Use *observation* and *experiments*

Scientific Method

1. Observe some aspect of the universe.
2. Invent a *hypothesis* that is consistent with the observation.
3. Use the hypothesis to make *predictions*.
4. Tests the predictions by experiments or observations and modify the hypothesis.
5. Repeat 3 and 4 to refine the hypothesis.

Scientific Method



TRAFFIC

TRAFFIC

Explicit Hypotheses

Hypothesis	The exec… causes a[0] = 0
Prediction	At L… should hold.
Experiment	… Line 37.
Observati…	…olds as predicted.
Conclusion	…hesis is confirmed.

Keeping everything in memory is like playing mastermind blind!

Explicit Hypotheses

T
R
A
F
F
I
C

Isolate

- We repeat the search for infection origins until we found the defect
- We proceed *systematically* along the scientific method
- *Explicit steps* guide the search – and make it repeatable at any time

T
R
A
F
F
I
C

Correction

Before correcting the defect, we must check whether the defect

- actually is an *error* and
- *causes* the failure

Only when we understood both, can we correct the defect

T
R
A
F
F
I
C

TRAFFIC

The Devil's Guide to Debugging

Find the defect by guessing:

- Scatter debugging statements everywhere
- Try changing code until something works
- Don't back up old versions of the code
- Don't bother understanding what the program should do

TRAFFIC

The Devil's Guide to Debugging

Don't waste time understanding the problem.

- Most problems are trivial, anyway.

TRAFFIC

The Devil's Guide to Debugging

Use the most obvious fix.

- Just fix what you see:

```
x = compute(y)
// compute(17) is wrong – fix it
if (y == 17)
    x = 25.15
```

Why bother going into compute()?

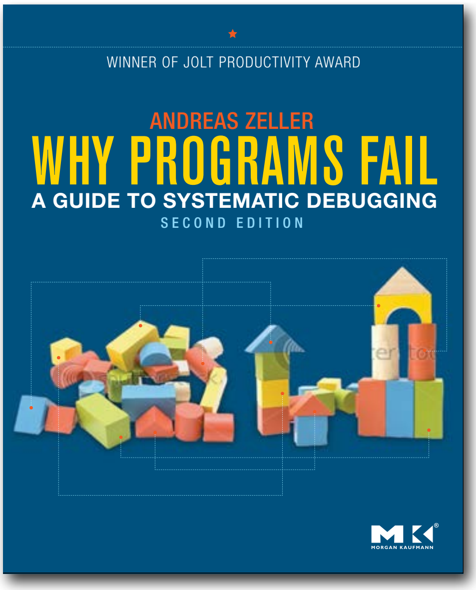
Successful Correction

Homework

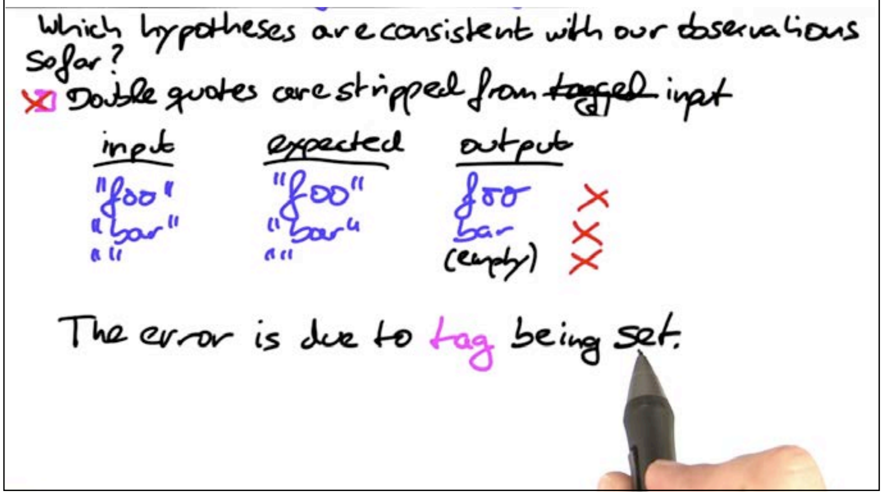
- Does the failure no longer occur?
  (If it does still occur, this should come as a big surprise)
- Did the correction introduce new problems?
- Was the same mistake made elsewhere?
- Did I commit the change to version control and problem tracking?

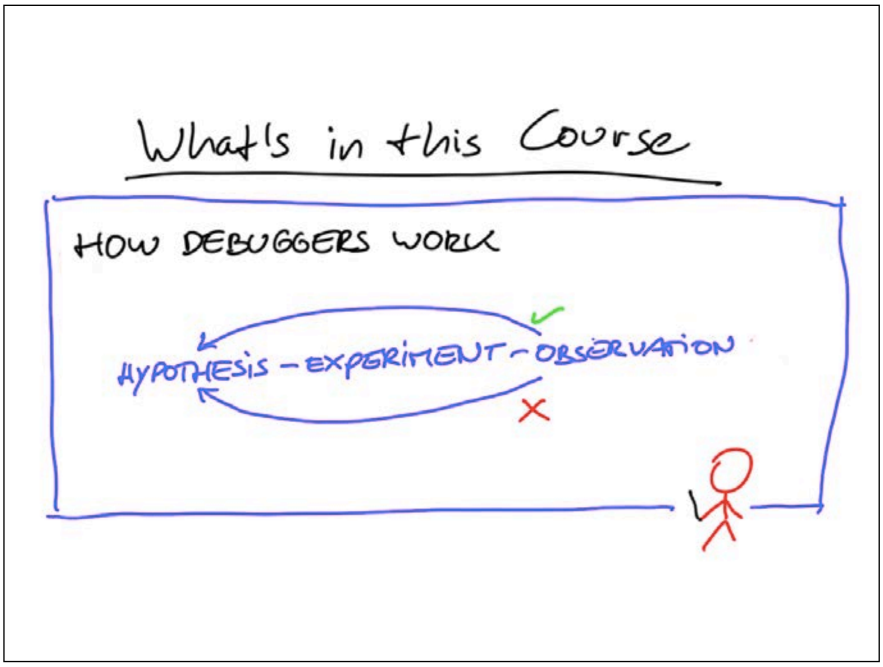
The Process

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect



Online Course on Debugging





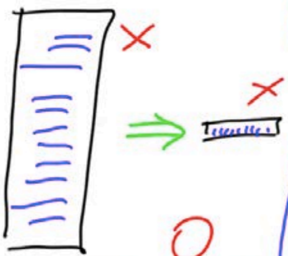
What's in this Course

HOW DEBUGGERS WORK
ASSERTING EXPECTATIONS

```
assert tree.hasNoCycles()
```

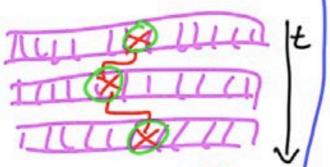
What's in this Course

HOW DEBUGGERS WORK
ASSERTING EXPECTATIONS
SIMPLIFYING FAILURES



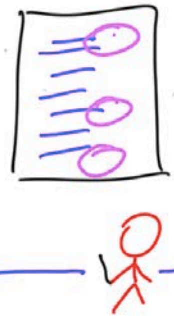
What's in this Course

HOW DEBUGGERS WORK
ASSERTING EXPECTATIONS
SIMPLIFYING FAILURES
TRACKING ORIGINS



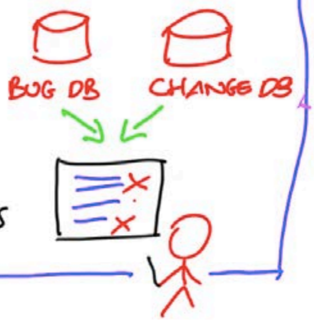
What's in this Course

- HOW DEBUGGERS WORK
- ASSERTING EXPECTATIONS
- SIMPLIFYING FAILURES
- TRACKING ORIGINS
- REPRODUCING FAILURES



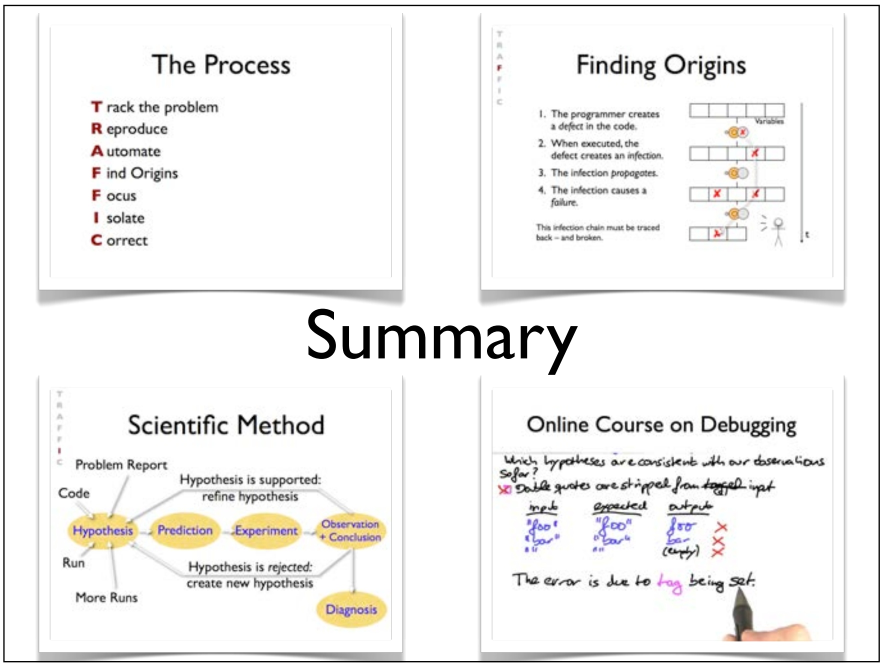
What's in this Course

- HOW DEBUGGERS WORK
- ASSERTING EXPECTATIONS
- SIMPLIFYING FAILURES
- TRACKING ORIGINS
- REPRODUCING FAILURES
- LEARNING FROM MISTAKES



What's in this Course

- HOW DEBUGGERS WORK
- ASSERTING EXPECTATIONS
- SIMPLIFYING FAILURES
- TRACKING ORIGINS
- REPRODUCING FAILURES
- LEARNING FROM MISTAKES



Summary
The Process
T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
Finding Origins
1. The programmer creates a defect in the code.
2. When executed, the defect creates an infection.
3. The infection propagates.
4. The infection causes a failure.
This infection chain must be traced back – and broken.
Variables
Scientific Method
Problem Report
Code
Run
More Runs
Hypothesis
Prediction
Experiment
Observation + Conclusion
Diagnosis
Hypothesis is supported: refine hypothesis
Hypothesis is rejected: create new hypothesis
Online Course on Debugging
The error is due to tag being set.