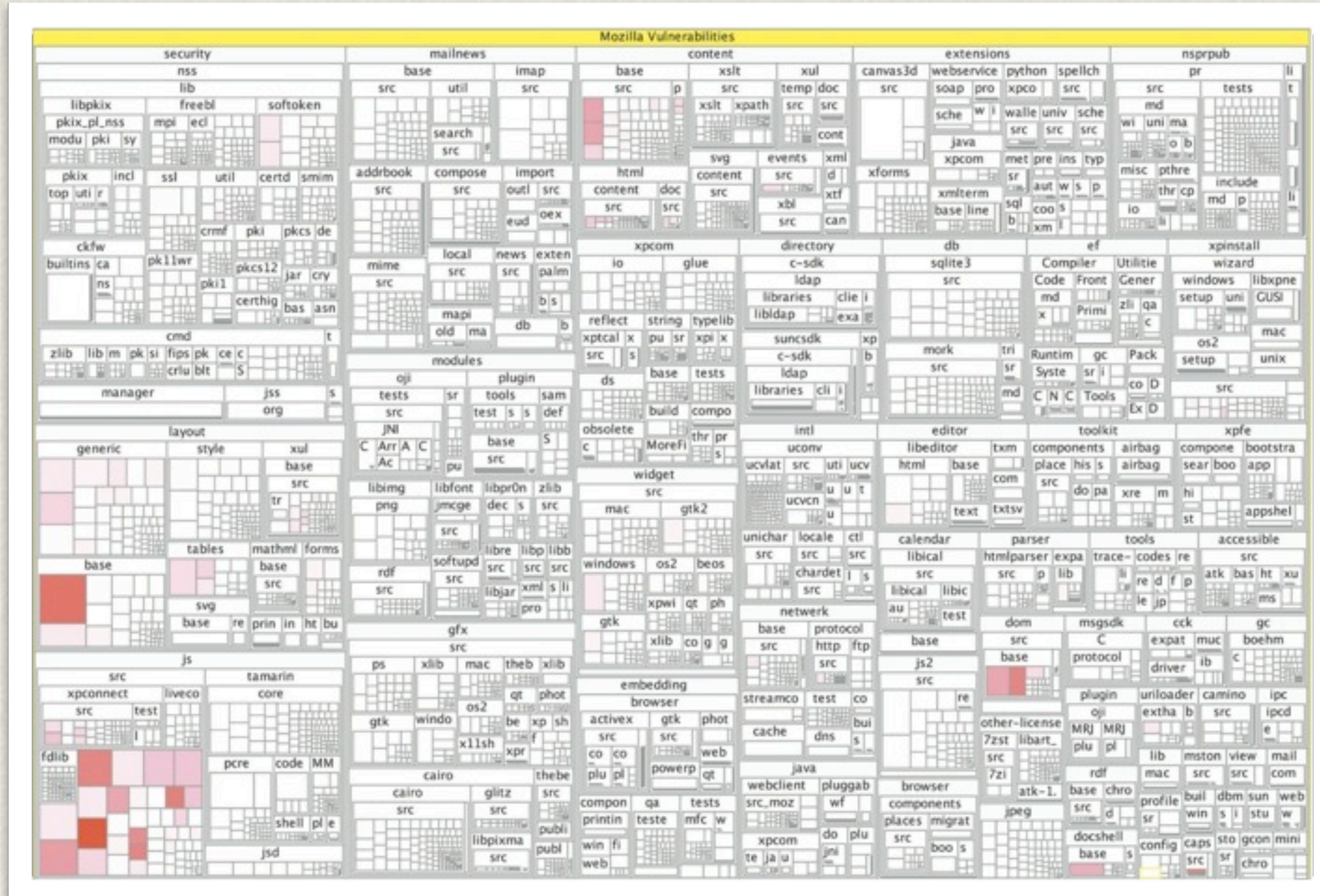
Failure is a four-letter word

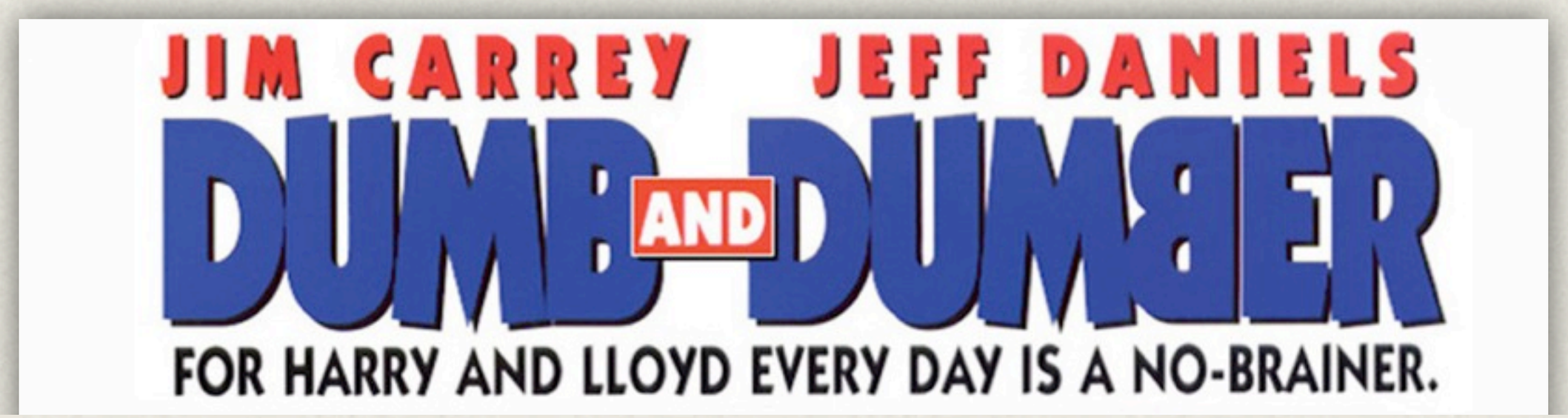
Andreas Zeller • Thomas Zimmermann • Christian Bird

PROMISE 2011, Banff, Canada

Software failures



Defect distributions

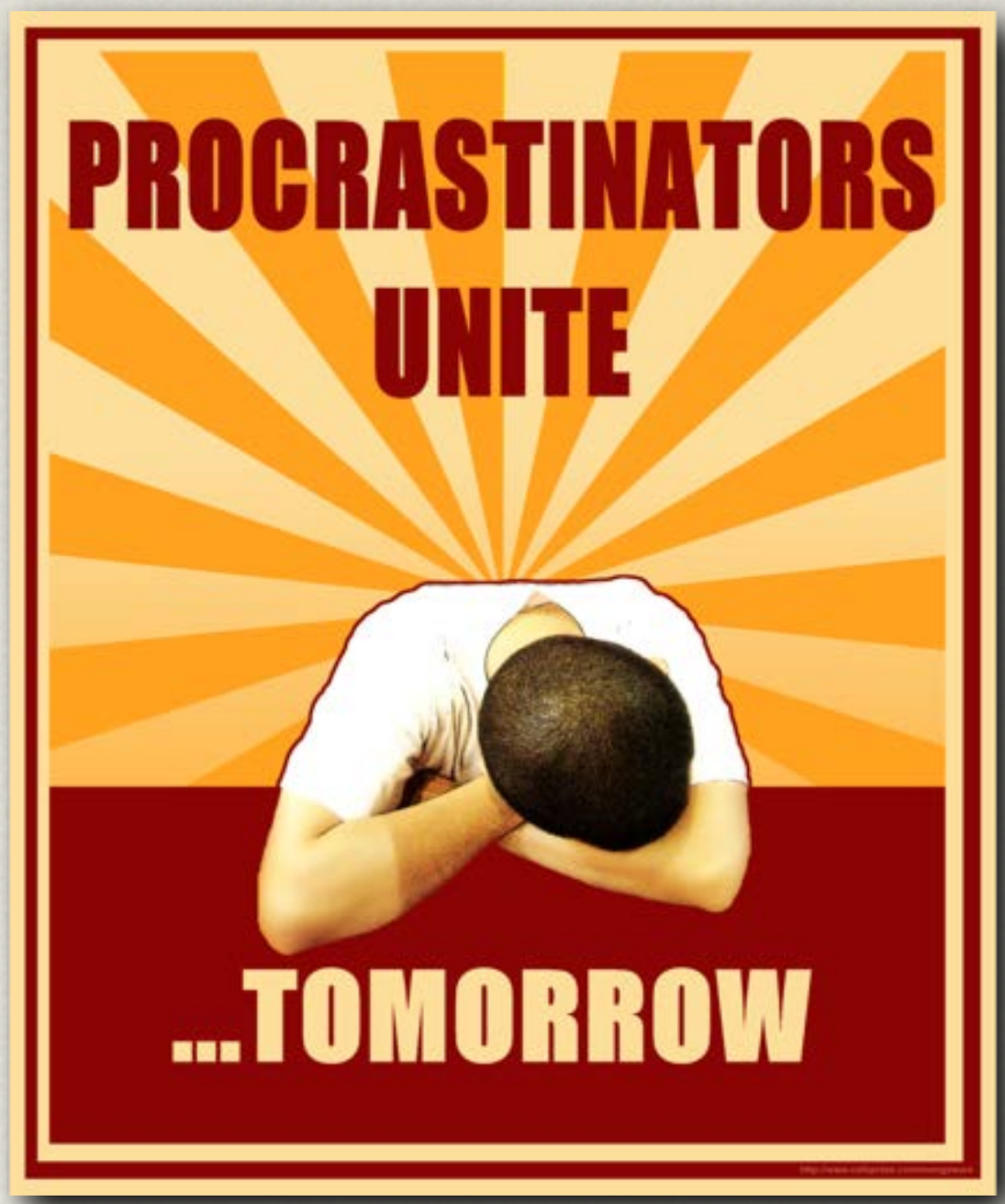


Failure causes

Failure causes

Failure causes

Failure causes



Cost of consequence

Back to basics

Back to basics

Basic actions

```
public class ImageViewerPlugin extends AbstractUIPlugin {

   //The shared instance.
   private static ImageViewerPlugin plugin;

   /**
    * The constructor.
    */
   public ImageViewerPlugin() {
      plugin = this;
   }

   /**
    * This method is called upon plug-in activation
    */
  public void start(BundleContext context) throws Exception {
      super.start(context);
  }
```

Basic actions

```
public class ImageViewerPlugin extends AbstractUIPlugin {

  //The shared instance.
  private static ImageViewerPlugin plugin;

  /**
   * The constructor.
   */
  public ImageViewerPlugin() {
     plugin = this;
  }

  /**
   * This method is called upon plug-in activation
   */
 public void start(BundleContext context) throws Exception {
     super.start(context);
 }
```

No abstraction

```
//////
********
aaaaaaaaaa
cccccccccc
d
eeeeeeeeeeeeeeeee
ggggggg
hh
iiiiiiiiiiiii
lllllllllllll
mmmm
nnnnnnnnnnnnnnn
ooooooooo
pppppppppp
rrrrrrrrrr
ssssssssssss
ttttttttttttttttttttt
uuuuuuuuuuuu
v
wwww
A
B
C
E
IIII
PPPP
TTT
U
VVV
{{{
}}}
```

No abstraction

Hypotheses

1. We can predict defects from programmer actions.

Hypotheses

1. We can predict defects from programmer actions.
2. We can isolate defect-prone programmer actions.

Hypotheses

1. We can predict defects from programmer actions.
2. We can isolate defect-prone programmer actions.
3. We can prevent defects by restricting programmer actions.

Hypotheses

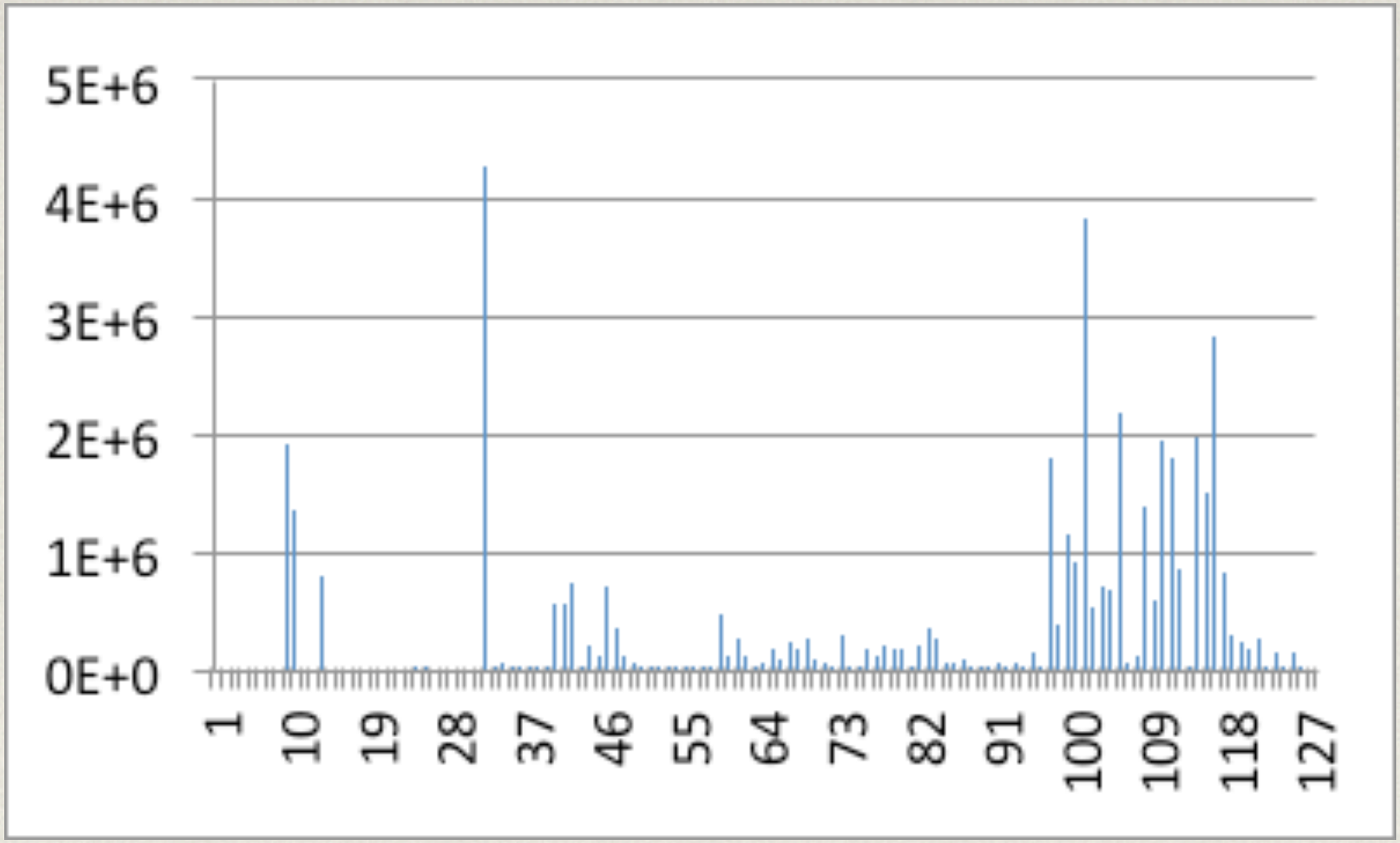
1. We can predict defects from programmer actions.
2. We can isolate defect-prone programmer actions.
3. We can prevent defects by restricting programmer actions.

Hypotheses

Release	Total chars	Total files	Files with defects
Eclipse 2.0	44,914,520	6,728	975 (14%)
Eclipse 2.1	56,068,650	7,887	854 (11%)
Eclipse 3.0	76,193,482	10,593	1,568 (15%)

Eclipse bug data

[PROMISE 2007]



Eclipse characters

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Precision

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Precision

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Precision

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.32	0.27	0.27	0.28
Eclipse 2.1	0.03	0.18	0.14	0.11
Eclipse 3.0	0.19	0.16	0.20	0.18
Average	0.18	0.20	0.20	0.19

Recall

1. We can predict defects from programmer actions.
2. We can isolate defect-prone programmer actions.
3. We can prevent defects by restricting programmer actions.

Hypotheses

1. We can predict defects from programmer actions.

2. We can isolate defect-prone programmer actions.
3. We can prevent defects by restricting programmer actions.

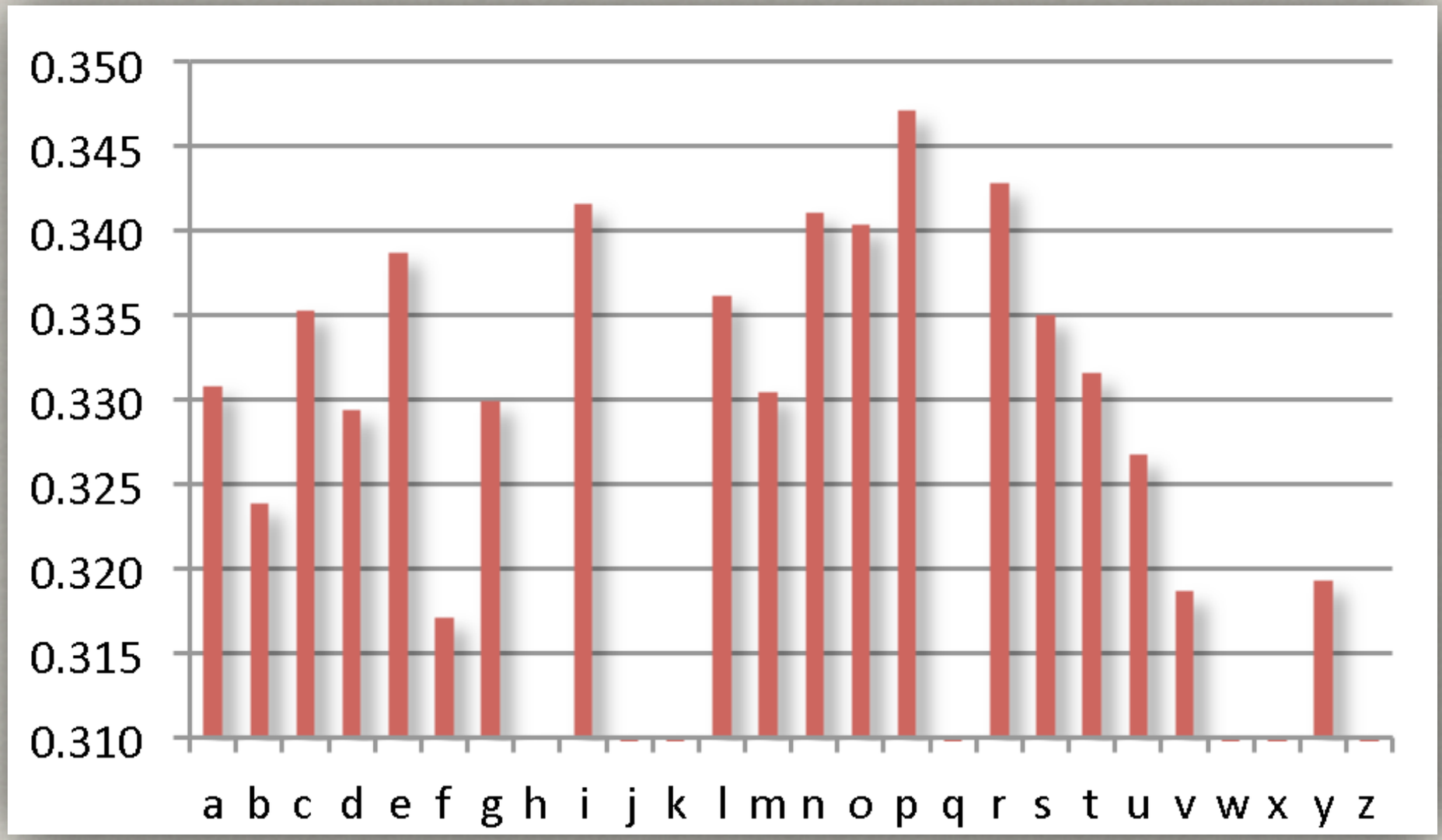
Hypotheses

1. We can predict defects from programmer actions. ✓

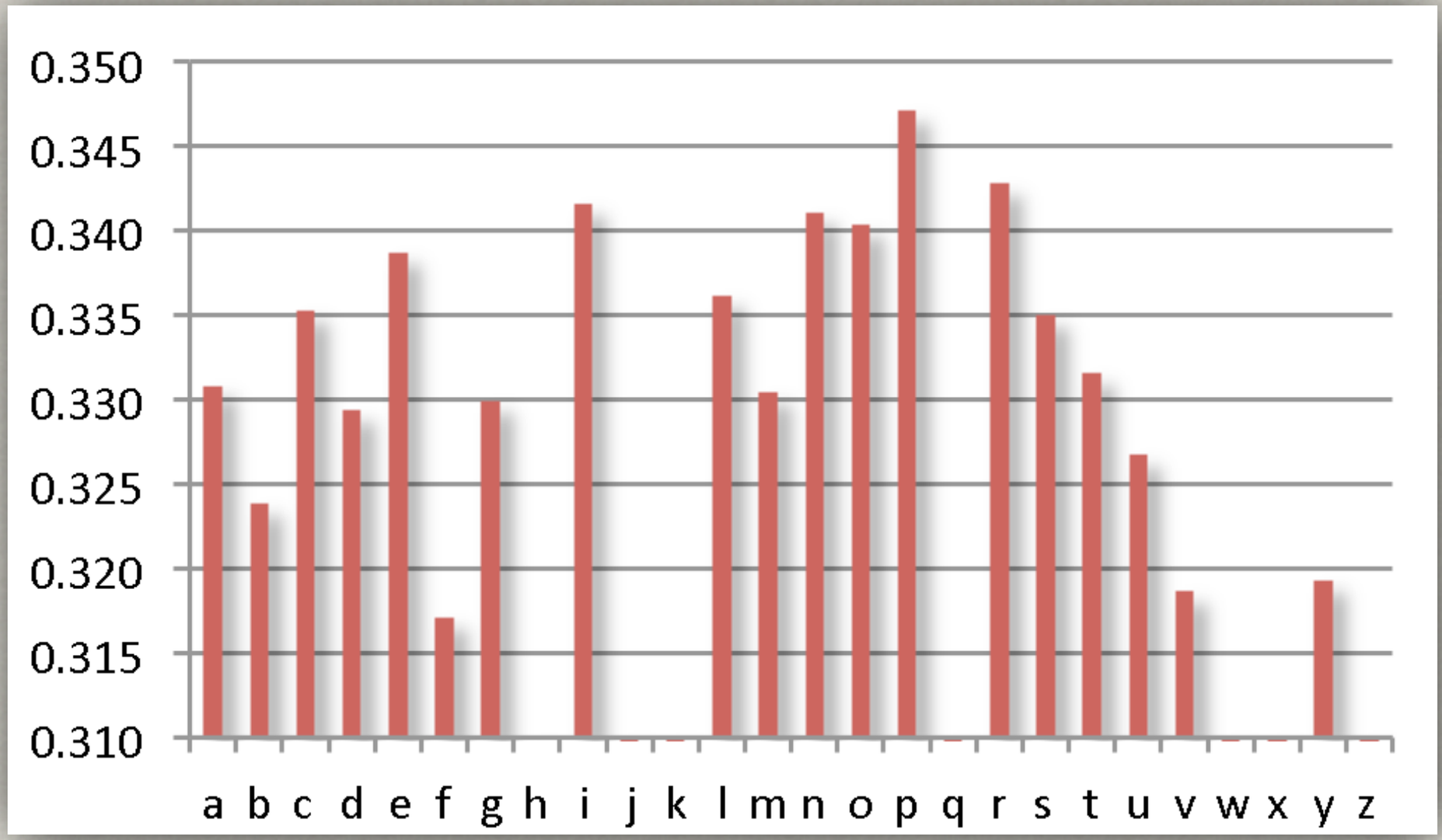
2. We can isolate defect-prone programmer actions.

3. We can prevent defects by restricting programmer actions.

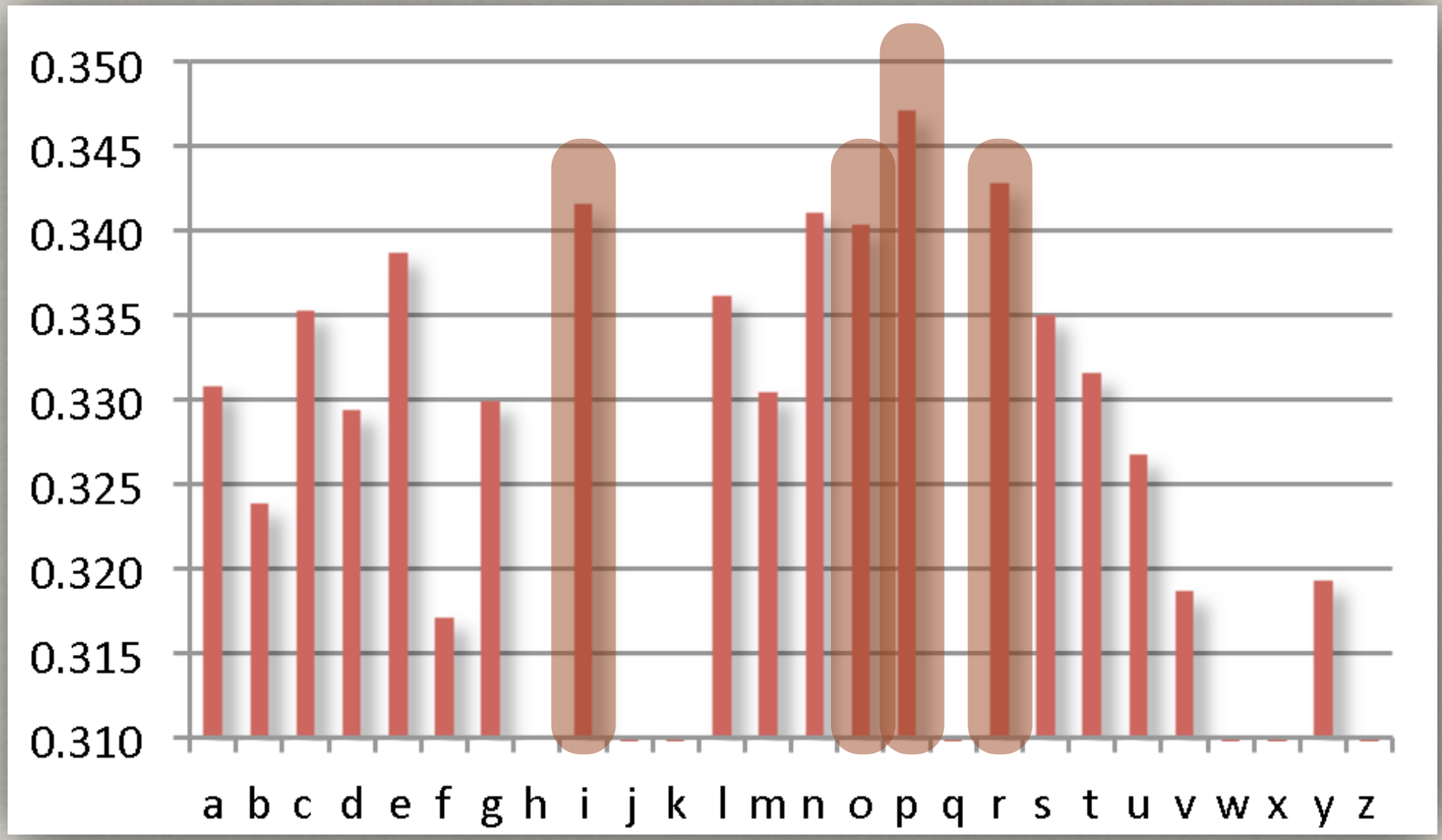
Hypotheses



Defect correlations



Defect correlations



Defect correlations

Defect correlations

TROI

Defect correlations

1. We can predict defects from programmer actions.

2. We can isolate defect-prone programmer actions.

3. We can prevent defects by restricting programmer actions.

Hypotheses

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions.

Hypotheses

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions.

Hypotheses

Explicit causes

Explicit causes

IROP keyboard

Coding standards

```
if (p != null)
   { int i; while (p[i] < 0) i++; return i; }
```

Coding standards

```
if (p != null)
   { int i; while (p[i] < 0) i++; return i; }
```

```
when (q != null)
   { num n; as (q[n] < 0) n++; handback n; }
```

Coding standards

```
when (q != null)
    { num n; as (q[n] < 0) n++; handback n; }
```

Coding standards

100% semantics preserving

```
when (q != null)
  { num n; as (q[n] < 0) n++; handback n; }
```

Coding standards

New habits

We can shun these set majuscules,

and the text stays just as swell as

antecedently. Let us just ban them!

New habits

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions.

Hypotheses

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions. ✓

Hypotheses

FAQs and threats

1. How about external validity?

   (findings based on ≥177,000,000 characters + 1,000s of defects; one of largest studies ever)

FAQs and threats

1. How about external validity?

   (findings based on ≥177,000,000 characters + 1,000s of defects; one of largest studies ever)

2. Are the correlations significant?

   (yes – all of them)

FAQs and threats

1. How about external validity?

   (findings based on ≥177,000,000 characters +
   1,000s of defects; one of largest studies ever)

2. Are the correlations significant?

   (yes – all of them)

3. Are the measures appropriate?

   (all code originally input as characters;
   no abstraction taken that could interfere)

FAQs and threats

Future work

- Automatic renamings

  (PROMISE → ENGAGEMENT, Eclipse → Eclse)

Future work

- Automatic renamings

  (PROMISE → ENGAGEMENT, Eclipse → Eclse)

- Abstraction

  (Failure / mistake / error / problem / bug report vs. success / fame)

Future work

- Automatic renamings

  (PROMISE → ENGAGEMENT, Eclipse → Eclse)

- Abstraction

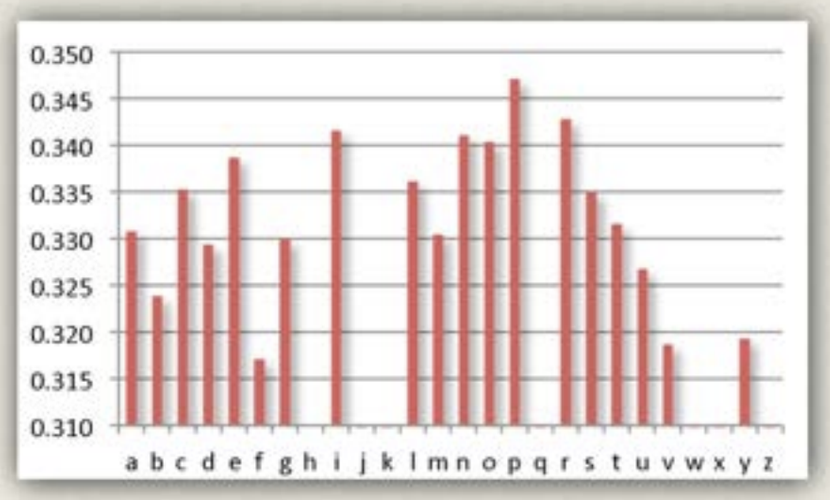
  (Failure / mistake / error / problem / bug report vs. success / fame)

- Generalization

  (ИРОП principle)

Future work

Failure is a four-letter word



Defect correlations

IROP

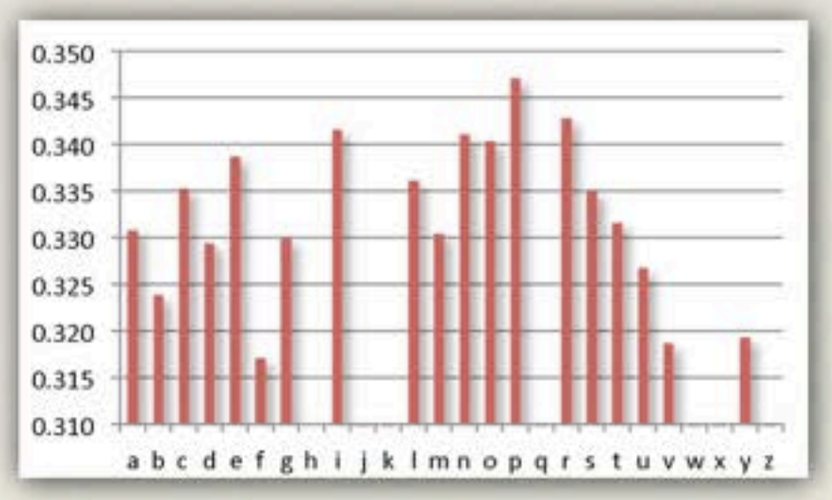
Defect correlations

Failure is a four-letter word

IROP keyboard

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions. ✓

Hypotheses



Defect correlations

IROP

Defect correlations

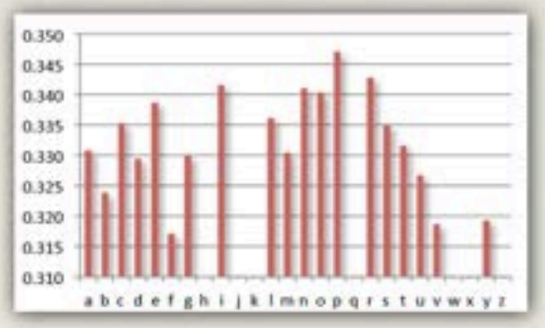
Failure is a four-letter word

IROP keyboard

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions. ✓

Hypotheses

Why all this is wrong



Defect correlations

IROP

Defect correlations

Failure is a four-letter word

IROP keyboard

1. We can predict defects from programmer actions. ✓
2. We can isolate defect-prone programmer actions. ✓
3. We can prevent defects by restricting programmer actions. ✓

Hypotheses

Correlation vs. Causation

```
//////
********
aaaaaaaaaa
ccccccccc
d
eeeeeeeeeeeeeeee
ggggggg
hh
iiiiiiiiiiii
lllllllllll
mmmm
nnnnnnnnnnnnnnn
ooooooooo
pppppppppp
rrrrrrrrrr
ssssssssssss
```

```
ttttttttttttttttttttt
uuuuuuuuuuuu
v
wwww
A
B
C
E
IIII
PPPP
TTT
U
VVV
{{{
}}}
```

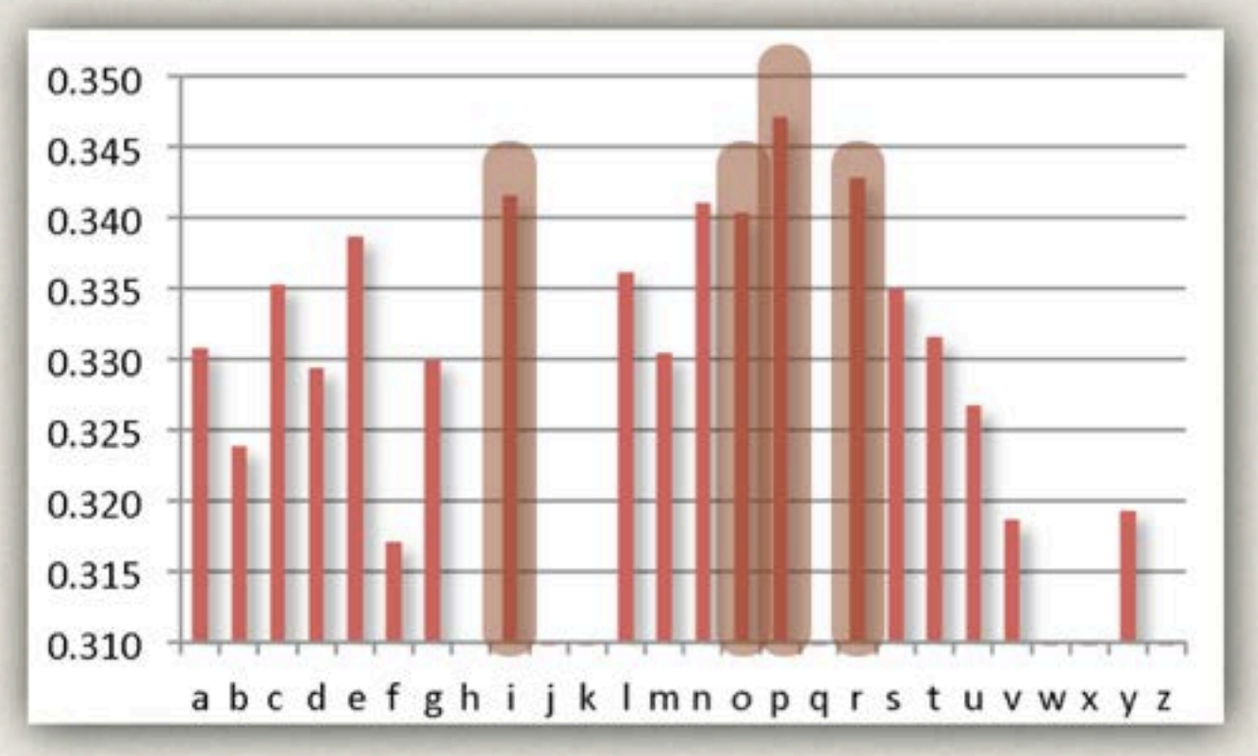
No abstraction

Machine Learning works

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Precision

Cherry Picking



Defect correlations

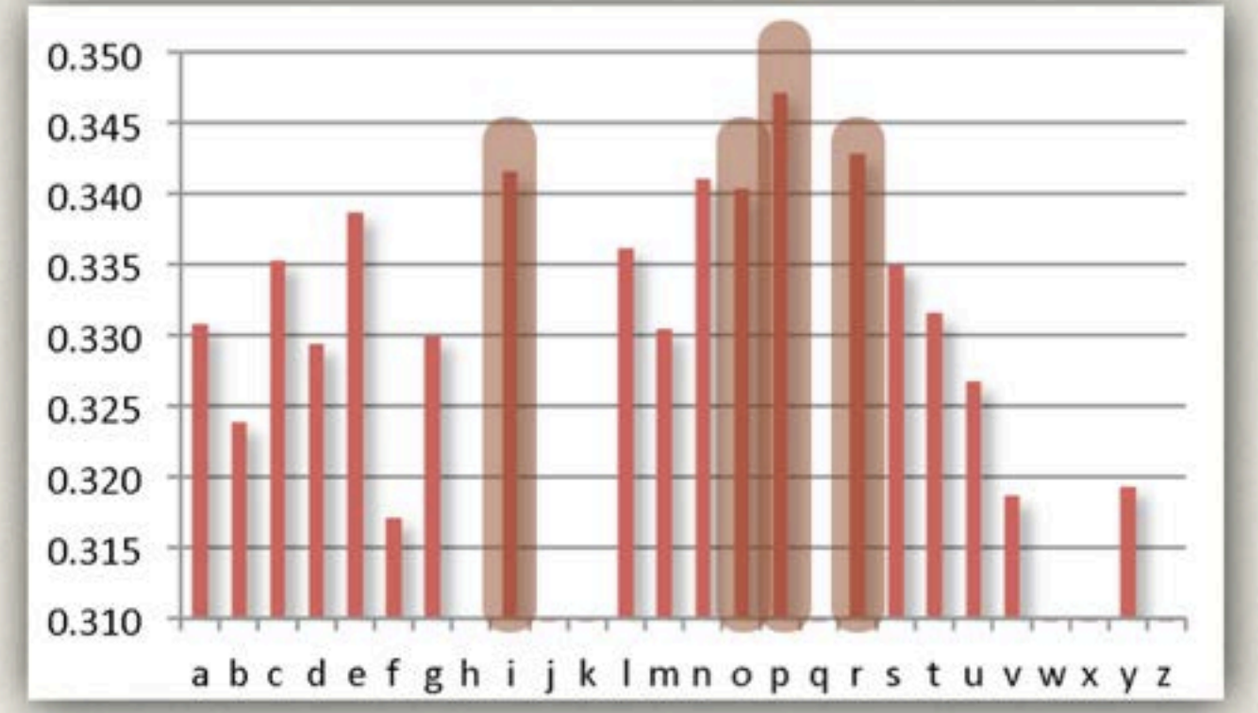
Fix Causes, not Symptoms

100%
semantics
preserving

```
when (q != null)
    { num n; as (q[n] < 0) n++; handback n; }
```

Coding standards

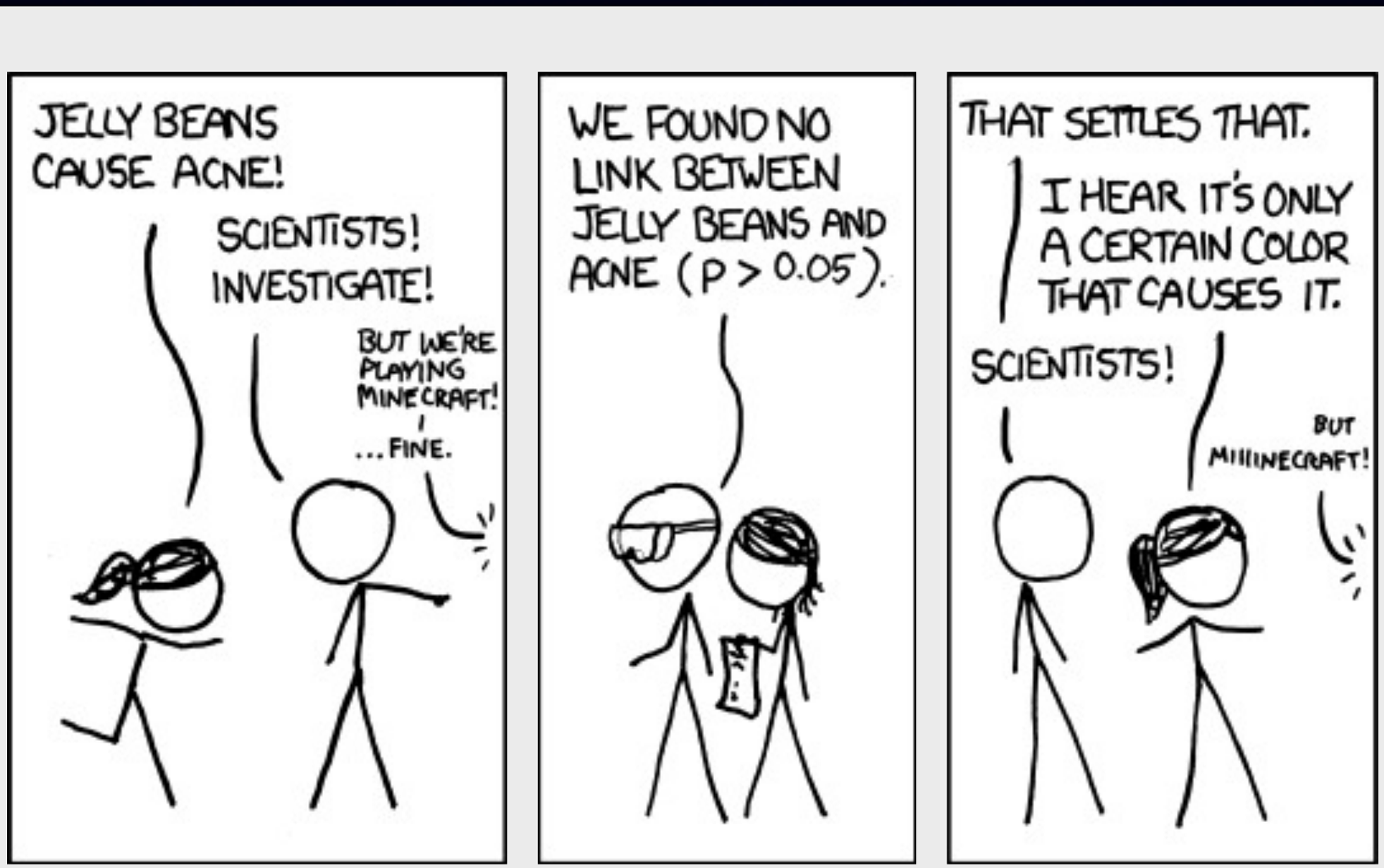
Actionable Findings

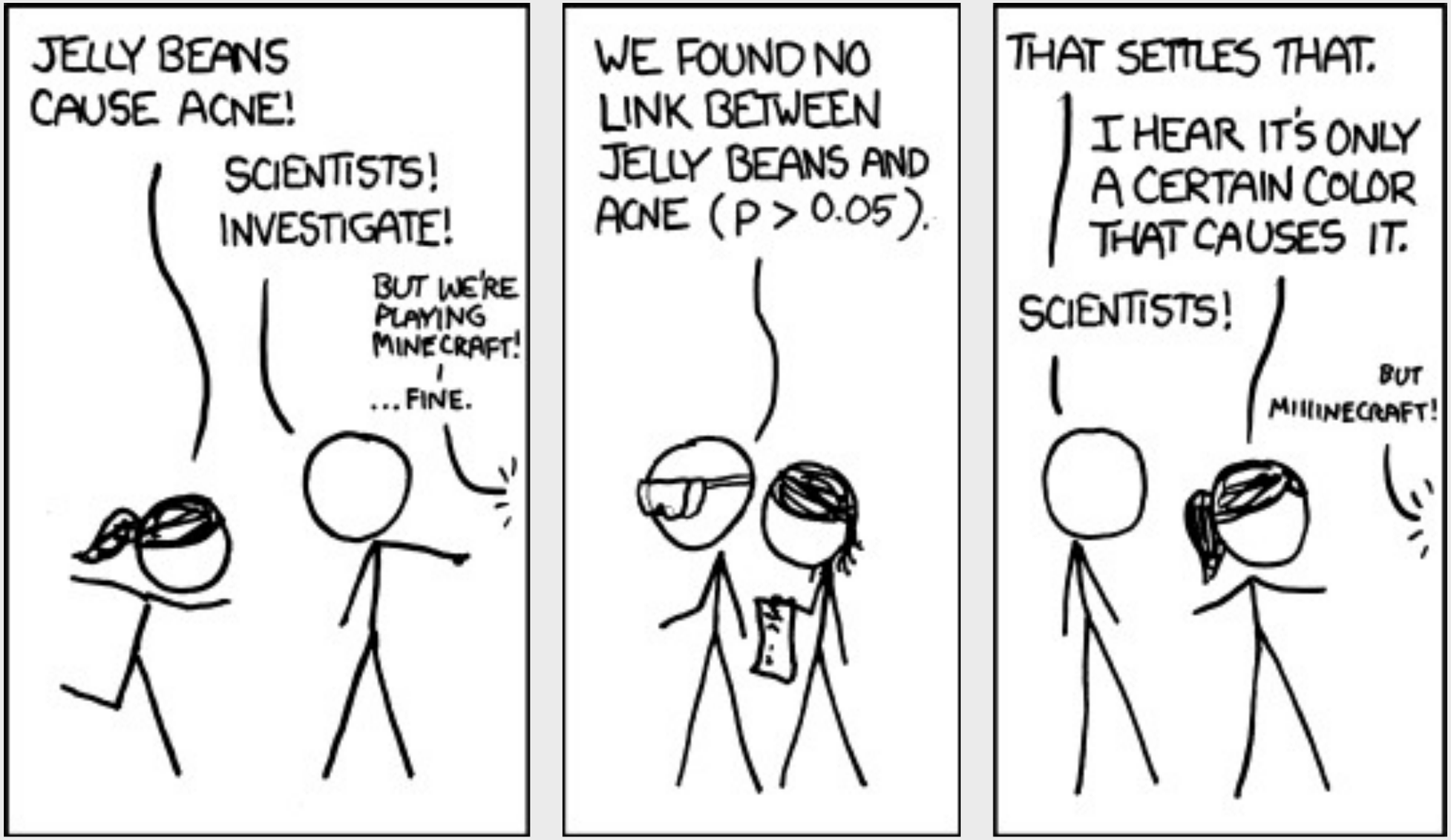


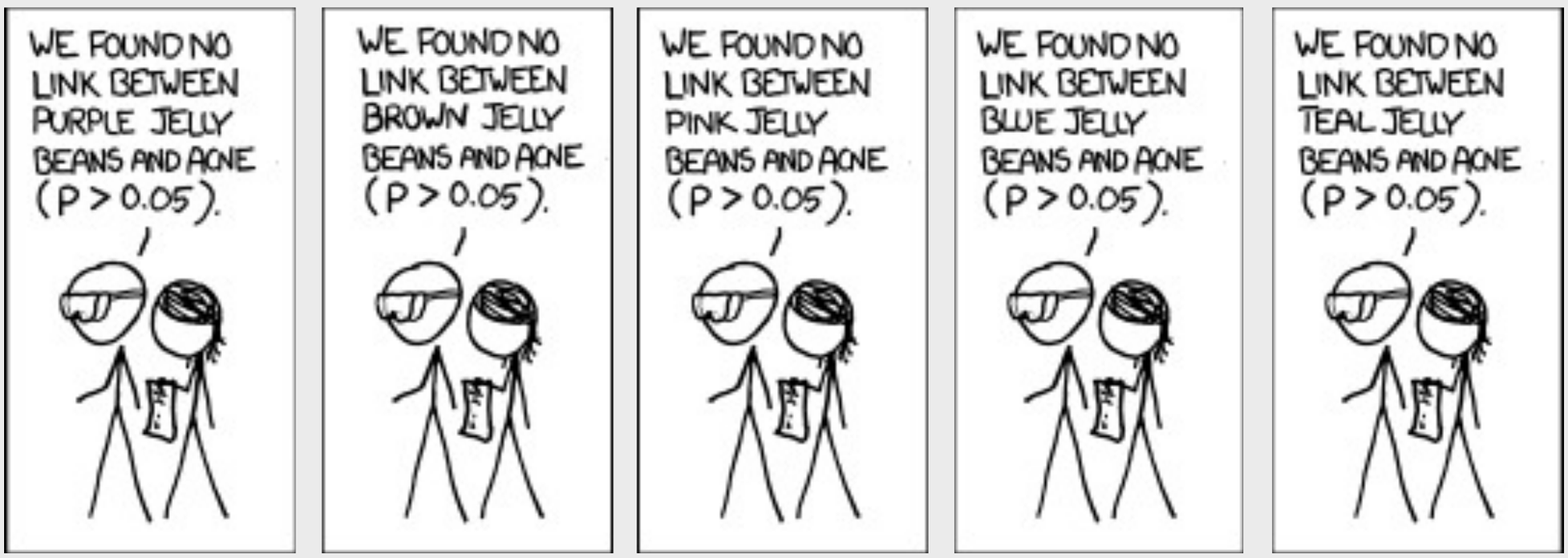
Defect correlations

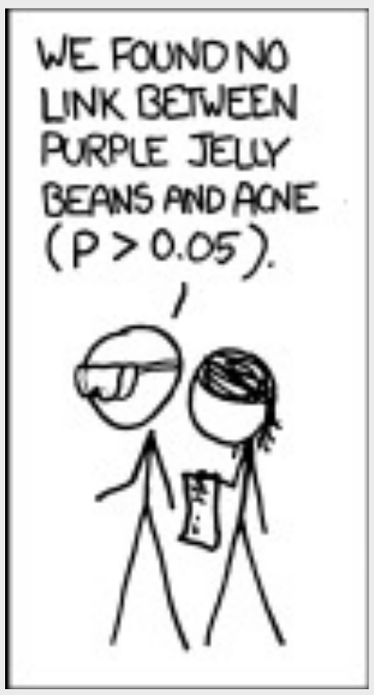
Our Inspiration

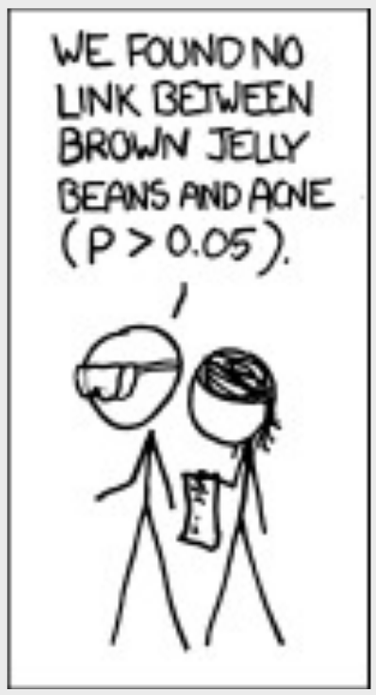
http://xkcd.com/882/

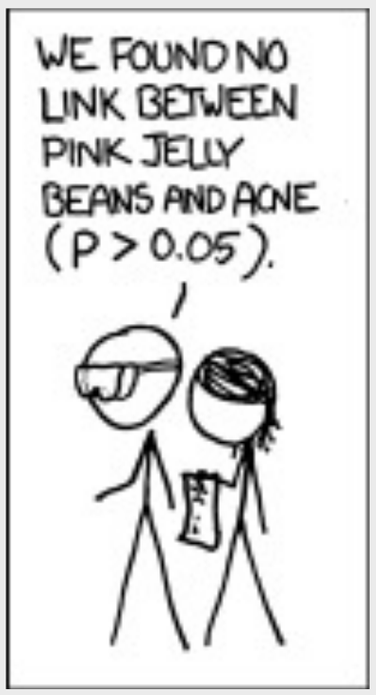


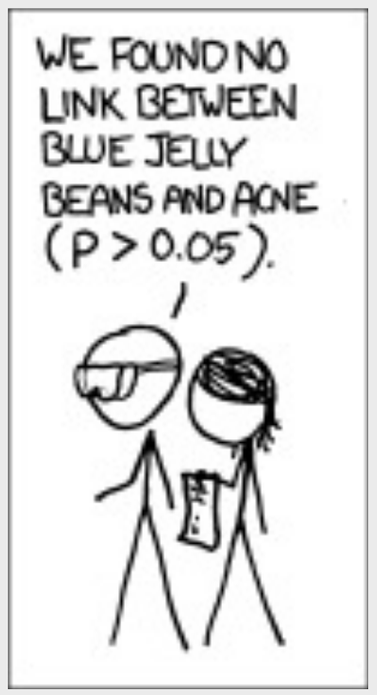

JELLY BEANS CAUSE ACNE!
SCIENTISTS! INVESTIGATE!
BUT WE'RE PLAYING MINECRAFT!
... FINE.
WE FOUND NO LINK BETWEEN JELLY BEANS AND ACNE (P > 0.05).
THAT SETTLES THAT.
I HEAR IT'S ONLY A CERTAIN COLOR THAT CAUSES IT.
SCIENTISTS!
BUT MIIIINECRAFT!

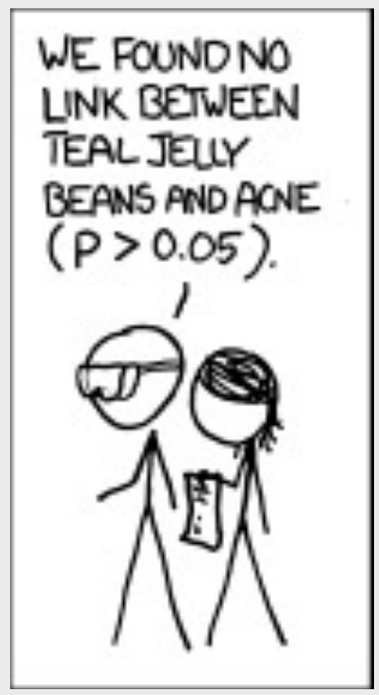

WE FOUND NO LINK BETWEEN PURPLE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BROWN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN PINK JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BLUE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TEAL JELLY BEANS AND ACNE (P > 0.05).

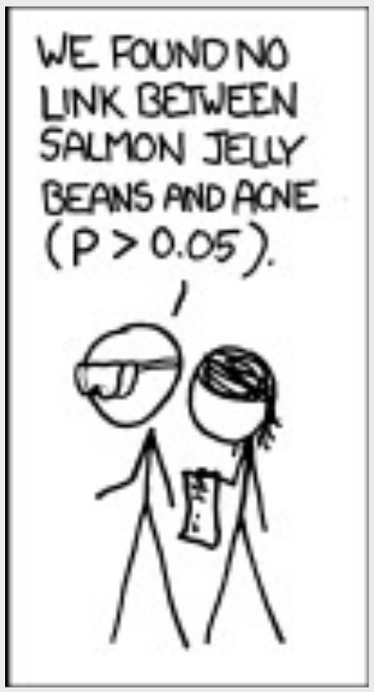

WE FOUND NO LINK BETWEEN PURPLE JELLY BEANS AND ACNE (P > 0.05).

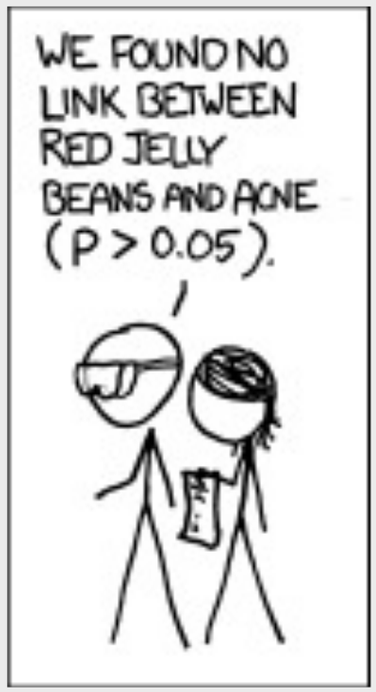

WE FOUND NO LINK BETWEEN BROWN JELLY BEANS AND ACNE (P > 0.05).

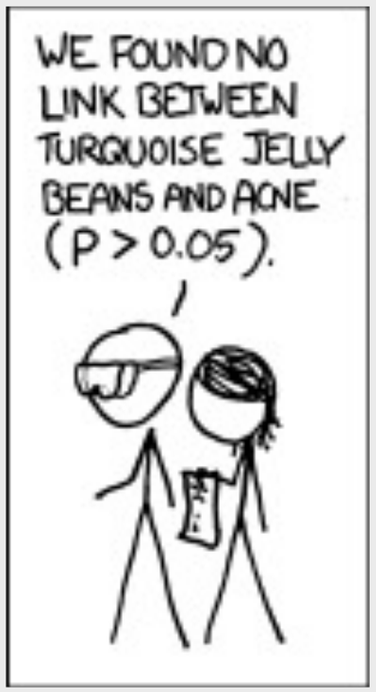

WE FOUND NO LINK BETWEEN PINK JELLY BEANS AND ACNE (P > 0.05).

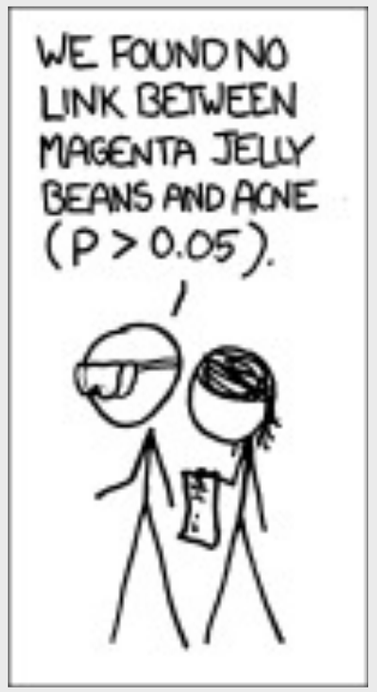

WE FOUND NO LINK BETWEEN BLUE JELLY BEANS AND ACNE (P > 0.05).

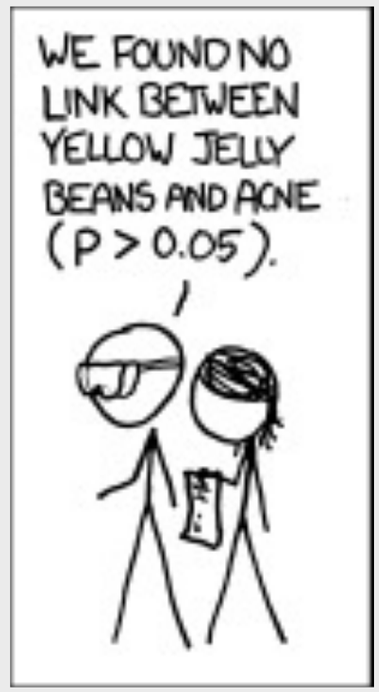

WE FOUND NO LINK BETWEEN TEAL JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN SALMON JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN RED JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN TURQUOISE JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN MAGENTA JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN YELLOW JELLY BEANS AND ACNE (P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

WE FOUND NO LINK BETWEEN SALMON JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN RED JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN TURQUOISE JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN MAGENTA JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN YELLOW JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN GREY JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN TAN JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN CYAN JELLY BEANS AND ACNE (P > 0.05).

WE FOUND A LINK BETWEEN GREEN JELLY BEANS AND ACNE (P < 0.05).

WE FOUND NO LINK BETWEEN MAUVE JELLY BEANS AND ACNE (P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

(P > 0.05).

WE FOUND NO LINK BETWEEN GREY JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN TAN JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN CYAN JELLY BEANS AND ACNE (P > 0.05).

WE FOUND A LINK BETWEEN GREEN JELLY BEANS AND ACNE (P < 0.05).

WHOA!

WE FOUND NO LINK BETWEEN MAUVE JELLY BEANS AND ACNE (P > 0.05).

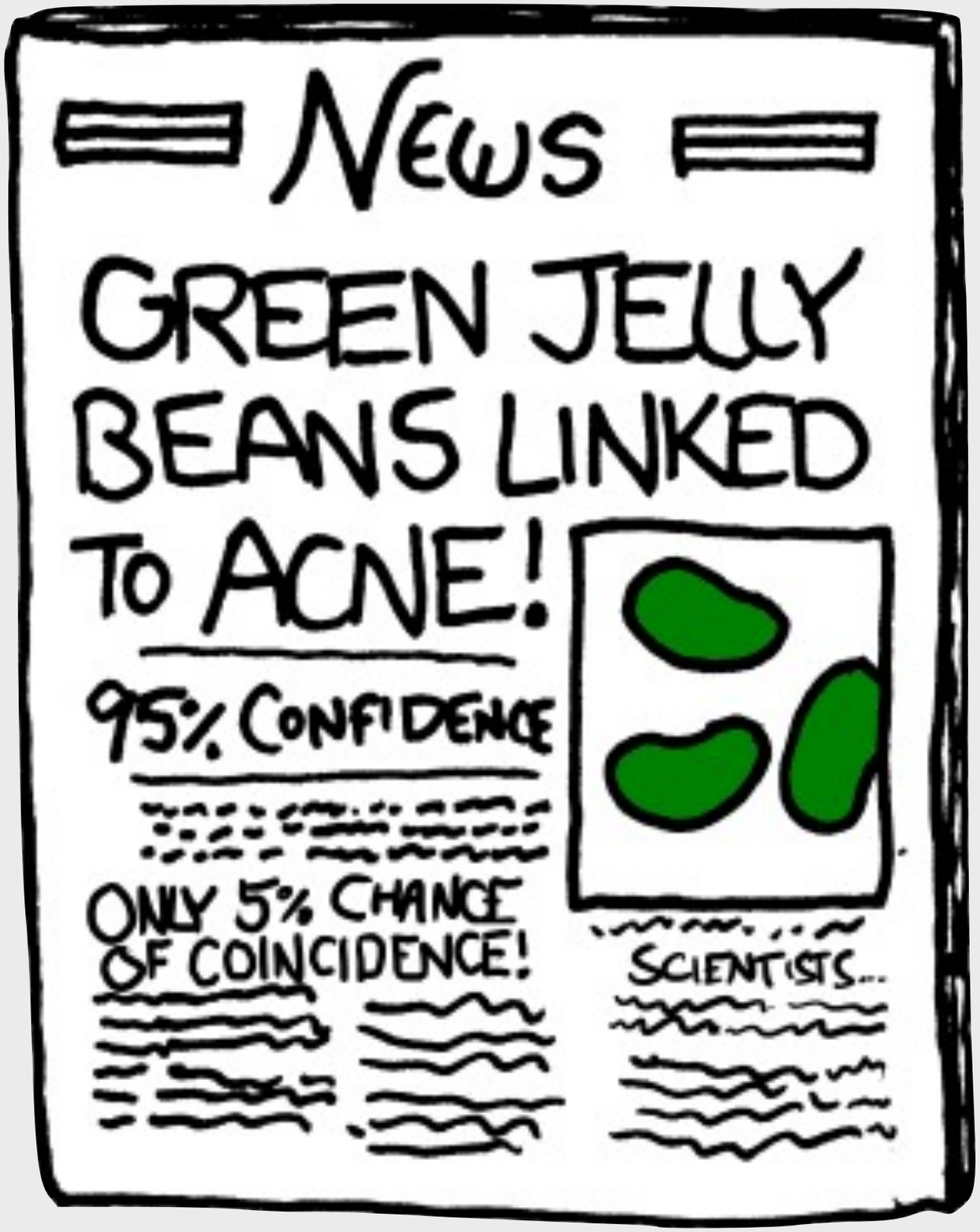
WE FOUND NO LINK BETWEEN BEIGE JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN LILAC JELLY BEANS AND ACNE (P > 0.05).

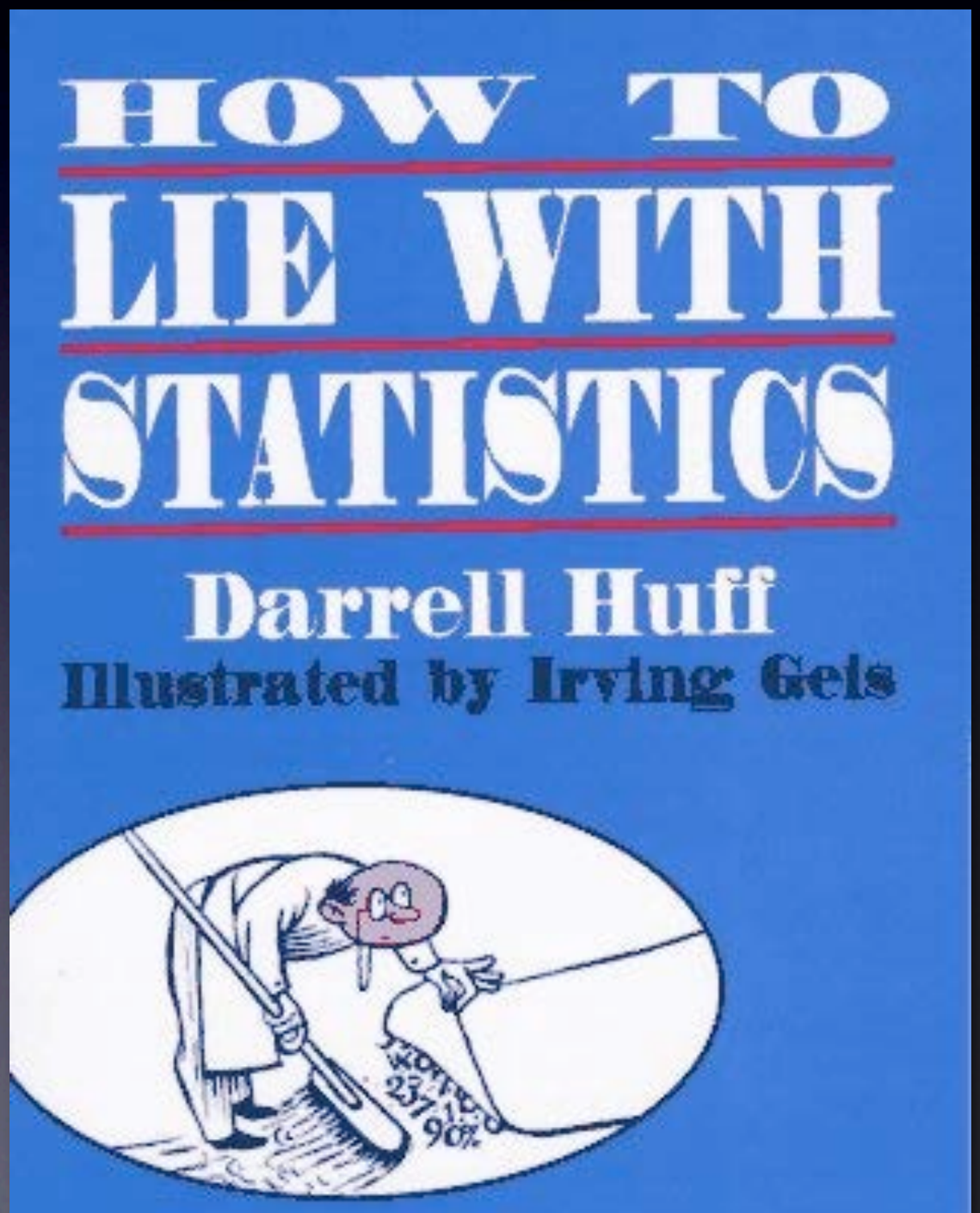
WE FOUND NO LINK BETWEEN BLACK JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN PEACH JELLY BEANS AND ACNE (P > 0.05).

WE FOUND NO LINK BETWEEN ORANGE JELLY BEANS AND ACNE (P > 0.05).


NEWS
GREEN JELLY BEANS LINKED TO ACNE!
95% CONFIDENCE
ONLY 5% CHANCE OF COINCIDENCE!
SCIENTISTS...

Use Book in Class



Use Paper in Class

Failure is a Four-Letter Word

– A Parody in Empirical Research –

Andreas Zeller*
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

Thomas Zimmermann
Microsoft Research
Washington, USA
tzimmer@microsoft.com

Christian Bird
Microsoft Research
Washington, USA
cbird@microsoft.com

ABSTRACT

Background: The past years have seen a surge of techniques predicting failure-prone locations based on more or less complex metrics. Few of these metrics are *actionable*, though.

Aims: This paper explores a simple, easy-to-implement method to predict and avoid failures in software systems. The IROP method links *elementary source code features* to known software failures in a lightweight, easy-to-implement fashion.

Method: We sampled the Eclipse data set mapping defects to files in three Eclipse releases. We used logistic regression to associate programmer actions with defects, tested the predictive power of the resulting classifier in terms of precision and recall, and isolated the most defect-prone actions. We also collected initial feedback on possible remedies.

Results: In our sample set, IROP correctly predicted up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. We isolated a set of four easy-to-remember recommendations, telling programmers precisely what to do to avoid errors. Initial feedback from developers suggests that these recommendations are straightforward to follow in practice.

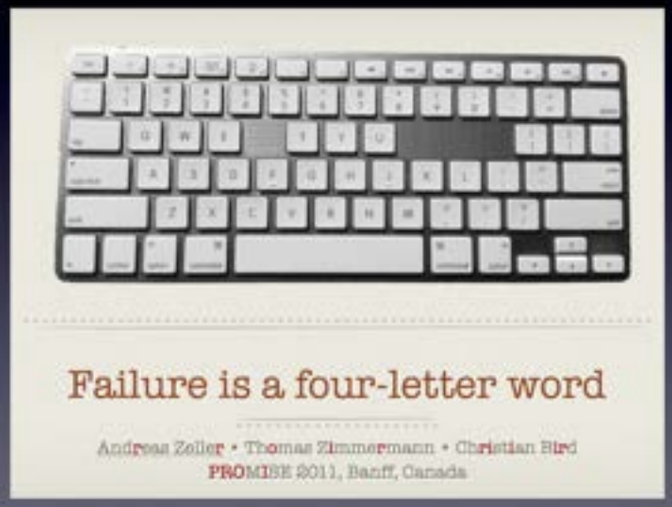
Conclusions: With the abundance of software development data, even the simplest methods can produce "actionable" results.

number of developers associated with a file. As elaborate as these approaches may be, they all share the same problem which we call *the cost of consequence*: If I know that a module is failure-prone because it frequently changes, should I stop changing it? If I know failures are related to complexity, should I rewrite it from scratch? Any of these measures induces a new risk – a risk which may be greater than the one originally addressed.

In this paper, we take a different approach. We predict failures from the most basic actions programmers undertake, focusing on the actions that introduce defects *as they are being made* – literally at the moment the source code is typed in. Our recommendations are *immediately actionable:* A simple visual representation associates actions with the likelihood of introducing defects – warning programmers before they might hit the wrong key. Our approach is both effective and efficient: In a case study on the Eclipse failure set, it correctly identified up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. Specifically, our contributions include:

1) A novel mechanism to associate programmer actions with software defects;
2) A predictor that is purely text-oriented, thus lightweight, real-time, easy to implement, and language-agnostic;

Why all this is wrong



Failure is a four-letter word

Andreas Zeller • Thomas Zimmermann • Christian Bird
PROMISE 2011, Banff, Canada

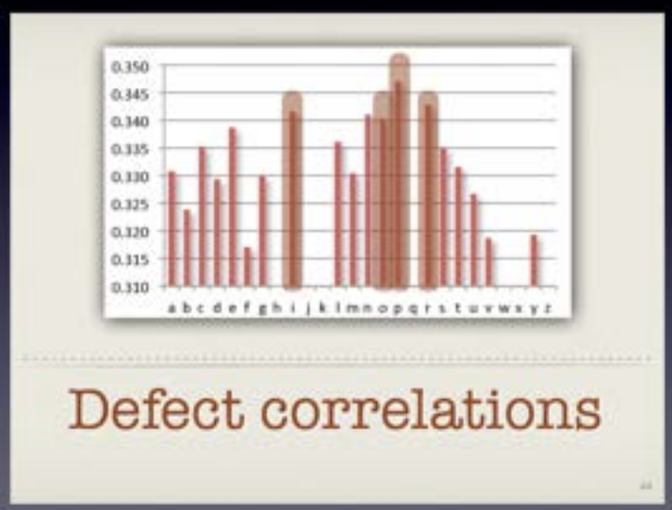
Machine Learning works

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Precision

http://www.st.cs.uni-saarland.de/softevo/irop/

Actionable Findings



Defect correlations

Use Paper in Class

Failure is a Four-Letter Word

– A Parody in Empirical Research –

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

Thomas Zimmermann
Microsoft Research
Washington, USA
tzimmer@microsoft.com

Christian Bird
Microsoft Research
Washington, USA
cbird@microsoft.com

ABSTRACT

Background: The past years have seen a surge of techniques predicting failure-prone locations based on more or less complex metrics. Few of these metrics are *actionable*, though.

Aims: This paper explores a simple, easy-to-implement method to predict and avoid failures in software systems. The IRQF method links *elementary source code features* to known software failures in a lightweight, easy-to-implement fashion.

Method: We sampled the Eclipse data set mapping defects to files in three Eclipse releases. We used logistic regression to associate programmer actions with defects, tested the predictive power of the resulting classifier in terms of precision and recall, and isolated the most defect-prone actions. We also collected initial feedback on possible remedies.

Results: In our sample set, IRQF correctly predicted up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. We isolated a set of four easy-to-remember recommendations, telling programmers precisely what to do to avoid errors. Initial feedback from developers suggests that these recommendations are straightforward to follow in practice.

Conclusion: With the abundance of software development data, even the simplest methods can produce "actionable" results.

number of developers associated with a file. As elaborate as these approaches may be, they all share the same problem which we call *the cost of consequence*: If I know that a module is failure-prone because it frequently changes, should I stop changing it? If I know failures are related to complexity, should I rewrite it from scratch? Any of these measures induces a new risk – a risk which may be greater than the one originally addressed.

In this paper, we take a different approach. We predict failures from the most basic actions programmers undertake, focusing on the actions that introduce defects *as they are being made* – literally at the moment the source code is typed in. Our recommendations are *immediately actionable*: A simple visual representation associates actions with the likelihood of introducing defects – warning programmers before they might hit the wrong key. Our approach is both effective and efficient: In a case study on the Eclipse failure set, it correctly identified up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. Specifically, our contributions include:

1) A novel mechanism to associate programmer actions with software defects;
2) A predictor that is purely text-oriented, thus lightweight, real-time, easy to implement, and language-agnostic;