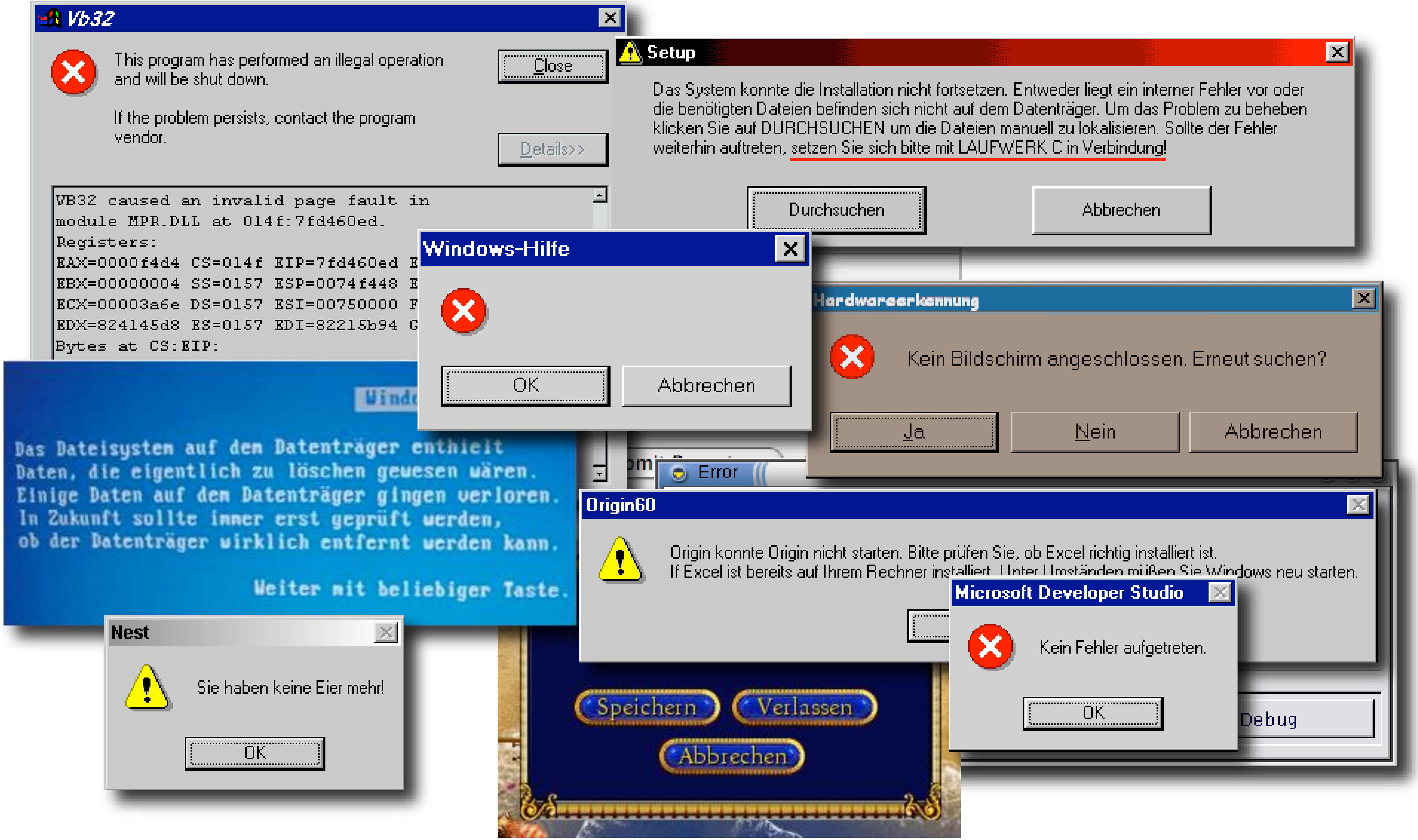
**F. Tip and**
**M. Weintraub**

# DEBUGGING

*Thanks go to Andreas Zeller for allowing incorporation of his materials*

Northeastern University
College of Computer and Information Science

# THE PROBLEM



This program has performed an illegal operation and will be shut down.

If the problem persists, contact the program vendor.

```
VB32 caused an invalid page fault in
module MPR.DLL at 014f:7fd460ed.
Registers:
EAX=0000f4d4 CS=014f EIP=7fd460ed E
EBX=00000004 SS=0157 ESP=0074f448 E
ECX=00003a6e DS=0157 ESI=00750000 F
EDX=824145d8 ES=0157 EDI=82215b94 G
Bytes at CS:EIP:
```

Das System konnte die Installation nicht fortsetzen. Entweder liegt ein interner Fehler vor oder die benötigten Dateien befinden sich nicht auf dem Datenträger. Um das Problem zu beheben klicken Sie auf DURCHSUCHEN um die Dateien manuell zu lokalisieren. Sollte der Fehler weiterhin auftreten, setzen Sie sich bitte mit LAUFWERK C in Verbindung!

Kein Bildschirm angeschlossen. Erneut suchen?

Das Dateisystem auf dem Datenträger enthielt
Daten, die eigentlich zu löschen gewesen wären.
Einige Daten auf dem Datenträger gingen verloren.
In Zukunft sollte immer erst geprüft werden,
ob der Datenträger wirklich entfernt werden kann.

Weiter mit beliebiger Taste.

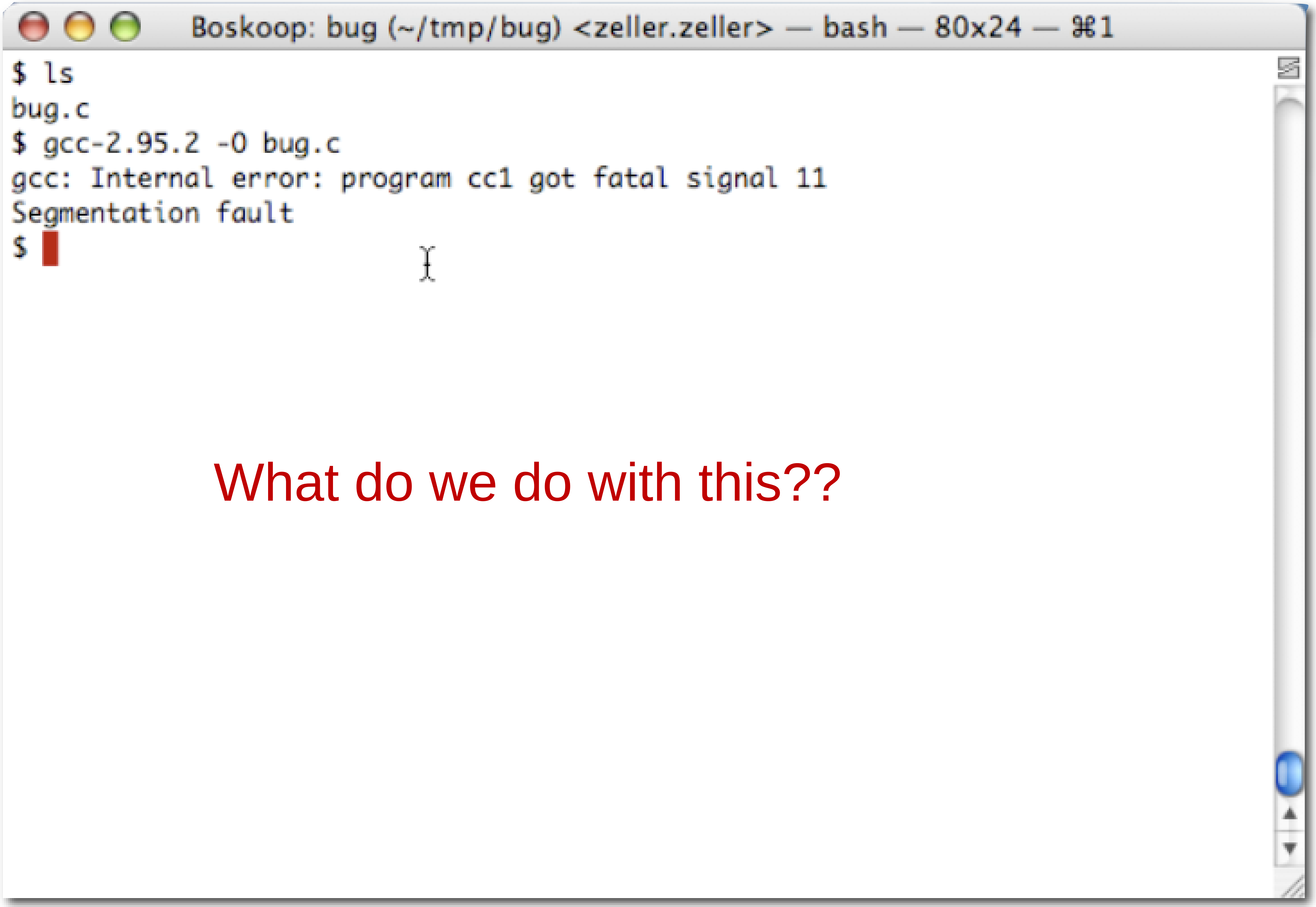
Sie haben keine Eier mehr!

Origin konnte Origin nicht starten. Bitte prüfen Sie, ob Excel richtig installiert ist.
If Excel ist bereits auf Ihrem Rechner installiert. Unter Umständen müßen Sie Windows neu starten.

Kein Fehler aufgetreten.

# FACTS ON DEBUGGING

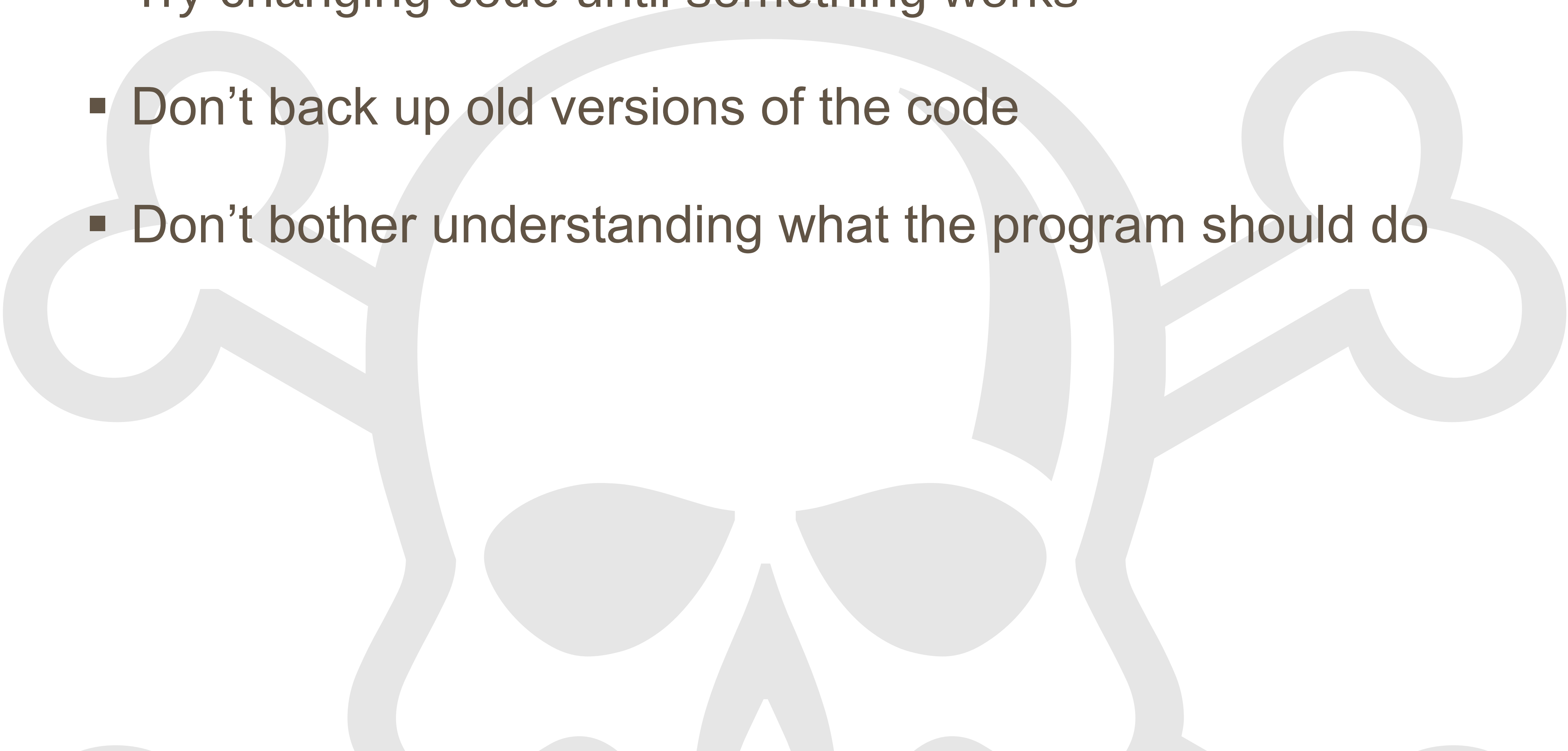
- Software bugs cost ~$60B per year in US
- Improvements could reduce cost by 30%
- Validation (including debugging) can easily eat up to 50-75% of the development time
- When debugging, some people are *three times* as efficient than others



```
$ ls
bug.c
$ gcc-2.95.2 -O bug.c
gcc: Internal error: program cc1 got fatal signal 11
Segmentation fault
$ 
```

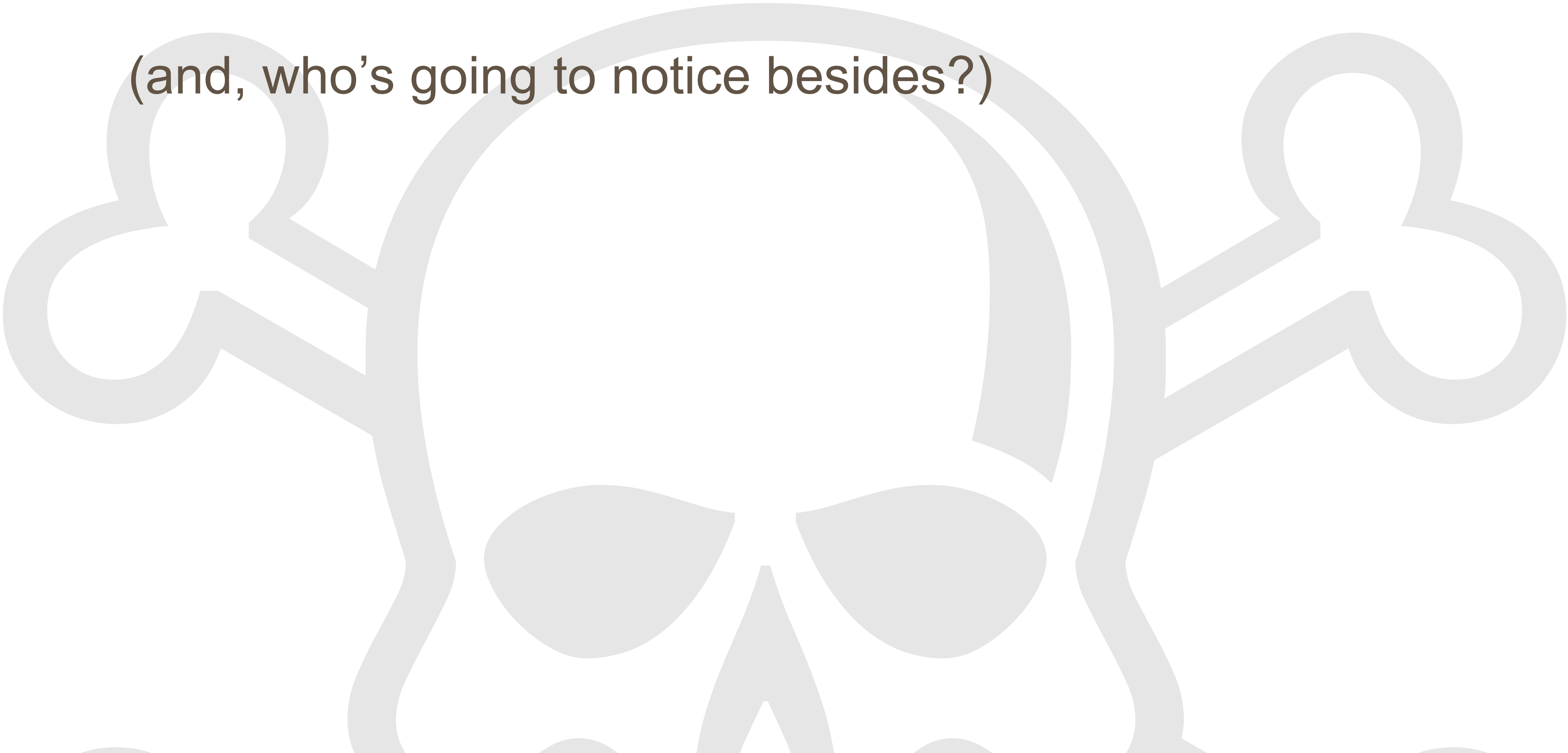
What do we do with this??

# THE DEVIL’S GUIDE TO DEBUGGING (OR, SADLY, HOW MANY APPROACH THE PROBLEM)



- 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

# THE DEVIL’S GUIDE TO DEBUGGING

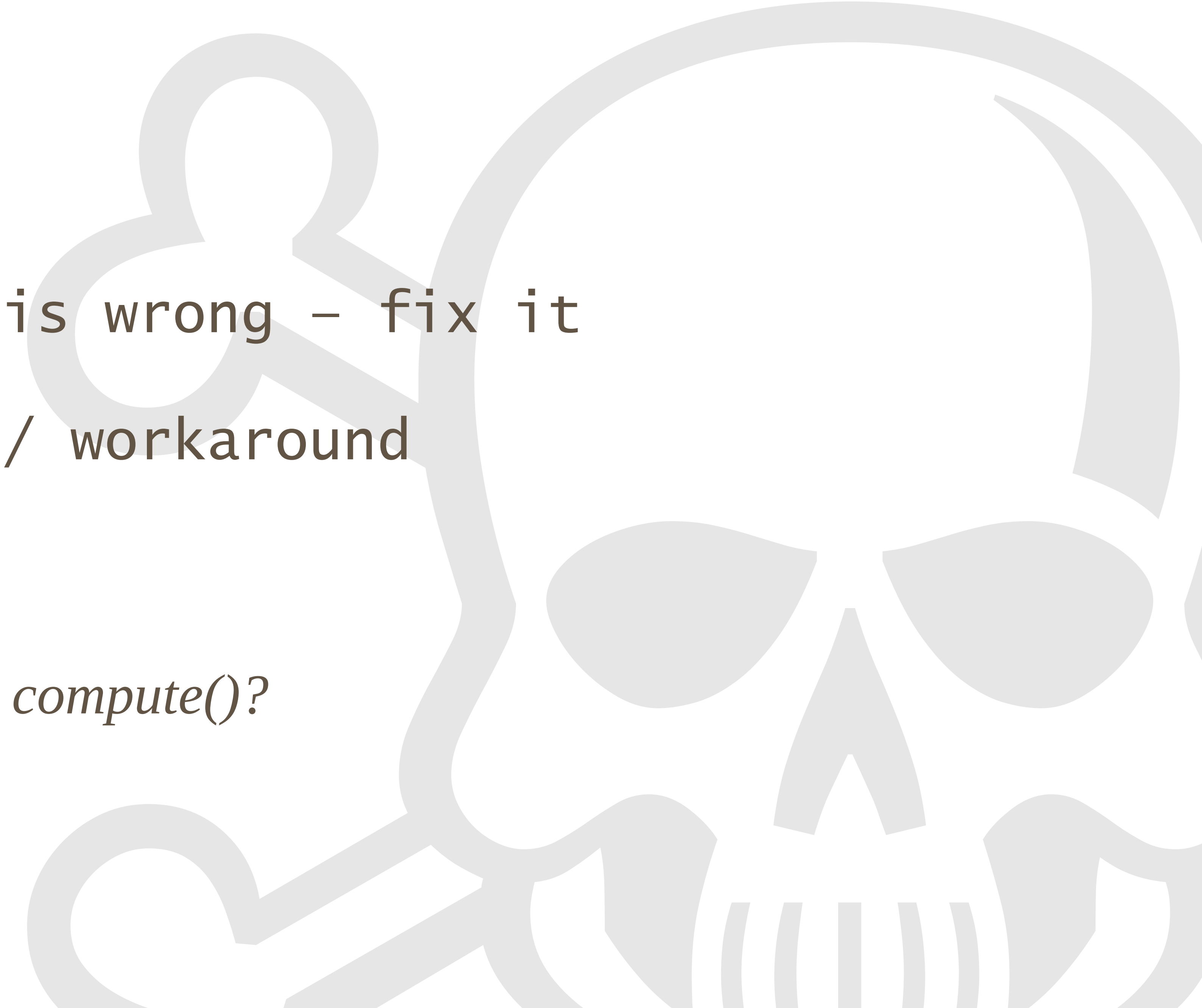


Don’t waste time understanding the problem.

Most problems are trivial, anyway.

(and, who’s going to notice besides?)

# THE DEVIL’S GUIDE TO DEBUGGING



Use the most obvious fix.

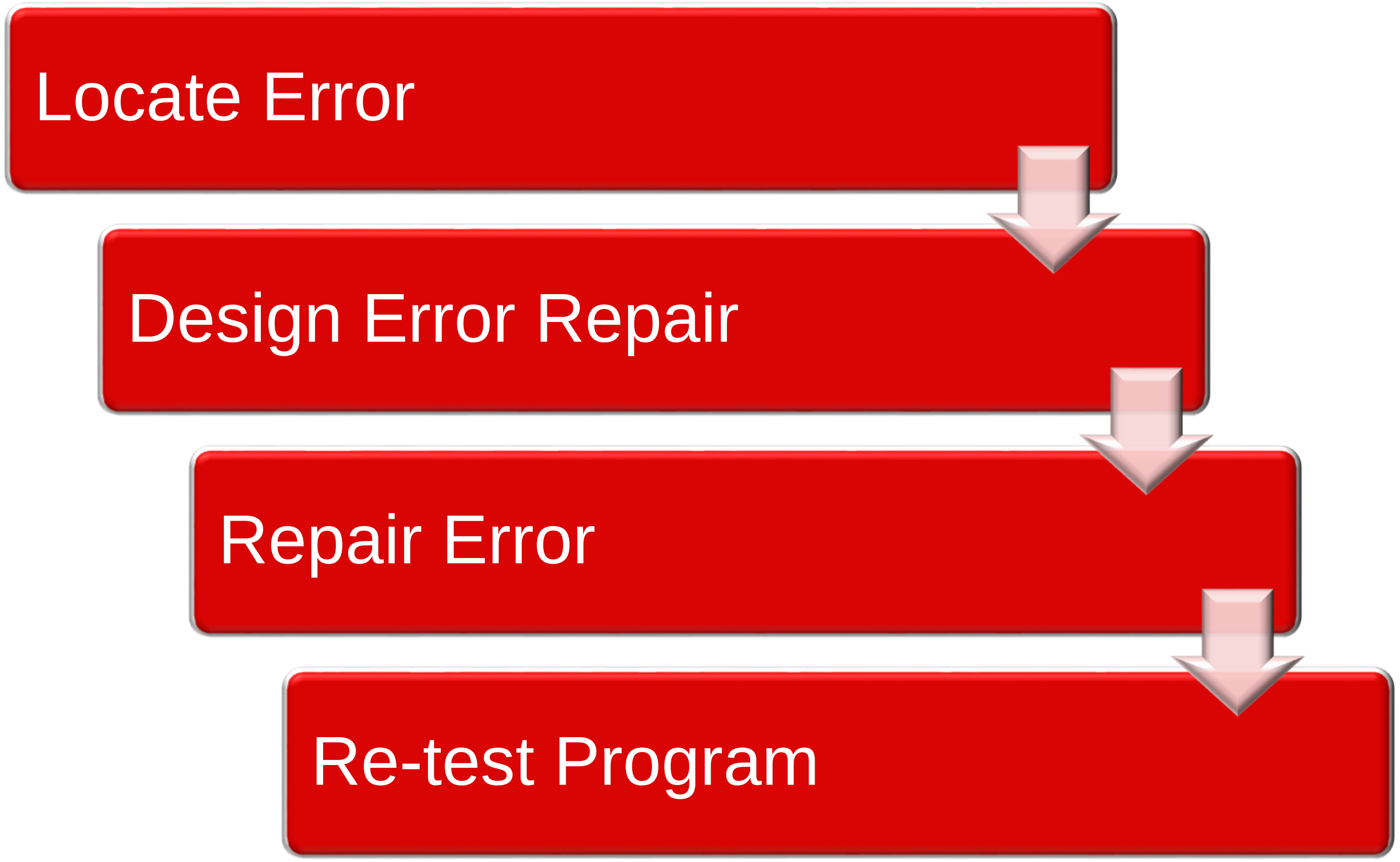
Just fix what you see:

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

*Why bother going into compute()?*

# HOW TO DEBUG

## (SOMMERVILLE 2004)



Locate Error

Design Error Repair

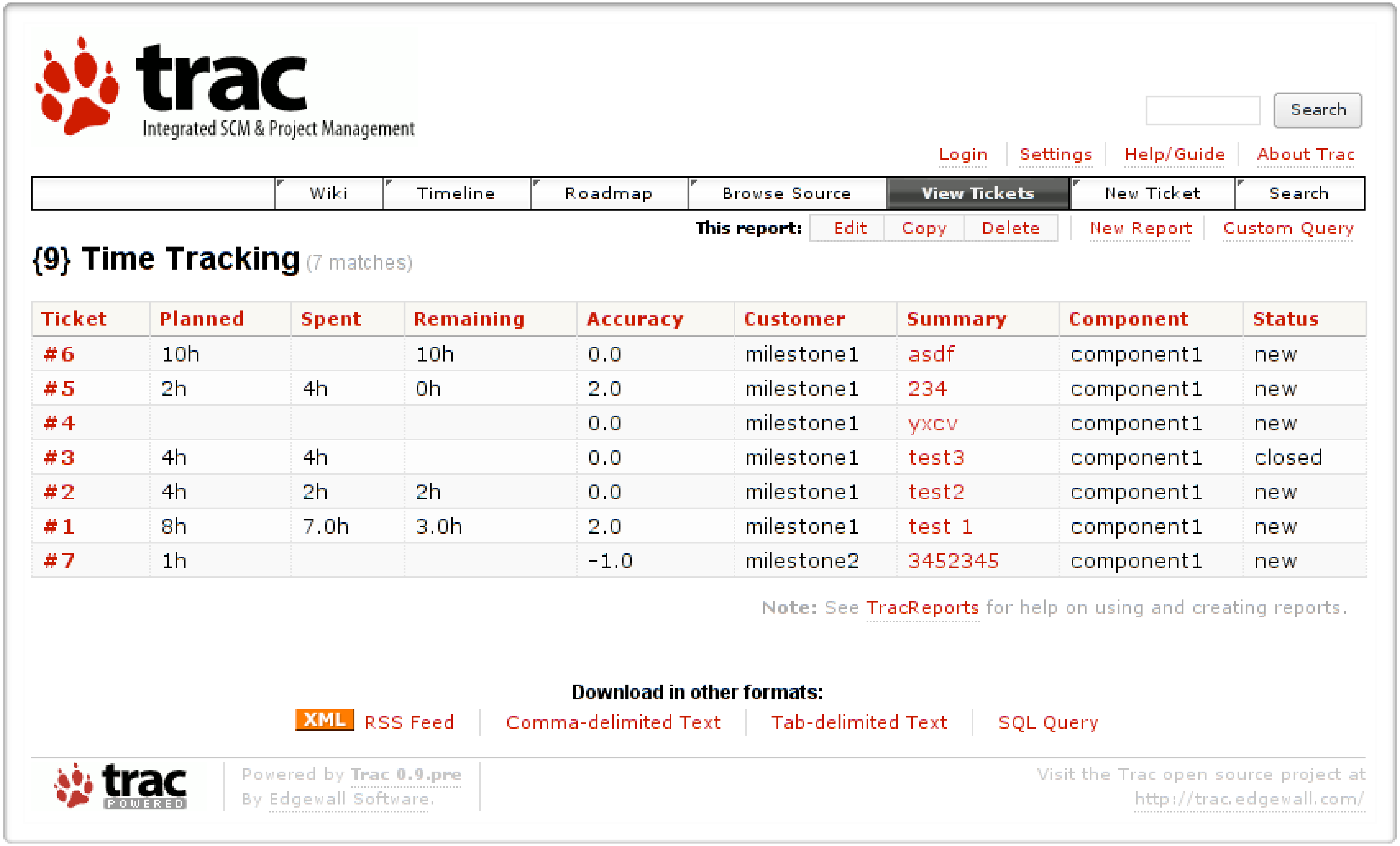
Repair Error

Re-test Program

## THE PROCESS

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

# TRACKING PROBLEMS



## {9} Time Tracking (7 matches)

| 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 |

**Note:** See TracReports for help on using and creating reports.

**Download in other formats:**

XML RSS Feed | Comma-delimited Text | Tab-delimited Text | SQL Query

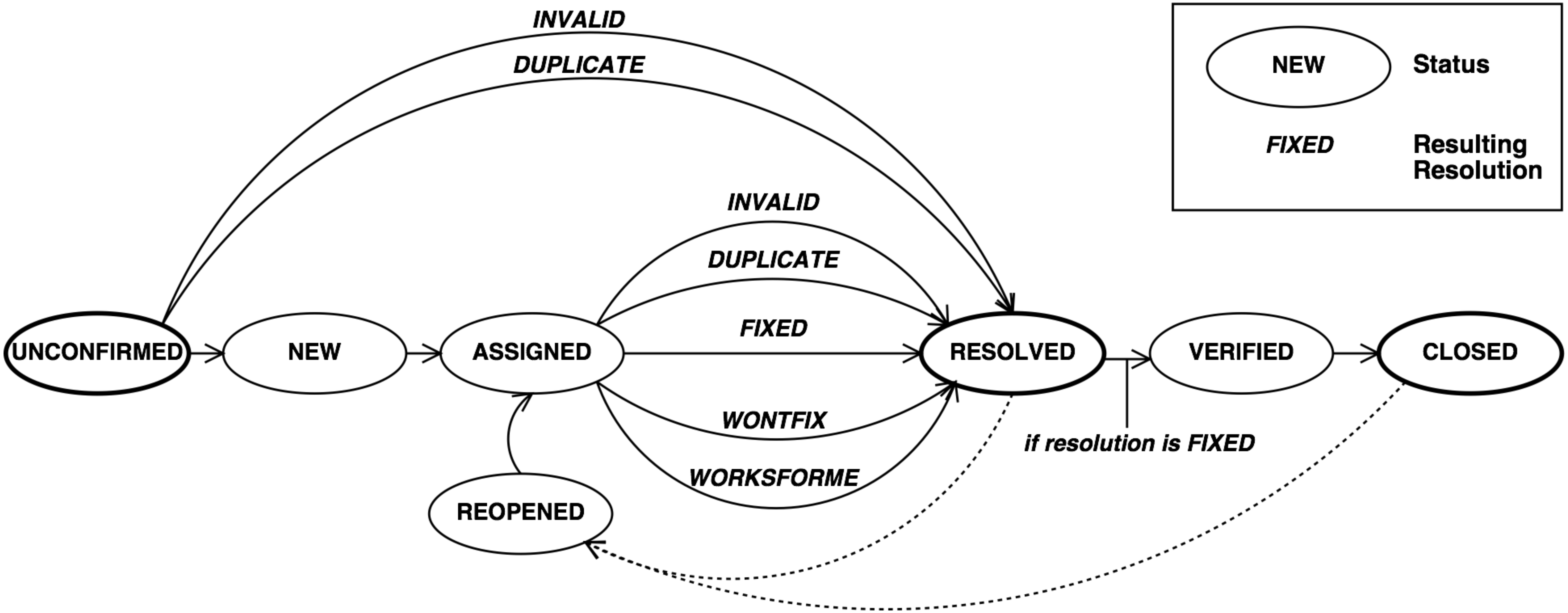
Powered by **Trac 0.9.pre**
By Edgewall Software.

Visit the Trac open source project at
http://trac.edgewall.com/

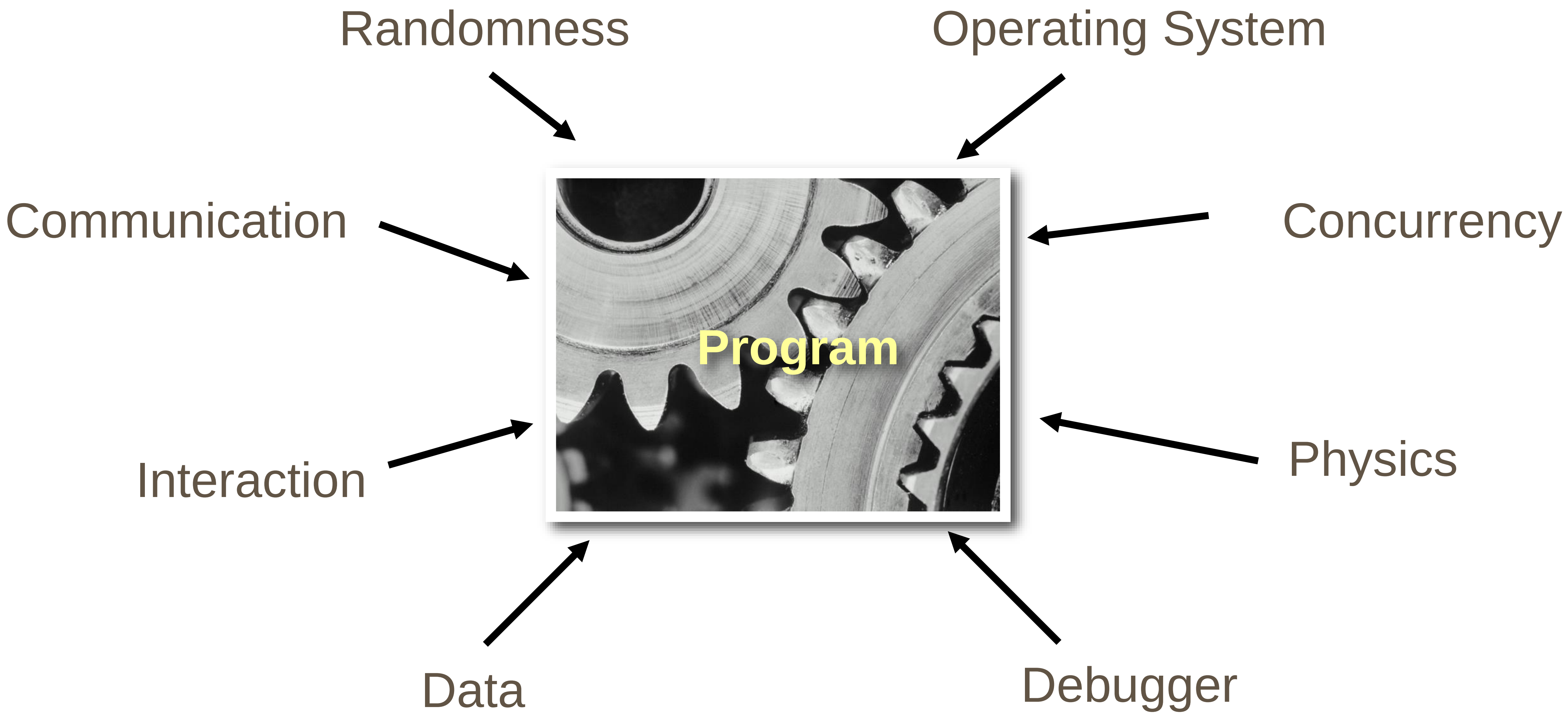
# TRACKING PROBLEMS

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

# PROBLEM LIFE CYCLE



# REPRODUCE



Randomness

Operating System

Communication

Concurrency

Program

Interaction

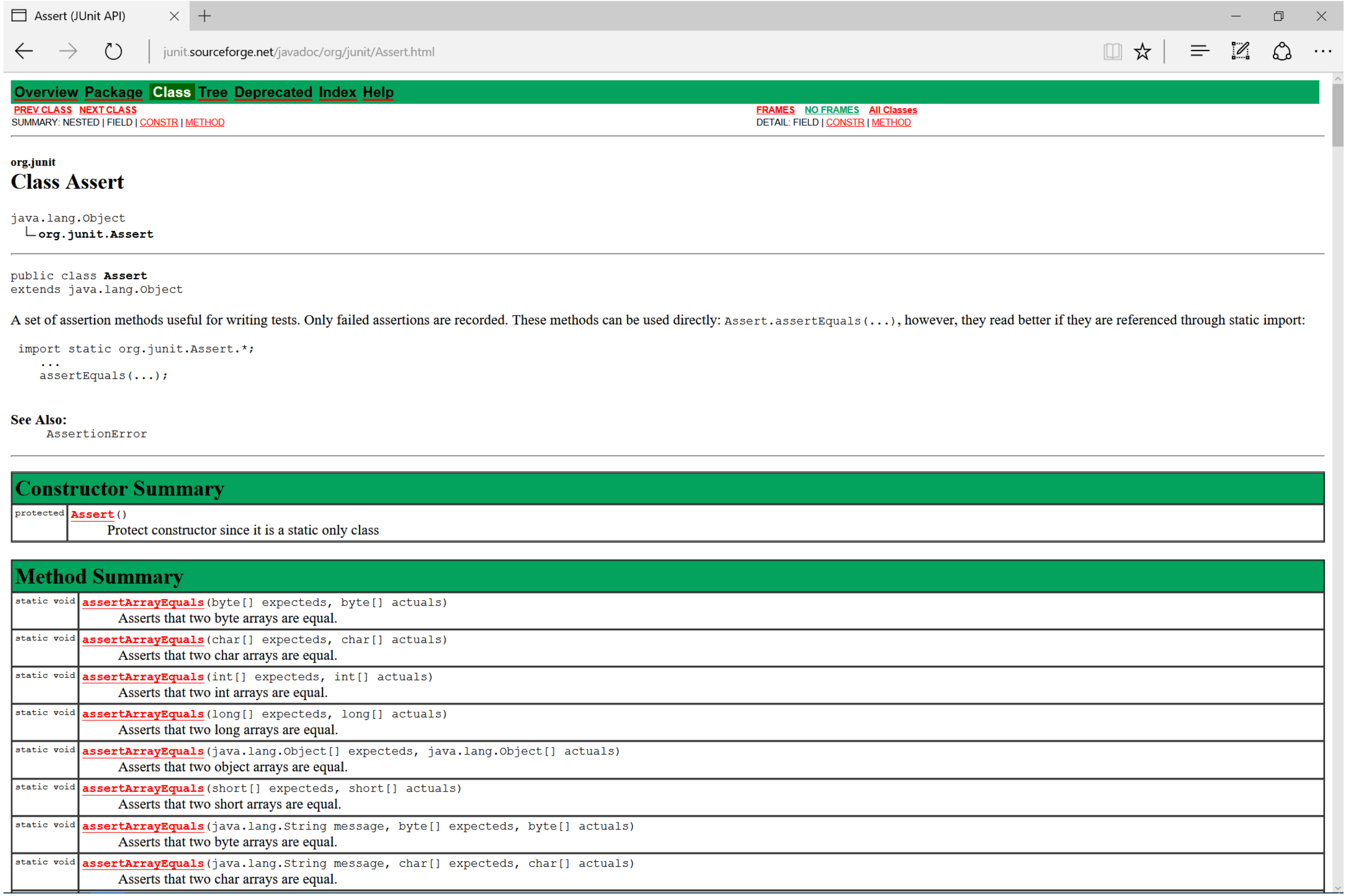
Physics

Data

Debugger

T
R
A
F
F
I
C

# AUTOMATE



org.junit

## Class Assert

```
java.lang.Object
  └org.junit.Assert
```

```
public class Assert
extends java.lang.Object
```

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: `Assert.assertEquals(...)`, however, they read better if they are referenced through static import:

```
 import static org.junit.Assert.*;
    ...
    assertEquals(...);
```

**See Also:**
`AssertionError`

## Constructor Summary

| | |
|---|---|
| `protected` | `Assert()`<br>Protect constructor since it is a static only class |

## Method Summary

| | |
|---|---|
| `static void` | `assertArrayEquals(byte[] expecteds, byte[] actuals)`<br>Asserts that two byte arrays are equal. |
| `static void` | `assertArrayEquals(char[] expecteds, char[] actuals)`<br>Asserts that two char arrays are equal. |
| `static void` | `assertArrayEquals(int[] expecteds, int[] actuals)`<br>Asserts that two int arrays are equal. |
| `static void` | `assertArrayEquals(long[] expecteds, long[] actuals)`<br>Asserts that two long arrays are equal. |
| `static void` | `assertArrayEquals(java.lang.Object[] expecteds, java.lang.Object[] actuals)`<br>Asserts that two object arrays are equal. |
| `static void` | `assertArrayEquals(short[] expecteds, short[] actuals)`<br>Asserts that two short arrays are equal. |
| `static void` | `assertArrayEquals(java.lang.String message, byte[] expecteds, byte[] actuals)`<br>Asserts that two byte arrays are equal. |
| `static void` | `assertArrayEquals(java.lang.String message, char[] expecteds, char[] actuals)`<br>Asserts that two char arrays are equal. |

T
R
A
F
F
I
C

# 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");
}
```

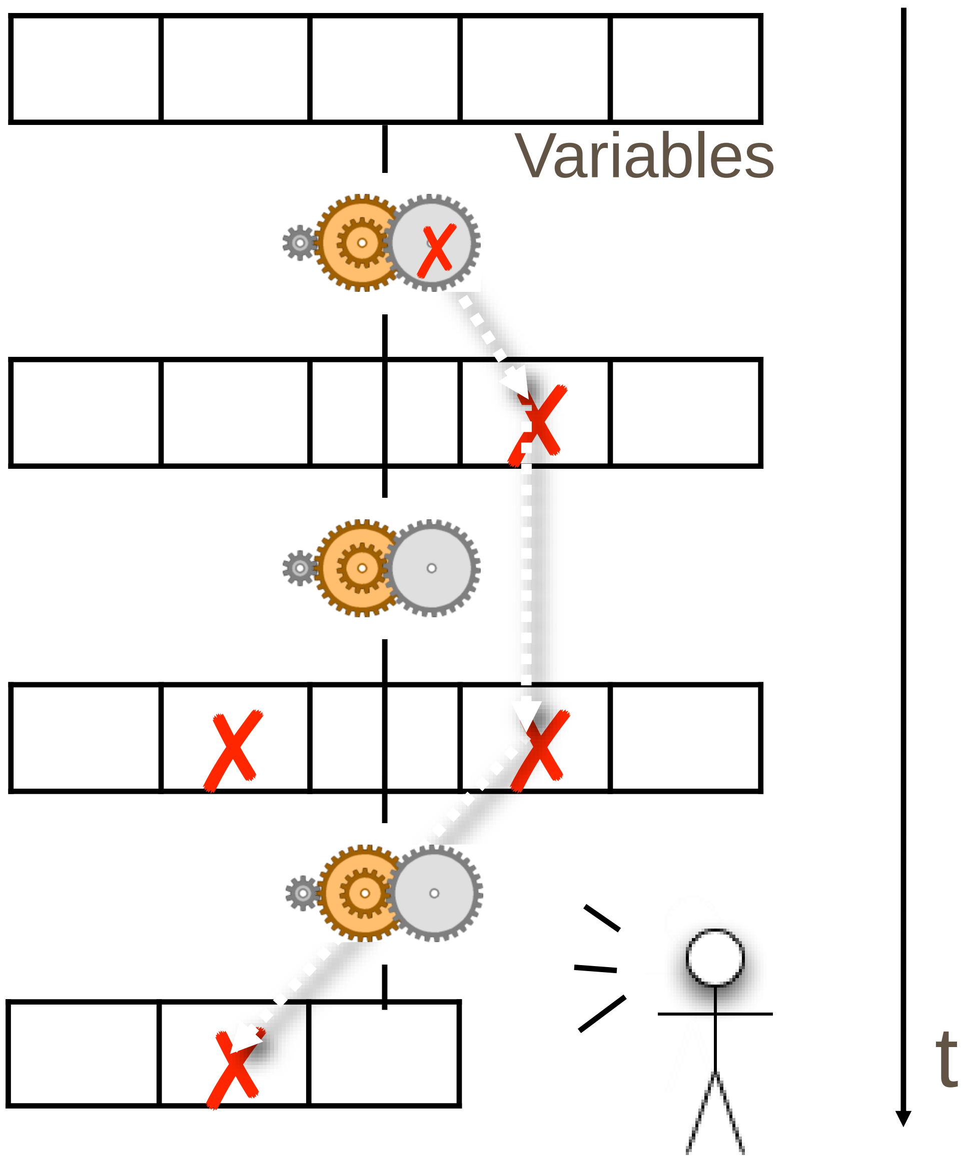
# AUTOMATE

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

# FINDING ORIGINS

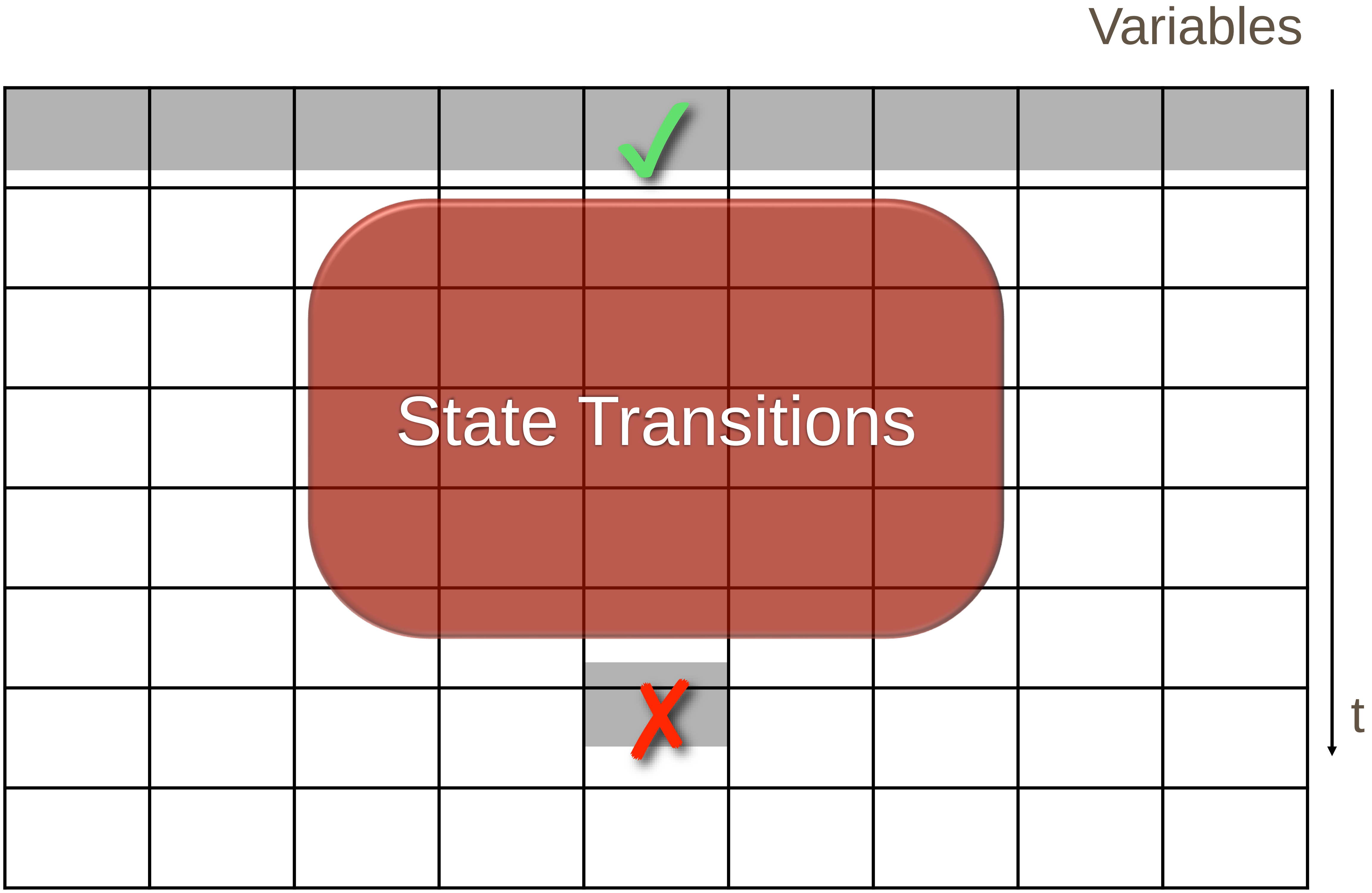
1. The programmer creates a ***defect*** in the code.
2. When executed, the defect creates an ***infection***.
3. The infection ***propagates***.
4. The infection causes a ***failure***.

This infection chain must be traced back – and broken.

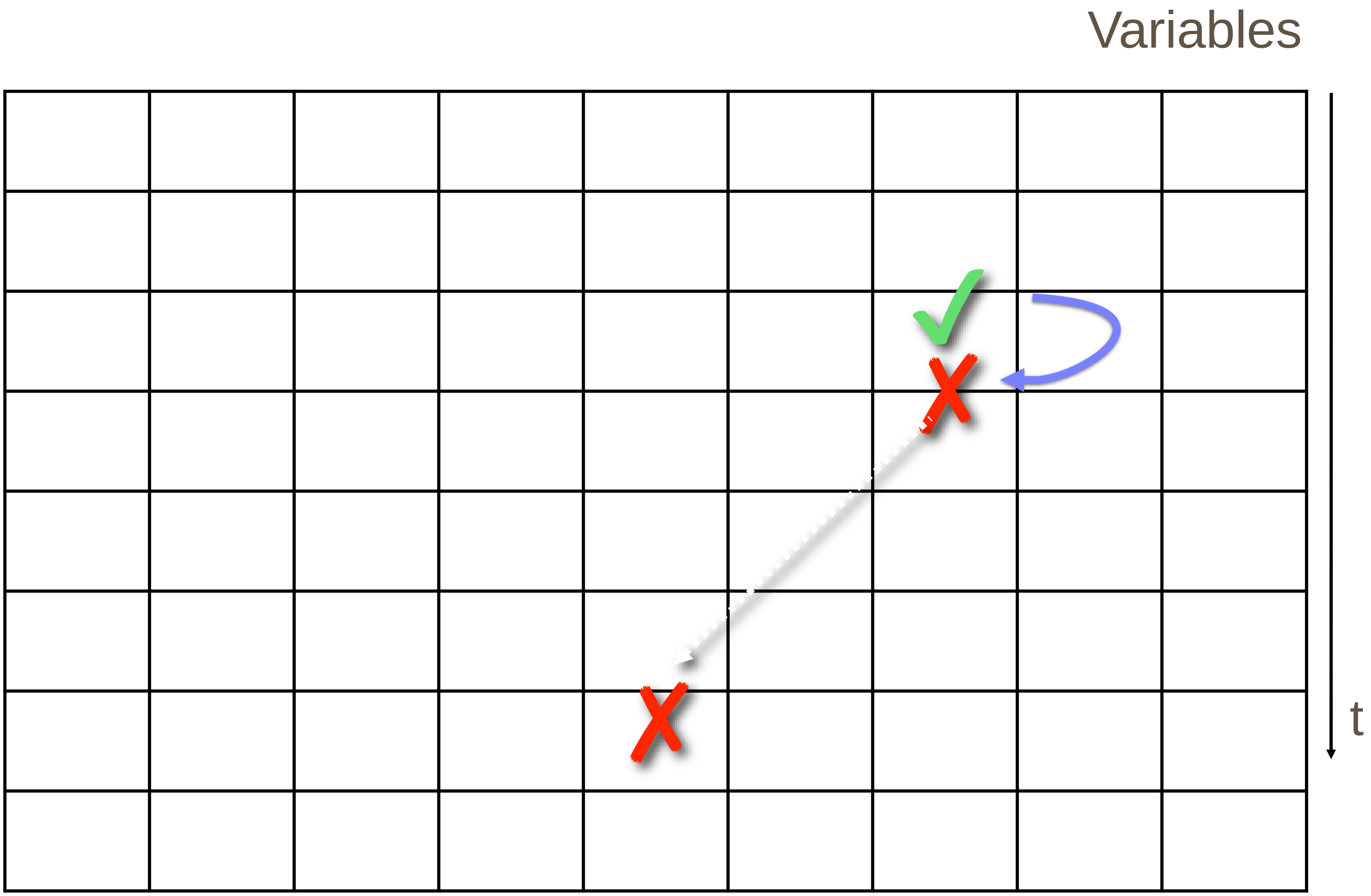


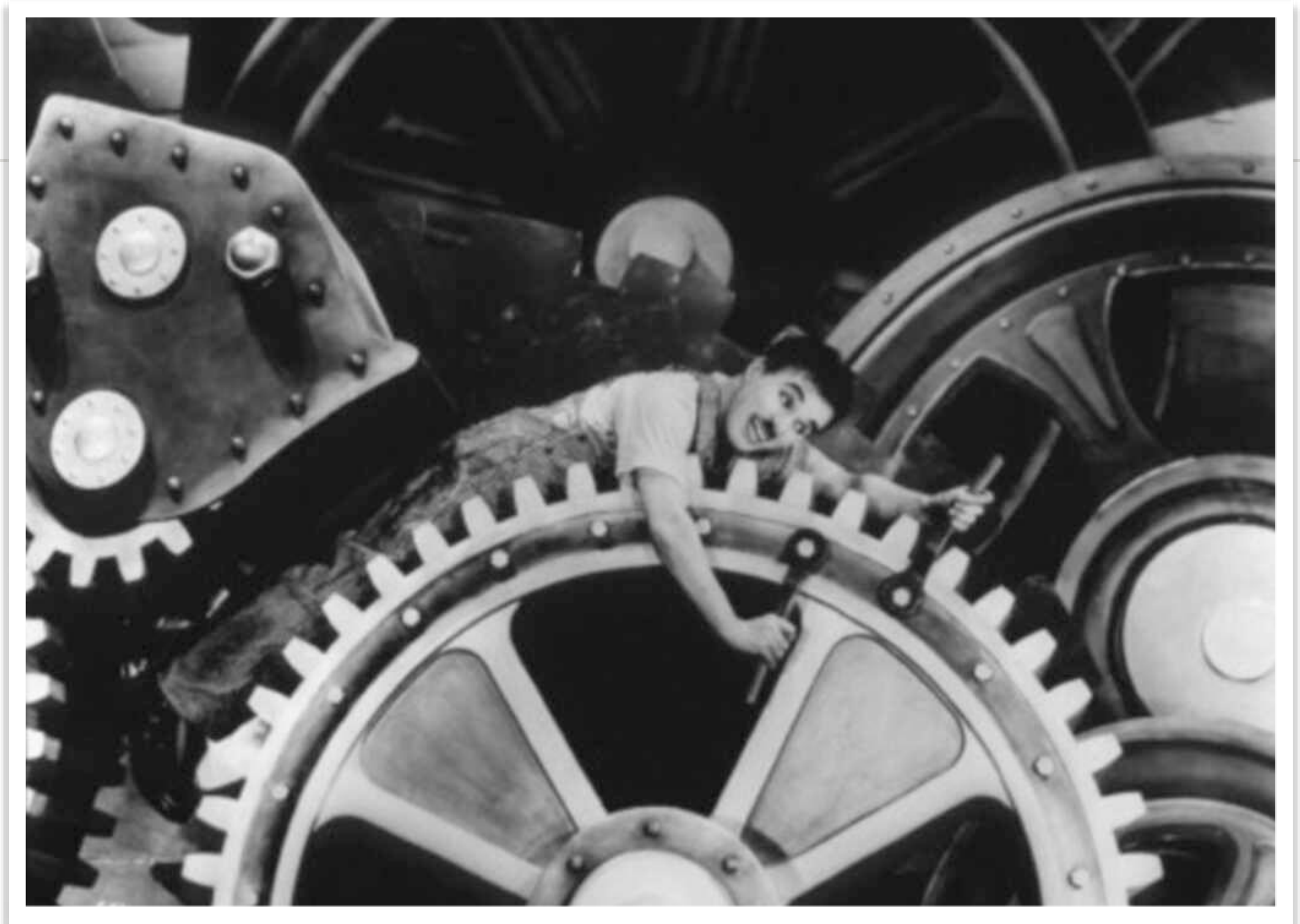
Not every defect creates an infection – not every infection results in a failure

# FINDING ORIGINS



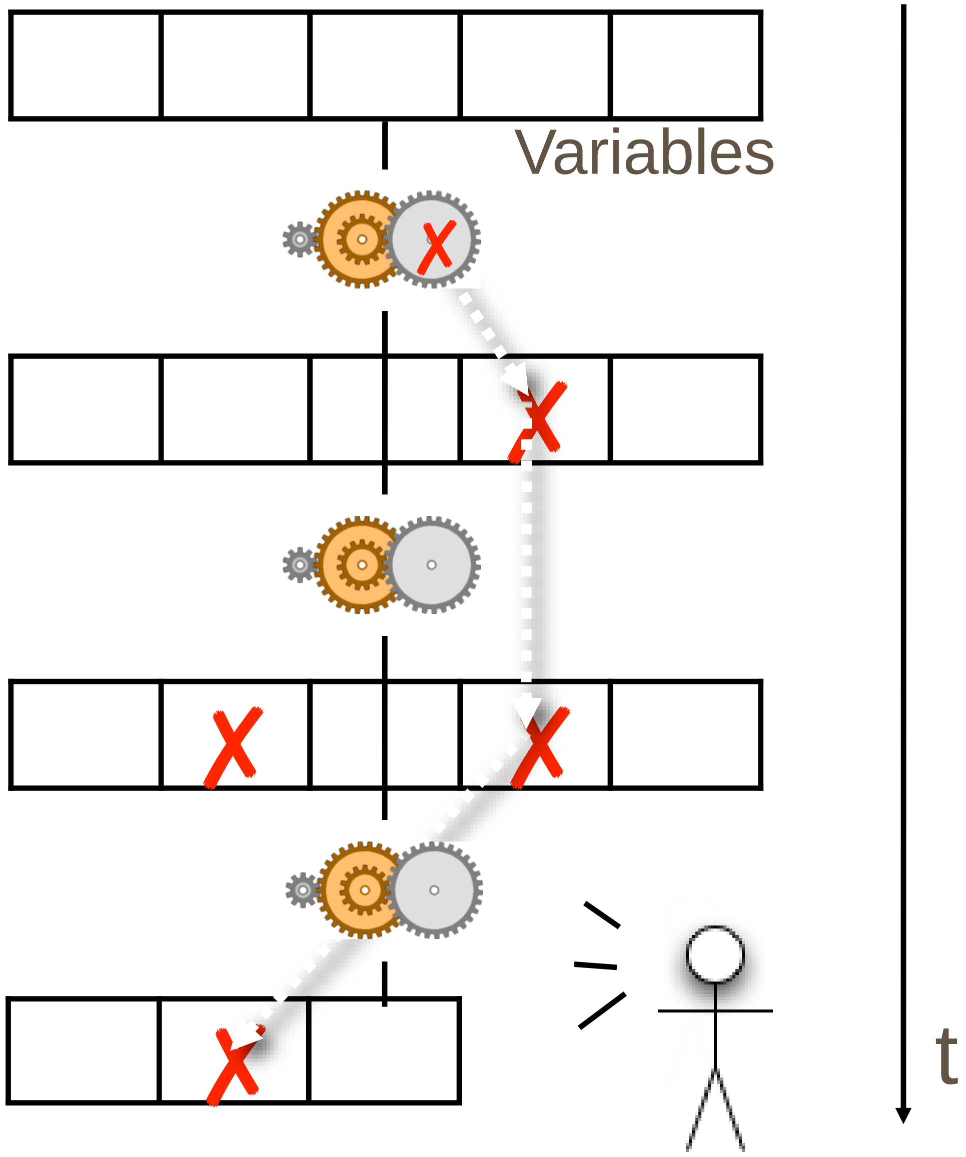
# THE DEFECT



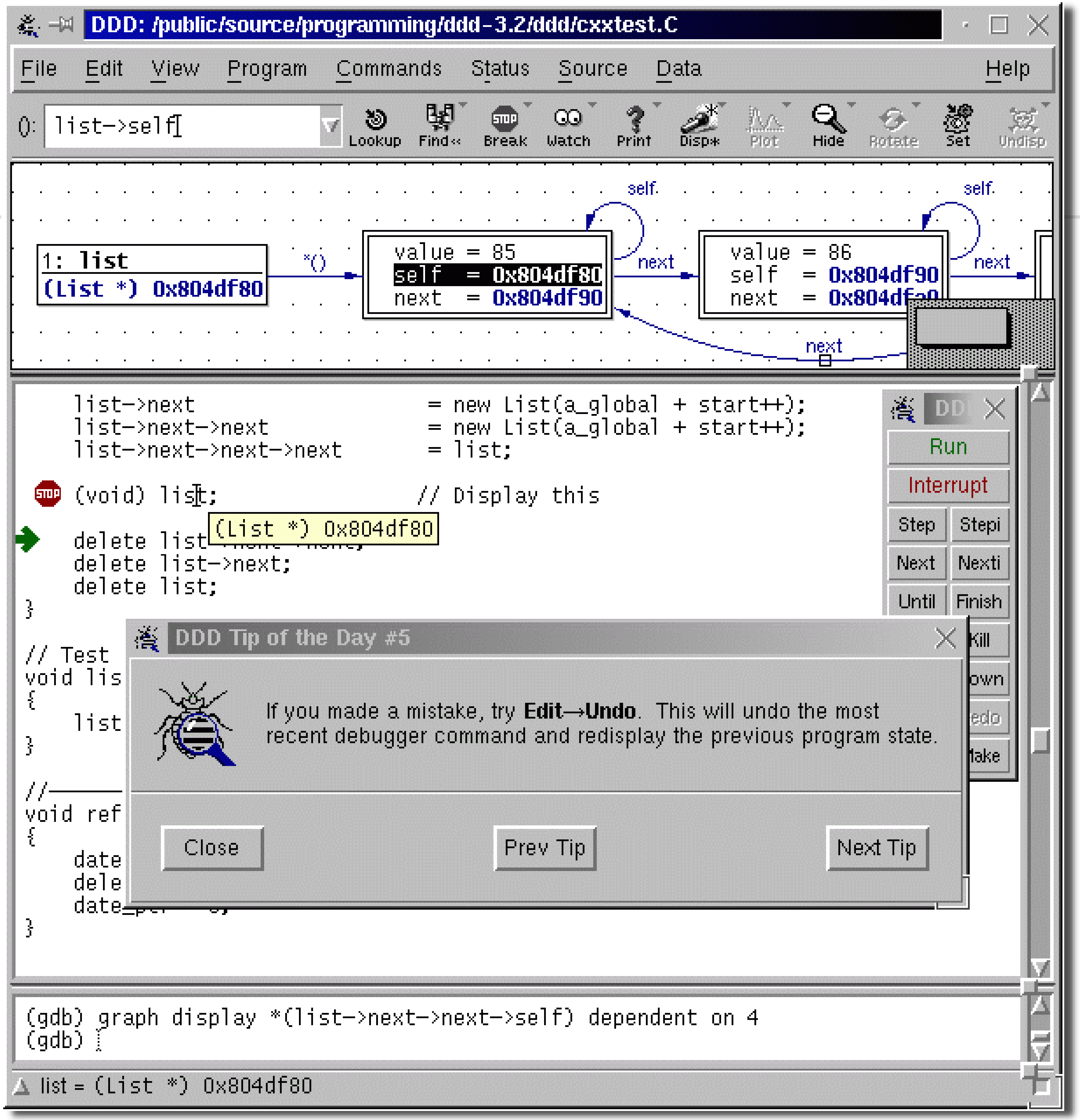


# FINDING ORIGINS

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



TRAFFIC



```
1: list
(List *) 0x804df80
```

```
value = 85
self  = 0x804df80
next  = 0x804df90
```

```
value = 86
self  = 0x804df90
next  = 0x804dfa0
```

```
    list->next               = new List(a_global + start++);
    list->next->next         = new List(a_global + start++);
    list->next->next->next   = list;

    (void) list;             // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date_
}
```

(List *) 0x804df80

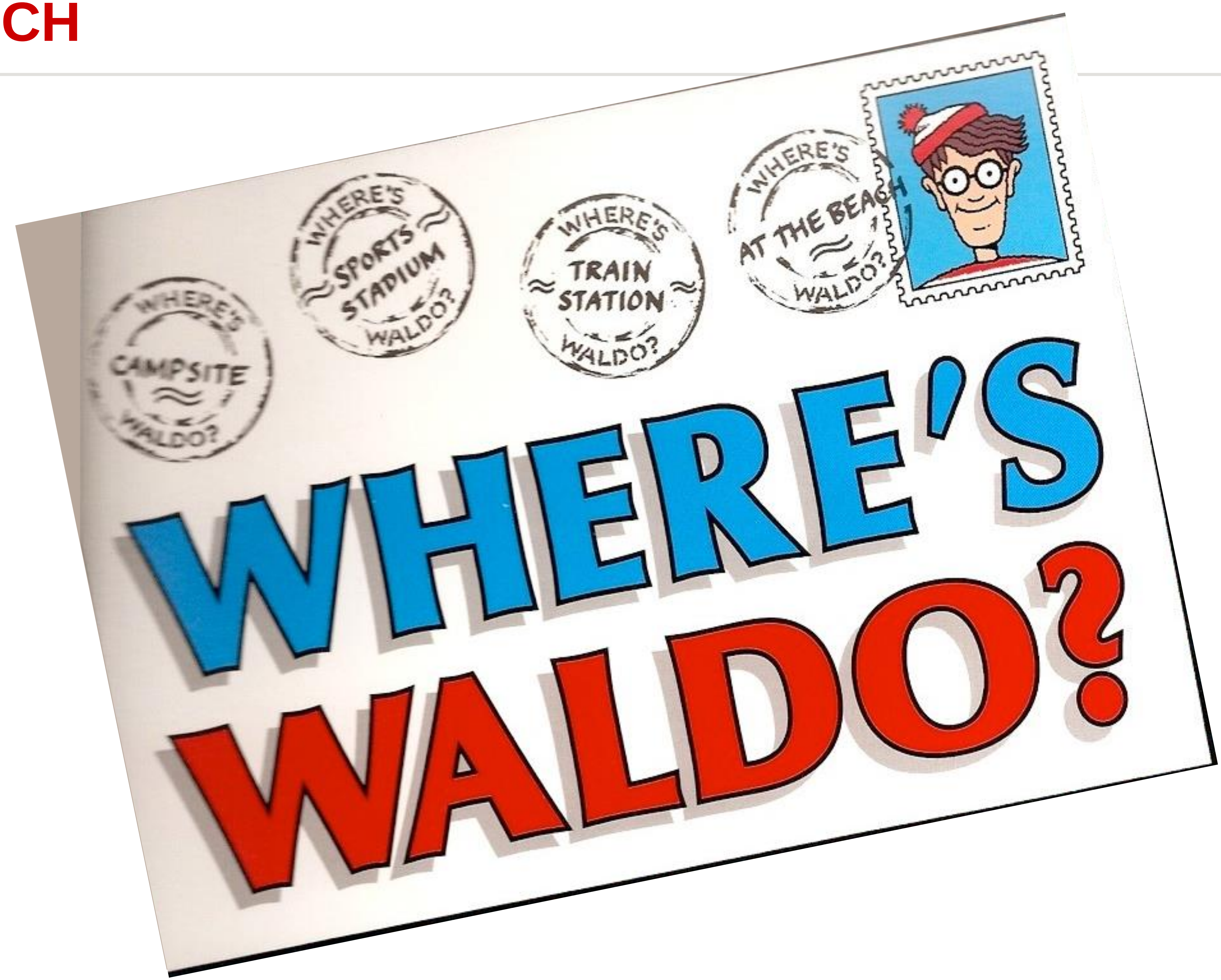
DDD Tip of the Day #5

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

## SEARCH


WHERE'S CAMPSITE WALDO?
WHERE'S SPORTS STADIUM WALDO?
WHERE'S TRAIN STATION WALDO?
WHERE'S AT THE BEACH WALDO?
WHERE'S
WALDO?

# FOCUS

During our search for infection, we focus upon locations that

1. *Are possibly wrong*
(e.g., because they were buggy before)

2. *Are explicitly wrong*
(e.g., because they violate an *assertion*)

Assertions are the best way to find infections!

# 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

# 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
}
```

# 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

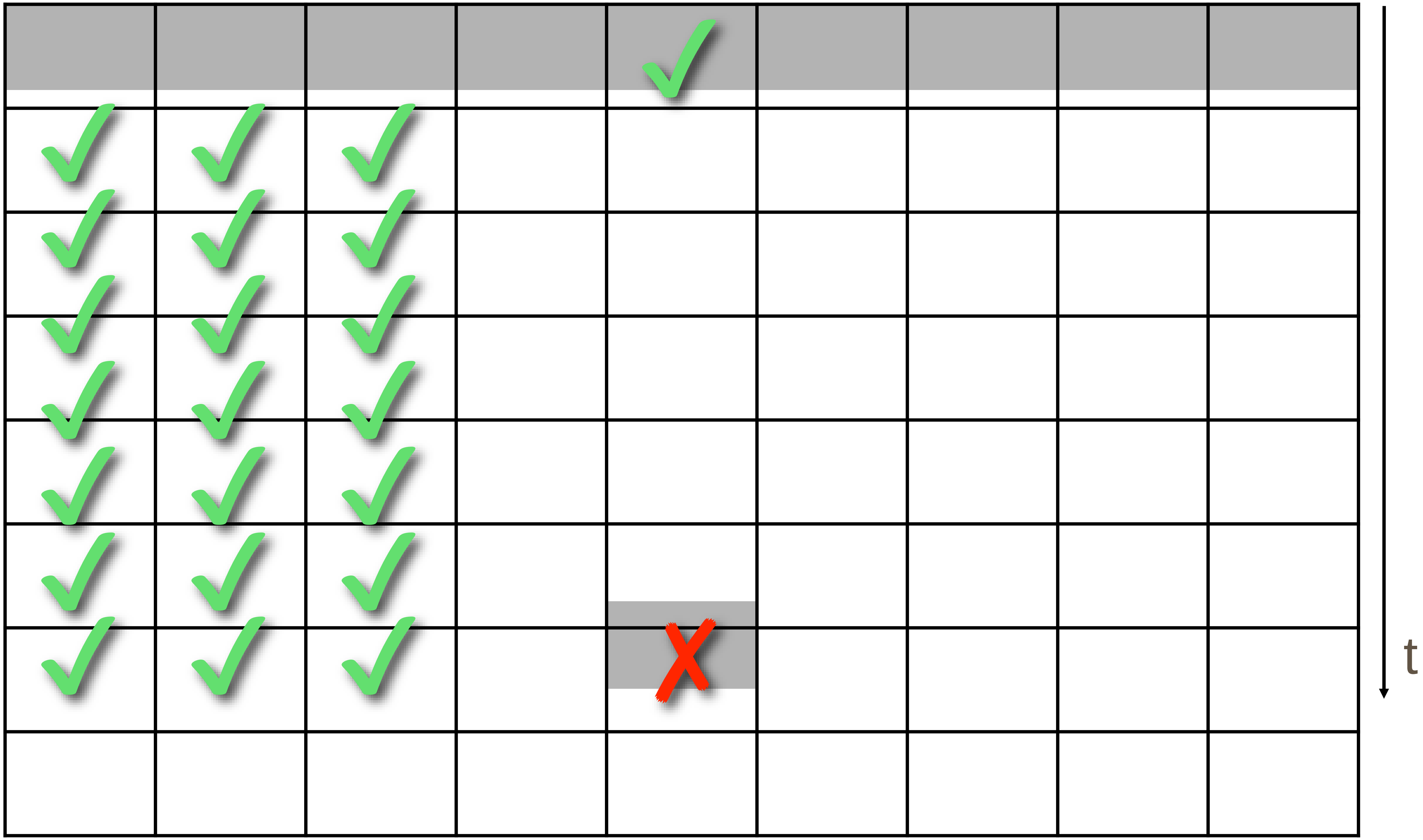
```
void Time::set_hour(int h)
{
    assert (sane());  // Precondition
    …
    assert (sane());  // Postcondition
}
```

# COMPLEX INVARIANTS

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

      return true;
  }
}
```

# ASSERTIONS



# FOCUSING

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

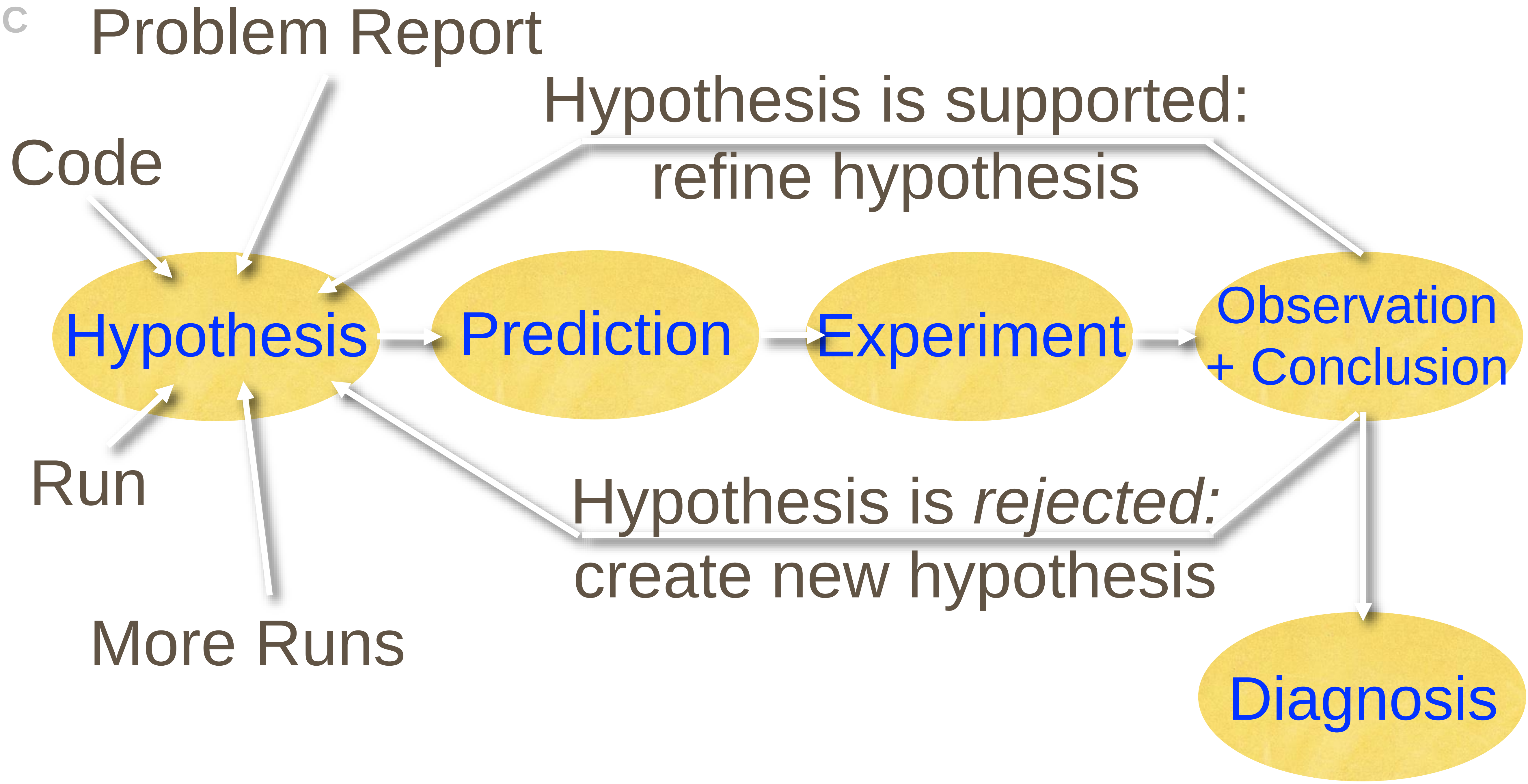
# ISOLATION

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

# USING OBSERVATIONS BASED ON EXPERIMENTS IS THE 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



Problem Report

Code

Hypothesis is supported: refine hypothesis

Hypothesis

Prediction

Experiment

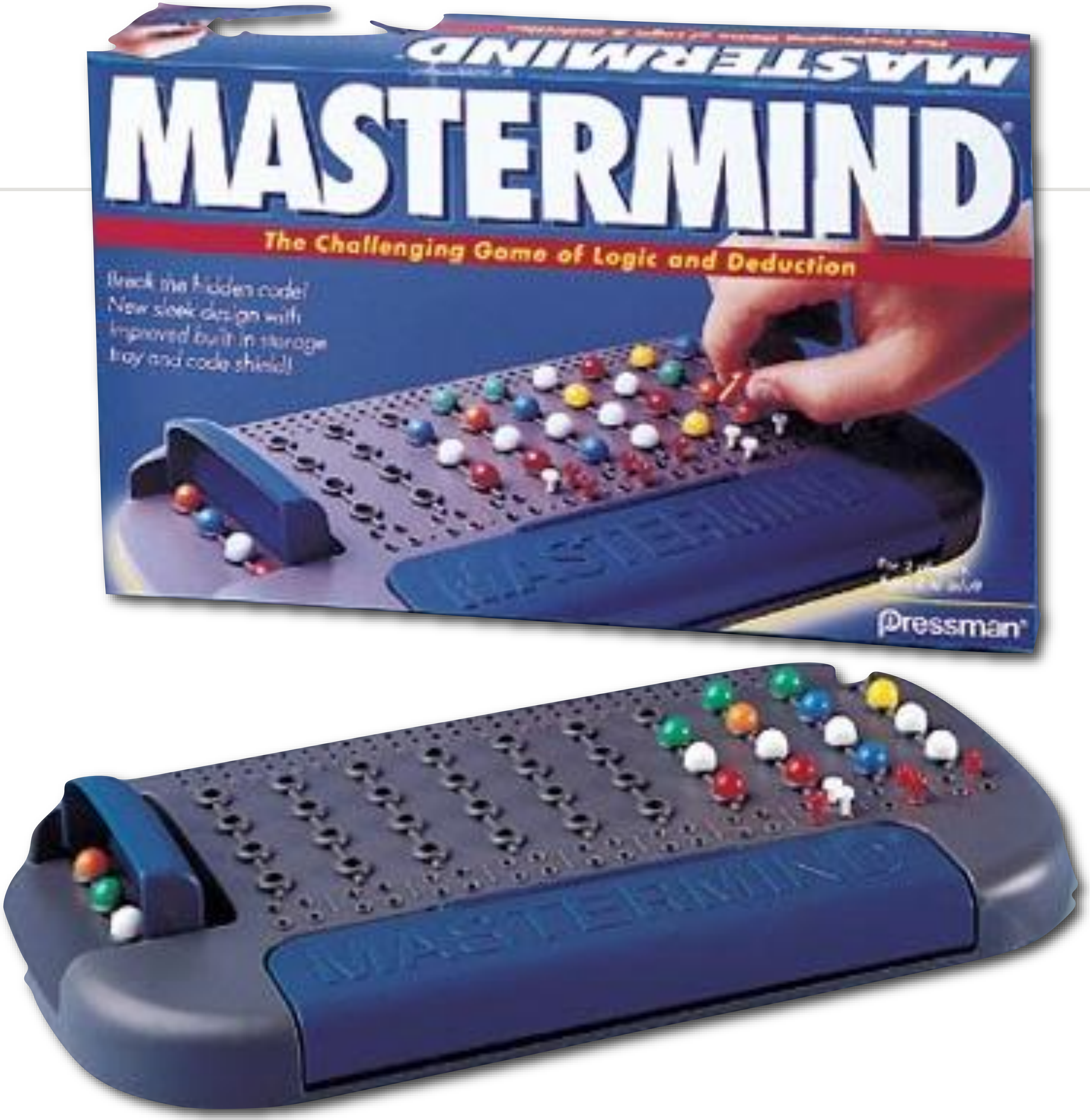
Observation + Conclusion

Run

Hypothesis is *rejected:* create new hypothesis

More Runs

Diagnosis


MASTERMIND
The Challenging Game of Logic and Deduction
Break the hidden code!
New sleek design with
improved built-in storage
tray and code shield!
Pressman
MASTERMIND

# EXPLICIT HYPOTHESES

| | |
|---|---|
| Hypothesis | The executi uses a[0] = 0 |
| Prediction | At Li ld hold. |
| Experiment | e 37. |
| Observatio | s as predicted. |
| Conclusion | esis is *confirmed.* |

Keeping everything in memory is like playing mastermind blind!

# ISOLATE

We repeat the search for infection origins until we found the defect

1. We proceed *systematically* along the scientific method
2. *Explicit steps* guide the search – and make it repeatable at any time

# 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

# SUCCESSFUL CORRECTION

# THE PROCESS

**T** rack the problem

**R** eproduce

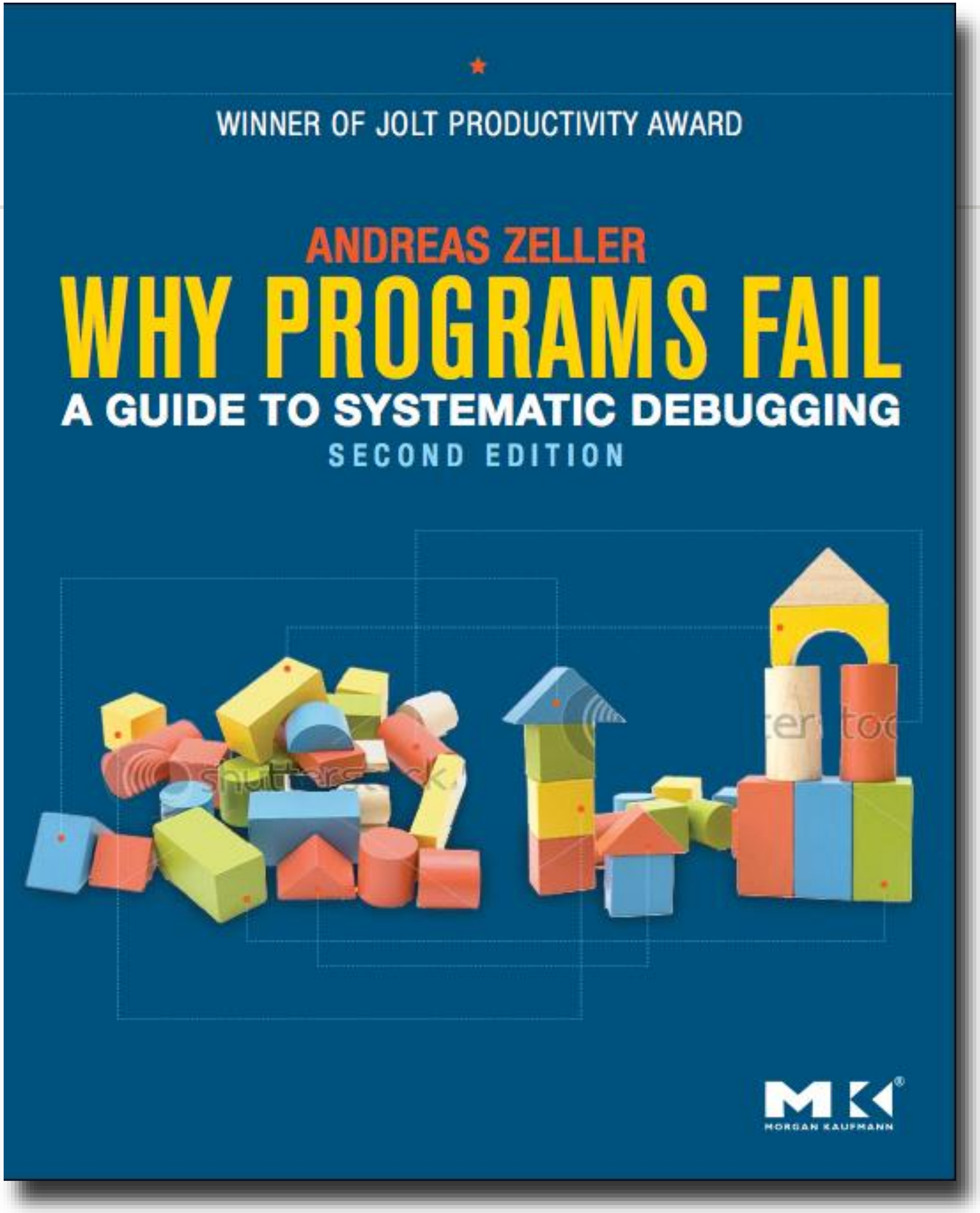
**A** utomate

**F** ind Origins

**F** ocus

**I** solate

**C** orrect



WINNER OF JOLT PRODUCTIVITY AWARD

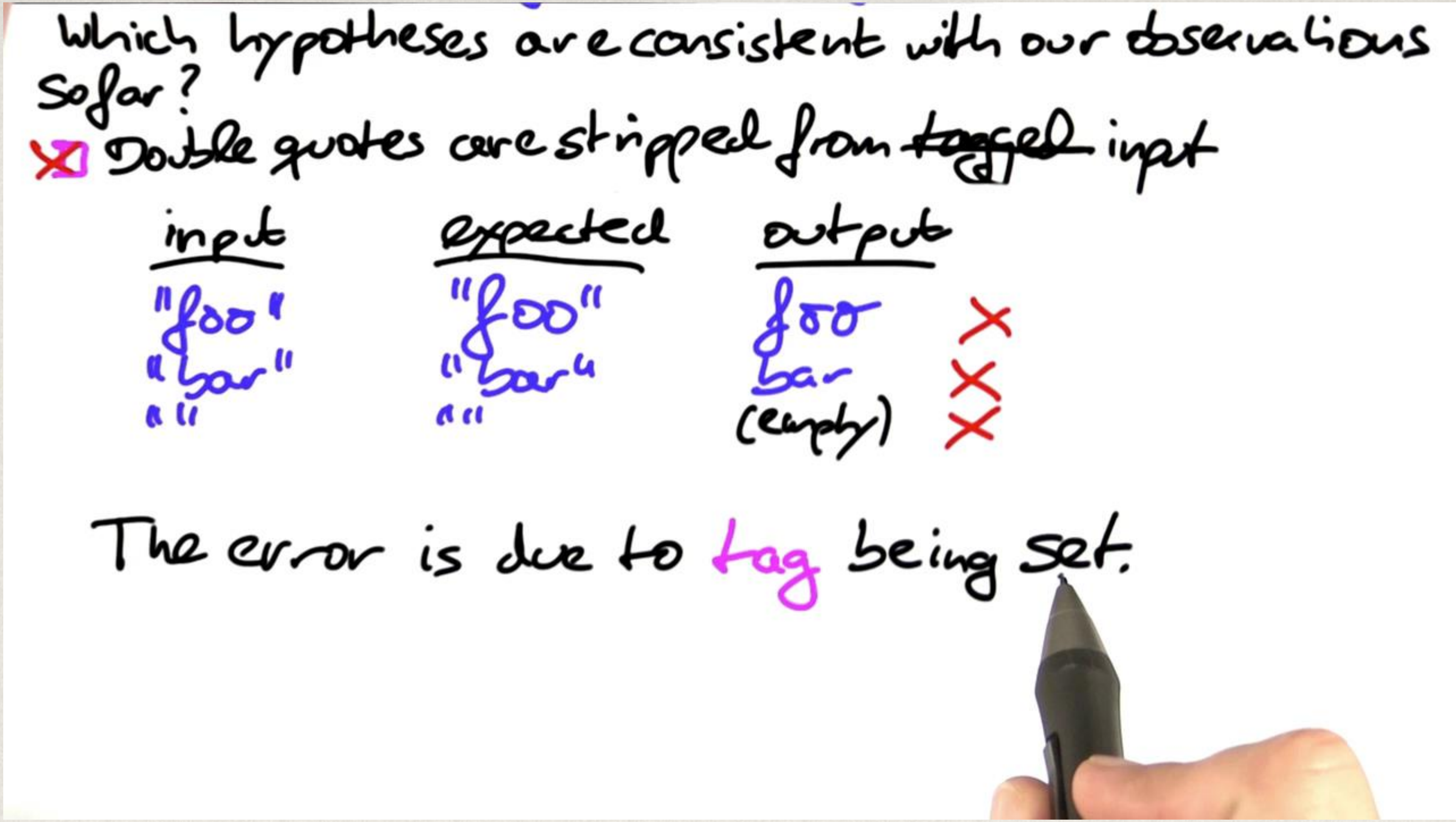
ANDREAS ZELLER

WHY PROGRAMS FAIL

A GUIDE TO SYSTEMATIC DEBUGGING

SECOND EDITION

MORGAN KAUFMANN



Which hypotheses are consistent with our observations so far?

☒ Double quotes are stripped from ~~tagged~~ input

| input | expected | output | |
|---|---|---|---|
| "foo" | "foo" | foo | ✗ |
| "bar" | "bar" | bar | ✗ |
| "" | "" | (empty) | ✗ |

The error is due to tag being set.

# AUTOMATED DEBUGGING (WS 2016/17)