Learning from Mistakes

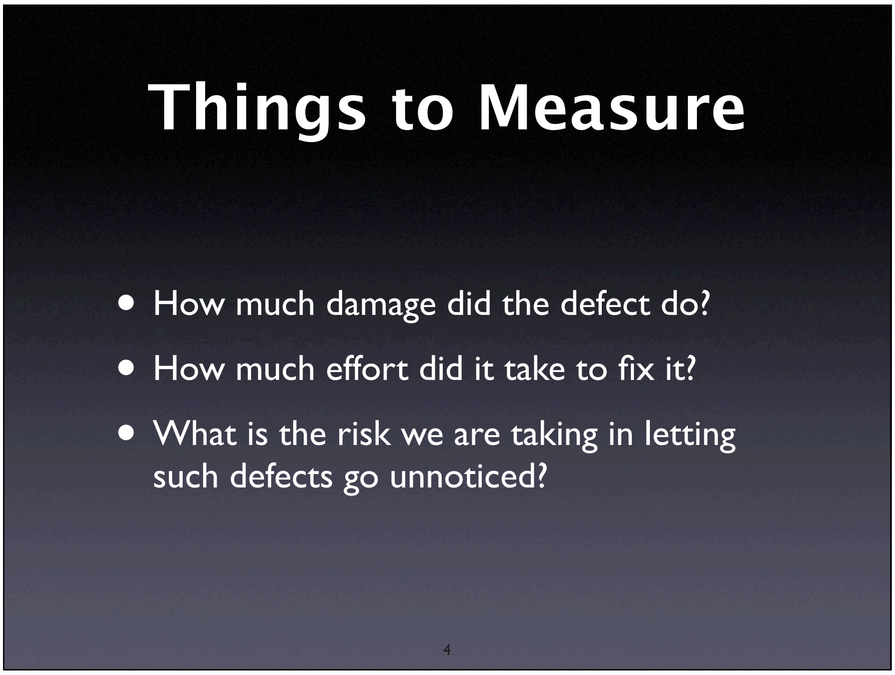
Andreas Zeller

Fixing the Process

- Any defect escaping into the wild should have been caught by local *quality assurance*
- Besides fixing the defect, we also must fix quality assurance!

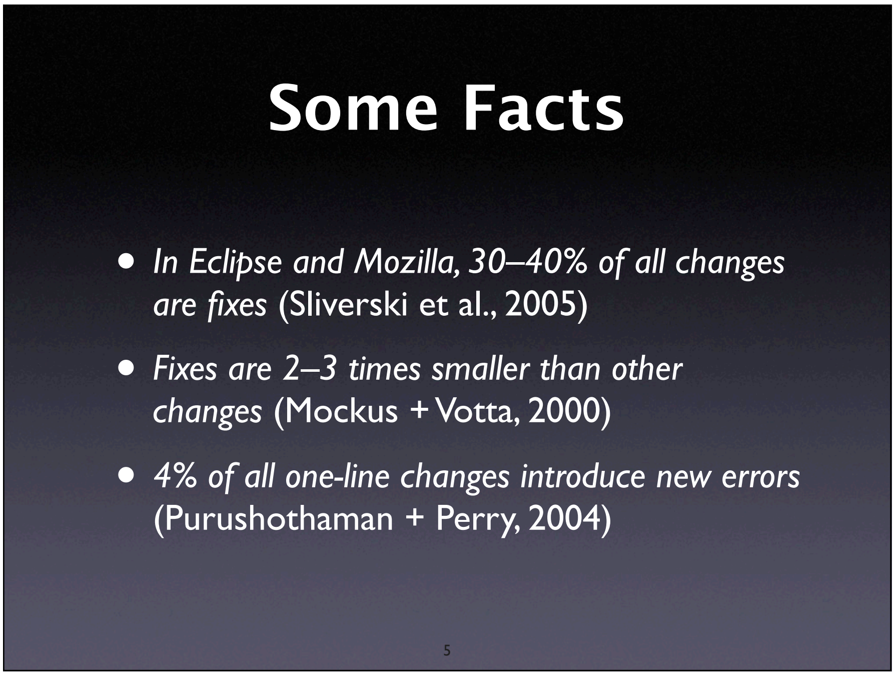
Things to do

- Improve your test suite
- Set up assertions
- Improve training
- Improve the software process
- Improve the analysis tools



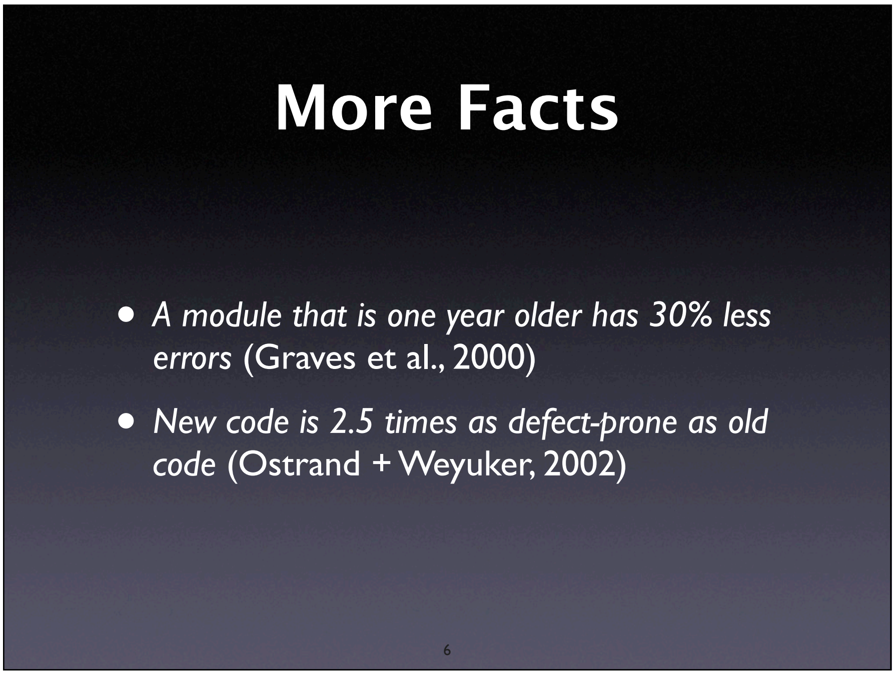
Things to Measure

- How much damage did the defect do?
- How much effort did it take to fix it?
- What is the risk we are taking in letting such defects go unnoticed?



Some Facts

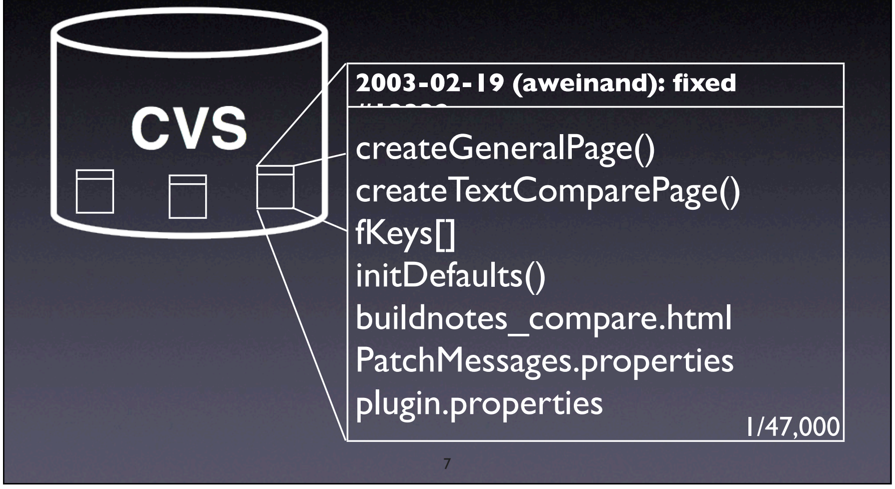
- *In Eclipse and Mozilla, 30–40% of all changes are fixes* (Sliverski et al., 2005)
- *Fixes are 2–3 times smaller than other changes* (Mockus + Votta, 2000)
- *4% of all one-line changes introduce new errors* (Purushothaman + Perry, 2004)



More Facts

- *A module that is one year older has 30% less errors* (Graves et al., 2000)
- *New code is 2.5 times as defect-prone as old code* (Ostrand + Weyuker, 2002)

Learning from History



2003-02-19 (aweinand): fixed #

createGeneralPage()
createTextComparePage()
fKeys[]
initDefaults()
buildnotes_compare.html
PatchMessages.properties
plugin.properties

1/47,000

The Risk of Change

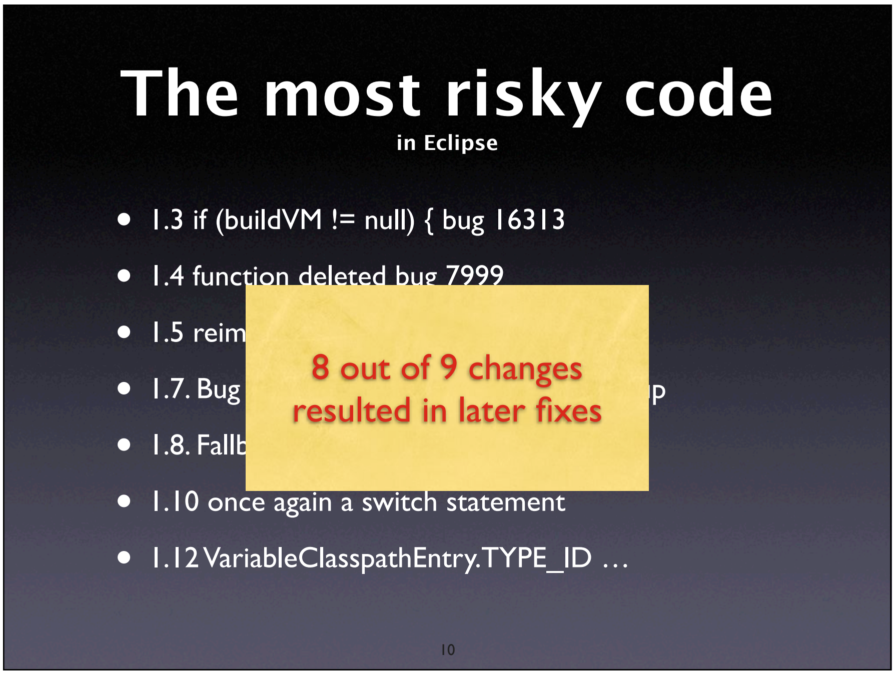
Change
Fix
Preferences
Workbench
Appearance
Compare/Patch
Editors
File Associations
Fonts
Keys
Label Decorations
Show additional compare information in the status line
Ignore white space
Open structure compare automatically

- Some locations in a program are *risky:* many changes result in a fix

The most risky code

in Eclipse

```
public IRuntimeClasspathEntry[] resolveClasspath(IRuntimeClasspathEntry[] entries,
                                                 ILaunchConfiguration configuration)
throws CoreException {
  List all = new ArrayList(entries.length);
  for (int i = 0; i < entries.length; i++) {
    switch (entries[i].getType()) {
      case IRuntimeClasspathEntry.PROJECT:
        all.add(entries[i]);
        break;
      case IRuntimeClasspathEntry.OTHER:
        IRuntimeClasspathEntry2 entry = (IRuntimeClasspathEntry2)entries[i];
        if (entry.getTypeId().equals(DefaultProjectClasspathEntry.TYPE ID)) {
          IRuntimeClasspathEntry[] children = entry.getRuntimeClasspathEntries(configuration);
          IRuntimeClasspathEntry[] res =
            JavaRuntime.resolveSourceLookupPath(children, configuration);
          for (int j = 0; j < res.length; j++) {
            all.add(res[j]);
          }
        }
        break;
      default:
        IRuntimeClasspathEntry[] resolved =
          JavaRuntime.resolveRuntimeClasspathEntry(entries[i], configuration);
        for (int j = 0; j < resolved.length; j++) {
          all.add(resolved[j]);
        }
        break;
    }
  }
  return (IRuntimeClasspathEntry[])all.toArray(new IRuntimeClasspathEntry[all.size()]);
}
```

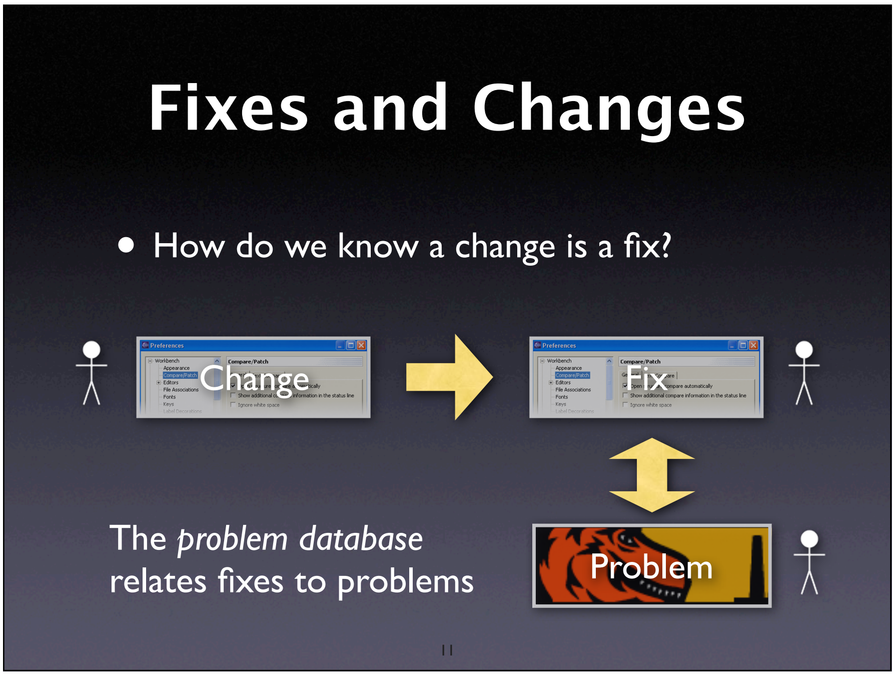


The most risky code

in Eclipse

- 1.3 if (buildVM != null) { bug 16313
- 1.4 function deleted bug 7999
- 1.5 reim
- 1.7. Bug p
- 1.8. Fallb
- 1.10 once again a switch statement
- 1.12 VariableClasspathEntry.TYPE_ID …

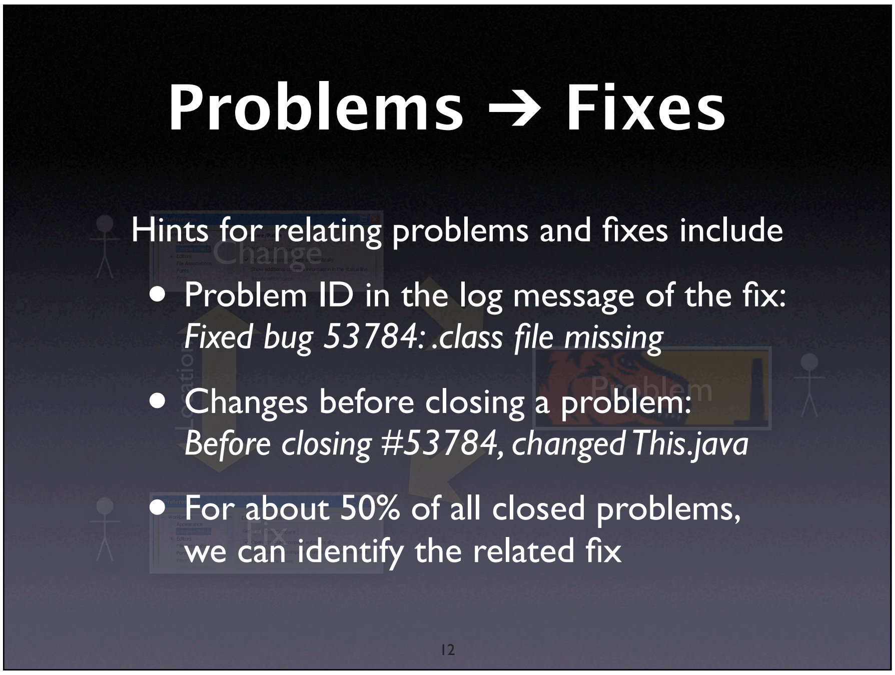
8 out of 9 changes resulted in later fixes



Fixes and Changes

- How do we know a change is a fix?

The *problem database* relates fixes to problems



Problems → Fixes

Hints for relating problems and fixes include

- Problem ID in the log message of the fix: *Fixed bug 53784: .class file missing*
- Changes before closing a problem: *Before closing #53784, changed This.java*
- For about 50% of all closed problems, we can identify the related fix

Fix-Inducing Changes

Change

Problem

- Can I predict the *risk of change?*
- Which are the *risky locations?*
- Do they have common *features?*

What makes changes risky?

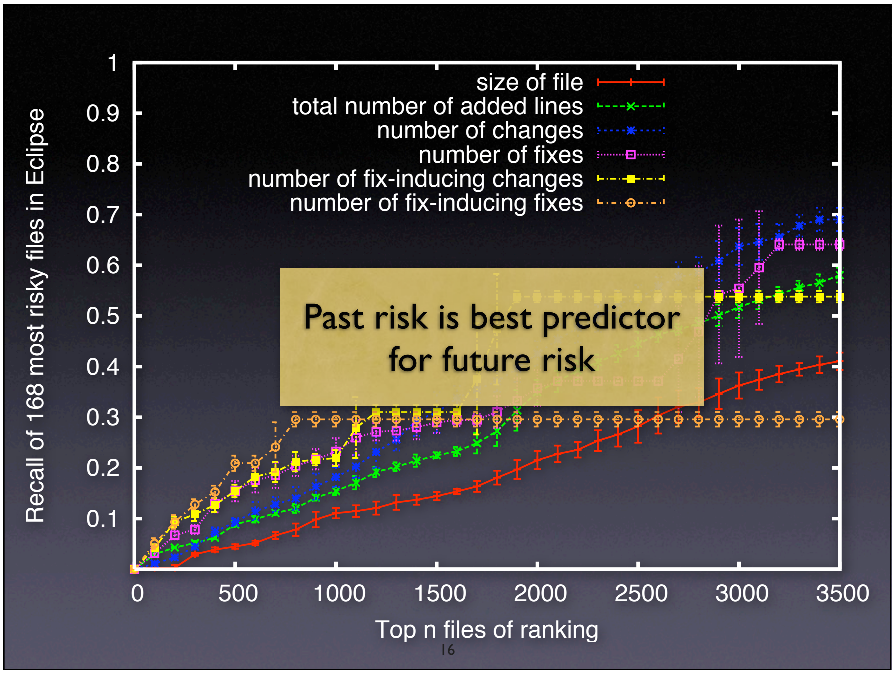
To determine whether changes induce risk, a number of *metrics* have been proposed:

size of file being changed	size of the change
number of changes so far	number of fixes so far

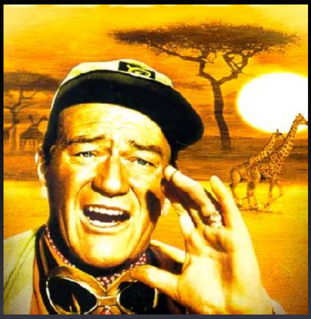
What makes changes risky?

Our claim: *past risk at the change location* is best predictor for future risk

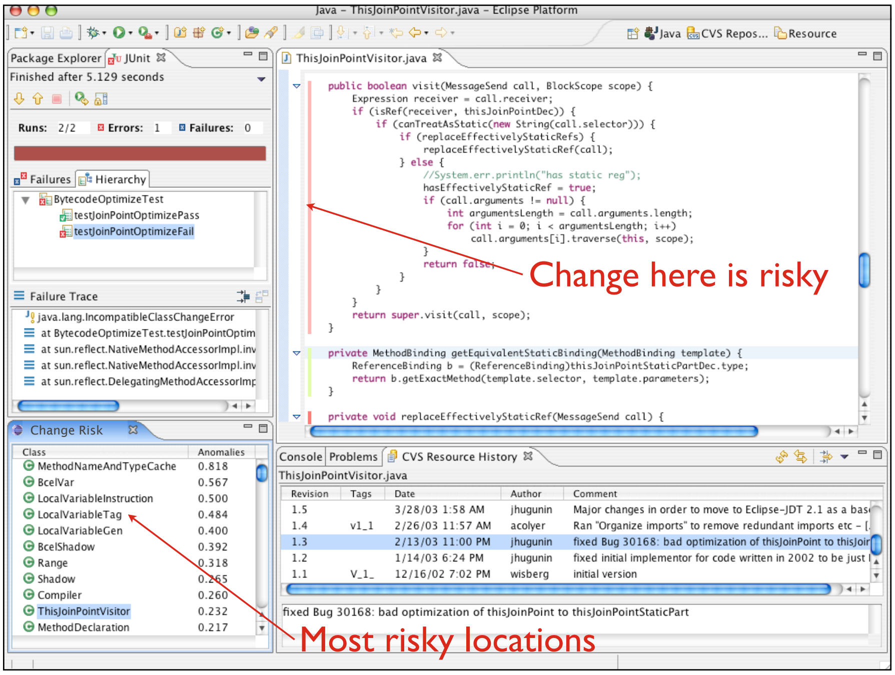
# of past *fix-inducing changes* at the change location	# of past *fix-inducing fixes* at the change location



Past risk is best predictor for future risk



Hatari



Finished after 5.129 seconds

Runs: 2/2 Errors: 1 Failures: 0

- BytecodeOptimizeTest
  - testJoinPointOptimizePass
  - testJoinPointOptimizeFail

```
java.lang.IncompatibleClassChangeError
at BytecodeOptimizeTest.testJoinPointOptim
at sun.reflect.NativeMethodAccessorImpl.inv
at sun.reflect.NativeMethodAccessorImpl.inv
at sun.reflect.DelegatingMethodAccessorImp
```

```
public boolean visit(MessageSend call, BlockScope scope) {
    Expression receiver = call.receiver;
    if (isRef(receiver, thisJoinPointDec)) {
        if (canTreatAsStatic(new String(call.selector))) {
            if (replaceEffectivelyStaticRefs) {
                replaceEffectivelyStaticRef(call);
            } else {
                //System.err.println("has static reg");
                hasEffectivelyStaticRef = true;
                if (call.arguments != null) {
                    int argumentsLength = call.arguments.length;
                    for (int i = 0; i < argumentsLength; i++)
                        call.arguments[i].traverse(this, scope);
                }
                return false;
            }
        }
    }
    return super.visit(call, scope);
}

private MethodBinding getEquivalentStaticBinding(MethodBinding template) {
    ReferenceBinding b = (ReferenceBinding)thisJoinPointStaticPartDec.type;
    return b.getExactMethod(template.selector, template.parameters);
}

private void replaceEffectivelyStaticRef(MessageSend call) {
```

Change here is risky

Class	Anomalies
MethodNameAndTypeCache	0.818
BcelVar	0.567
LocalVariableInstruction	0.500
LocalVariableTag	0.484
LocalVariableGen	0.400
BcelShadow	0.392
Range	0.318
Shadow	0.265
Compiler	0.260
ThisJoinPointVisitor	0.232
MethodDeclaration	0.217

ThisJoinPointVisitor.java

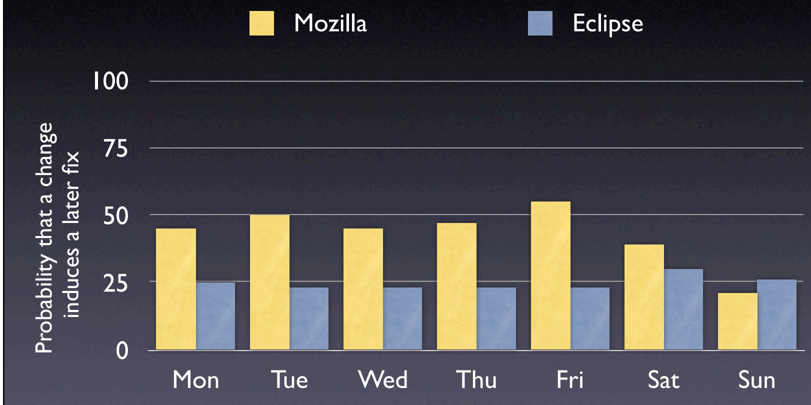
Revision	Tags	Date	Author	Comment
1.5		3/28/03 1:58 AM	jhugunin	Major changes in order to move to Eclipse-JDT 2.1 as a bas
1.4	v1_1	2/26/03 11:57 AM	acolyer	Ran "Organize imports" to remove redundant imports etc – [.
1.3		2/13/03 11:00 PM	jhugunin	fixed Bug 30168: bad optimization of thisJoinPoint to thisJoir
1.2		1/14/03 6:24 PM	jhugunin	fixed initial implementor for code written in 2002 to be just
1.1	V_1_	12/16/02 7:02 PM	wisberg	initial version

fixed Bug 30168: bad optimization of thisJoinPoint to thisJoinPointStaticPart

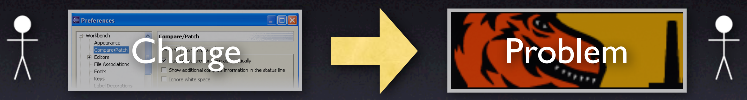
Most risky locations

CAUTION
HOT FUNCTION DON'T TOUCH!
DANGER
ONLY AUTHORIZED PROGRAMMERS MAY CHANGE FUNCTION
NOTICE
DON'T EVEN THINK OF CHANGING
DANGER
DO NOT PROGRAM ON FRIDAYS!

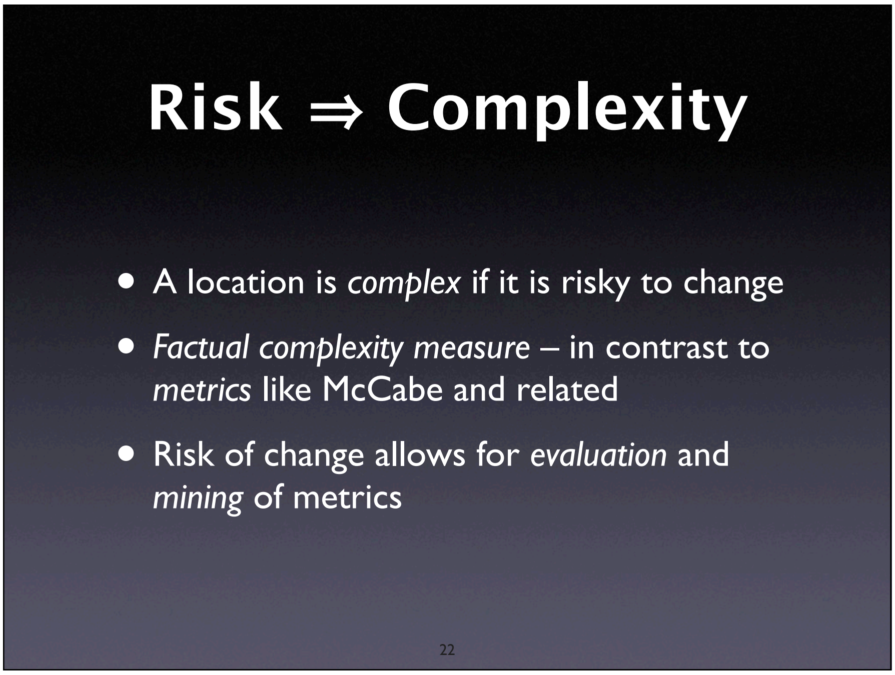
Risk along the Week



What makes changes risky?

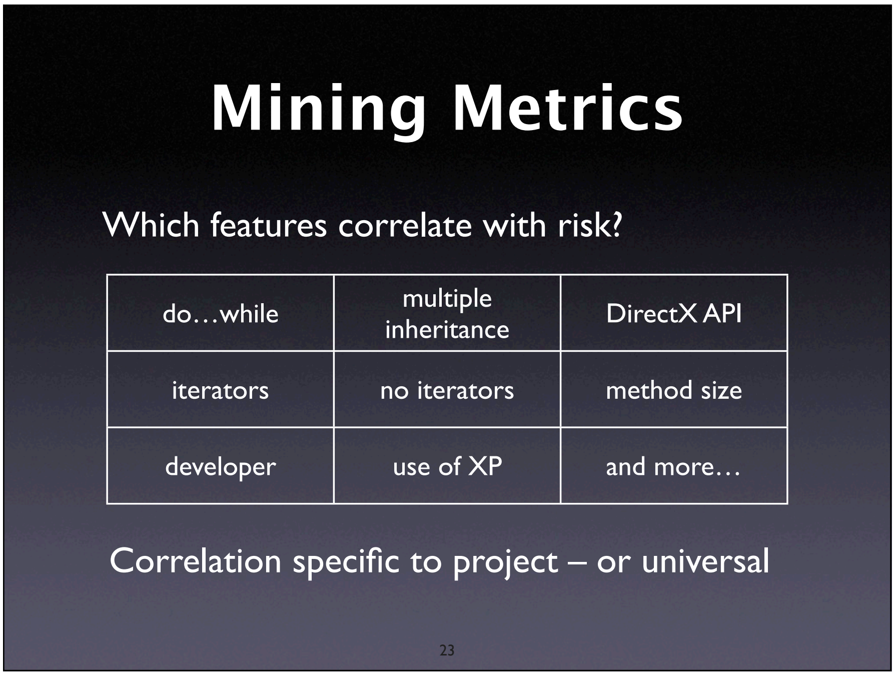


- Past risk at the location
- The day of the week
- Properties of the code?



Risk ⇒ Complexity

- A location is *complex* if it is risky to change
- *Factual complexity measure* – in contrast to *metrics* like McCabe and related
- Risk of change allows for *evaluation* and *mining* of metrics

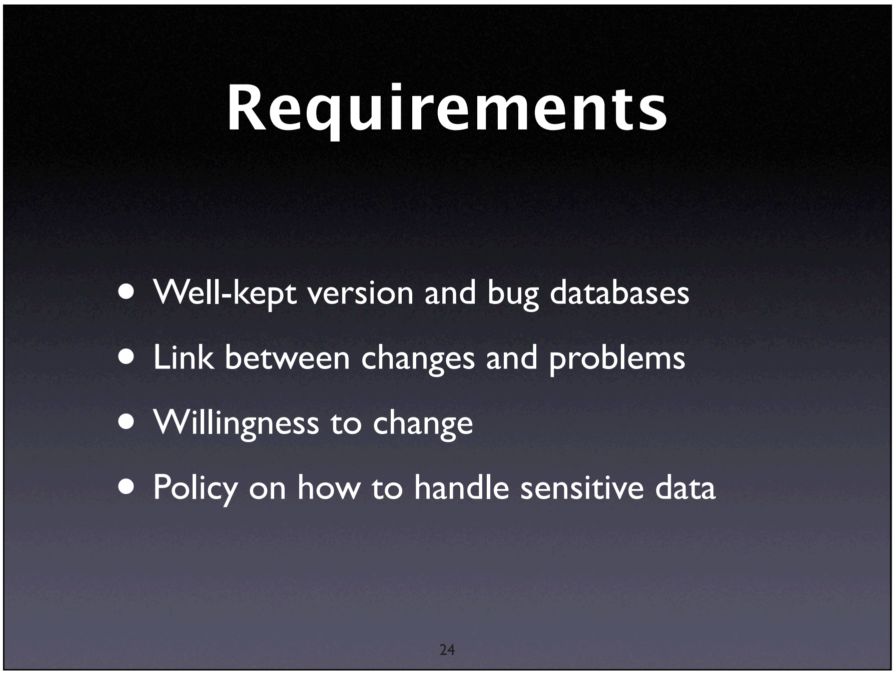


Mining Metrics

Which features correlate with risk?

do…while	multiple inheritance	DirectX API
iterators	no iterators	method size
developer	use of XP	and more…

Correlation specific to project – or universal



Requirements

- Well-kept version and bug databases
- Link between changes and problems
- Willingness to change
- Policy on how to handle sensitive data

Space Shuttle Software

Problem Tracking

- When was the error discovered? How? Who? What flight?
- How was the error introduced? Why wasn't it caught?
- How was the error corrected? Are there similar errors?
- What can we learn from previous errors?

The Process

- Software error = *an error in the process*
- Planning the software carefully in advance
- Reducing risk at all stages
- Keeping record of all activities
- "Not even rocket science" – just standard practice in engineering