Where do Bugs come from?

Andreas Zeller • Saarland University

Woher kommen Software-Fehler?

Jeder Programmierer kennt die Situation: Ein Programm läuft nicht so, wie es soll. Ich stelle Techniken vor, die automatisch

(a) die Ursachen eines Fehlverhaltens finden - indem wir genau die Aspekte isolieren, die das Zustandekommen eines Fehlers verursachen;
(b) Programmfehler finden - indem wir aus dem Code "normale" Anweisungsfolgen lernen und nach Abweichungen suchen; und
(c) vorhersagen, wo in Zukunft Fehler auftreten werden - indem wir maschinell lernen, welche Code- und Prozesseigenschaften bisher mit Fehlern korrelierten.

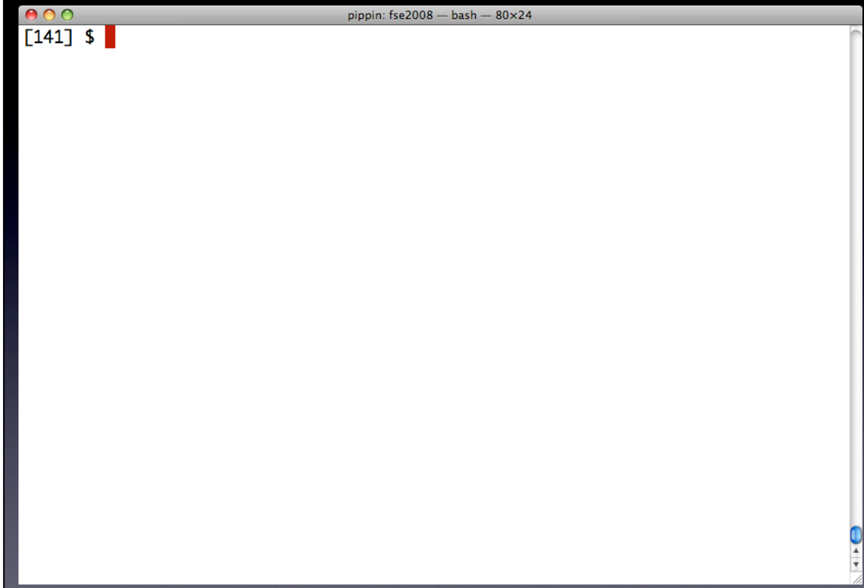
Fallstudien an echten Programmen mit echten Fehlern, von AspectJ über Firefox zu Windows demonstrieren die Praxistauglichkeit der vorgestellten Verfahren.

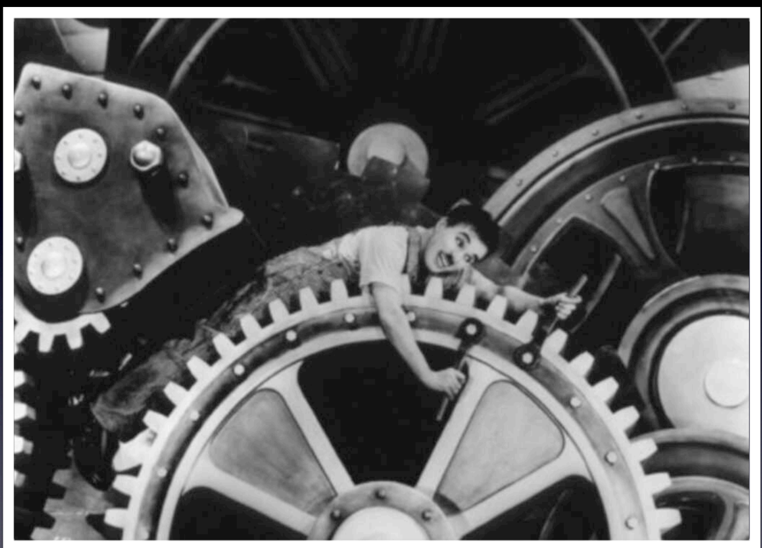
Andreas Zeller ist Professor für Softwaretechnik an der Universität des Saarlandes in Saarbrücken. Sein Forschungsgebiet ist die Analyse großer Software-Systeme und deren Fehler. Sein Buch "Why Programs Fail - A Guide to Systematic Debugging" wurde 2006 mit dem Jolt Software Development Productivity Award ausgezeichnet.

aspectj *crosscutting objects for better modularity*

bug.aj

```
@interface A {}

aspect Test {
  declare @field : @A int var* : @A;
  declare @field : int var* : @A;

  interface Subject {}

  public int Subject.vara;
  public int Subject.varb;
}

class X implements Test.Subject {}
```



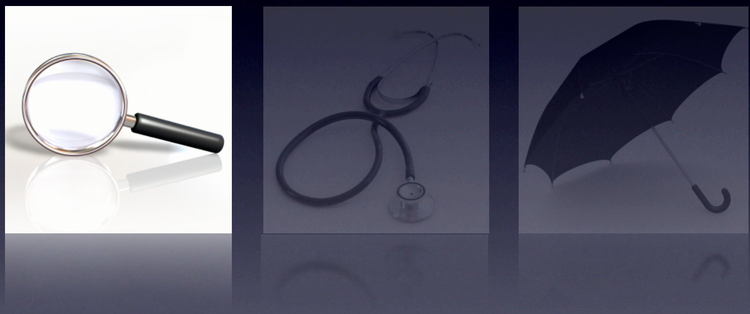


Diagnosis

Detection

Prevention

Diagnosis Detection Prevention



bug.aj

```
@interface A {}

aspect Test {
   declare @field : @A int var* : @A;
   declare @field : int var* : @A;

   interface Subject {}

   public int Subject.vara;
   public int Subject.varb;
}

class X implements Test.Subject {}
```

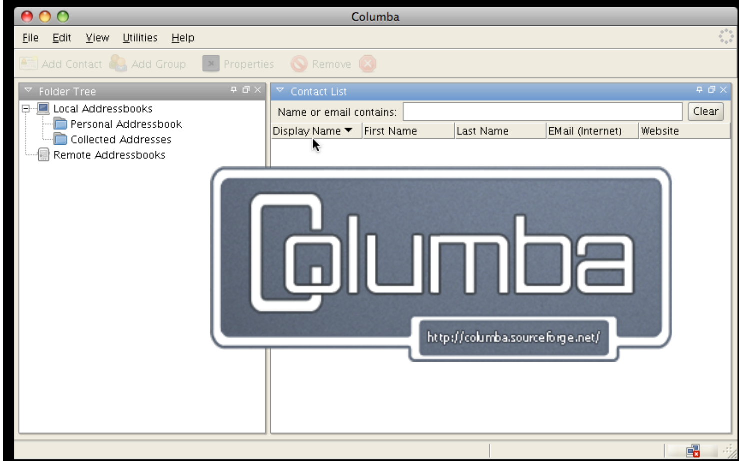
ajc Stack Trace

```
java.util.NoSuchElementException
   at java.util.AbstractList$Itr
      .next(AbstractList.java:427)
   at org.aspectj.weaver.bcel.BcelClassWeaver
      .weaveAtFieldRepeatedly
      (BcelClassWeaver.java:1016)
```

We can fix this by looking at the stack trace.

weaveAtFieldRepeatedly

```
for (Iterator iter = itdFields.iterator();
      iter.hasNext();) {
    ...
    for (Iterator iter2 = worthRetrying.iterator();
         iter.hasNext();) {
        ...
    }
}
```



mozilla.csv

```
Lenhof, Hans-Peter,, +49 681 302-64701, lenhof@cs.uni-sb.de
Lindig, Christian, +49 681 9358406, +49 681 302 5590, lindig@cs.uni-sb.de
Mehlmann, Martin,,, mehlmann@st.cs.uni-sb.de
Meyer zu Tittingdorf, Frie[illegible] 681 302-58099,
meyer@cs.uni-sb.de
Mileva, Yana,, +49 681 302-64020, mileva@cs.uni-sb.de
Müller-Perich, Elisabeth,, +49 681 302-5070, sekr.techfak@rz.uni-sb.de
Nir-Bleimling, Naomi,, +49 68130264011, naomi@wjpserver.cs.uni-sb.de
Offergeld, Thilo,, +49 681 302-6594, t.offergeld@univw.uni-sb.de
PC, CC 2006,,, cc2006pc@st.cs.uni-sb.de
Paul, Wolfgang, +4968171825, +49 6813022436, wjp@cs.uni-sb.de
Premraj, Rahul, +44 7796953511, +49 681 302-64013, premraj@cs.uni-sb.de
Reindel, Erich, +49 6371 912842, +49 681 302-58091, reindel@cs.uni-sb.de
Schuler, David,, +49 681 302-64026, schuler@st.cs.uni-sb.de
Schuler, Erika,, +49 6813025069, schuler@tf.uni-sb.de
Schäfer, Christa, +49 6897 51165, +49 68130264011,
Security, AG,,, security@st.cs.uni-sb.de
Seidel, Raimund, +49 6894 383698, +49 681 302-4513, rseidel@cs.uni-sb.de
Sekretariat, Sekretariat,, +49 681 302-64011, office@st.cs.uni-sb.de
Sliwerski, Jacek, +491741333208,, sliwers@st.cs.uni-sb.de
Slusallek, Philipp, +49 6826 1 88 71 32, +49 681 302-3830, slusallek@cs.uni-sb.de
Slusallek USA, Philipp, +1 650 391 9186, +1 408 486 2788, slusallek@cs.uni-sb.de
Smolka, Gert, +49 681 582770, +49 681 302-5311, smolka@ps.uni-sb.de
Software-Evolution, AG,,, softevo@st.cs.uni-sb.de
Thiel, Frank,,, hausmeister@cs.uni-sb.de
Weiß, Cathrin,,, weiss@st.cs.uni-sb.de
Wilhelm, Reinhard,, +49 681 302-4399, wilhelm@cs.uni-sb.de
Zeller, Andreas,,, zeller@cs.uni-sb.de
Zeller, Andreas, +49 681 3710465, +49 681 302-64011, zeller@cs.uni-sb.de
Zimmermann, Tom, +49 851 51542 (Eltern), +1 403 210 9470, zimmerth@cs.uni-sb.de
```

Now, the idea is that we can easily automate the whole process.

Problem:
Simplifying manually
is inhuman.

Delta Debugging

Delta Debugging *isolates failure causes automatically:*

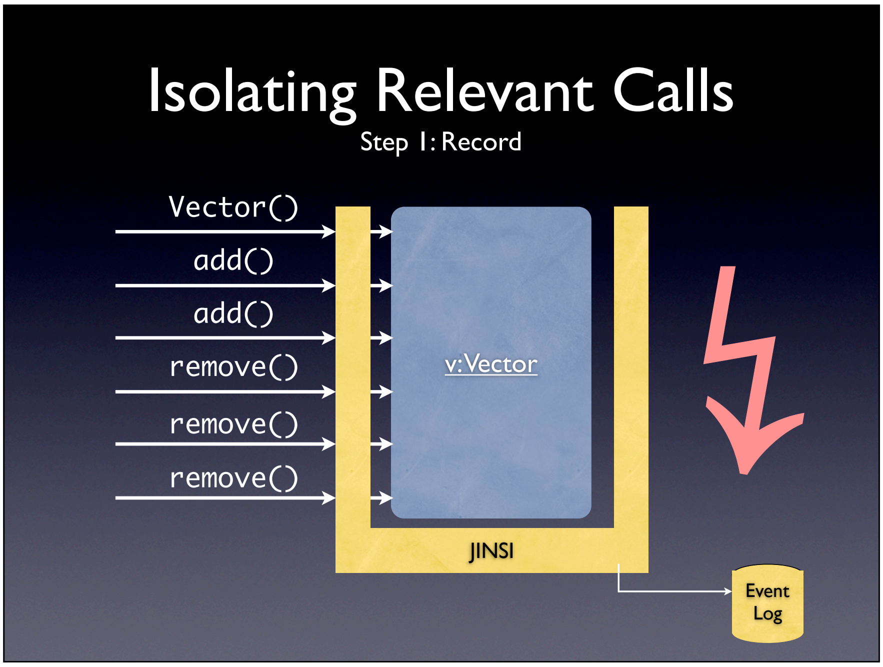
Inputs: 1 of 436 Columba contacts

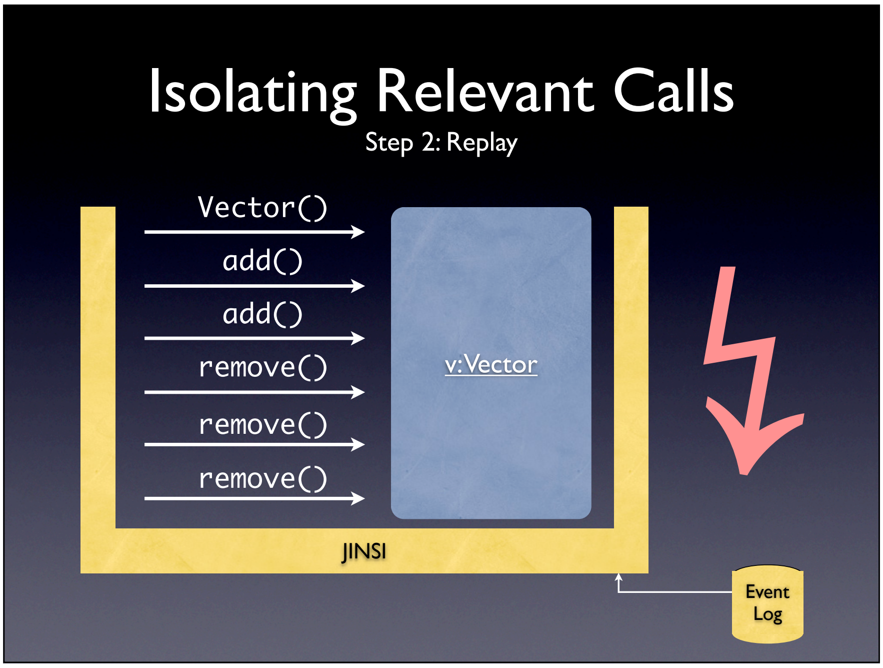
Code changes: 1 of 8,721 code changes in GDB

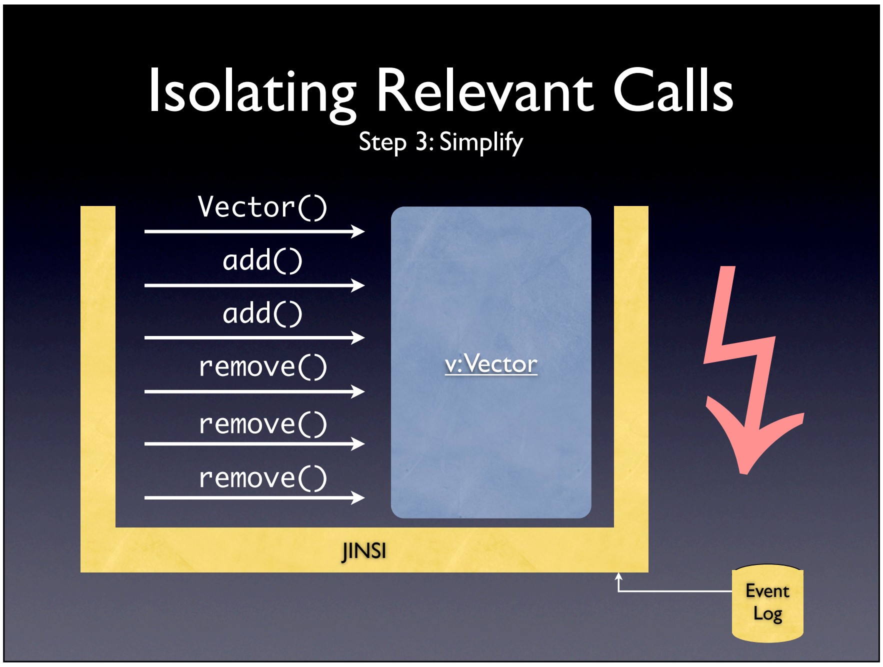
Threads: 1 of 3.8 bln thread switches in Scene.java

Fully automatic + purely test-based

Problem:
Simulating user interaction
is cumbersome.

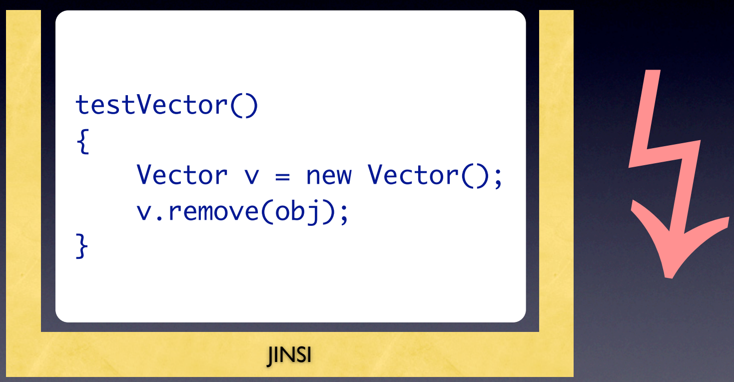

Isolating Relevant Calls
Step 1: Record
Vector()
add()
add()
remove()
remove()
remove()
v:Vector
JINSI
Event Log


Isolating Relevant Calls
Step 2: Replay
Vector()
add()
add()
remove()
remove()
remove()
v:Vector
JINSI
Event Log


Isolating Relevant Calls
Step 3: Simplify
Vector()
add()
add()
remove()
remove()
remove()
v:Vector
JINSI
Event Log

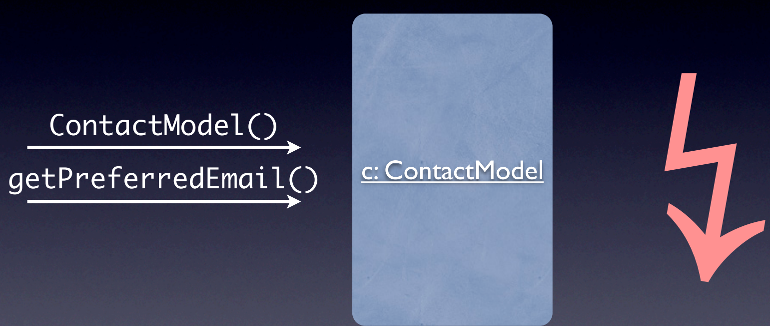
Isolating Relevant Calls

Step 4: Create Unit Test



Columba ContactModel

Columba ContactModel



Unit Test

```
testContactModel()
{
    ContactModel c = new ContactModel();
    String s = c.getPreferredEmail();
}
```

getPreferredEmail

```
public String getPreferredEmail() {
   Iterator it = getEmailIterator();

   // get first item
   IEmailModel model = (IEmailModel) it.next();

   // backwards compatiblity
   // -> its not possible anymore to create a
   // contact model without email address
   if (model == null)
      return null;

   return model.getAddress();
}
```

Delta Debugging

Delta Debugging *isolates failure causes automatically:*

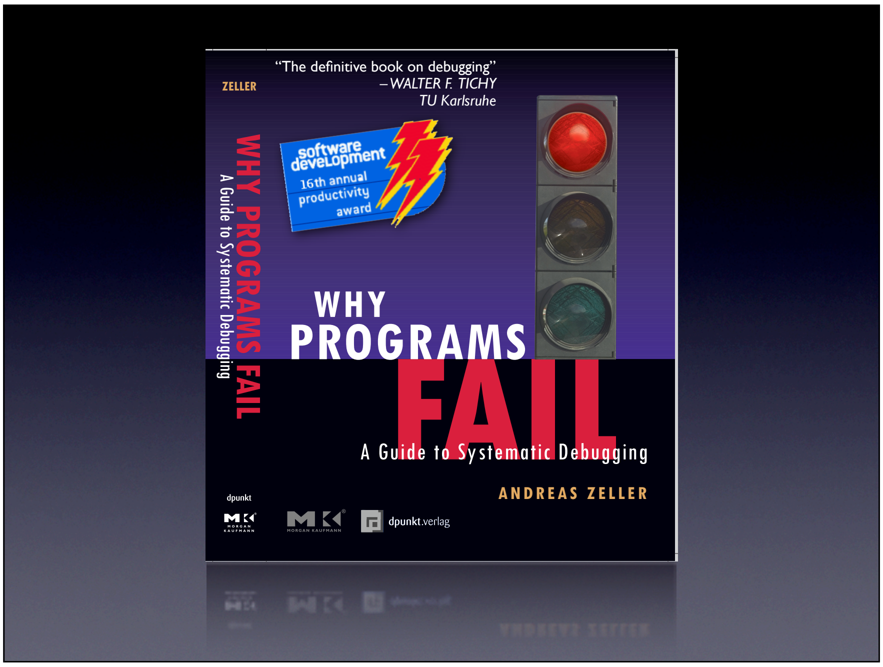
Inputs: 1 of 436 Columba contacts

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

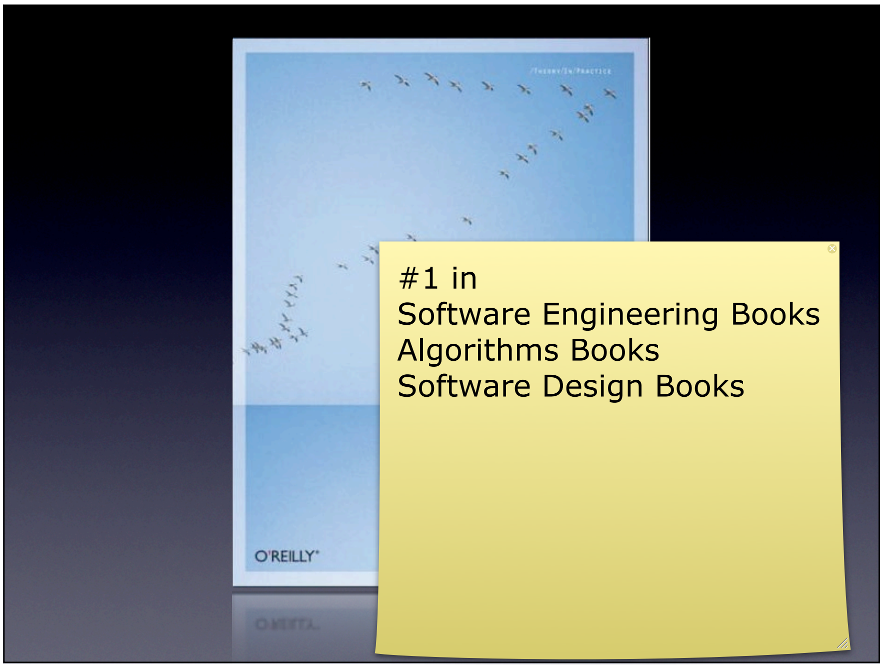
Threads: 1 of 3.8 bln thread switches in Scene.java

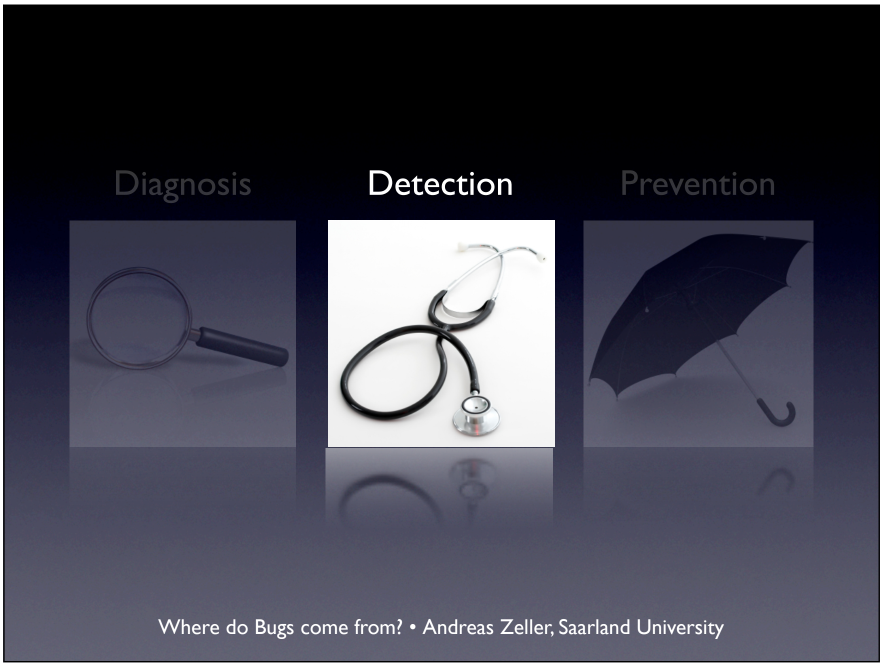
Calls: 2 of 18738 method calls

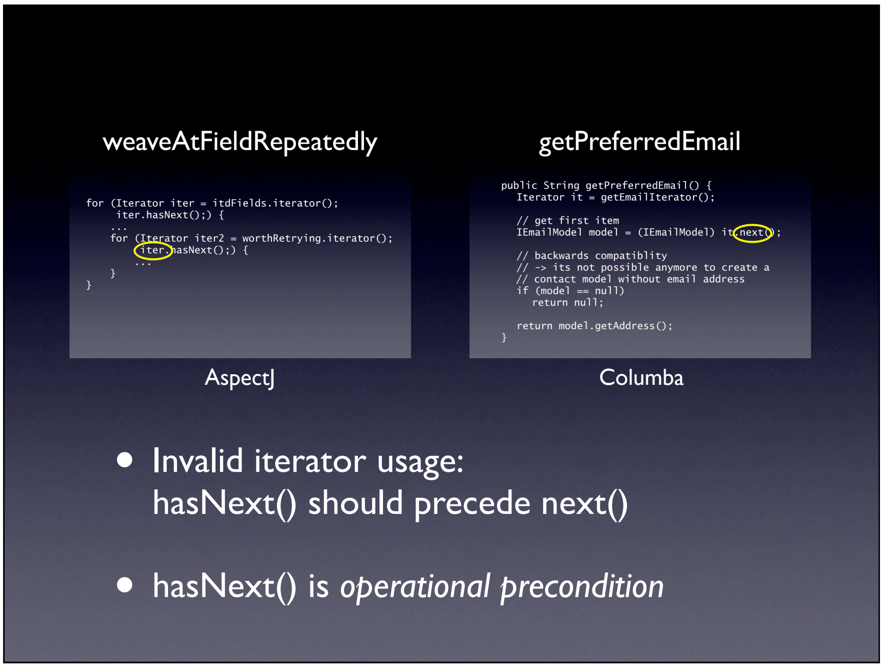
Fully automatic + purely test-based



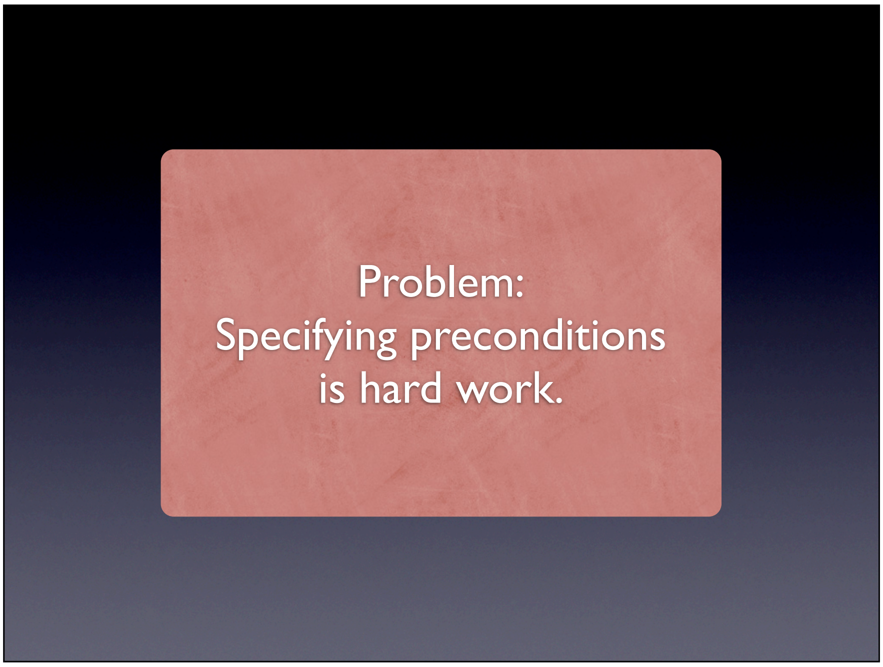
And if you need such a toolbox, I have written all these techniques down in a textbook.



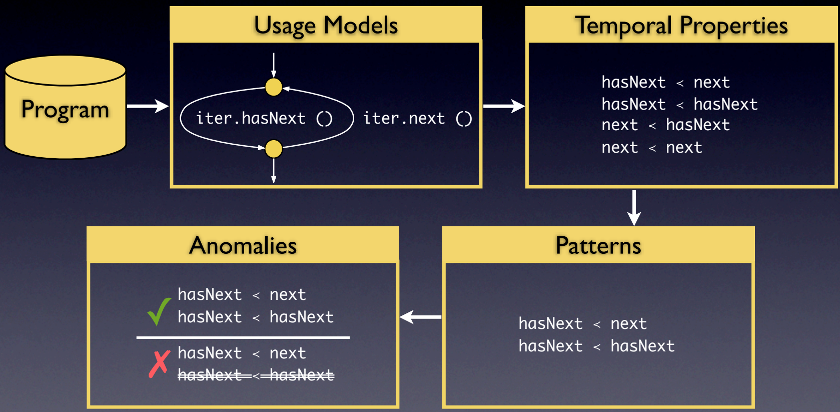




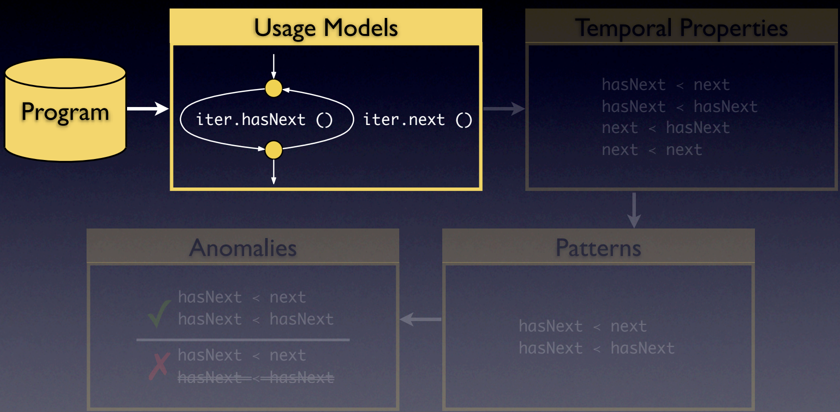
Kann man spezifizieren – eleganter ist aber das Extrahieren aus Code



OP-Miner



OP-Miner

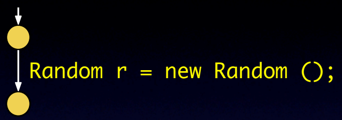


Method Models

```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```

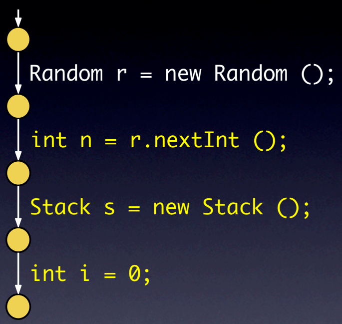
Method Models

```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```



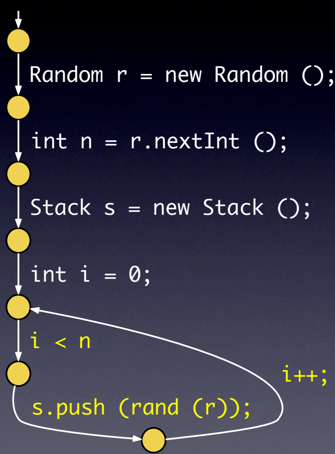
Method Models

```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```



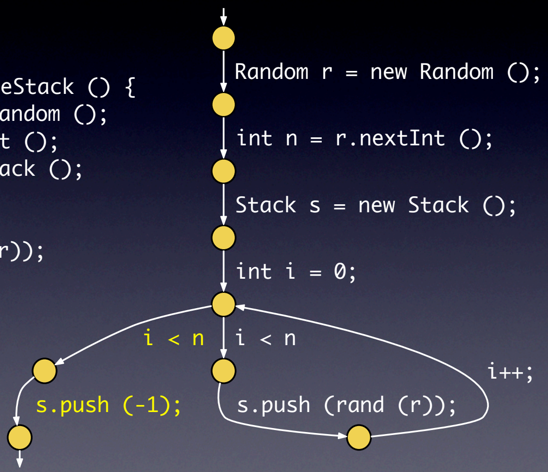
Method Models

```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```



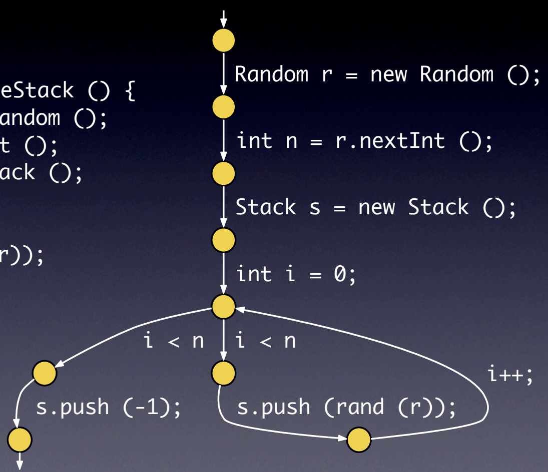
Method Models

```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```

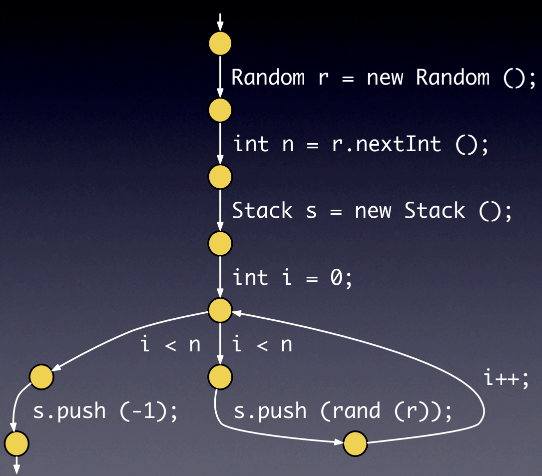


Method Models

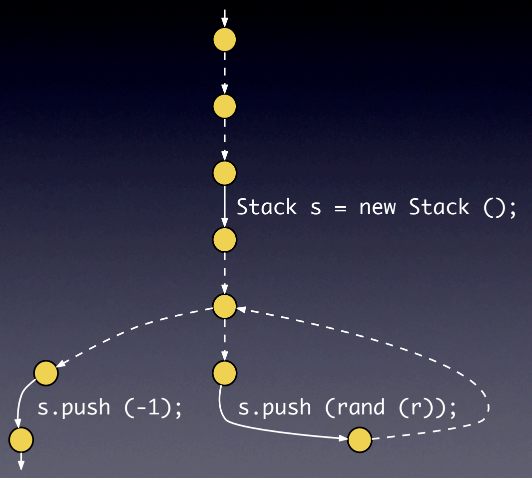
```
public Stack createStack () {
  Random r = new Random ();
  int n = r.nextInt ();
  Stack s = new Stack ();
  int i = 0;
  while (i < n) {
    s.push (rand (r));
    i++;
  }
  s.push (-1);
  return s;
}
```



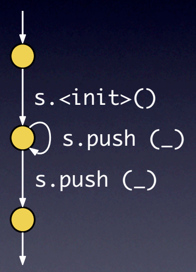
Usage Models



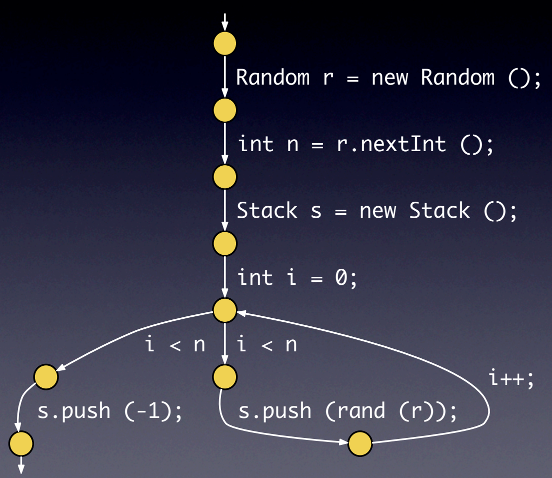
Usage Models

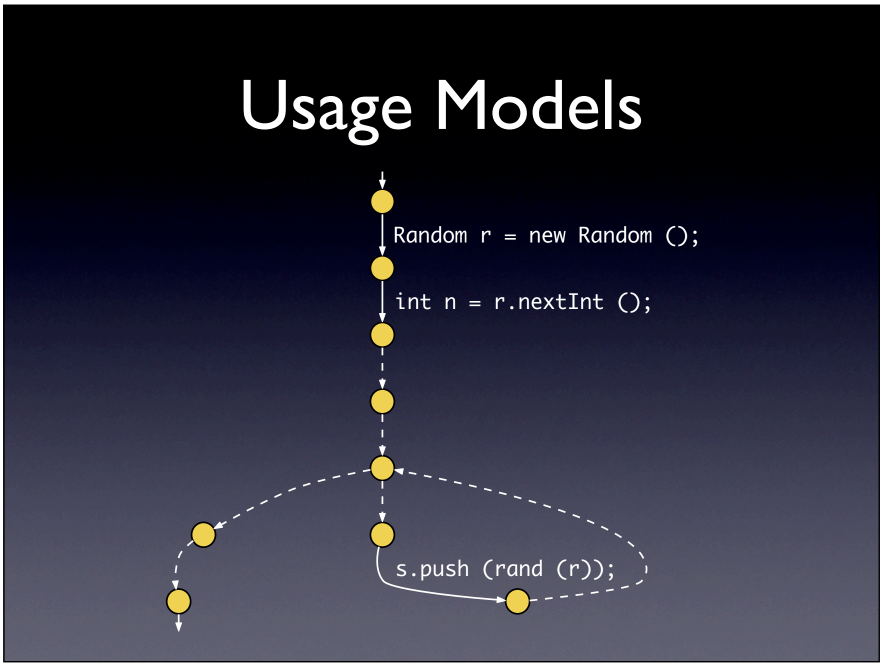


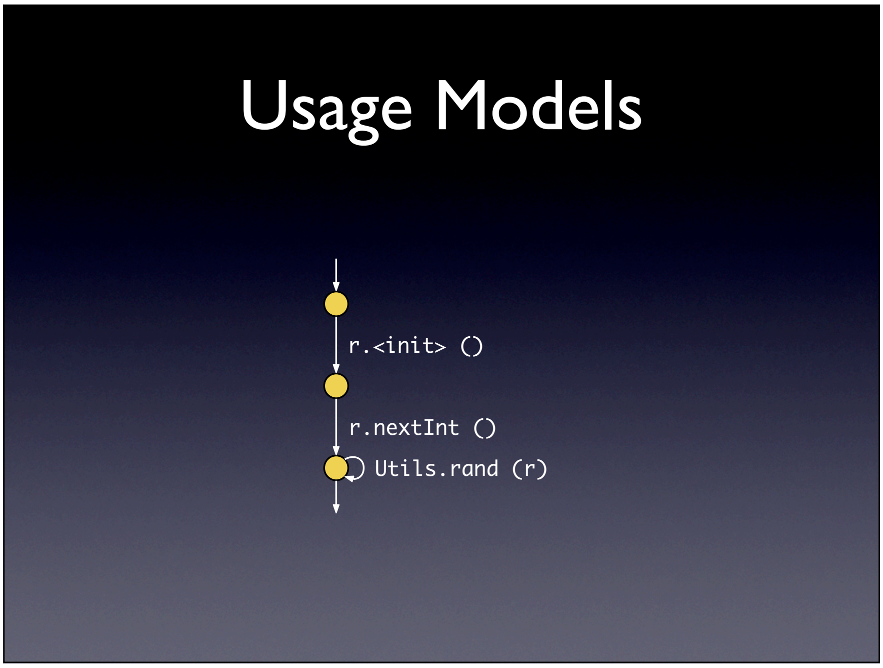
Usage Models

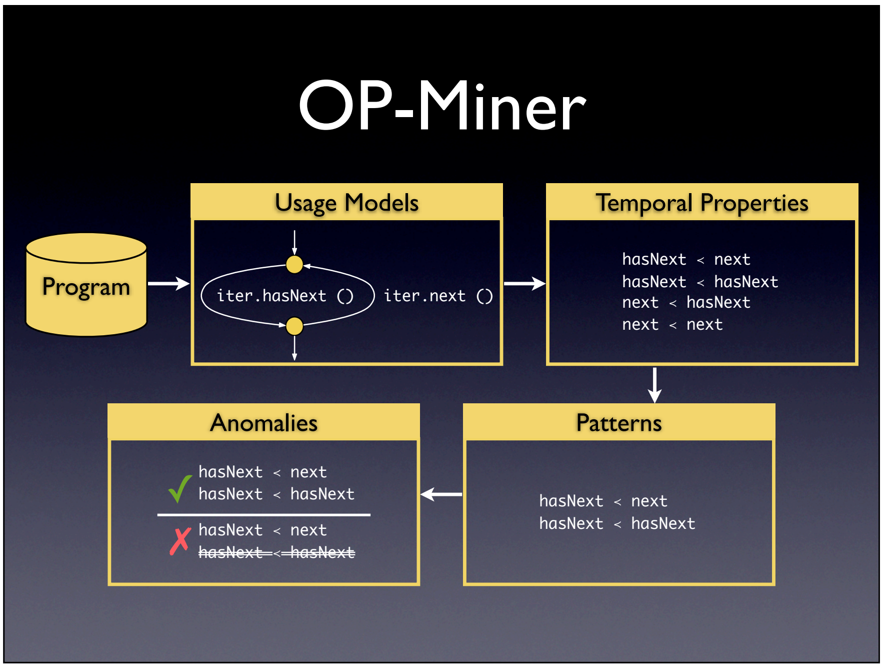


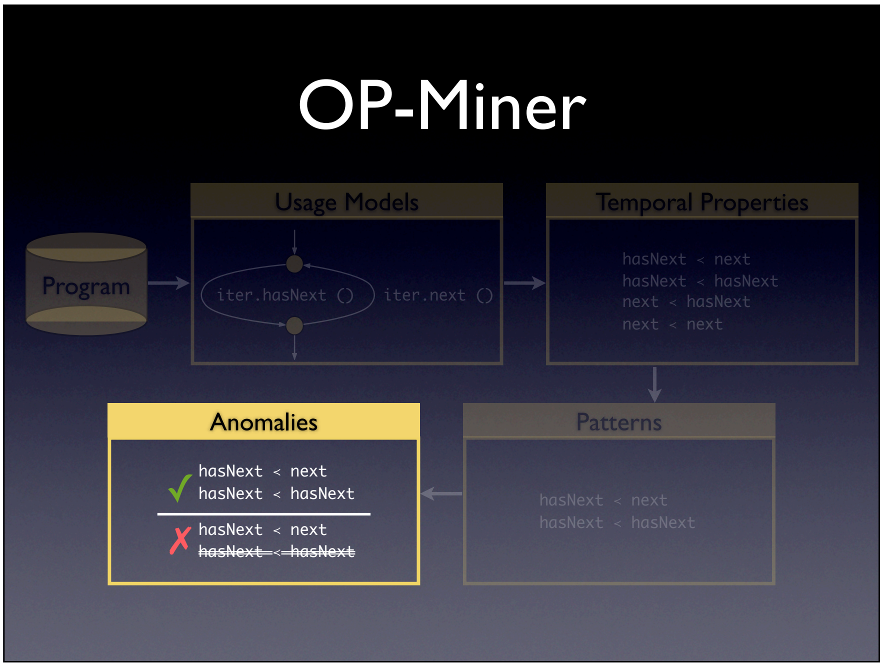
Usage Models

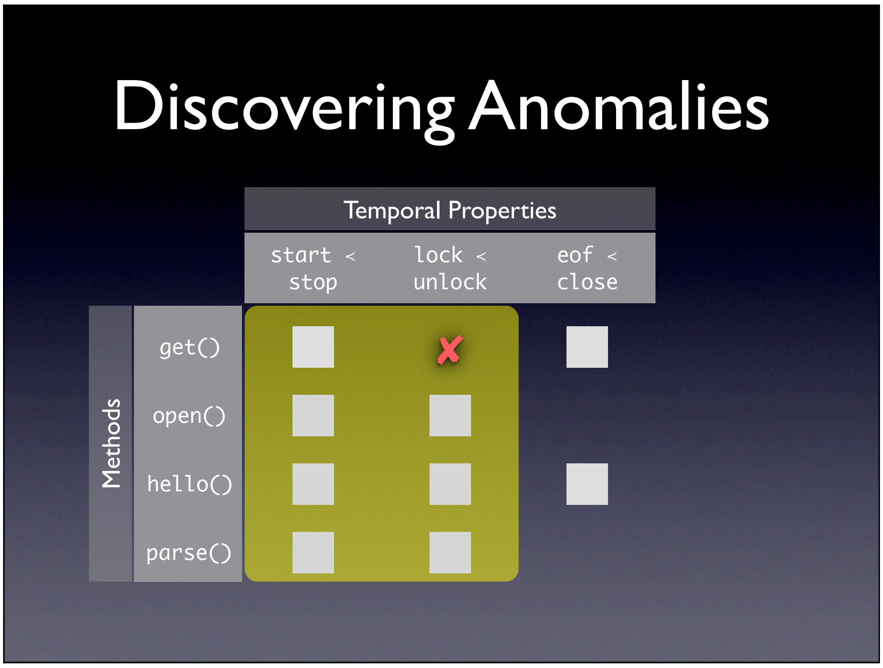



Usage Models
Random r = new Random ();
int n = r.nextInt ();
s.push (rand (r));

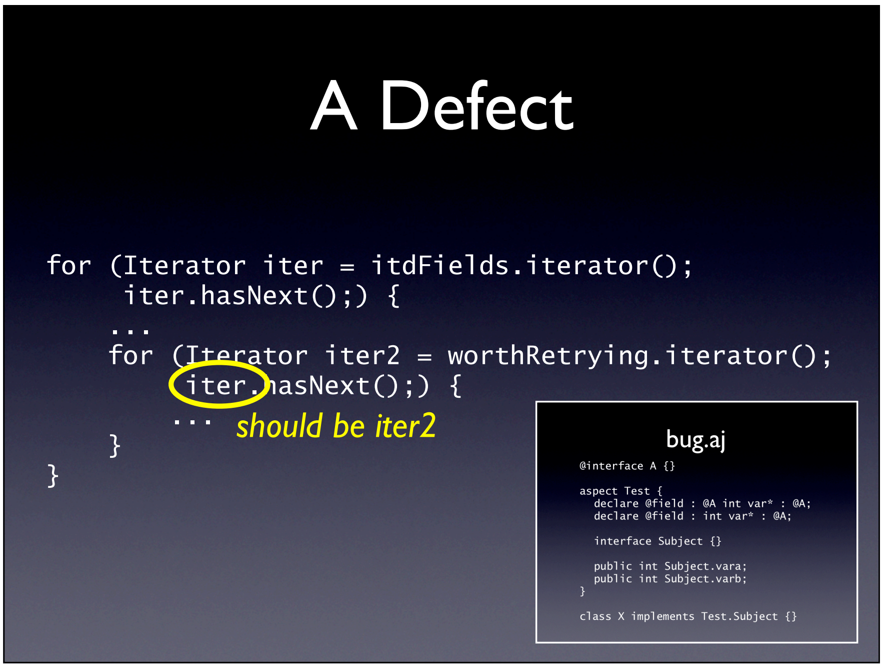

Usage Models
r.<init> ()
r.nextInt ()
Utils.rand (r)


OP-Miner
Program
Usage Models
iter.hasNext ()
iter.next ()
Temporal Properties
hasNext < next
hasNext < hasNext
next < hasNext
next < next
Patterns
hasNext < next
hasNext < hasNext
Anomalies
✓ hasNext < next
hasNext < hasNext
✗ hasNext < next
~~hasNext < hasNext~~





This would be a pattern, if it were not for the missing element



Another Defect

```
public void visitNEWARRAY (NEWARRAY o) {
    byte t = o.getTypecode ();
    if (!((t == Constants.T_BOOLEAN) ||
            (t == Constants.T_CHAR) ||
             ...
           (t == Constants.T_LONG))) {
        constraintViolated (o, "(...)'+t+'(...)");
    }
}
```

should be double quotes

On encountering a wrong typecode, \<visitNEWARRAY()> should report the typecode to the user. However, it fails to do so, as it uses \<'+t+'> instead of \<"+t+"> when

A False Positive

```
Name internalNewName (String[] identifiers)
  ...
  for (int i = 1; i < count; i++) {
    SimpleName name = new SimpleName(this);
    name.internalSetIdentifier(identifiers[i]);
    ...
  }
  ...
}
```

should stay as is

In 48 cases: argument comes from String() constructor;
only in 3 cases: from array

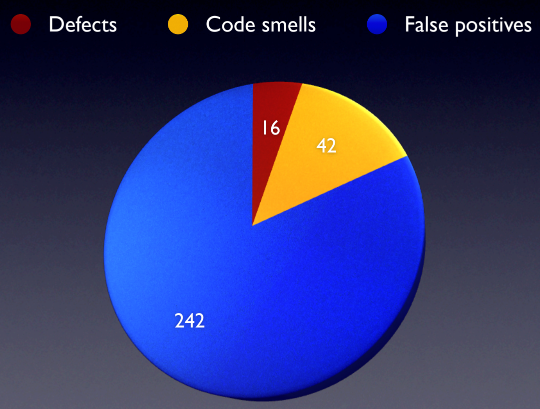
A Code Smell

```
public String getRetentionPolicy ()
{
    ...
    for (Iterator it = ...; it.hasNext();)
    {
        ... = it.next();
        ...
        return retentionPolicy;
    }
    ...
}
```

should be fixed

Hint → if fixed, would improve program
Code smell → does not result in errors, but may cause maintainability problems
Defects → reported & verified

AspectJ



1 out of 5 is a defect or code smell
2.5 minutes per violation – one new defect after 10 minutes
Defects → reported & confirmed

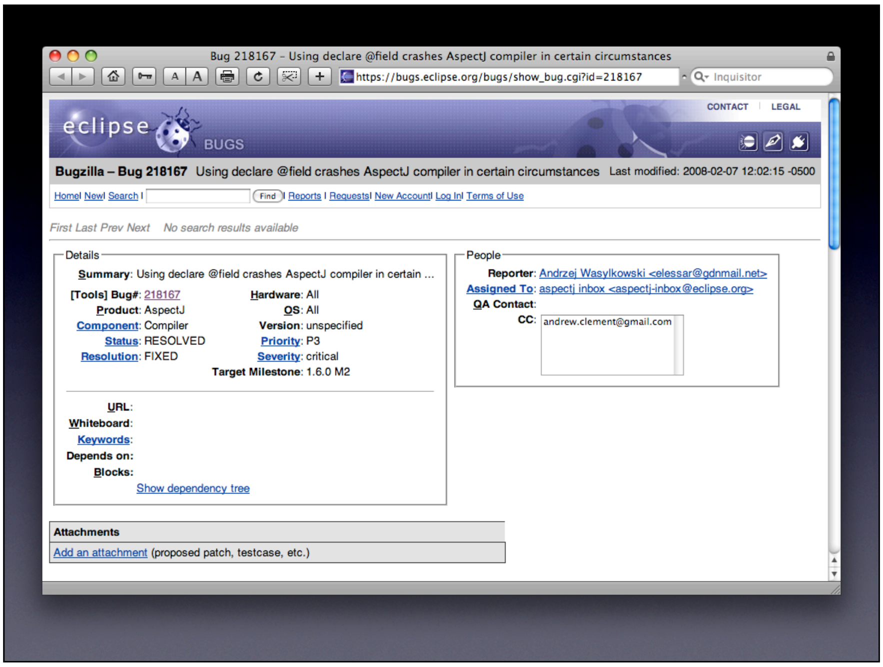
More Results

Program	# Violations Total	# Violations Investigated	# Defects	# Code smells	# False positives	Efficiency
ACT-RBOT 0.8.2	25	25	2	13	10	**60%**
APACHE TOMCAT 6.0.16	55	55	0	9	46	**16%**
ARGOUML 0.24	305	28	0	12	16	**43%**
ASPECTJ 1.5.3	300	300	16	42	242	**19%**
AZUREUS 2.5.0.0	315	85	1	26	58	**32%**
COLUMBA 1.2	57	57	4	15	38	**33%**
JEDIT 4.2	11	11	0	4	7	**36%**
	1,068	**562**	**23**	**121**	**417**	**26%**

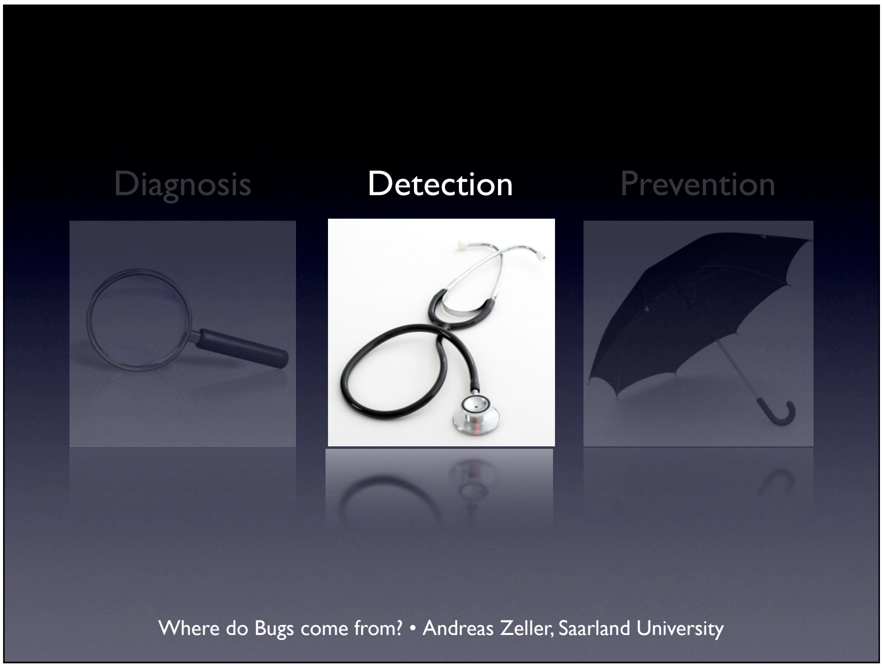
All in all, 1 out of 4 violations is a problem
Lots of subtle defects in production code
Unclear whether these would be found by other means

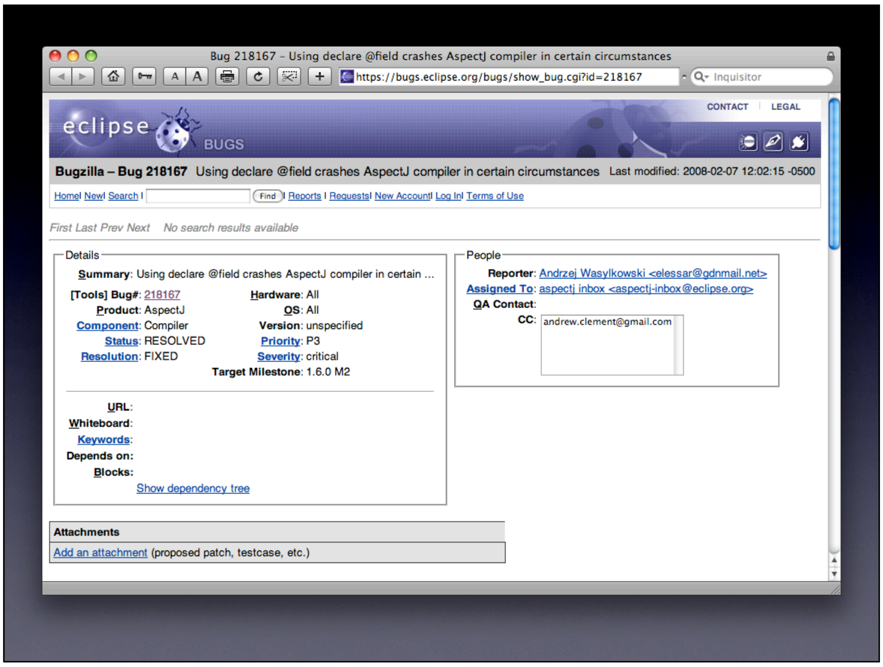
OP-Miner

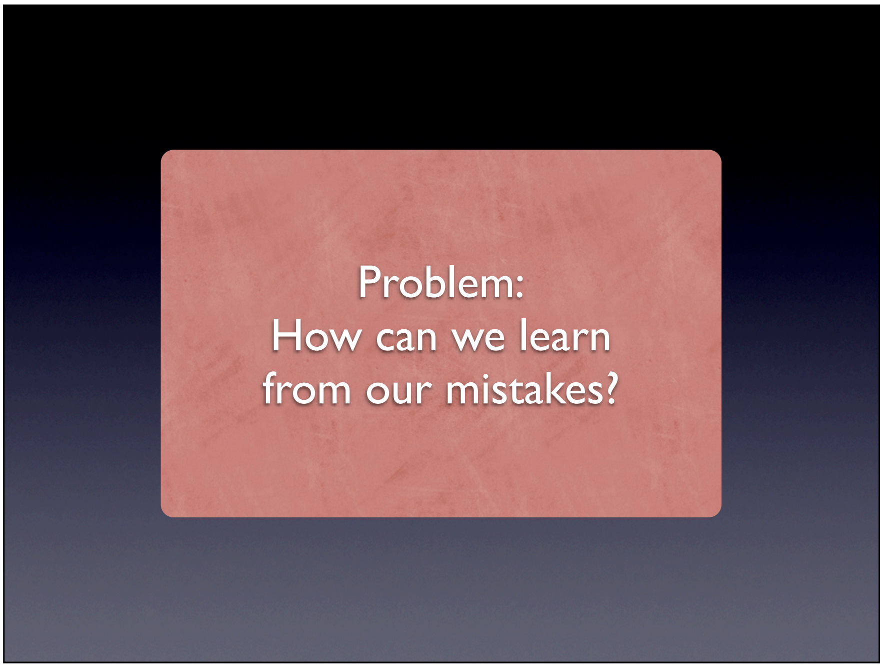
- ★ OP-Miner learns *operational preconditions* – i.e., how to typically construct arguments
- ★ learns from normal usage – for specific projects or across projects
- ★ Fully automatic
- ★ Found dozens of verified defects



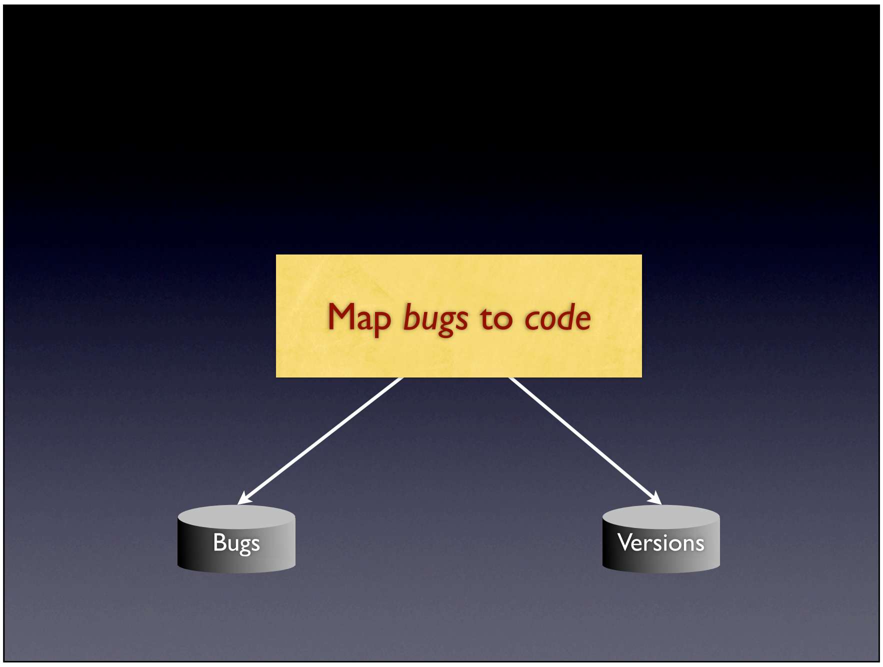
ProgramBugs sind eingereicht und bestätigt



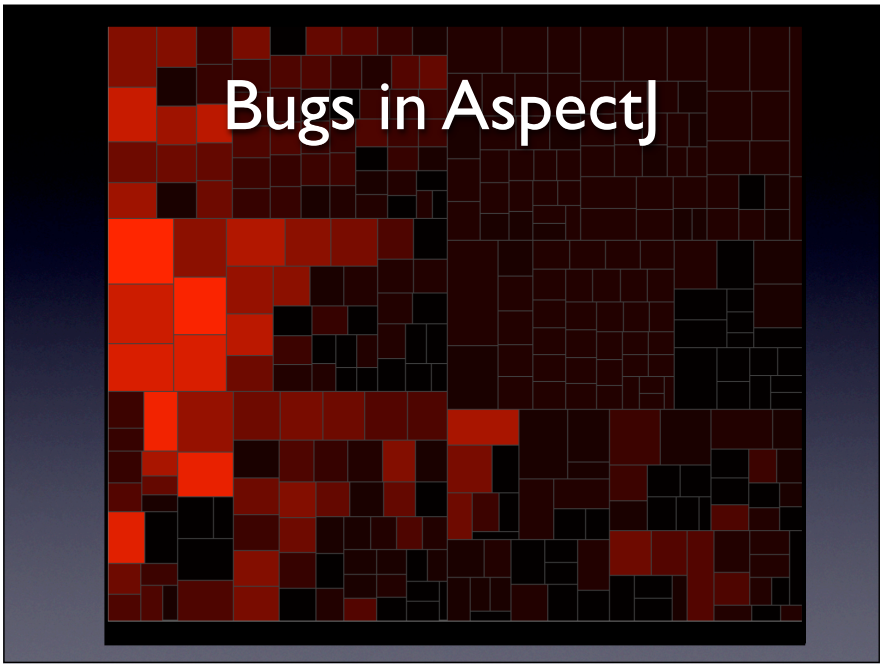




Such software archives are being used in practice all the time. If you file a bug, for instance, the report is stored in a bug database, and the resulting fix is stored in the version archive.



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.



Is it the Developers?

Does experience matter?

Bug density correlates with experience!

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!

Ah! Language features?

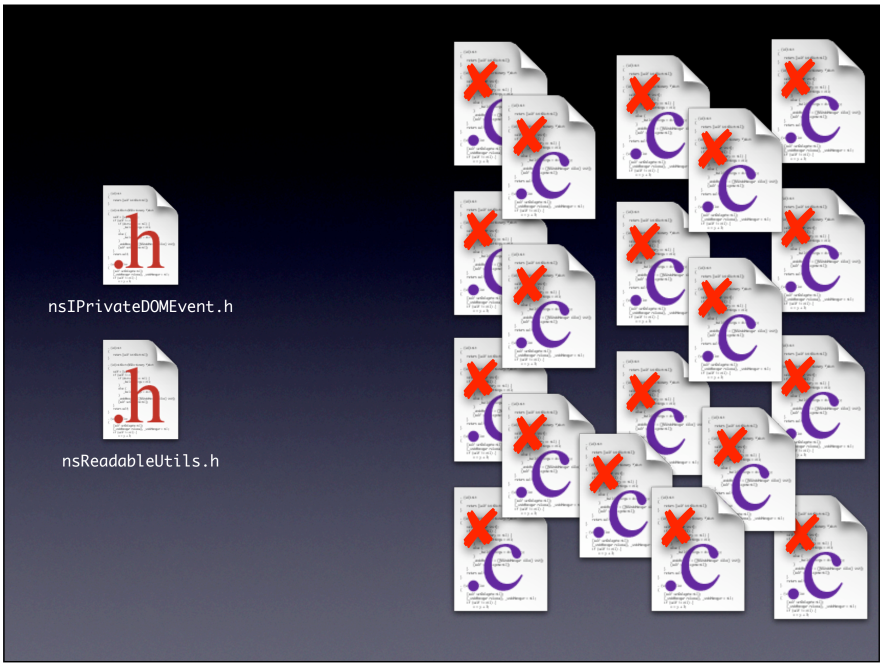
Are gotos harmful?

No correlation!

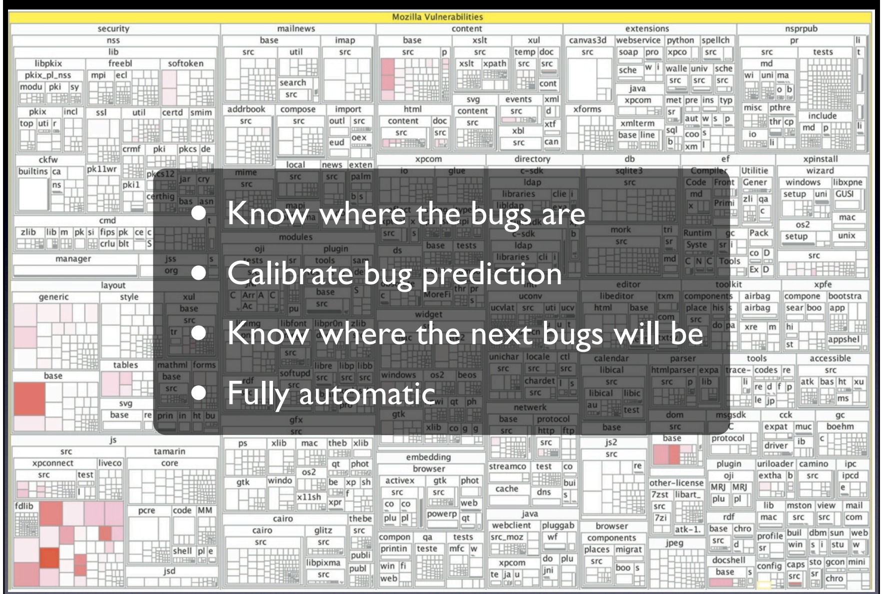
Ok. Problem Domain?

Which tokens do matter?

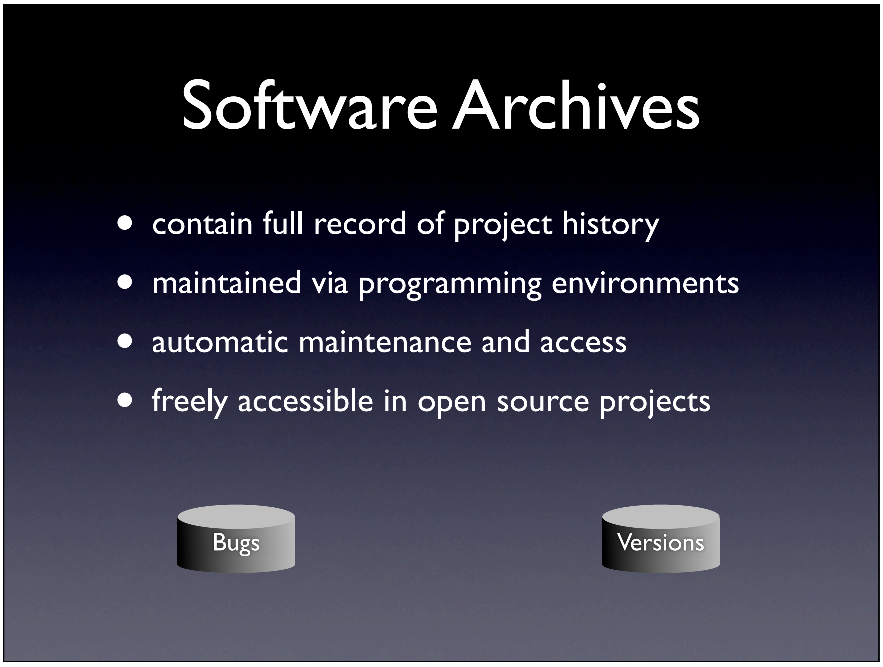
import • extends • implements



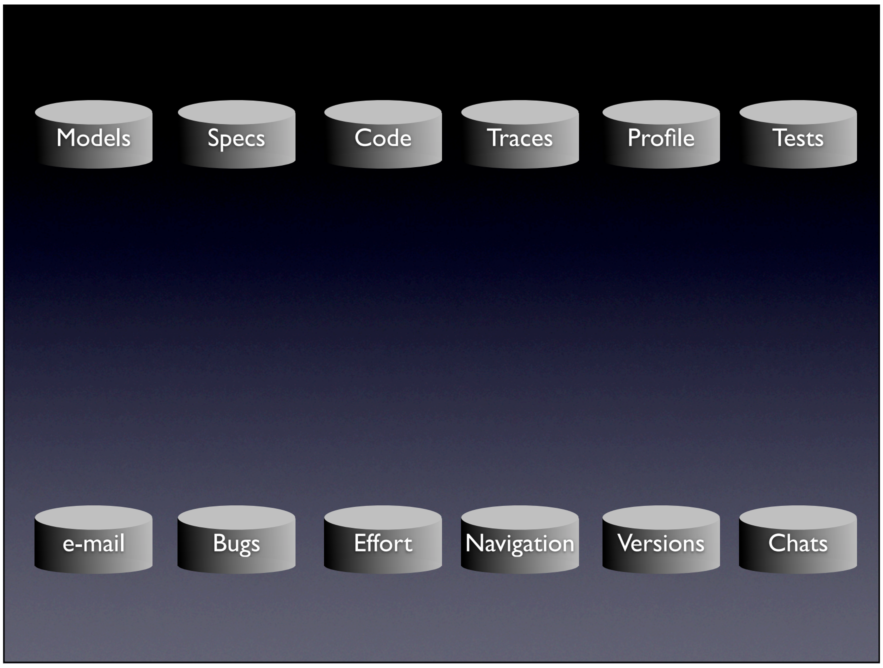
Prediction	Component	Fact
1	nsDOMClassInfo	3
2	SGridRowLayout	95
3	xpcprivate	6
4	jsxml	2
5	nsGenericHTMLElement	8
6	jsgc	3
7	nsISEnvironment	12
8	jsfun	1
9	nsHTMLLabelElement	18
10	nsHttpTransaction	35



- Know where the bugs are
- Calibrate bug prediction
- Know where the next bugs will be
- Fully automatic

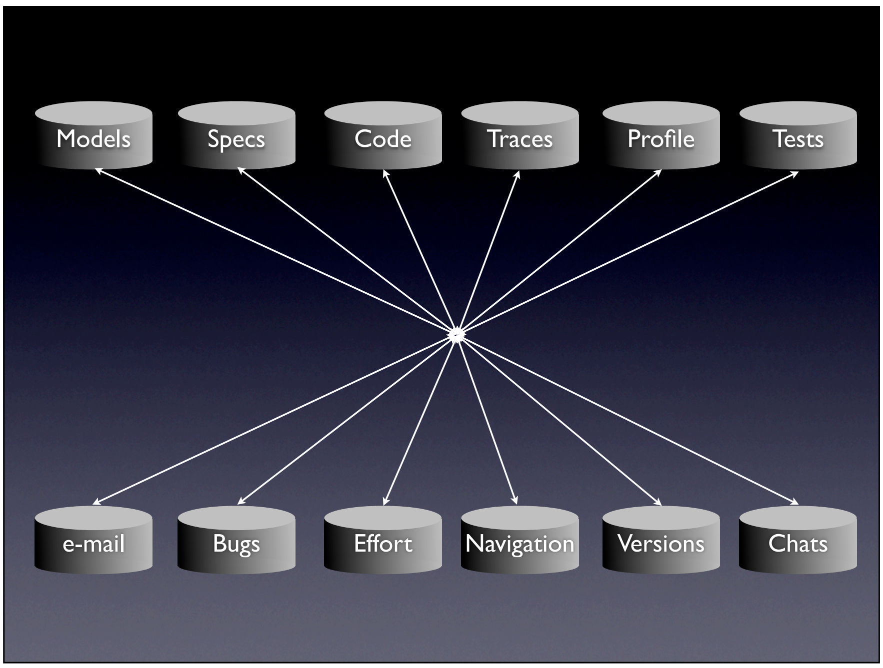


This was just a simple example. So, the most important aspect that software archives give you is automation. They are maintained automatically ("The data comes to you"), and they can be evaluated automatically ("Instantaneous results"). For researchers, there are plenty open source archives available,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



Tools can only work together if they draw on different artefacts

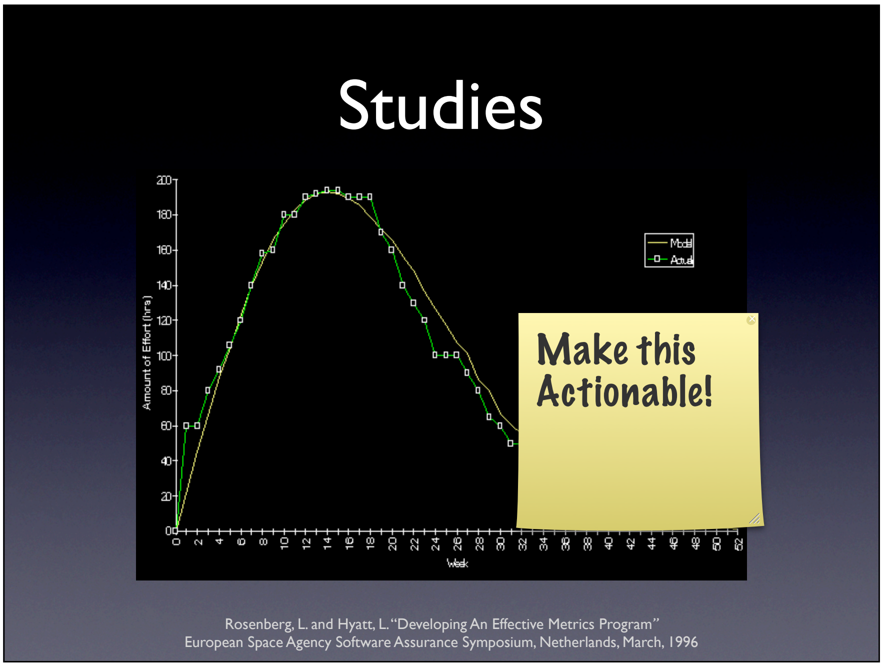
What are we working on in SE – we are constantly producing and analyzing artefacts: code, specs, etc.



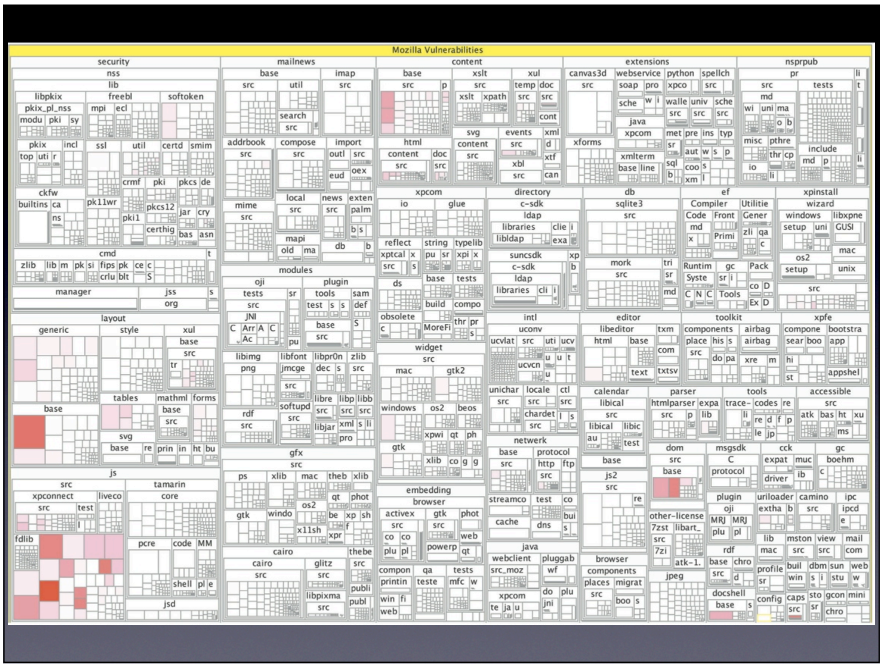
Combining these sources will allow us to get this "waterfall effect" – that is, being submerged by data; having more data than we could possibly digest.

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.



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




Tom Mens
Serge Demeyer (Eds.)
Software Evolution
Springer
Windows Server System
DirectX 9
Windows NetMeeting
eclipse
aspectj


Diagnosis
Detection
Prevention
Where do Bugs come from? • Andreas Zeller, Saarland University


Diagnosis
Detection
Prevention
Isolating Relevant Calls
OP-Miner
Why Programs Fail
Beautiful Code
Software Evolution
Where do Bugs come from? • Andreas Zeller, Saarland University