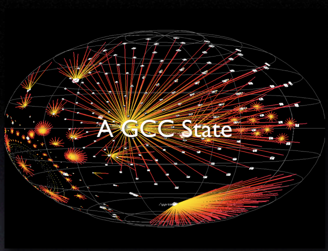
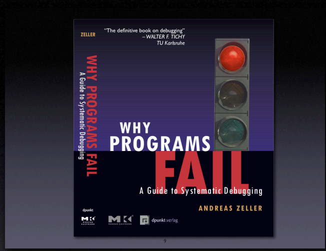
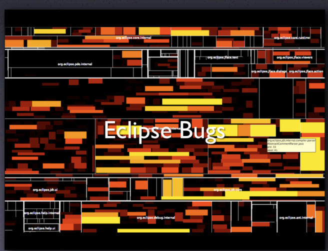
Where do Bugs Come From?

Andreas Zeller
Saarland University



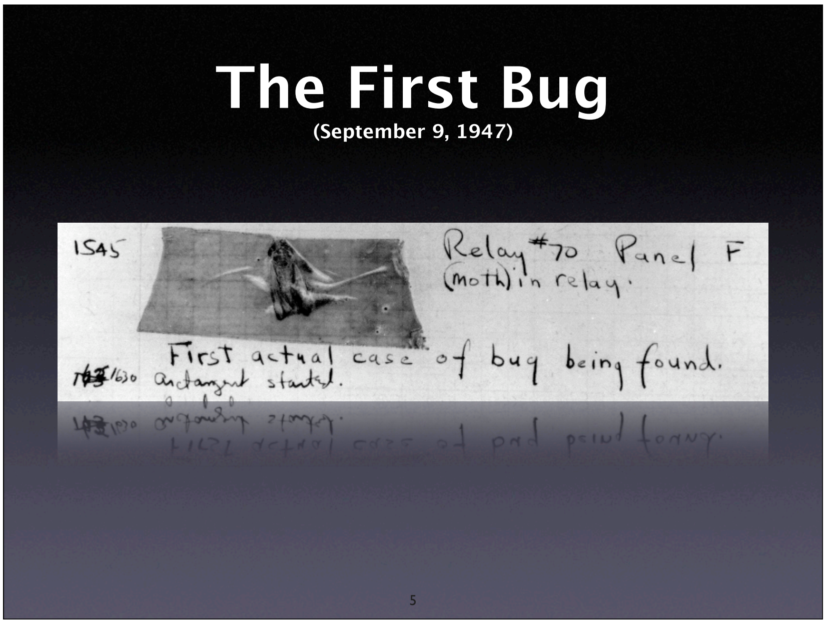




bug.c

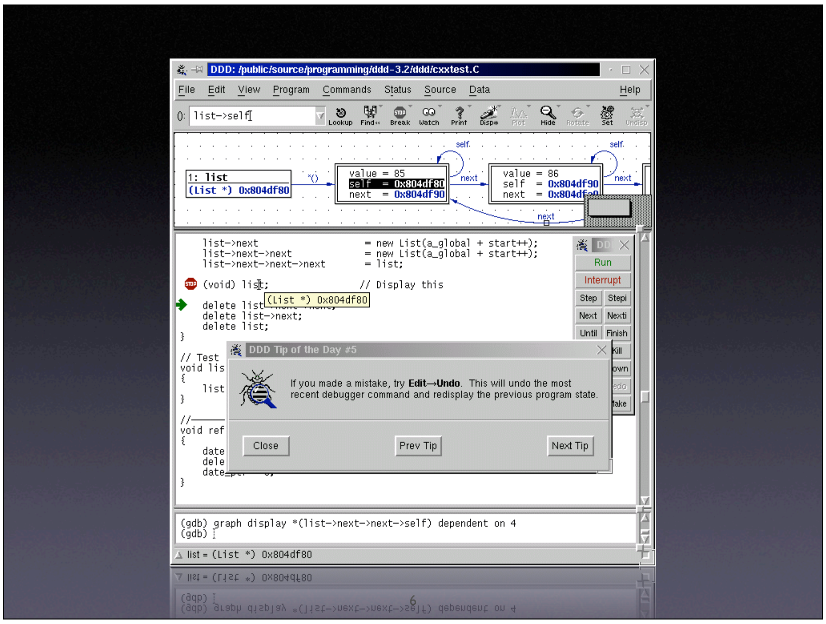
```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

What do we do now? We can follow Platon and say: Hey, let’s just verify this compiler, let’s do more abstraction, let’s do more of the same. (This is what I learned in school: The state of the art is bad, but if only people would do it our way, than the world would be a



Retrieved by a technician from the Harvard Mark II machine on September 9, 1947.

Now on display at the Smithsonian, Washington

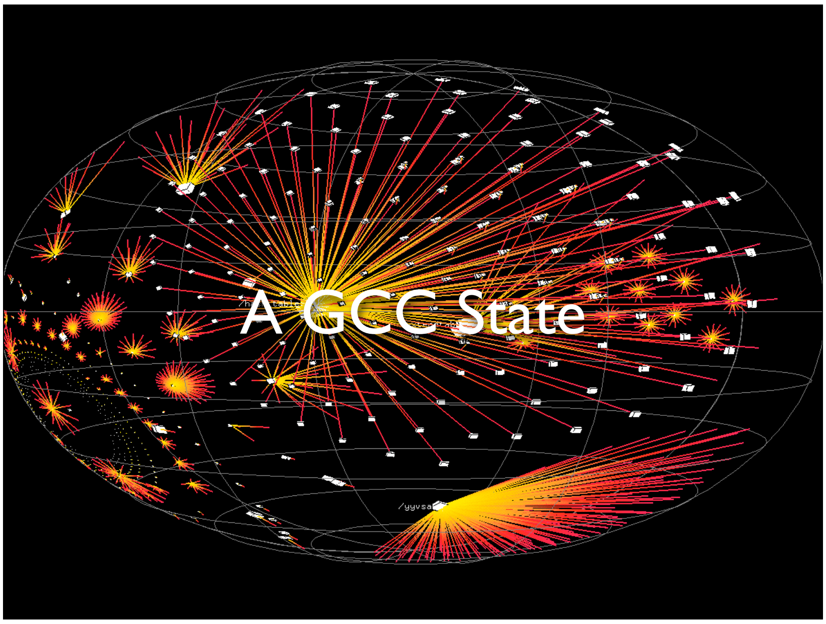


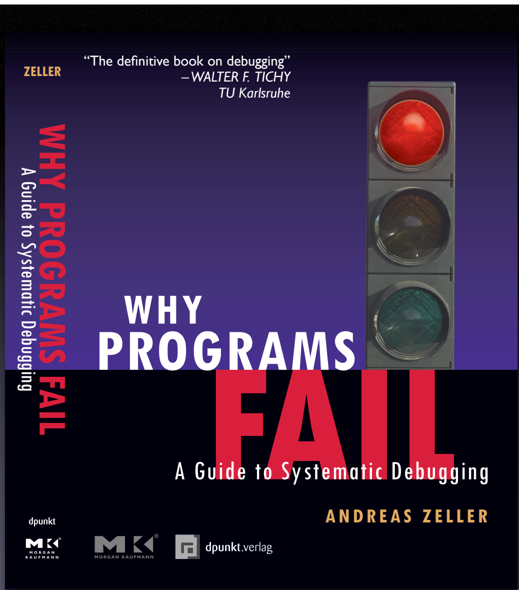
How to Debug

(Sommerville 2004)

Locate error → Design error repair → Repair error → Re-test program

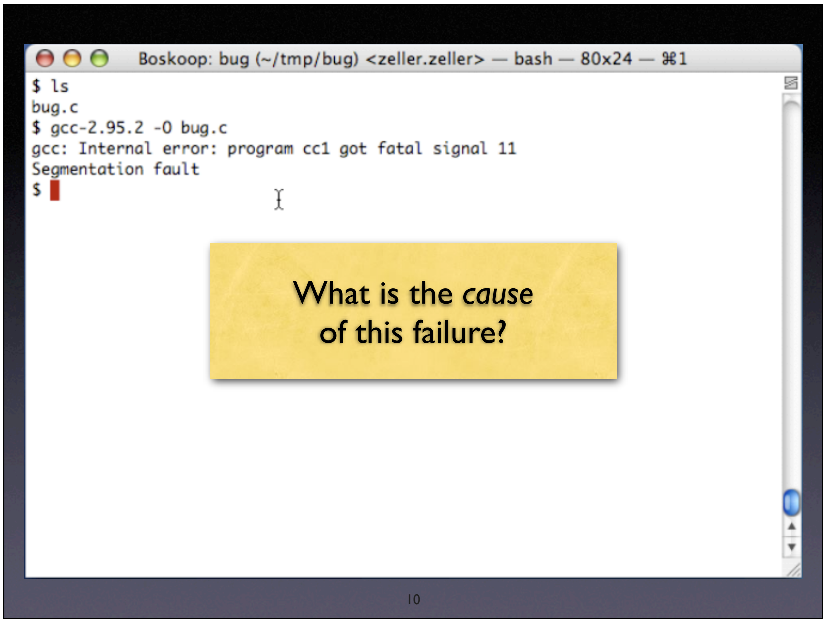
7



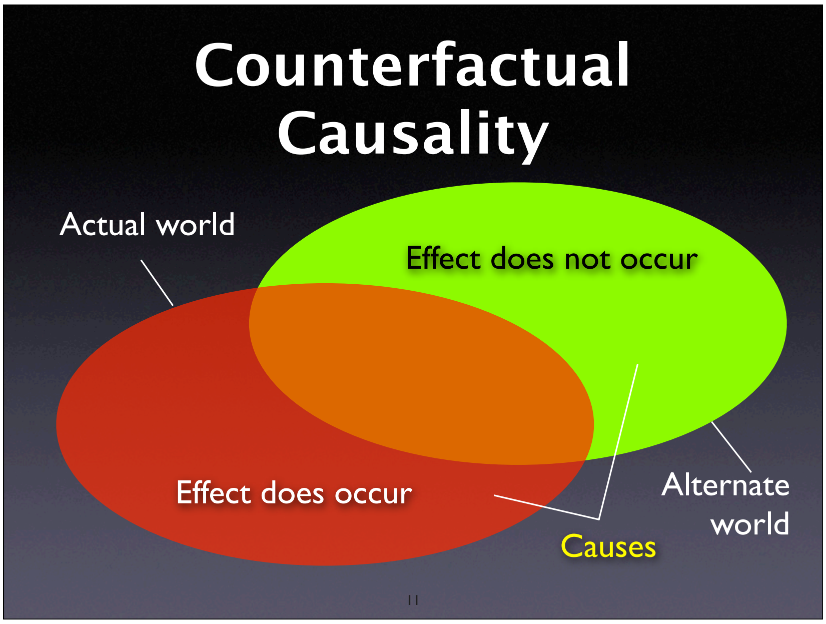


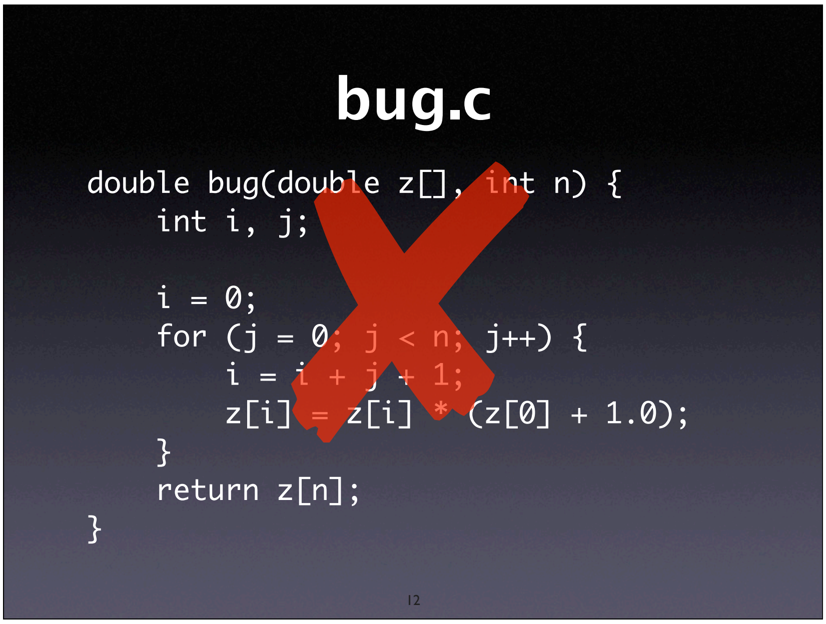
9

And if you need such a toolbox, I have written all these techniques down in a textbook.



What do we do now? We can follow Platon and say: Hey, let's just verify this compiler, let's do more abstraction, let's do more of the same. (This is what I learned in school: The state of the art is bad, but if only people would do it our way, than the world would be a

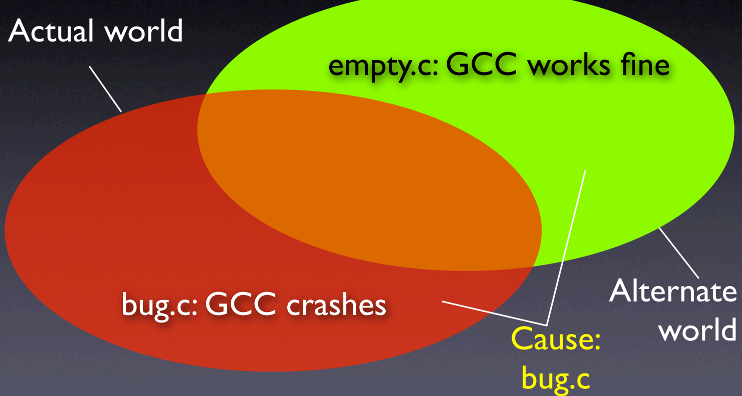




empty.c

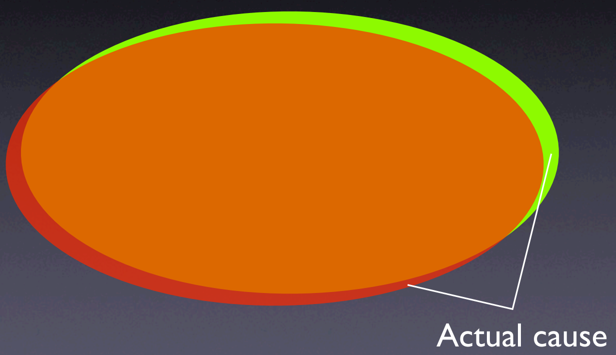
```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```


Causes as Differences



Actual Causes

"The" cause (*actual cause*) is a *minimal difference*



Isolating Causes

```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

16

Isolating Causes

```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

17

Isolating Causes

```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

18

Isolating Causes

```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Actual cause narrowed down

Isolating Causes

```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```


Isolating Causes

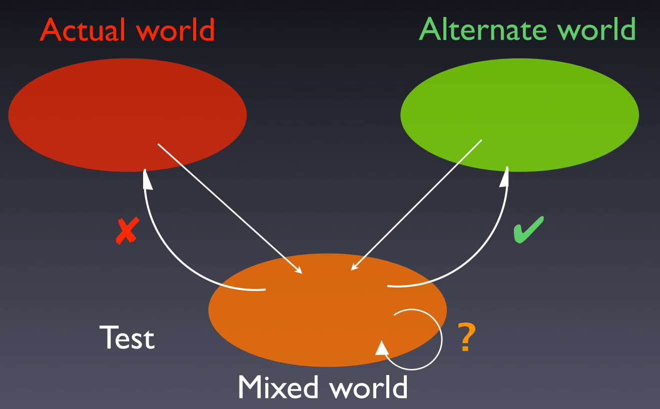
```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Isolating Causes

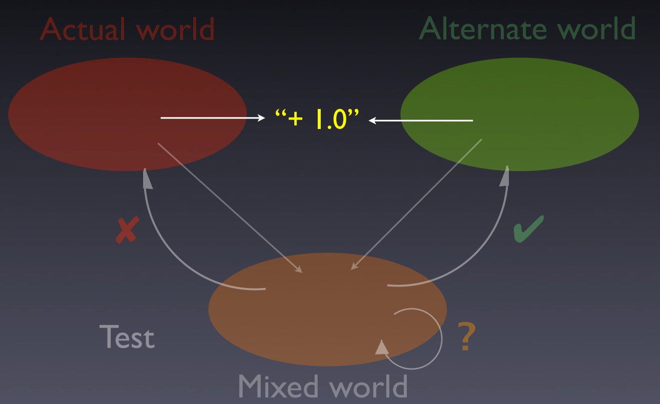
```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Actual cause of the GCC crash

Isolating Causes



Isolating Causes



Delta Debugging

Delta Debugging *isolates failure causes automatically:*

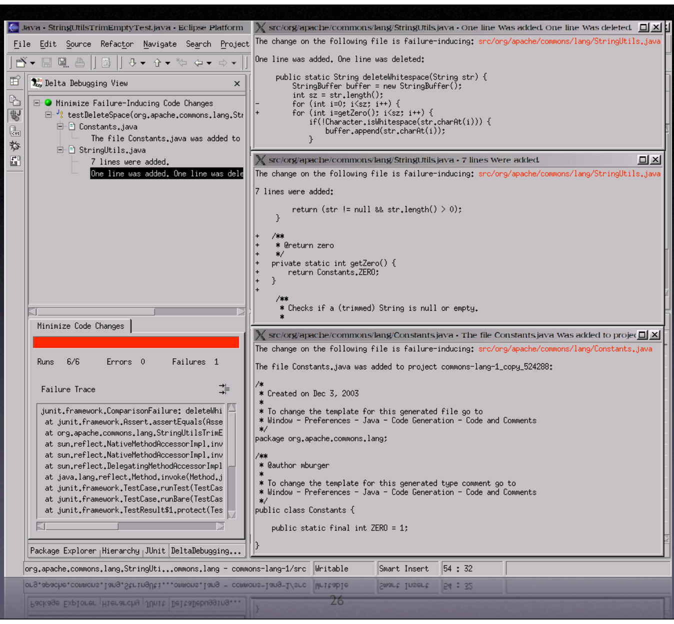
Inputs: 1 of 900 HTML lines in Mozilla

Code changes: 1 of 8,721 code changes in GDB

Threads: 1 of 3.8 bln thread switches in Scene.java

Messages: 2 of 347 Java method calls

Fully automatic + purely test-based

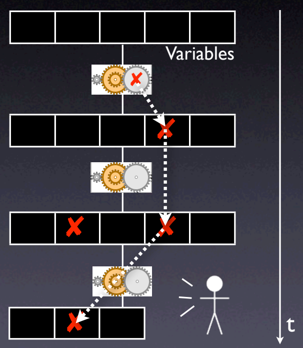


Automated Test
→ Automated Debugging

From Defect to Failure

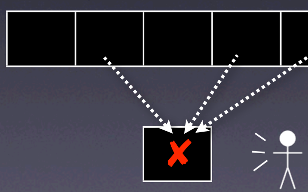
1. The programmer creates a *defect* in the code.
2. When executed, the defect creates an *infection*.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

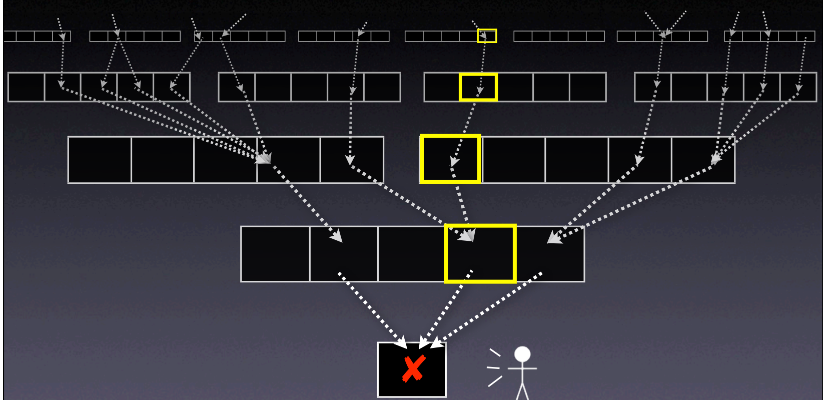


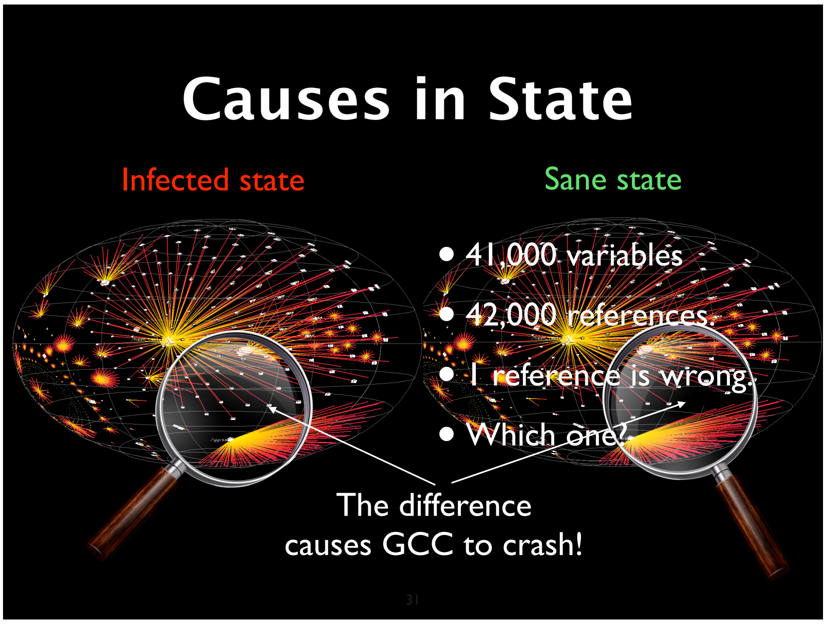
Tracing Infections

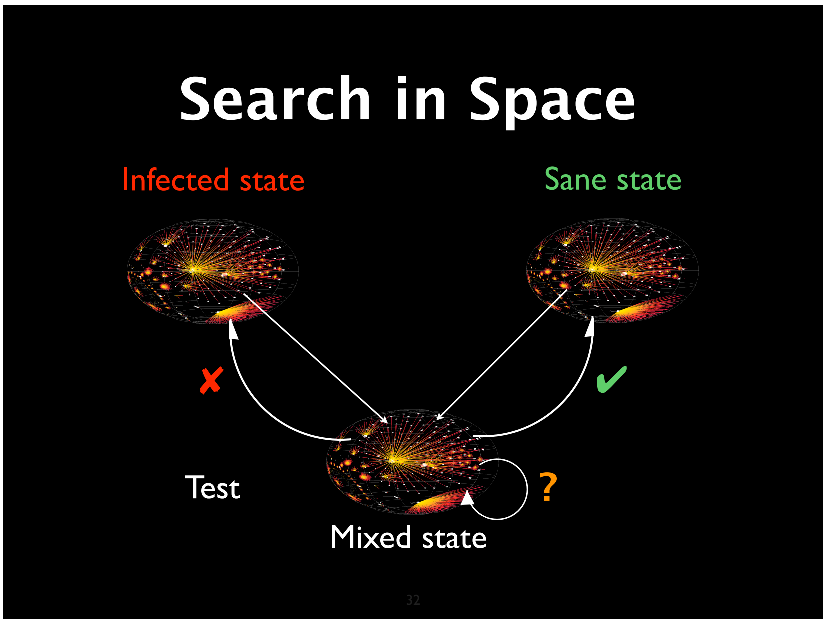
- For every infection, we must find the *earlier infection* that *causes* it.
- Program analysis tells us possible causes

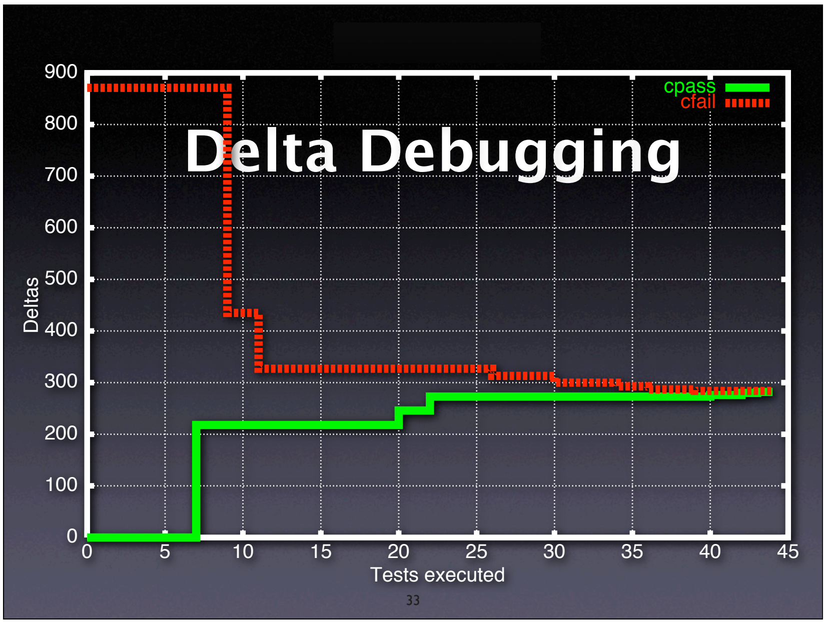


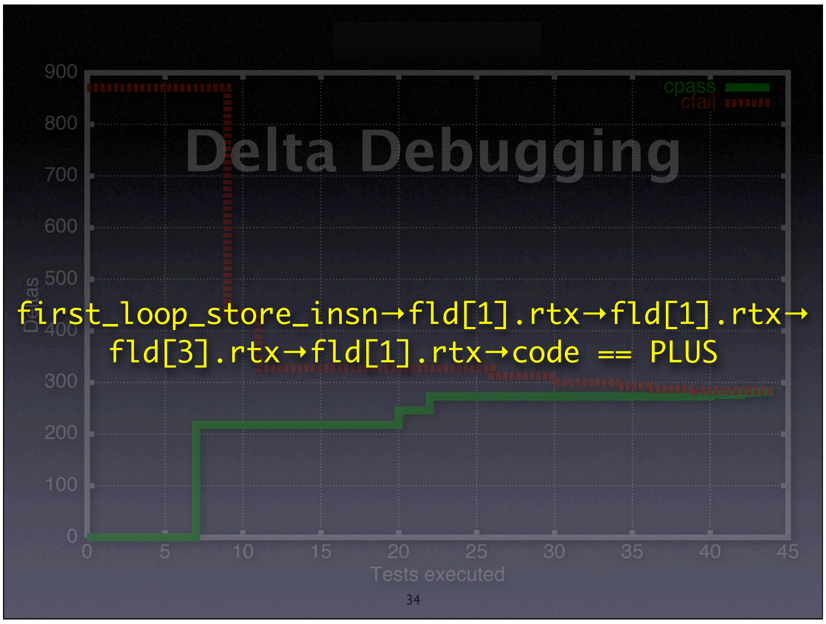
Tracing Infections

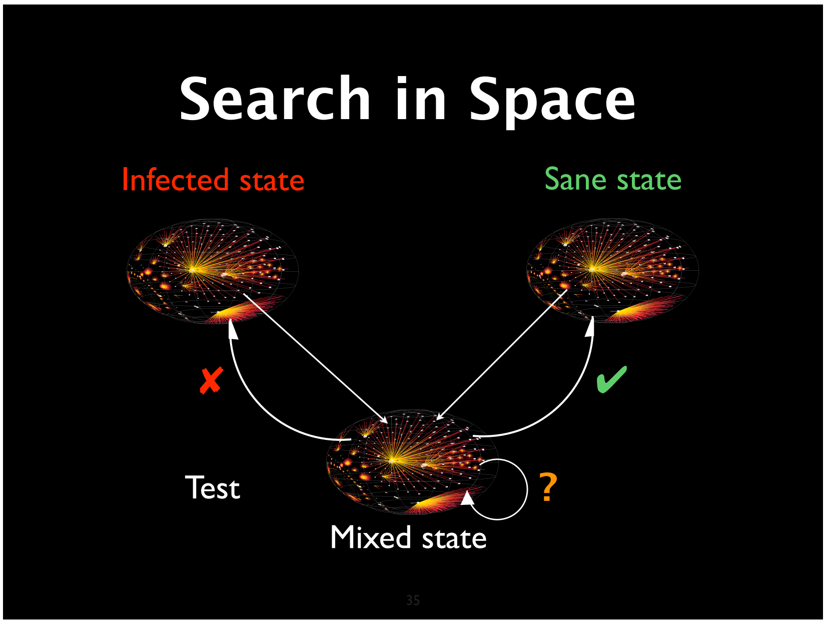


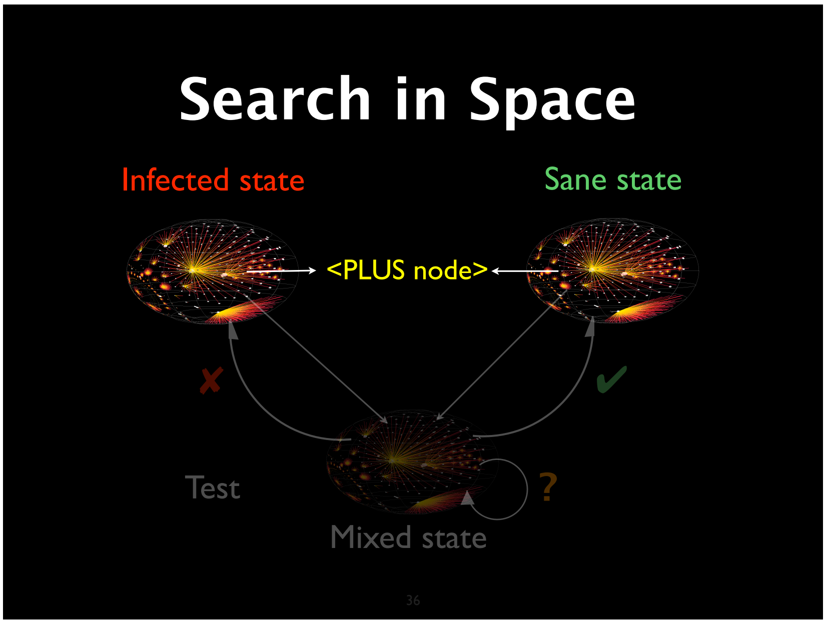

Causes in State
Infected state
Sane state
• 41,000 variables
• 42,000 references
• 1 reference is wrong
• Which one?
The difference
causes GCC to crash!
31

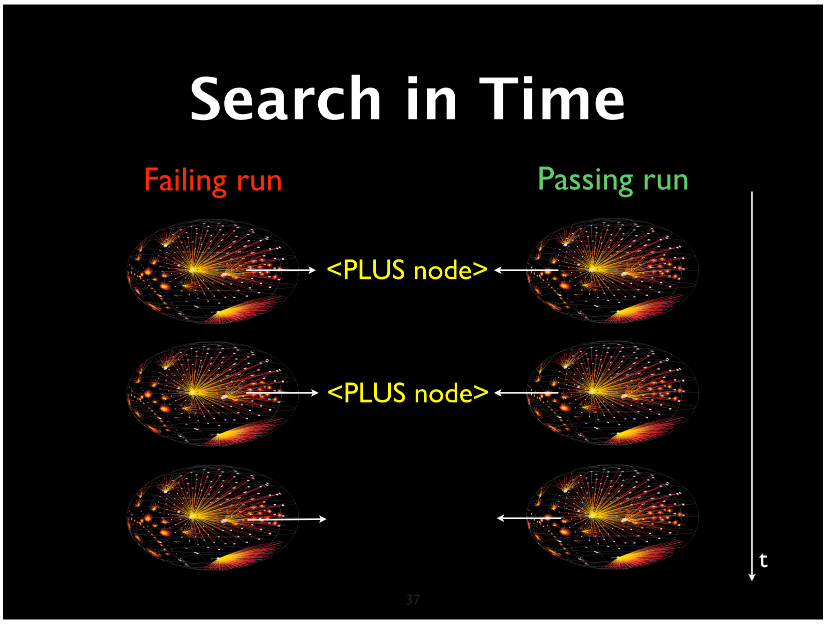

Search in Space
Infected state
Sane state
✘
✔
Test
?
Mixed state
32

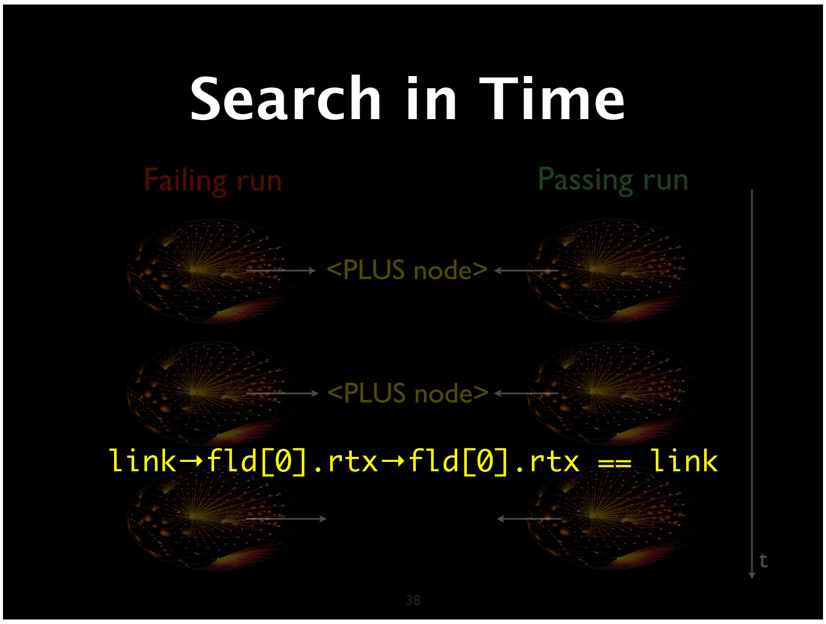

Delta Debugging
cpass
cfail
Deltas
900
800
700
600
500
400
300
200
100
0
0
5
10
15
20
25
30
35
40
45
Tests executed
33

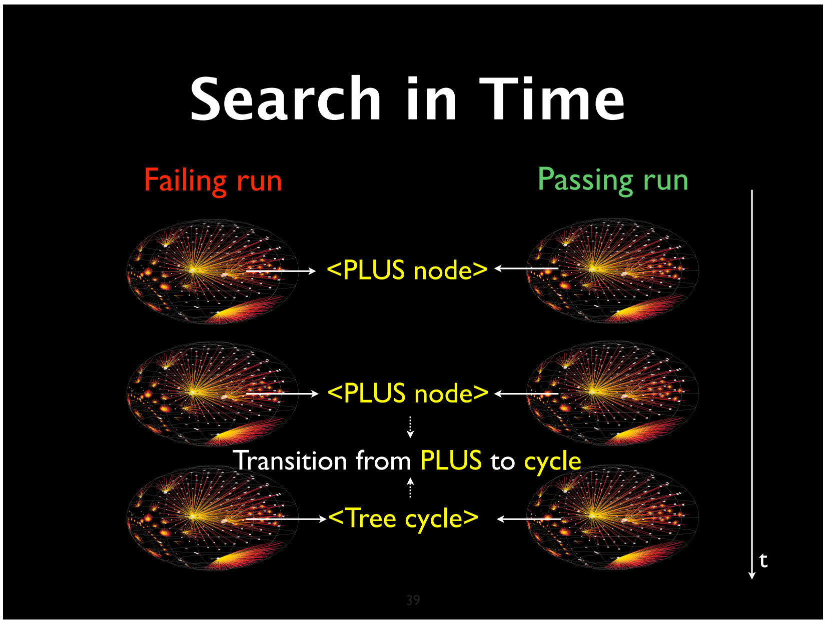

Delta Debugging
900
800
700
600
500
400
300
200
100
0
0
5
10
15
20
25
30
35
40
45
Tests executed
cpass
cfail
first_loop_store_insn→fld[1].rtx→fld[1].rtx→
fld[3].rtx→fld[1].rtx→code == PLUS
34


Search in Space
Infected state
Sane state
✘
✔
Test
?
Mixed state
35


Search in Space
Infected state
Sane state
<PLUS node>
✘
✔
Test
?
Mixed state
36


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
t
37


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
link→fld[0].rtx→fld[0].rtx == link
t
38


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
Transition from PLUS to cycle
<Tree cycle>
t
39

Why Transitions?

- Each failure cause in the program state is caused by some statement
- These statements are executed at cause transitions
- Cause transitions thus are statements that cause the failure

All GCC Transitions

Location	New cause at transition
<Start>	argv[3]
toplev.c:4755	name
toplev.c:2909	dump_base_name
c-lex.c:187	finput→_IO_buf_base
c-lex.c:1213	nextchar
c-lex.c:1213	yyssa[41]
c-typeck.c:3615	yyssa[42]
c-lex.c:1213	last_insn→fld[1].rtx→…→fld[1].rtx.code
c-decl.c:1213	sequence_result[2]→…→fld[1].rtx.code
combine.c:4271	x→fld[0].rtx→fld[0].rtx

combine.c

```
if (GET_CODE (XEXP (x, 0)) == PLUS {
    x = apply_distributive_law
      (gen_binary (PLUS, mode,
                                          (x, 0), 0),
                                          ),
                                 XEXP (XEXP (x, 0), 1),
                                 XEXP (x, 1))));

    if (GET_CODE (x) != MULT)
        return x;
}
```

2 lines out of 338,000

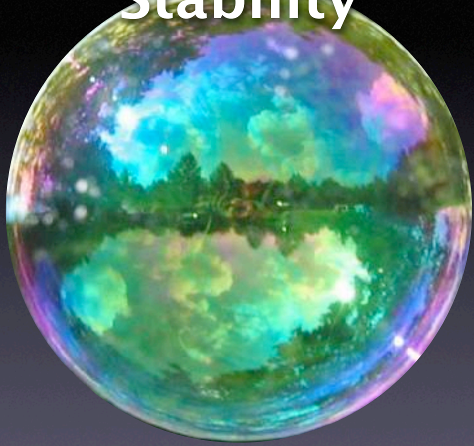
Should be copy_rtx()

Implementations

C	Java	Python
Web service + command line	Eclipse plug-in	Module
24 months	12 months	2 days

43

Stability

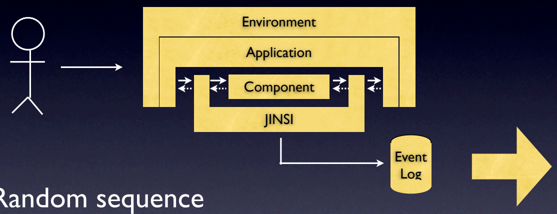


44

Stability

45

Isolating Relevant Calls



Random sequence of 28 operations

Event log contains 32 interactions

```
<init>
insert(Coin(25))
insert(Coin(10))
insert(Coin(25))
insert(Coin(25))
vend()
vend()
```

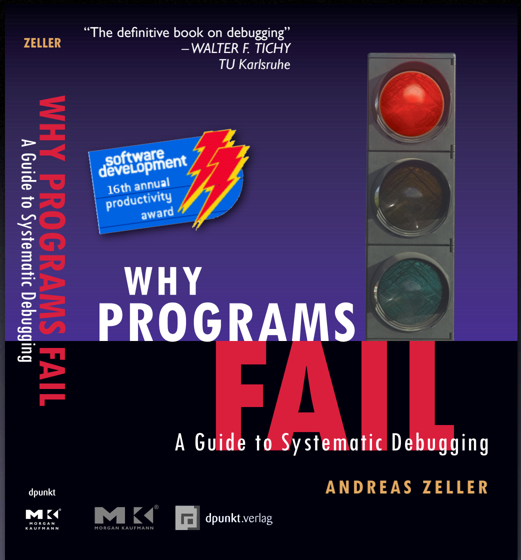
Minimal set to reproduce the failure: 7 interactions

IN: In order to show the feasibility of our JINSI tool, we have implemented a proof of concept.

OUT: These results are very

The Traffic Principle

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



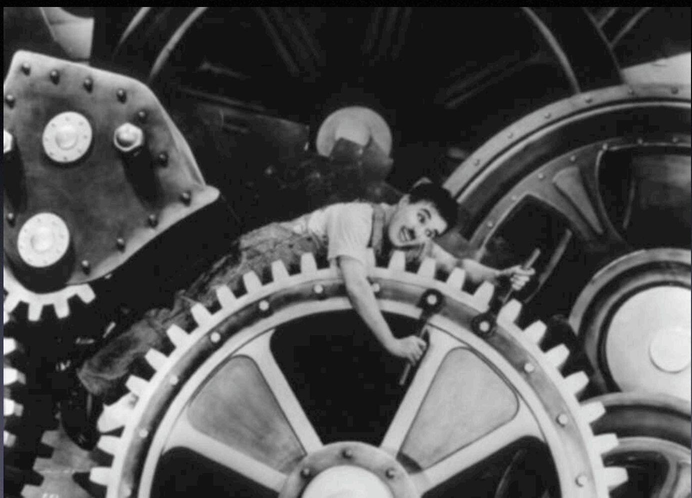
And if you need such a toolbox, I have written all these techniques down in a textbook.

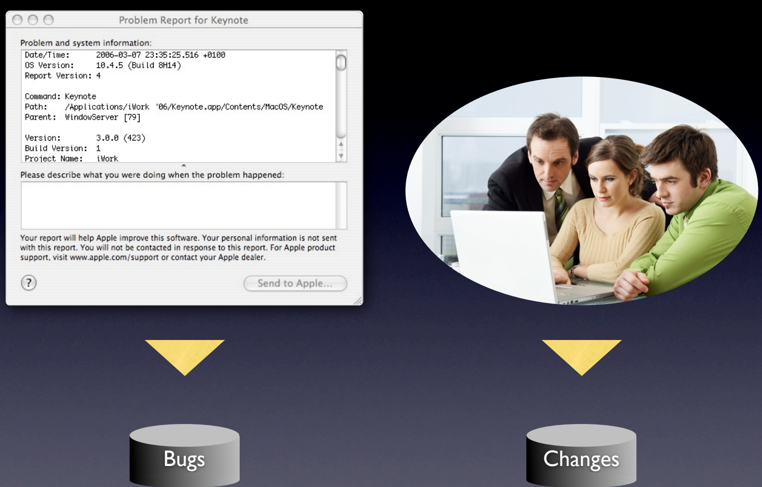
combine.c

```
if (GET_CODE (XEXP (x, 0)) == PLUS {
    x = apply_distributive_law
      (gen_binary (PLUS, mode,
                            (x, 0), 0),
                            ),
                  XEXP (XEXP (x, 0), 1),
                  XEXP (x, 1))));

     if (GET_CODE (x) != MULT)
         return x;
}
```

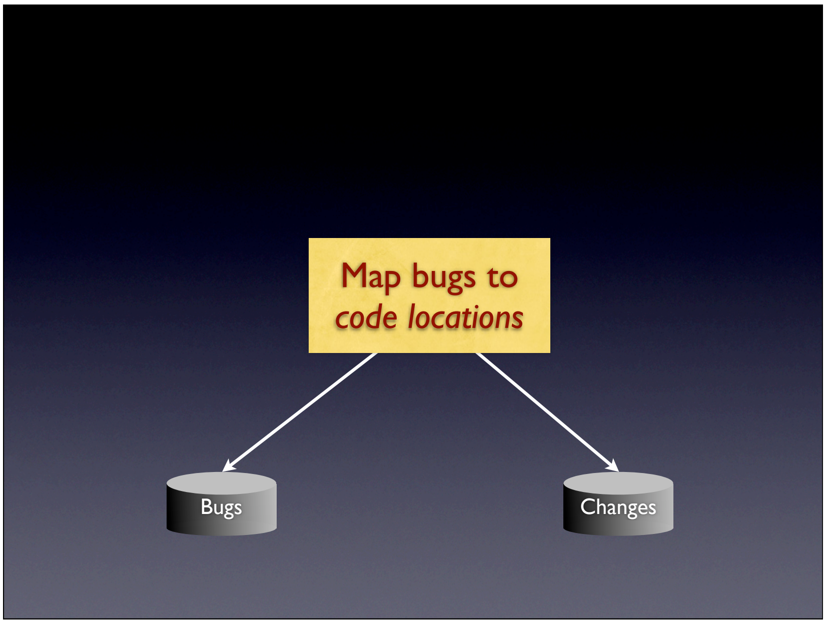
What is the cause of this error?

Should be copy_rtx()



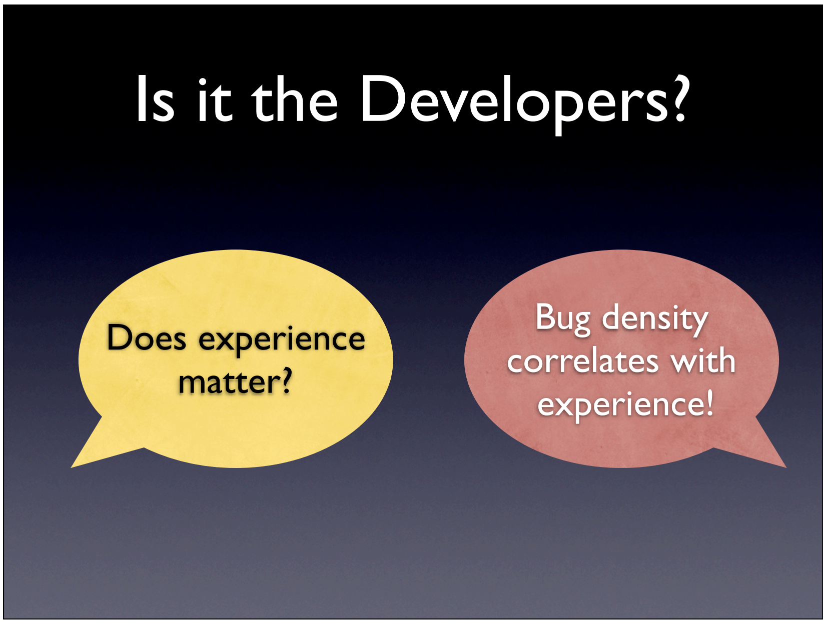


Such software archives are being used in practice all the time. If you file a bug, for instance, the report is stored in a bug database, and the resulting fix is stored in the version archive.



These databases can then be mined to extract interesting information. From bugs and changes, for instance, we can tell how many bugs were fixed in a particular location.

This is what you get when doing such a mapping for eclipse. Each class is a rectangle in here (the larger the rectangle, the larger its code); the colors tell the defect density – the brighter a rectangle, the more defects were fixed in here. Interesting question: Why are come modules so much more defect-prone than others? This is what has kept us busy for years now.



Is it History?

I found lots of bugs here. Will there be more?

Yes! (But where did these come from?)

How about metrics?

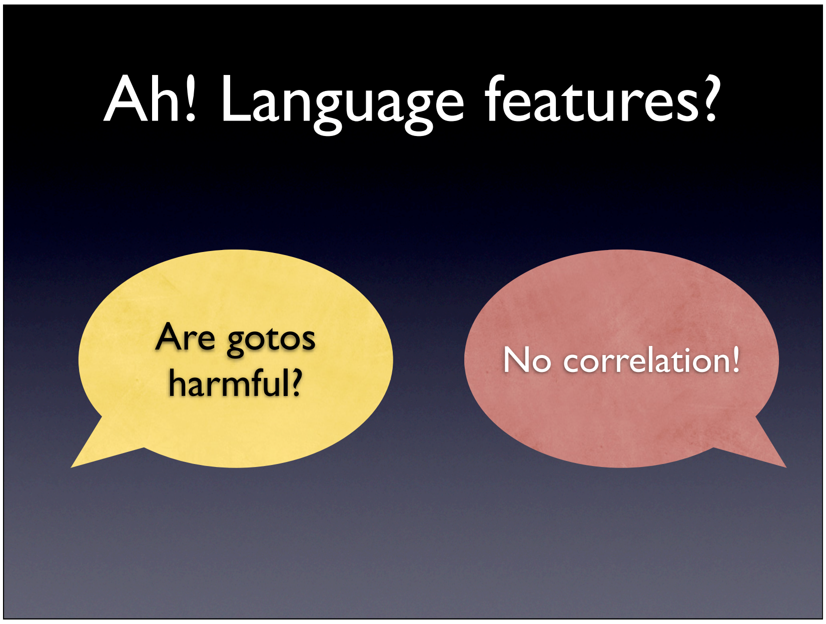
Do *code metrics* correlate with bug density?

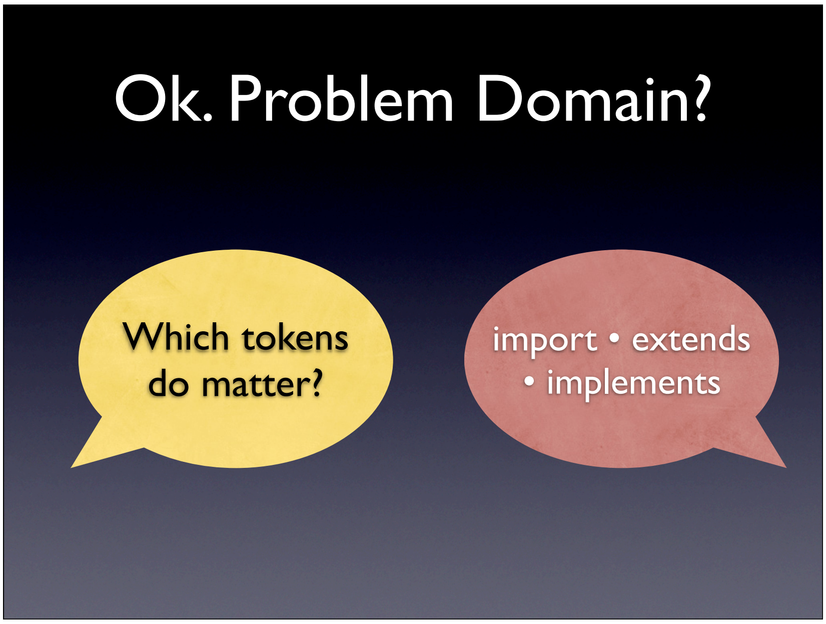
Sometimes!

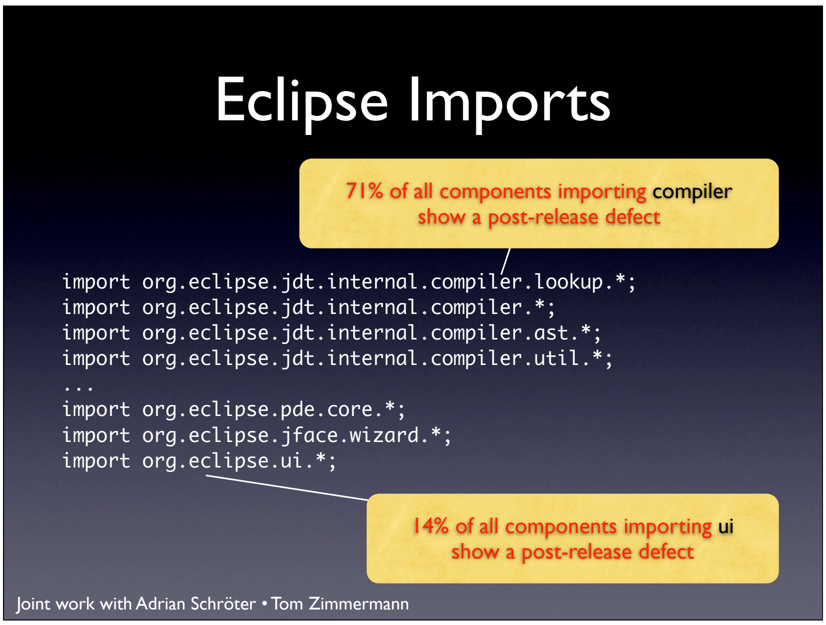
Uh. Coverage?

Does *test coverage* correlate with bug density?

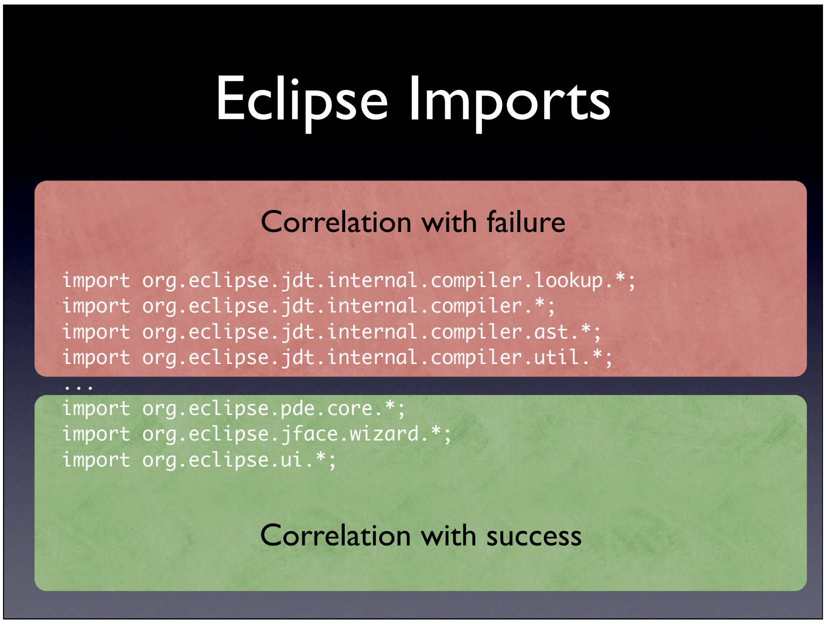
Yes –
the more coverage,
the more bugs!



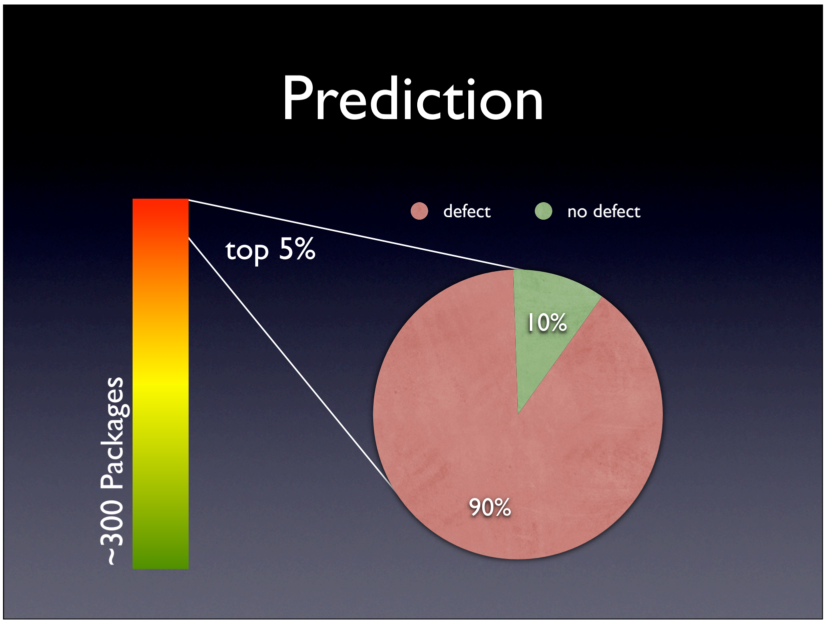




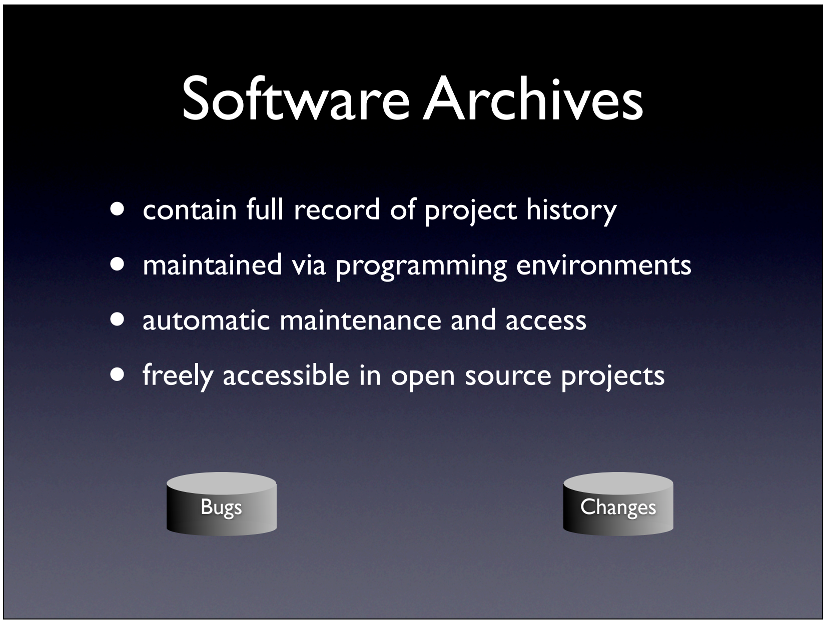
The best hint so far what it is that determines the defect-proneness is the import structure of a module. In other words: “What you eat determines what you are” (i.e. more or less defect-prone).



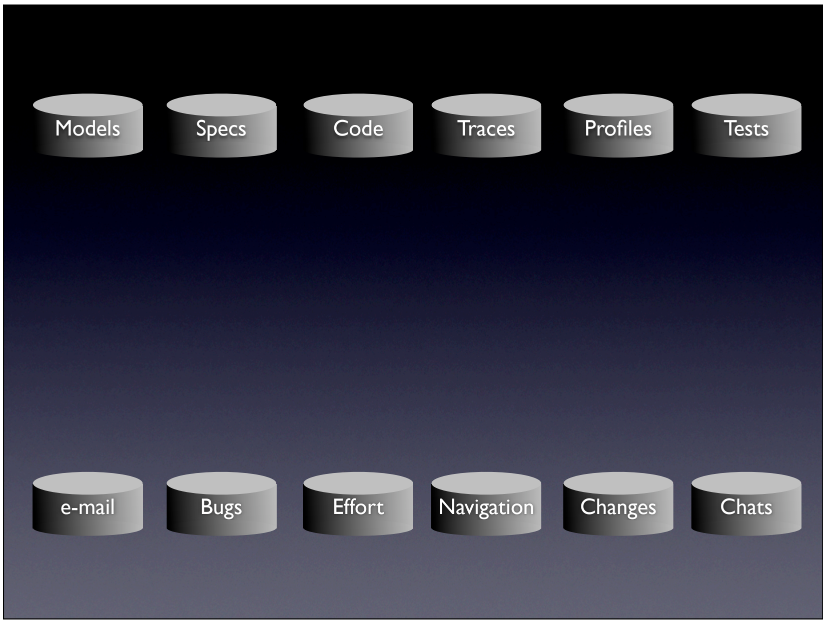
For instance, if your code is related to compilers, it is much more defect-prone, than, say, code related to user interfaces.



…and this is what we get if we rank 300 packages according to our predictor (which has learned from the remaining modules): if we look at the top 5%, 90% actually are defective. A random pick would have gotten us only 36%.

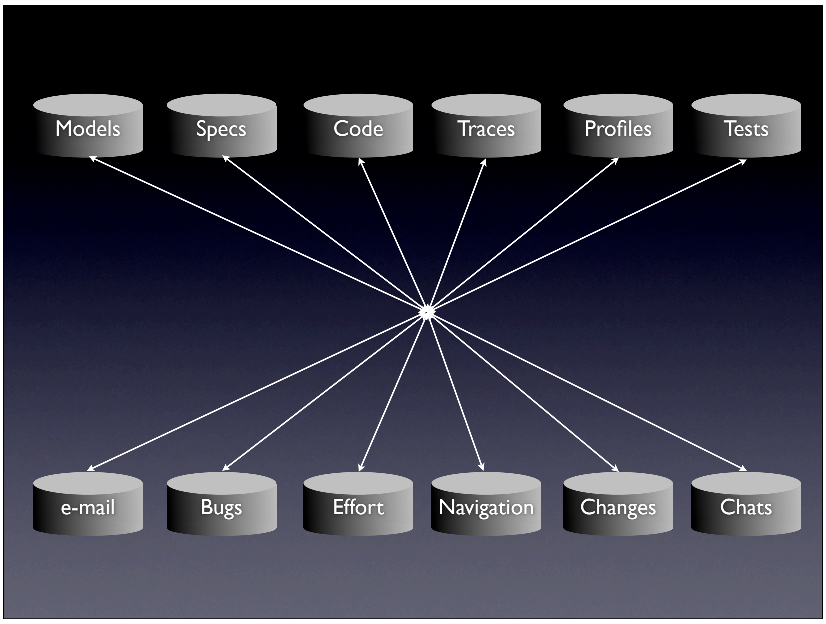


This was just a simple example. So, the most important aspect that software archives give you is automation. They are maintained automatically ("The data comes to you"), and they can be evaluated automatically ("Instantaneous results"). For researchers, there are plenty open source archives available, allowing us to test, compare, and evaluate

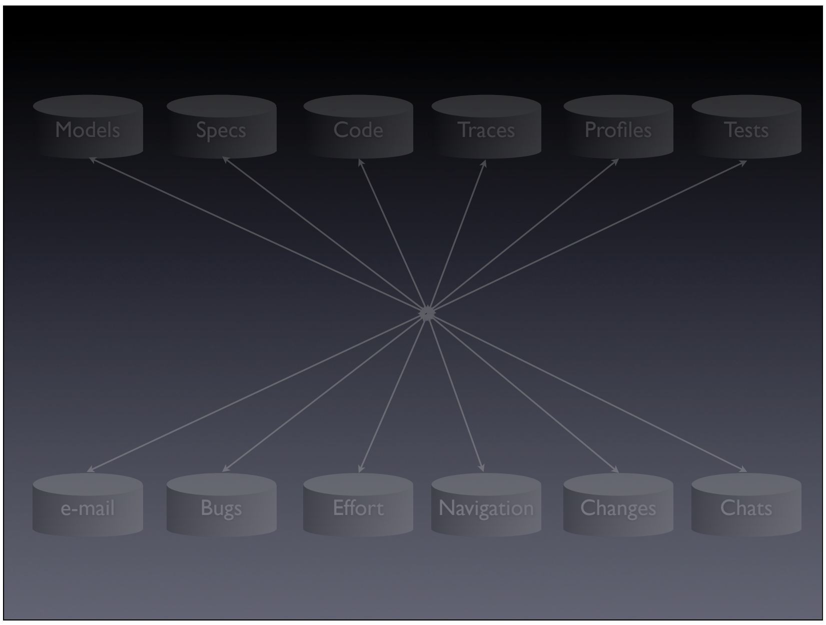


Tools can only work together if they draw on different artefacts

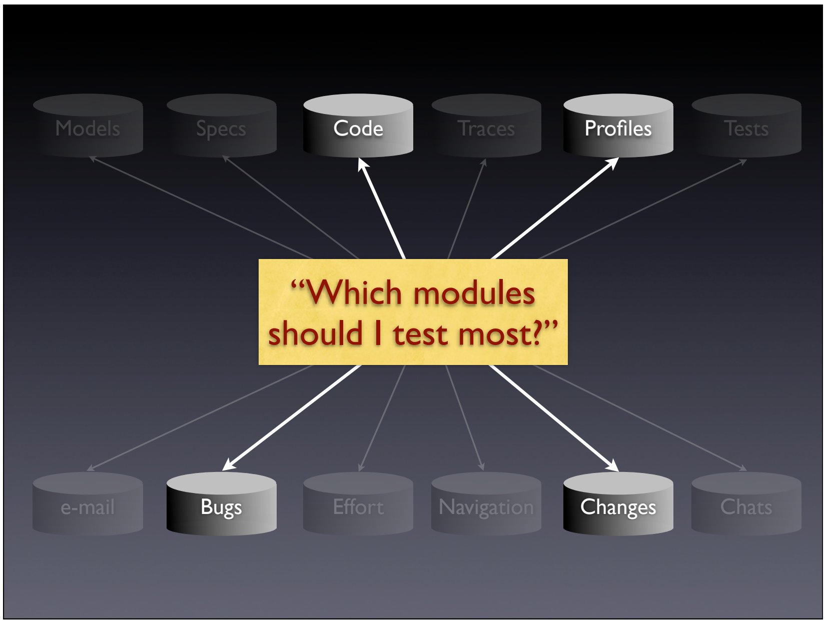
What are we working on in SE – we are constantly producing and



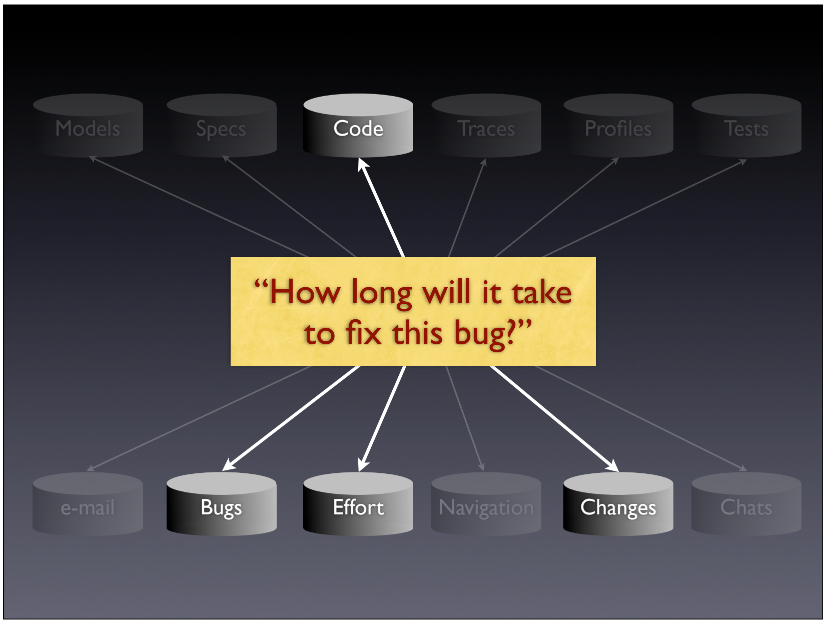
Combining these sources will allow us to get this “waterfall effect” – that is, being submerged by data; having more data than we could possibly digest.



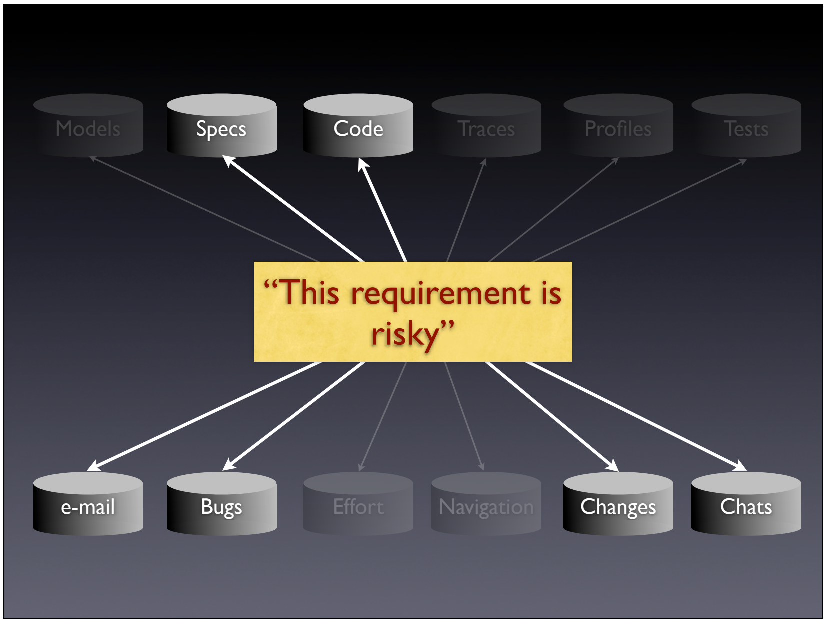
This is the oldest example, referring to work by Tom Zimmermann et al. at ICSE 2004 (and the work of Annie Ying et al. at the same time): You change one function – which others should be changed? This is easy to mine drawing on the change history and the code.



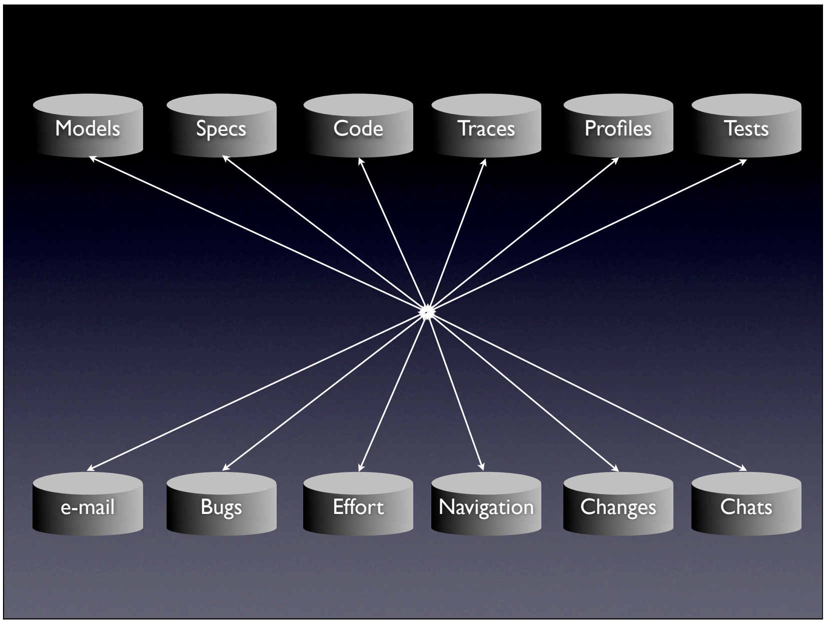
Defect density data as sketched before can be used to decide where to test most – of course, where the most defects are. If one additionally takes profiles (e.g. usage data) into account, one can even allocate test efforts to minimize the predicted potential damage optimally.



If one has effort data, one can tell how long it takes to fix a bug. Cathrin Weiß has a talk on this topic right after this keynote.



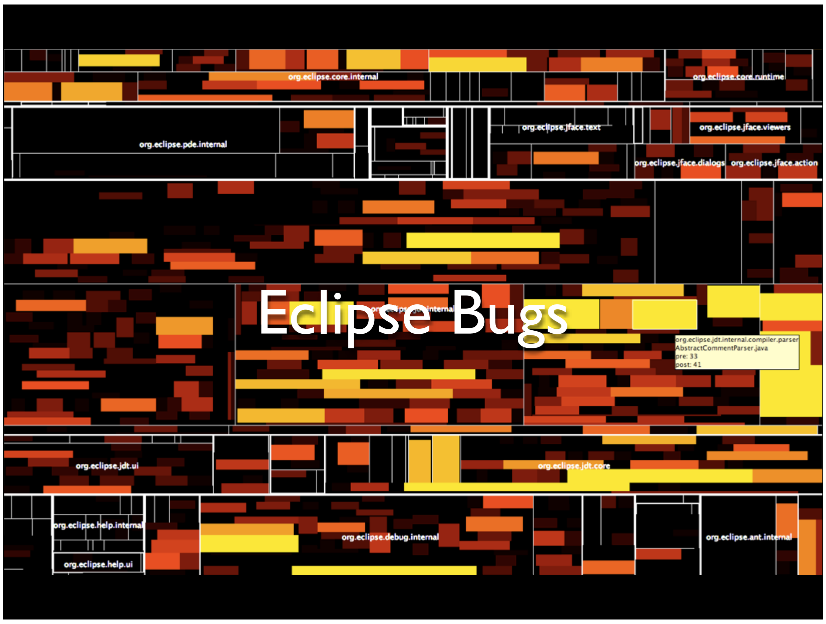
Finally, a glimpse into the future, taking natural language resources into account. The idea is to associate specs with (natural language) topics, and to map these topics to source code. What you then get is an idea of how specific topics (or keywords) influence failure probability, and this will allow you making predictions for specific requirements.



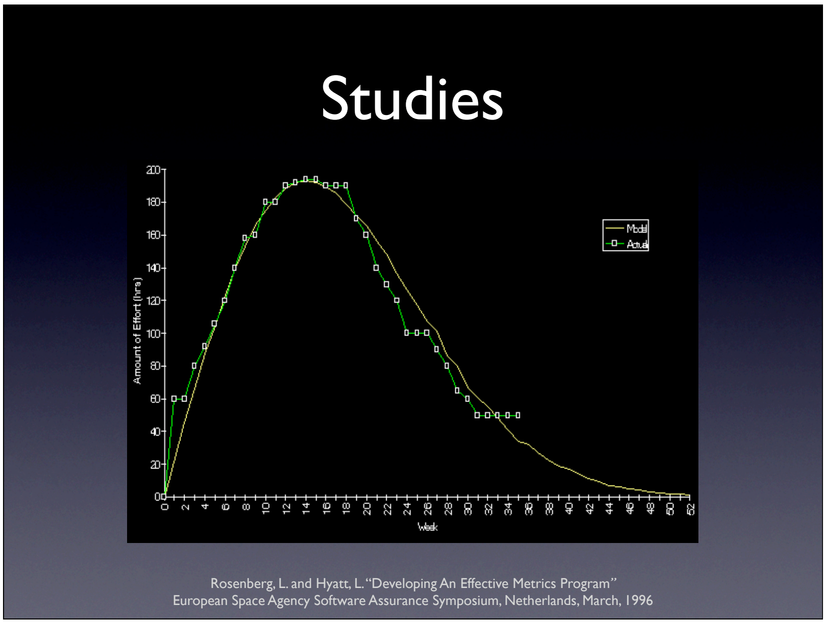
Combining these sources will allow us to get this "waterfall effect" – that is, being submerged by data; having more data than we could possibly digest.

The dirty story about this data is that it is frequently collected manually. In fact, the company phone book is among the most important tools of an empirical software engineering researchers. One would phone one developer after the other, and question them – say, "what was your effort", or "how often did you test module 'foo'?", and tick in the appropriate form. In other words, data is scarce, and as it is being collected from humans after the fact, is prone to errors, and prone to bias.

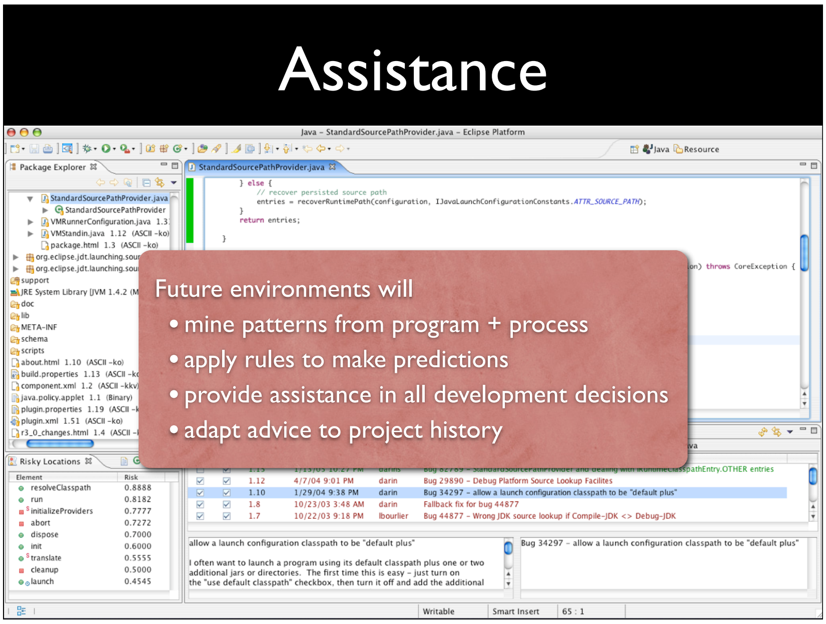
Combining these sources will allow us to get this “waterfall effect” – that is, being submerged by data; having more data than we could possibly digest.



This is what you get when doing such a mapping for eclipse. Each class is a rectangle in here (the larger the rectangle, the larger its code); the colors tell the defect density – the brighter a rectangle, the more defects were fixed in here. Interesting question: Why are come modules so much more defect-prone than others? This is what has kept us busy for years now.



Let’s now talk about results. What should our tools do? Should they come up with nice reports, and curves like this one?



Programming environments also are the tools that allow us to collect, maintain, and integrate all this project data. This is where the waterfall becomes imminent. In pair programming, you have a navigator peering over your shoulder, giving you advice whether what you are doing is good or bad. We want the environment peer over your shoulder – as an automated "developer's buddy". Whatever we do must stand the test of the developers – if they accept it, it will be good enough.