The Paradox of Software Quality

Why More Bugs Indicate Better Software?

Audris Mockus

Avaya Labs Research
211 Mt Airy Rd
Basking Ridge, NJ 07920
audris@avaya.com

Nov 7, 2013

Motivation

Are there fundamental time-lag relationships among software production factors?

Can they be harnessed to improve software development?

Empirical Studies of Software Development

Typical investigated relationships in software, e.g., size and defects

- Are caused by external-to-software factors
  - Short term trends: product adoption, extent of usage
  - Long term trends: world economy, business practice, technology
- Have unclear mechanism of action
- Typically not usable in practice

Empirical Studies of Software Development

Typical investigated relationships in software, e.g., size and defects

- Are caused by external-to-software factors
  - Short term trends: product adoption, extent of usage
  - Long term trends: world economy, business practice, technology
- Have unclear mechanism of action
- Typically not usable in practice

`Investigating SW evolution by observing only software` $\equiv$ `Divining reality from shadows on a cave wall`

Proposed Solution

- Investigate short-term and recurring relationships with a clear mechanism originating from the way software is created and used
- Use information from outside software development cave
- Answer actual software engineering questions
  - How to to evaluate the effectiveness of QA practices?
    - e.g., by comparing two releases of software
  - Do easy-to-get measures, e.g., defects, approximate quality?

Approach

- Start from clear assumptions
- Observe fundamental relationships
- Validate
- Build more complex propositions using validated relationships

Define: *Bug*

A user-observed (and reported) program behavior (e.g., failure) that results in a code change.

Define: *Action Will Introduce a Bug*

Action will increase the chances of a *Bug* occurring in the future.

Assumed background knowledge

Developers **create** software by making *changes* to code

- All changes **are recorded** by a Version Control System
- A release of software is simply a dynamic superposition of changes

Before:

```
int i = n;
while(i++)
    prinf(" %d", i--);
```

After:

```
//print n integers
int i = n;
while(i++ && i > 0)
    prinf(" %d", i--);
```

one line deleted
two lines added
two lines unchanged

Assumed background knowledge

Developers **create** software by making *changes* to code

- All changes **are recorded** by a Version Control System
- A release of software is simply a dynamic superposition of changes

Before:

```
int i = n;
while(i++)
    prinf(" %d", i--);
```

After:

```
//print n integers
int i = n;
while(i++ && i > 0)
    prinf(" %d", i--);
```

one line deleted
two lines added
two lines unchanged

Assumed background knowledge

Developers **create** software by making *changes* to code

- All changes **are recorded** by a Version Control System
- A release of software is simply a dynamic superposition of changes

Before:

```
int i = n;
while(i++)
    prinf(" %d", i--);
```

After:

```
//print n integers
int i = n;
while(i++ && i > 0)
    prinf(" %d", i--);
```

one line deleted
two lines added
two lines unchanged

Assumed background knowledge

Developers **create** software by making *changes* to code

- All changes **are recorded** by a Version Control System
- A release of software is simply a dynamic superposition of changes

Before:

```
int i = n;
while(i++)
    prinf(" %d", i--);
```

After:

```
//print n integers
int i = n;
while(i++ && i > 0)
    prinf(" %d", i--);
```

one line deleted
two lines added
two lines unchanged

Assumed background knowledge

Developers **create** software by making *changes* to code

- All changes **are recorded** by a Version Control System
- A release of software is simply a dynamic superposition of changes

Before:

```
int i = n;
while(i++)
    prinf(" %d", i--);
```

After:

```
//print n integers
int i = n;
while(i++ && i > 0)
    prinf(" %d", i--);
```

one line deleted
two lines added
two lines unchanged

Other attributes: date, developer, defect number,
submit comment: e.g, "Fix bug 3987 - crashing when menu item is selected"

First Fundamental Law of Software Evolution

Formulation

`Code change will introduce bugs`

Mechanism

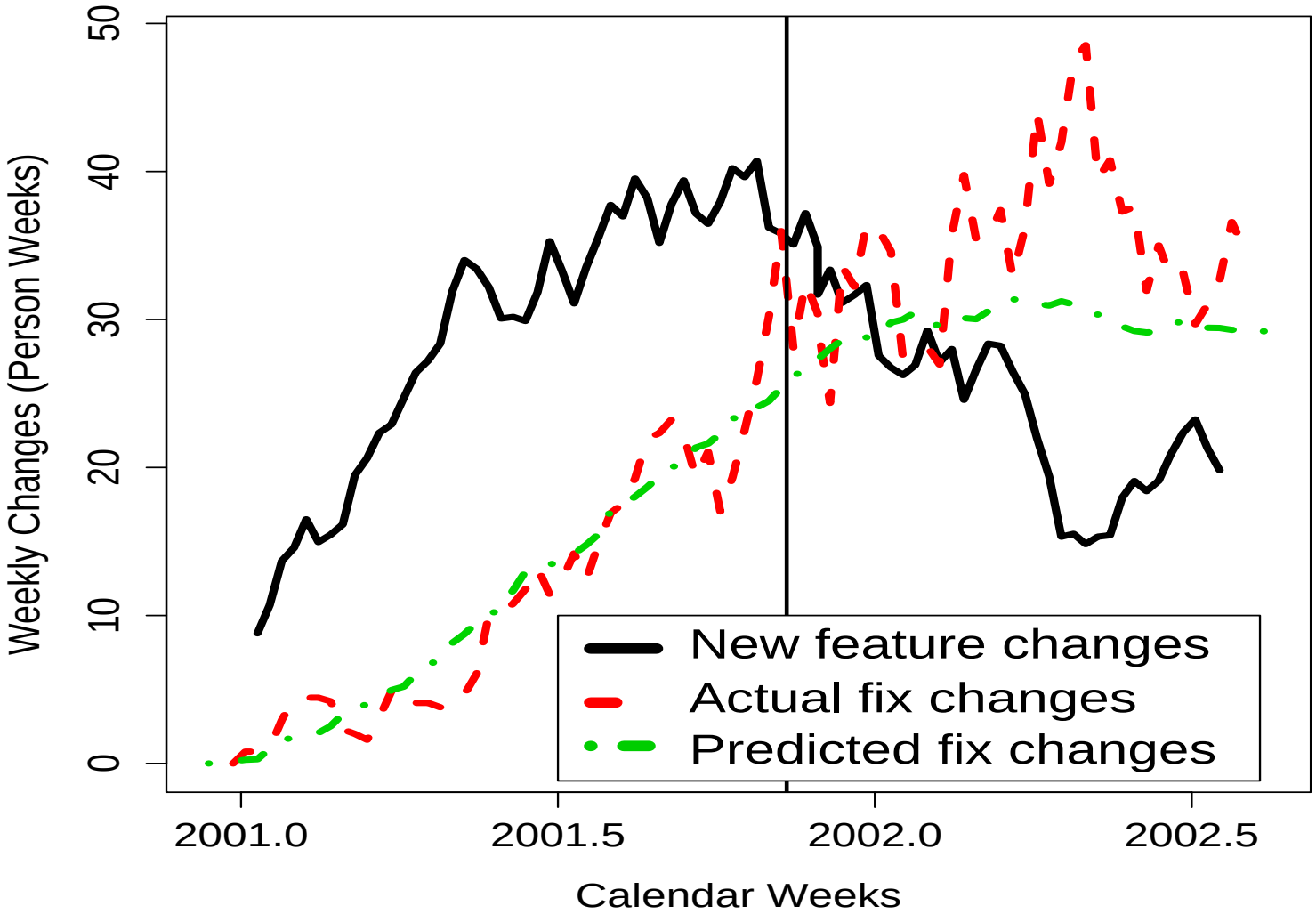
- New code has defects
- New code exercises existing code differently
- Program behavior changes

Note: platform changes cause code changes

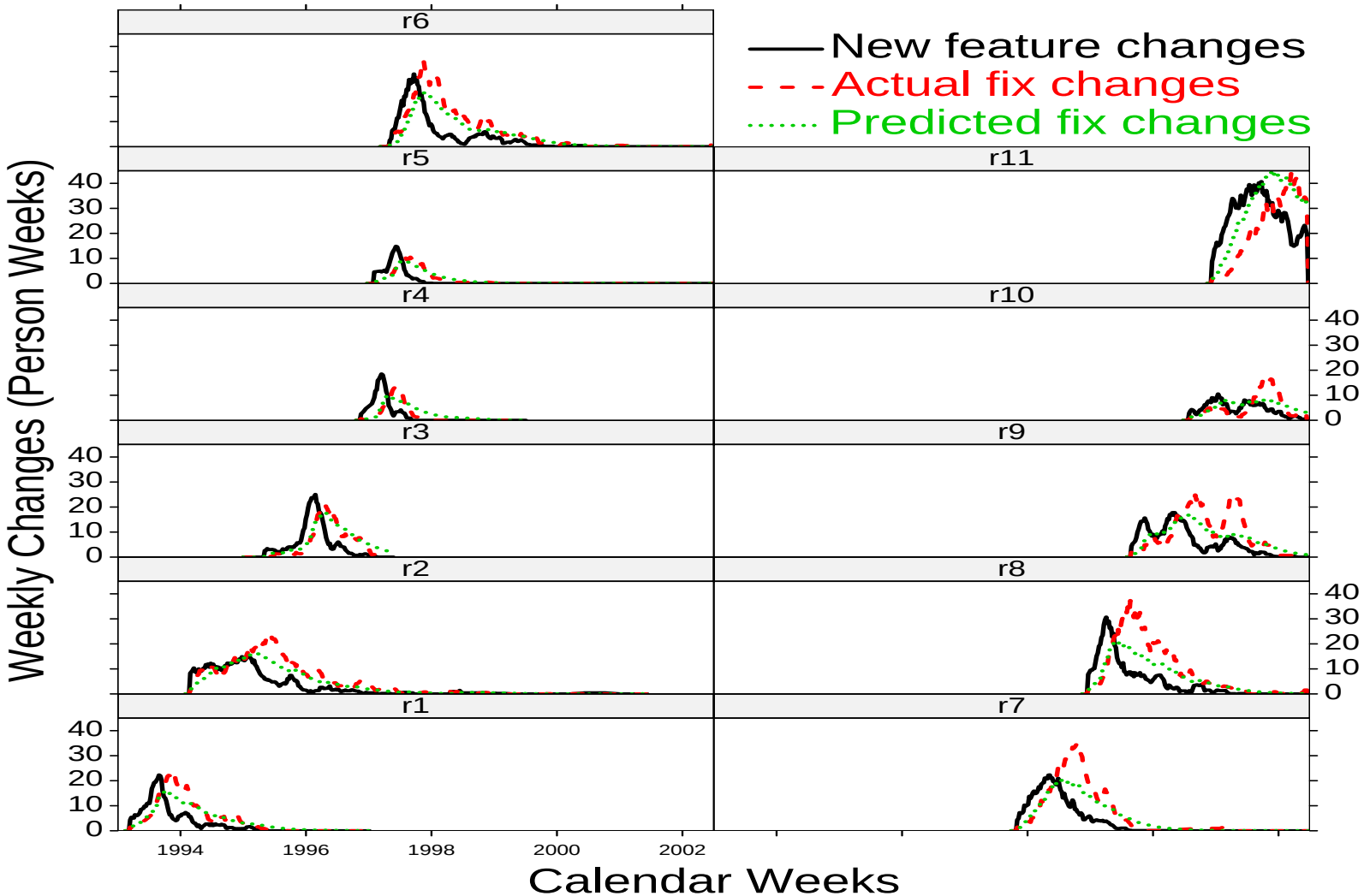
Evidence

- New releases bring new bugs
- Model: a business-driven feature implementation code change leads to $N \sim Poisson(\lambda)$ fixes with delay $T \sim Exp(\mu)$ [1]

Model prediction for one release



Model prediction for 11 releases (using earlier release)



Corollary 1: Need to Normalize by Change to Obtain Quality

How to normalize by change?

- Divide by the number of pre-release changes
- Divide by the LOC added or changed

Hypothesis 1

Increase ↑ in the number of customer-found defects per pre-release change (a simple-to-obtain measure) affects users' perception of software quality negatively ↓

Qualitative evidence: No

Quotes from a quality manager

"**we tried to improve quality**: get most experienced team members to test, do code inspections, conduct root cause analysis, ..."

"**Did it work?** I.e., is this release better than previous one?"

Everyone uses **defect density** (e.g.,customer reported defects per 1000 changes or lines of code), but "it **does not reflect** the feedback from customers."

Let's Peek Outside the Software Development Cave

Does the increase in

the number of users and
the amount of usage

introduce bugs?

Second Fundamental Law of Software Evolution

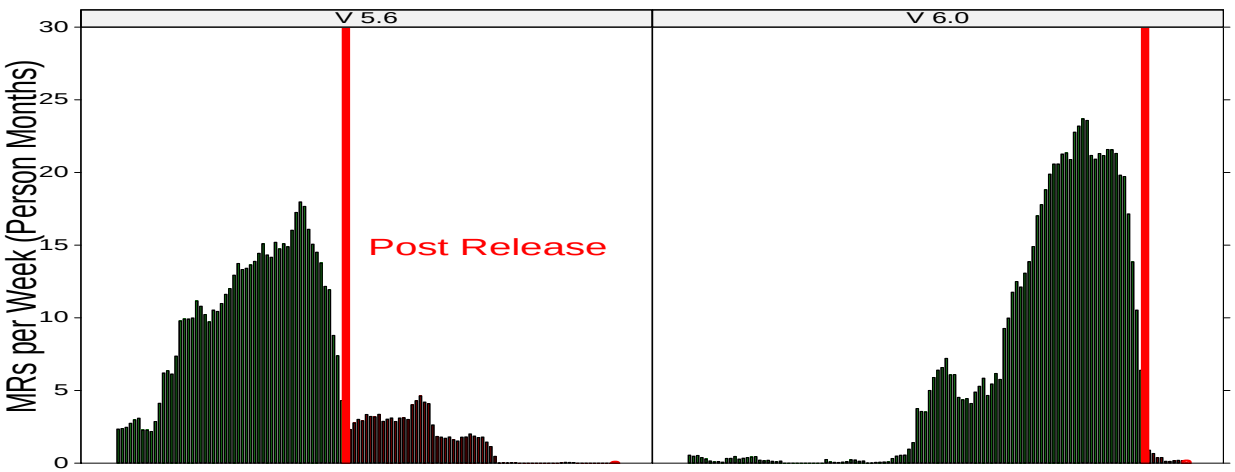
Formulation

Deploying to more users will introduce bugs

Mechanism

- New use profiles
- Different environments

Evidence



Release with no users

has no bugs

Third Fundamental Law of Software Evolution

Formulation

`Longer (and heavier) use will introduce bugs`

Mechanism

- New inputs and use cases are encountered over longer periods
- More extreme environmental conditions happen over longer periods

Evidence

- Bugs tend to be encountered even after year(s) of usage
- See Commandments below

Third Fundamental Law of Software Evolution

Formulation

`Longer (and heavier) use will introduce bugs`

Mechanism

- New inputs and use cases are encountered over longer periods
- More extreme environmental conditions happen over longer periods

Evidence

- Bugs tend to be encountered even after year(s) of usage
- See Commandments below

Does every user and every year of usage introduce the same number of bugs?

Commandment 1: Don't Install Right After the Release Date

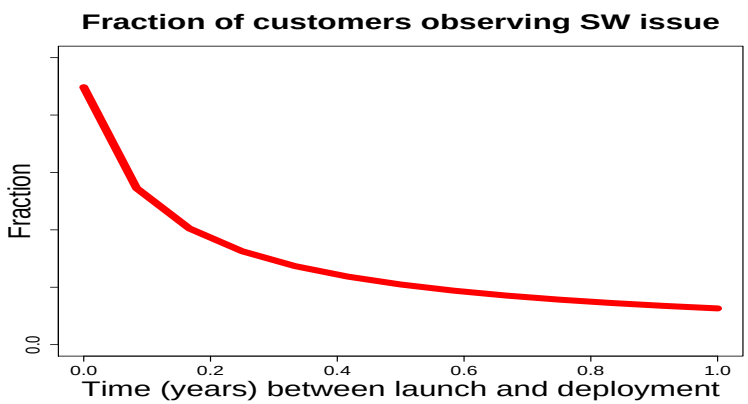
Formulation

`Users who install close to the release date will introduce more bugs`

Mechanism

- Later users get builds with patches
- Services team understands better how to install/configure properly
- Workarounds for many issues are discovered

Evidence



- Quality ↑ with time after the launch, and is an order of magnitude better one year later [2]

Commandment 2: Don't Panic After Install/Upgrade

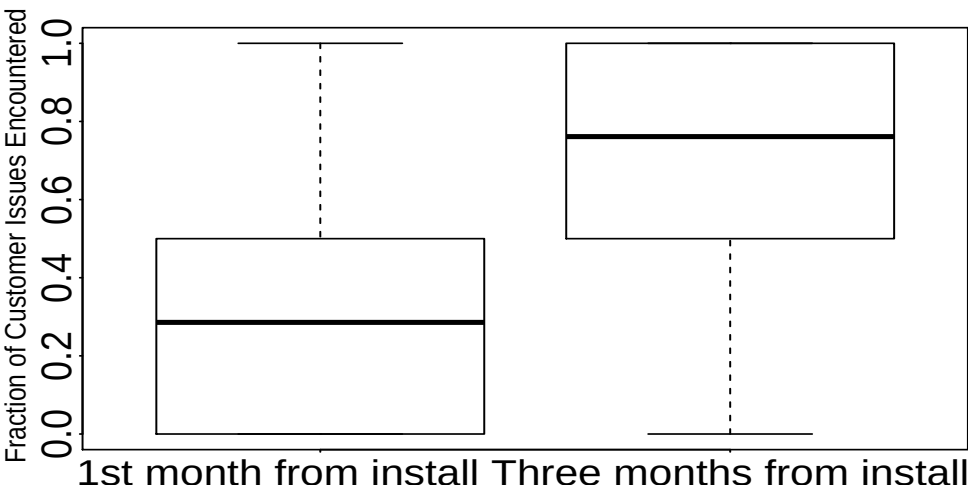
Formulation

`A user will introduce more bugs close to their install/upgrade date`

Mechanism

- Software is not hardware: parts do not wear off
- Misconfiguration or incompatibility with the environment

Evidence



- Two thirds of customer issues (leading to a software fix) are reported within three months of install
- Sample: 87 release/product combinations

Corollary 1: Customer Quality

Formulation

Software release quality from users perspective is the fraction of:

- The number of users reporting a bug shortly after the installation *over*
- The number of users who install soon after the release date

Corollary 1: Customer Quality

Formulation

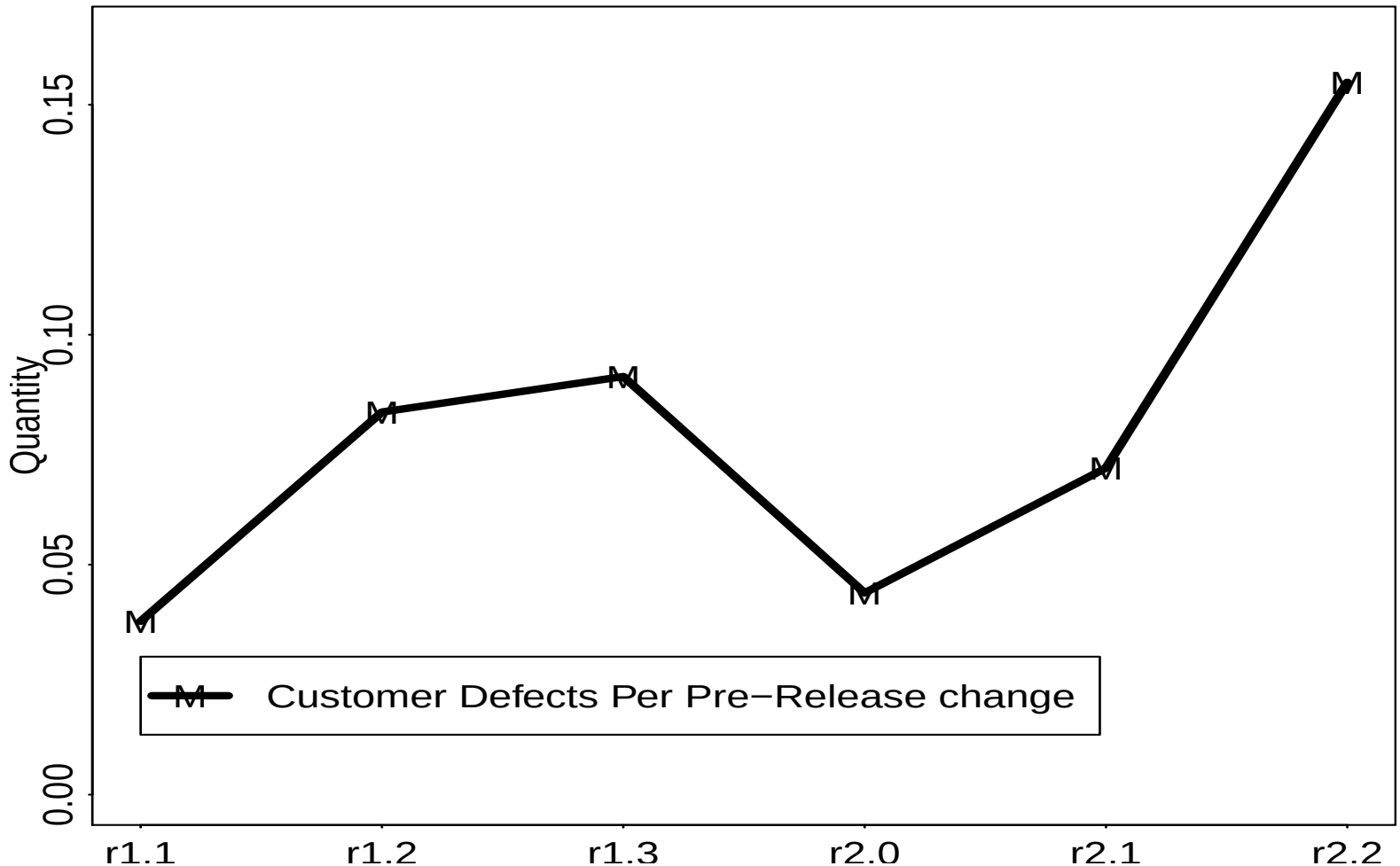
Software release quality from users perspective is the fraction of:

- The number of users reporting a bug shortly after the installation *over*
- The number of users who install soon after the release date

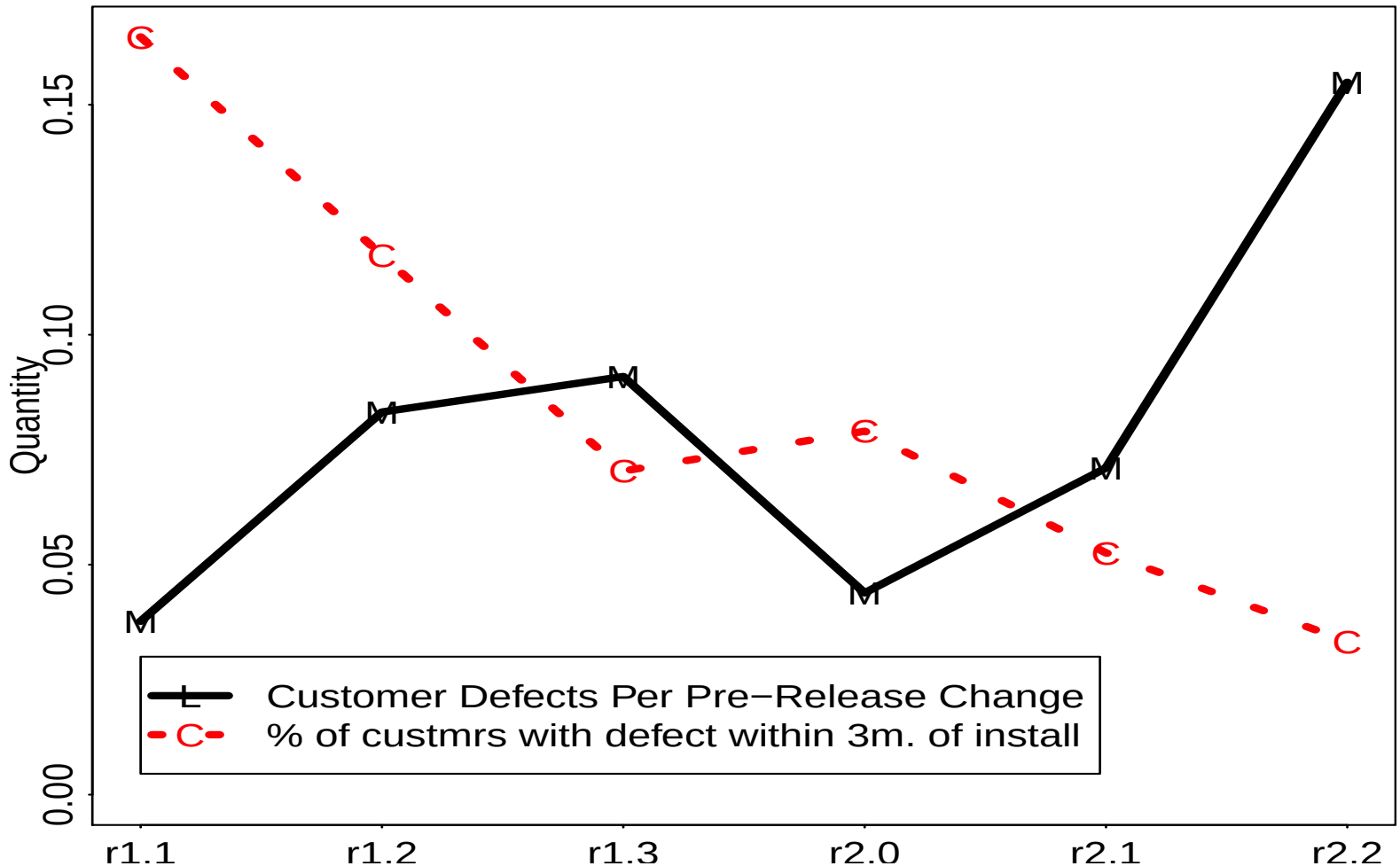
"We live or die by this measure"

VP for quality

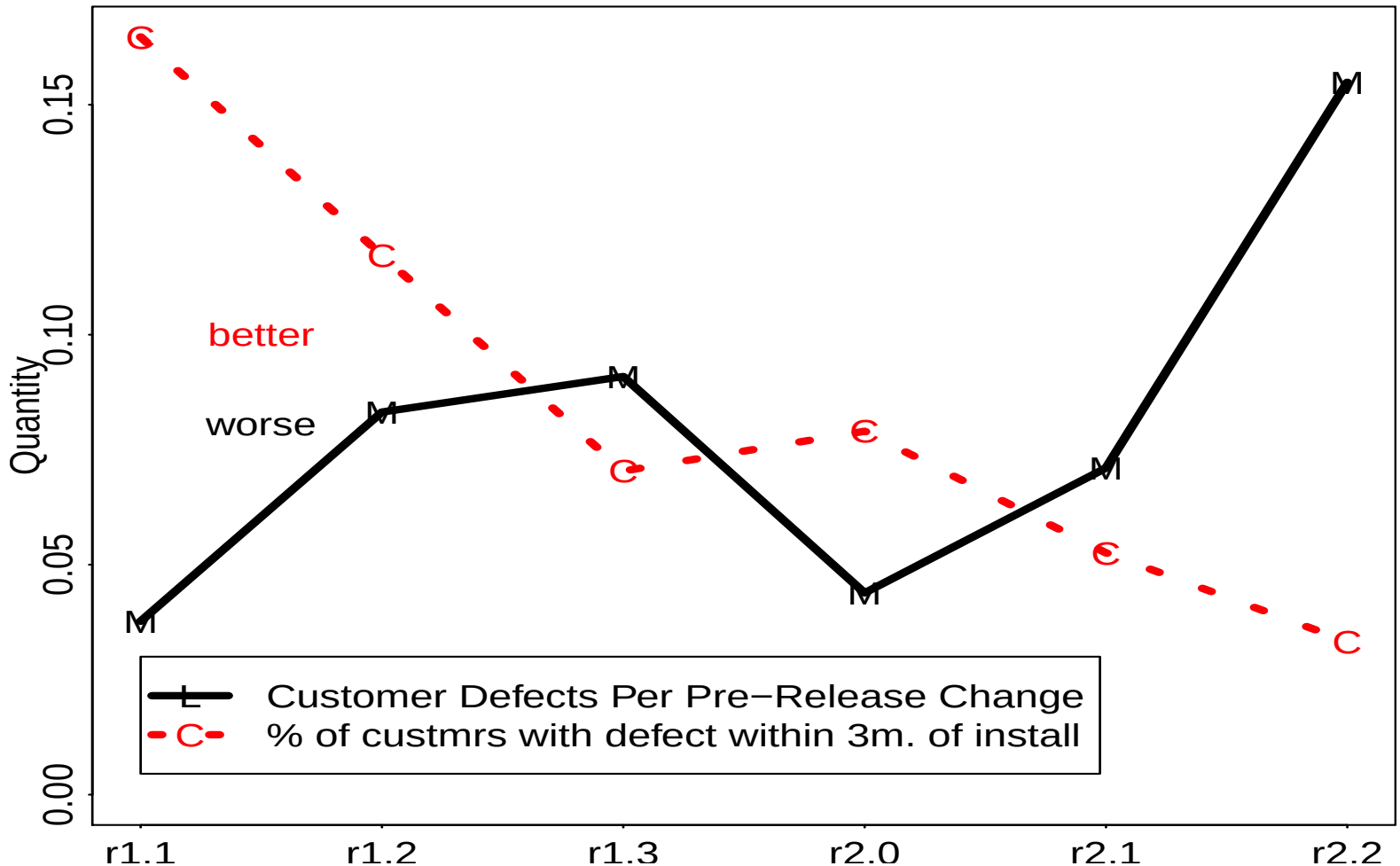
Testing Hypothesis 1: Defect Density Reflects Customer Quality



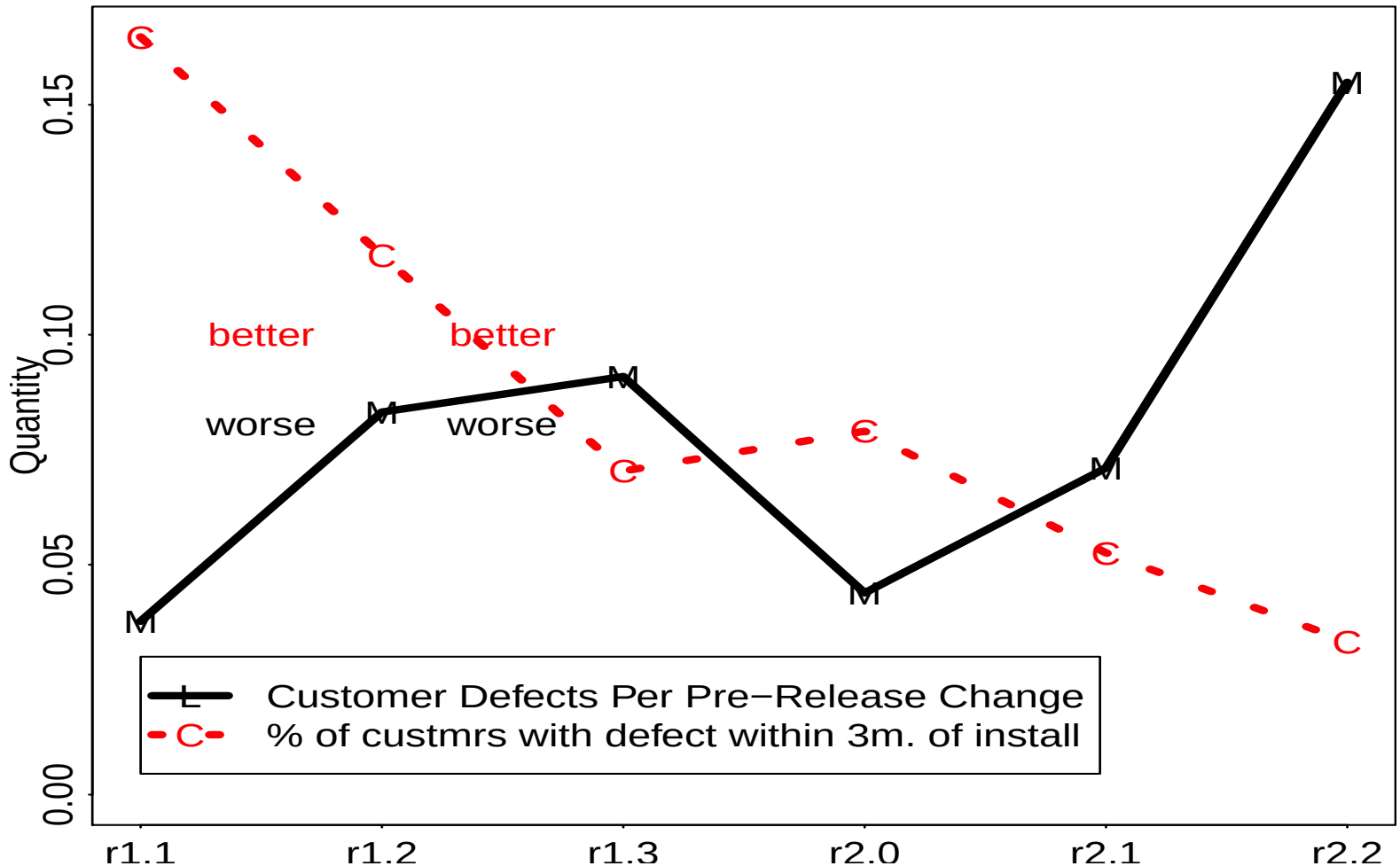
Testing Hypothesis 1: Defect Density Reflects Customer Quality



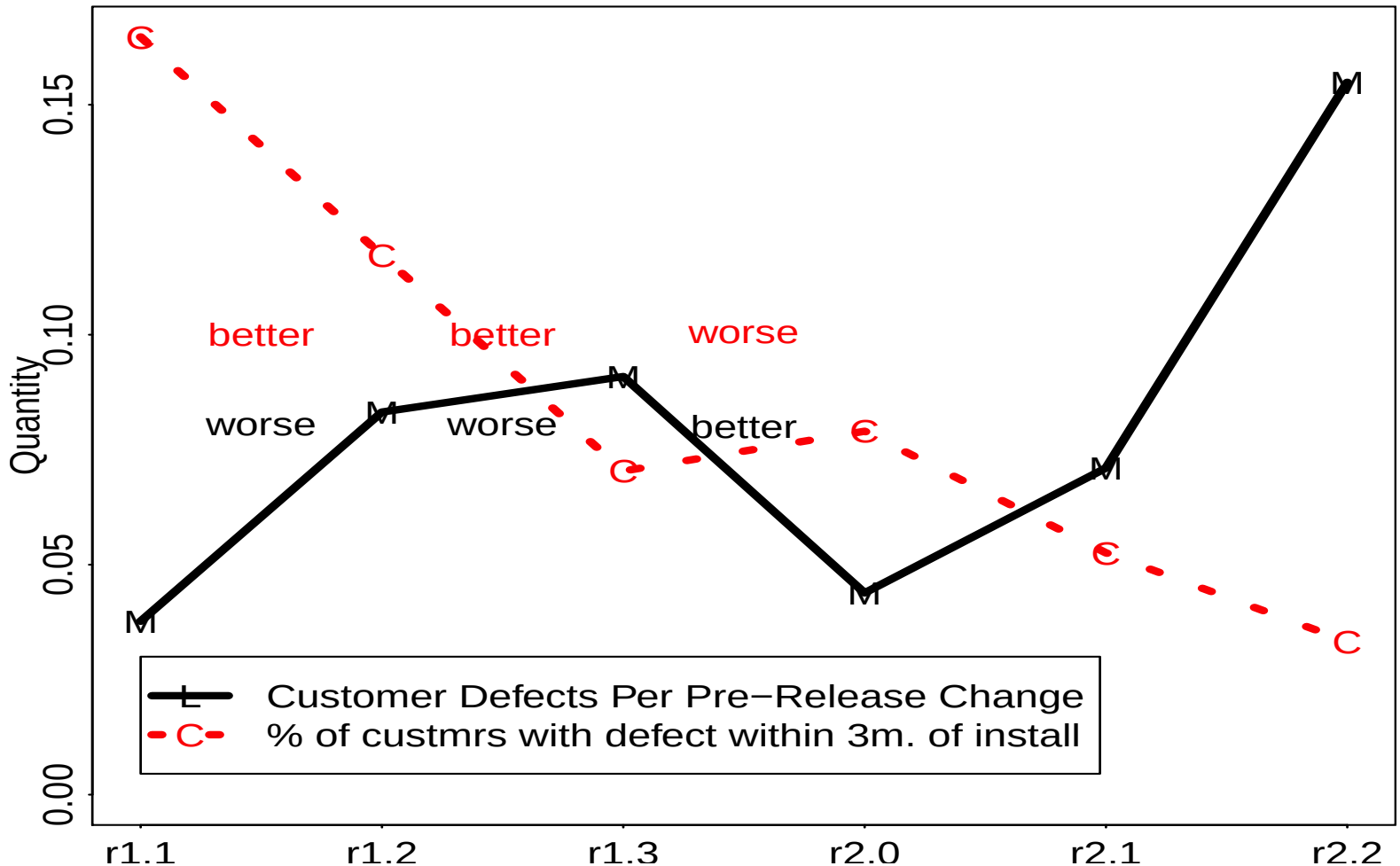
Testing Hypothesis 1: Defect Density Reflects Customer Quality



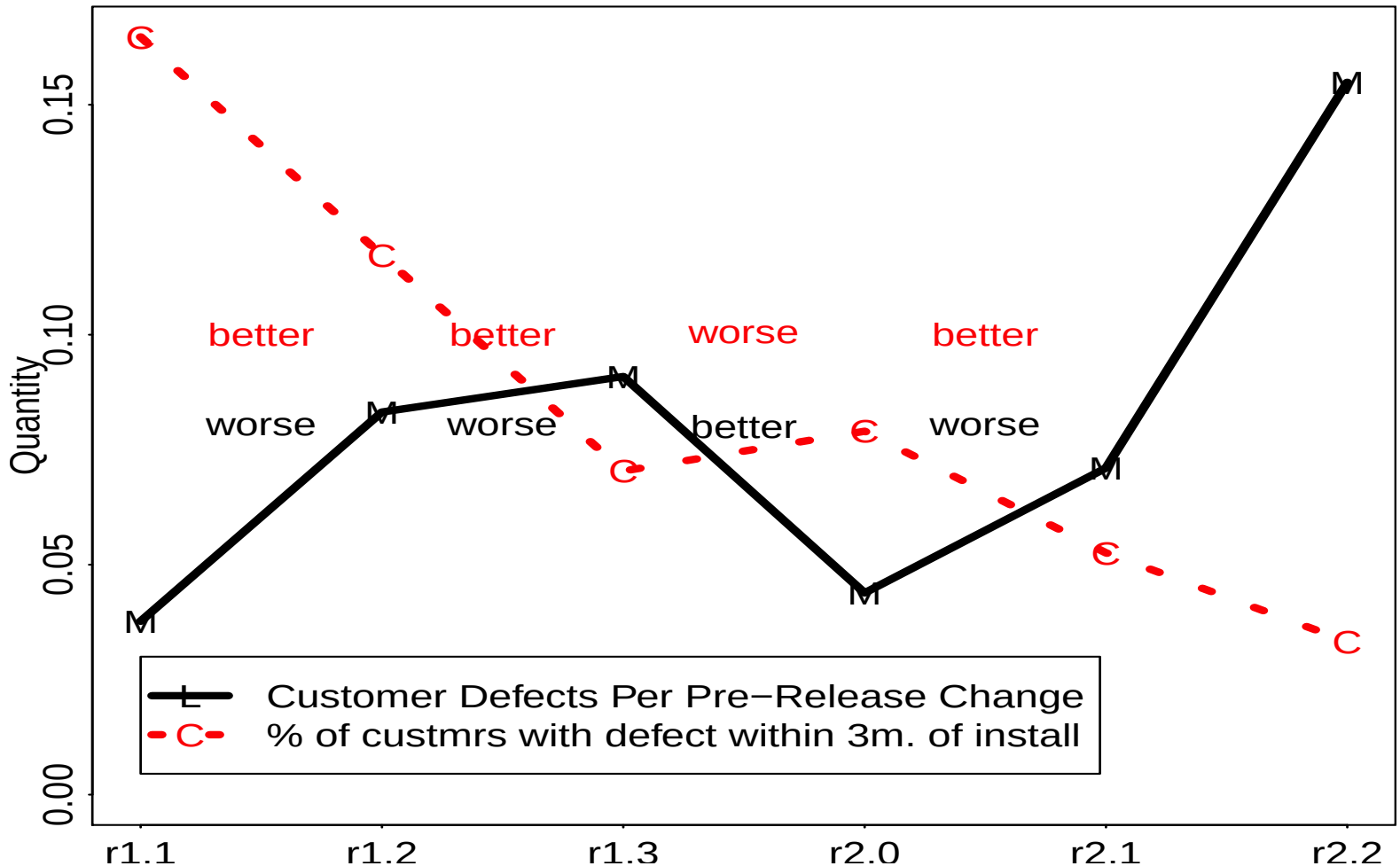
Testing Hypothesis 1: Defect Density Reflects Customer Quality



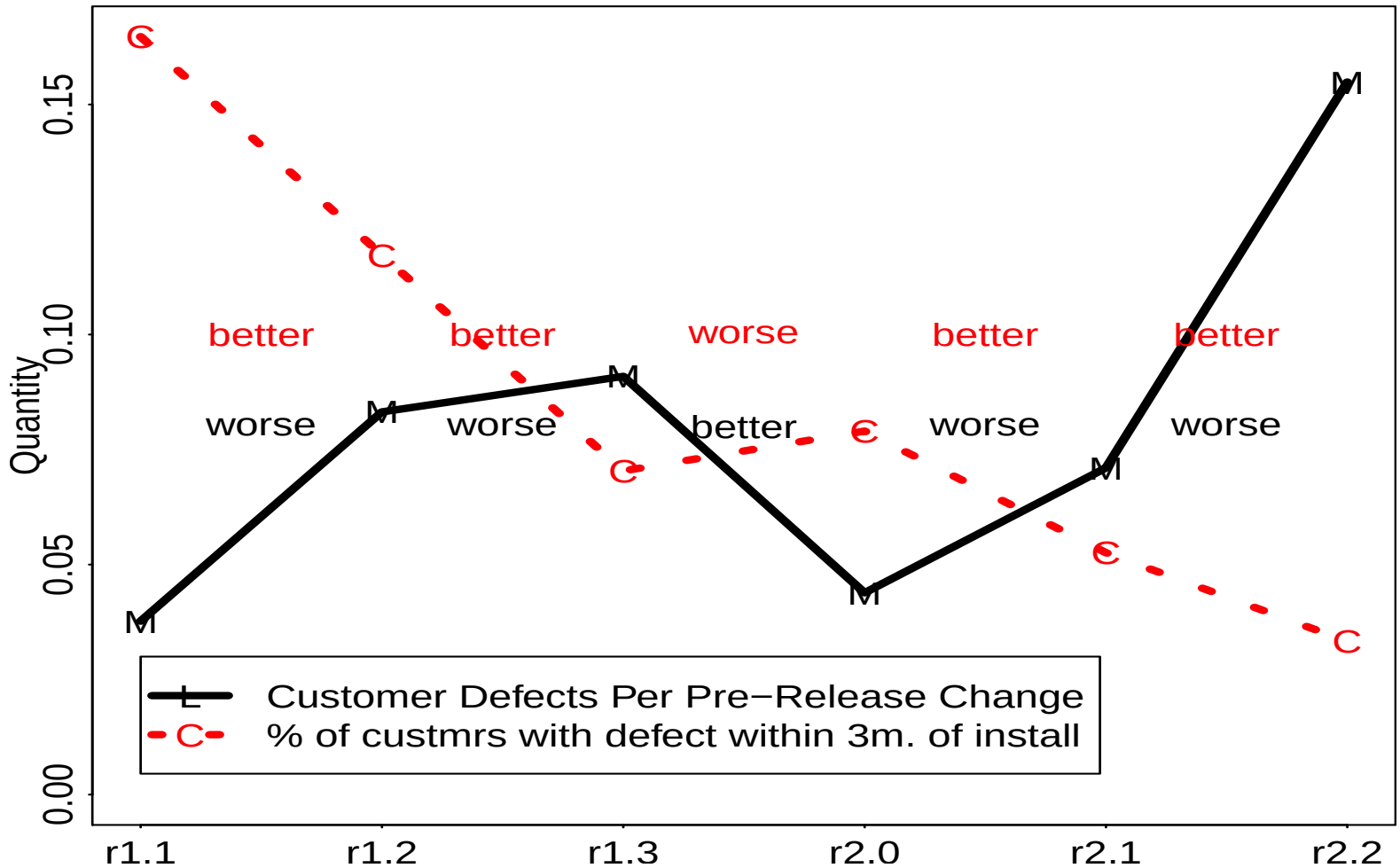
Testing Hypothesis 1: Defect Density Reflects Customer Quality



Testing Hypothesis 1: Defect Density Reflects Customer Quality

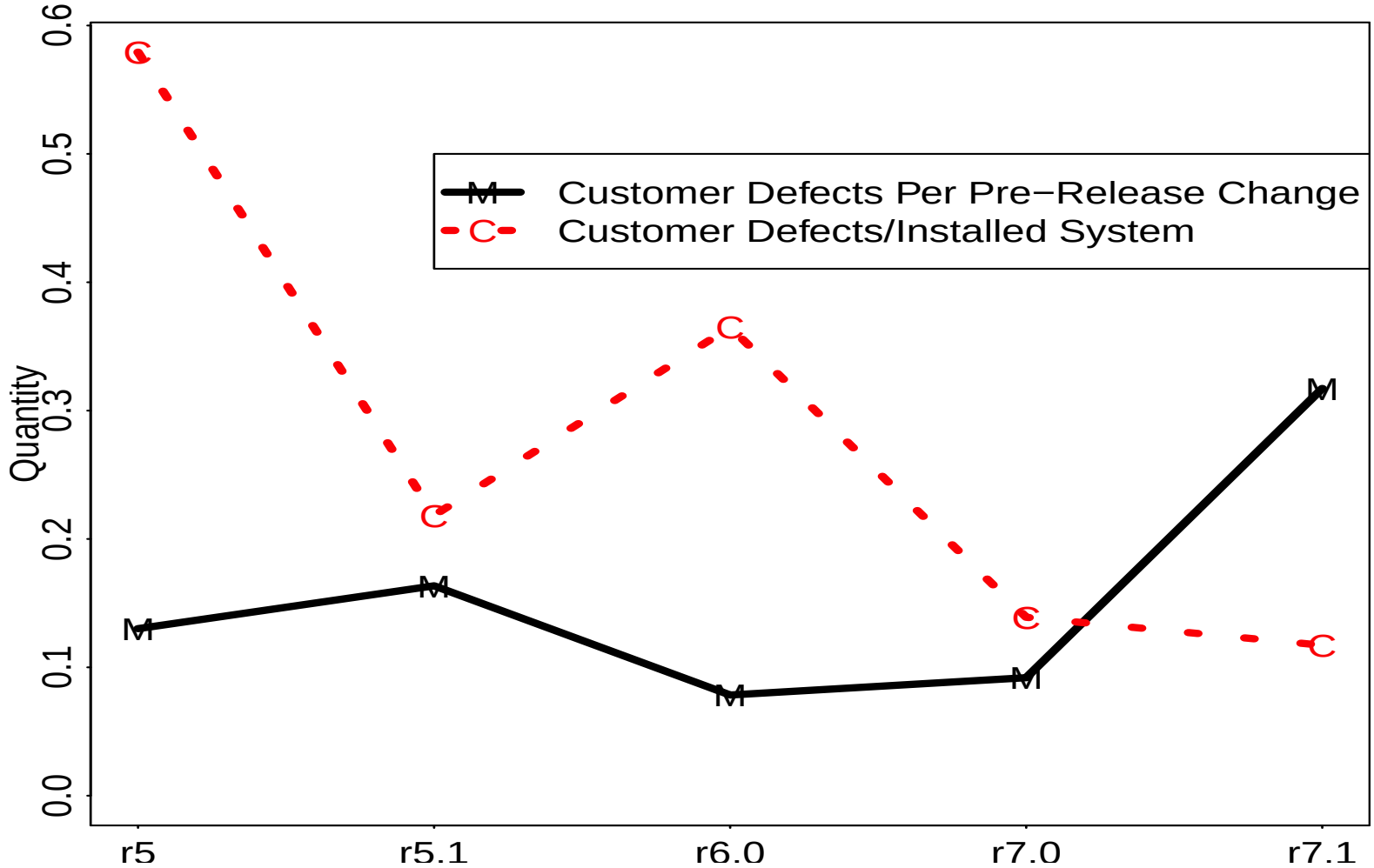


Testing Hypothesis 1: Defect Density Reflects Customer Quality

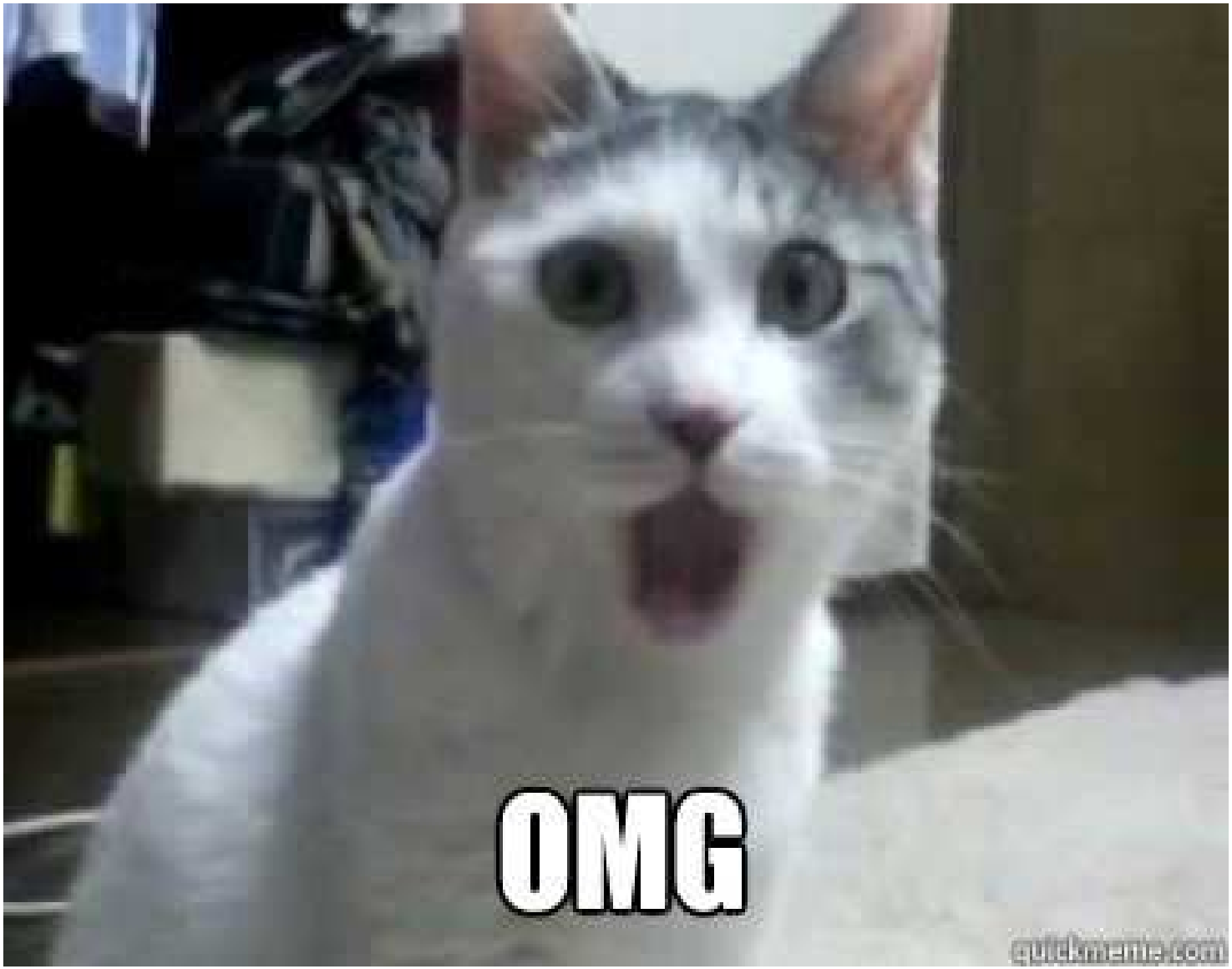


Perfect anti-correlation?!

Trying Another Product



Perfect anti-correlation again?!



Why customers like high defect density?

Why customers like high defect density?

Why customers like high defect density?

Customers don't care about defect density

- Most customers try to avoid bugs
  - By not jumping to a major dot zero release
  - By not installing immediately when new release is available

Software salesmen don't care about defect density

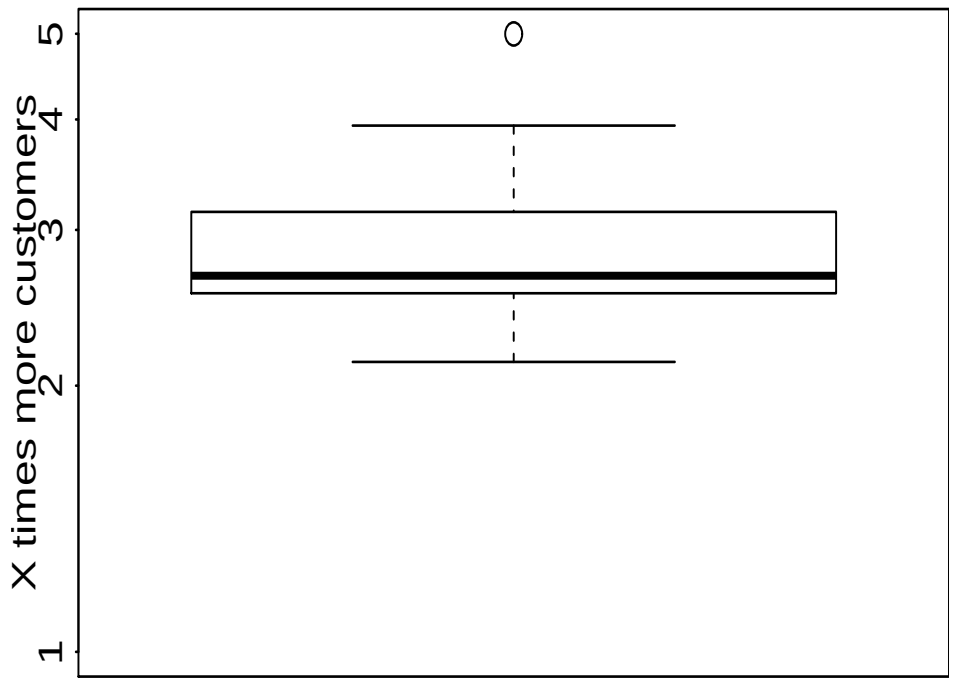
- They want their customers to avoid bugs
  - By warning about products that are likely to cause problems

Software support people don't care about defect density

- They want their customers to report as few problems as possible
  - By delaying wide installation of new releases

Lemma 1: Major Releases Have Few Customers

Minor releases have two to five times more customers



Note: based on 38 major and 49 minor releases in 22 products

Commandment 3
Thou Shell Have a Constant Rate of Customer Issues

Mechanism

- The only thing customers like less than a Bug is

Commandment 3
Thou Shell Have a Constant Rate of Customer Issues

Mechanism

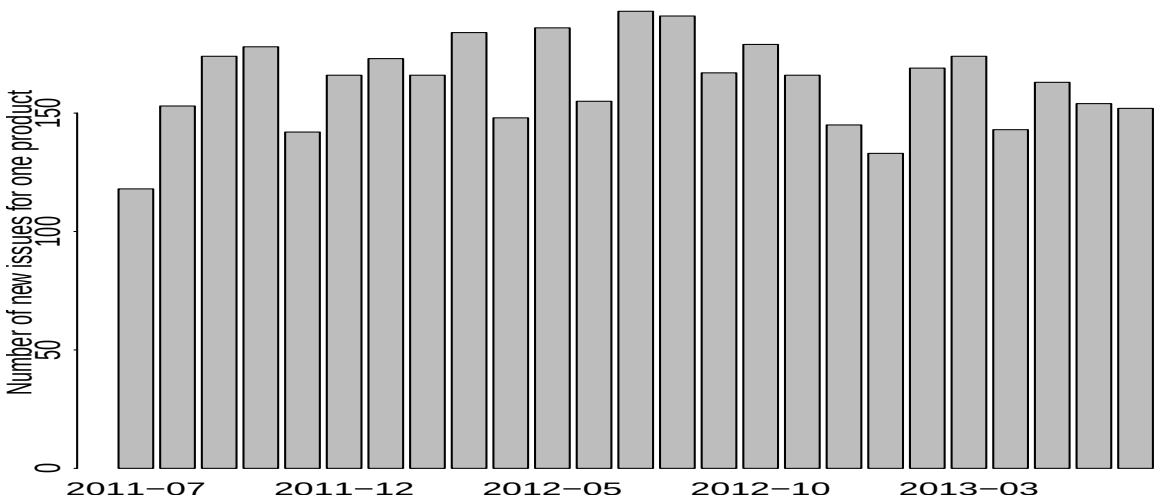
- The only thing customers like less than a Bug is
  - The bug that does not get fixed for a long time

Commandment 3
Thou Shell Have a Constant Rate of Customer Issues

Mechanism

- The only thing customers like less than a Bug is
  - The bug that does not get fixed for a long time
- Team handling customer issues can not expand and collapse instantaneously and has limited throughput

Evidence



Monthly numbers of new customer issues is relatively **constant**

Law of Minor Release

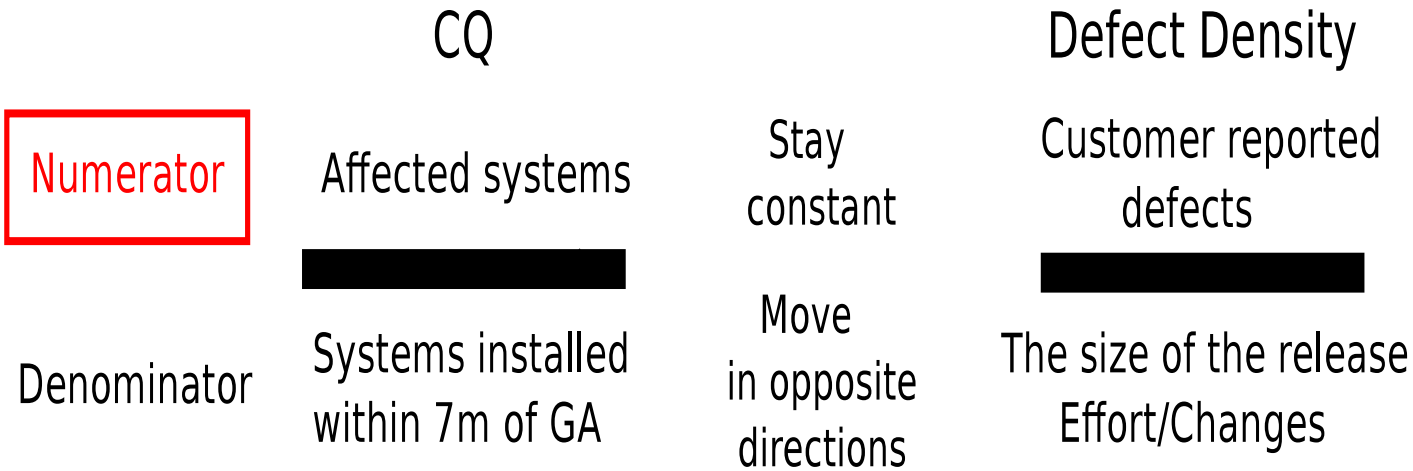
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

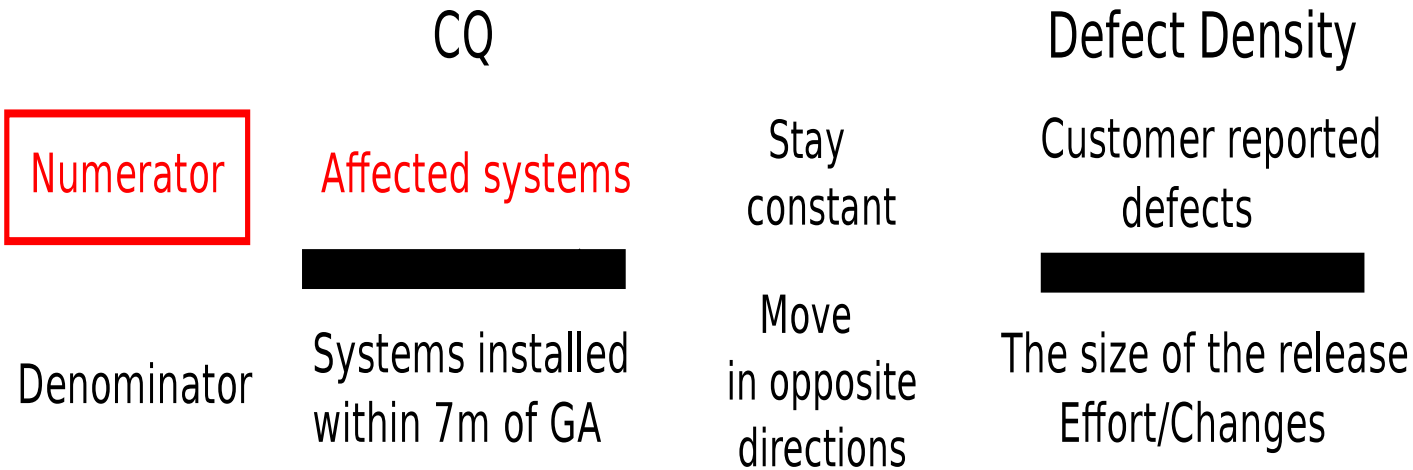
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

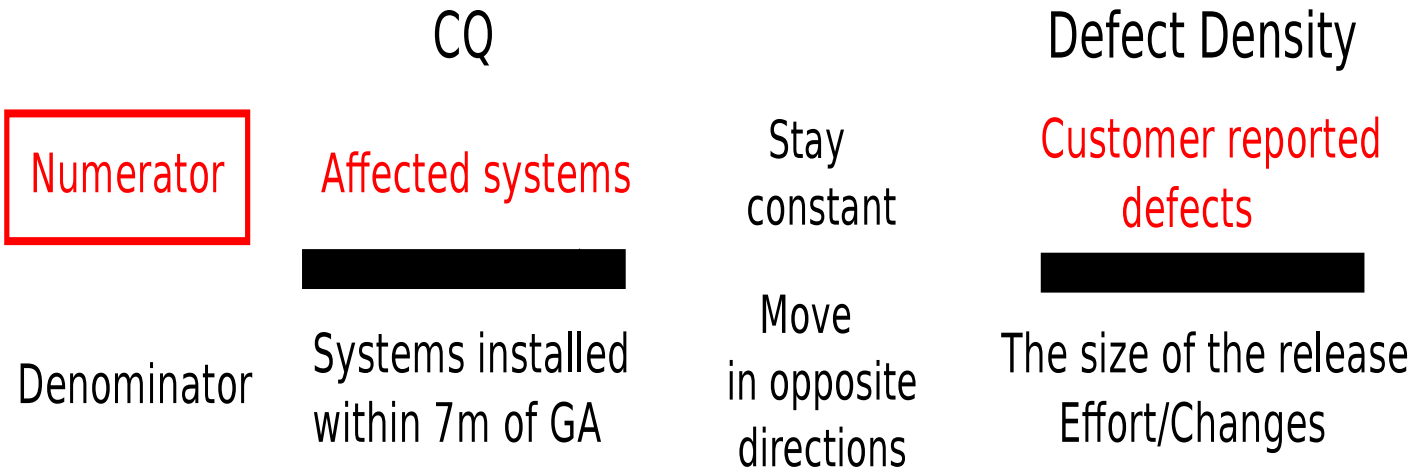
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

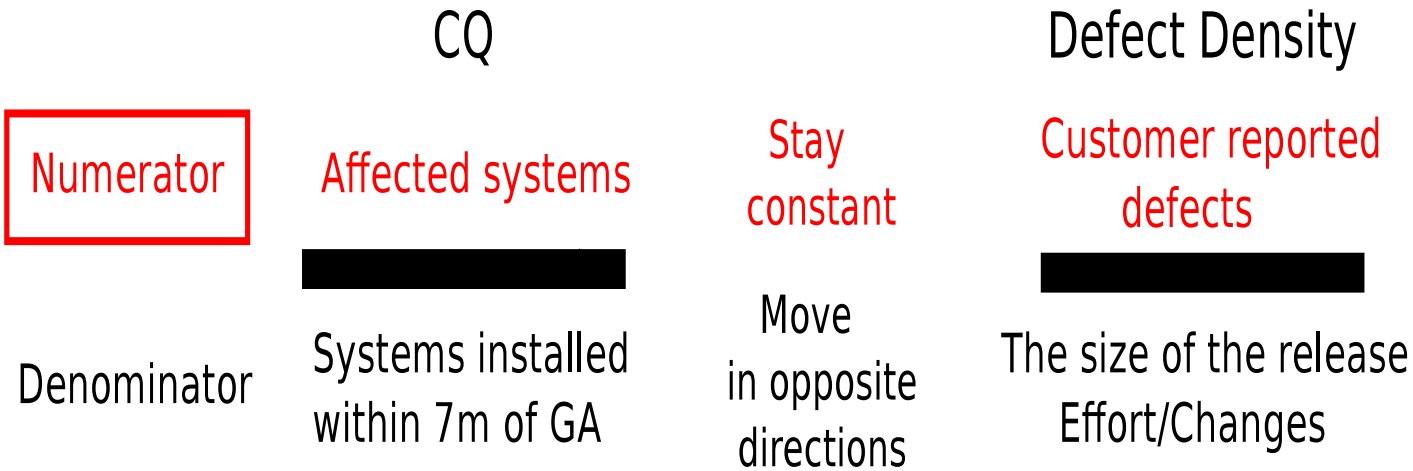
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

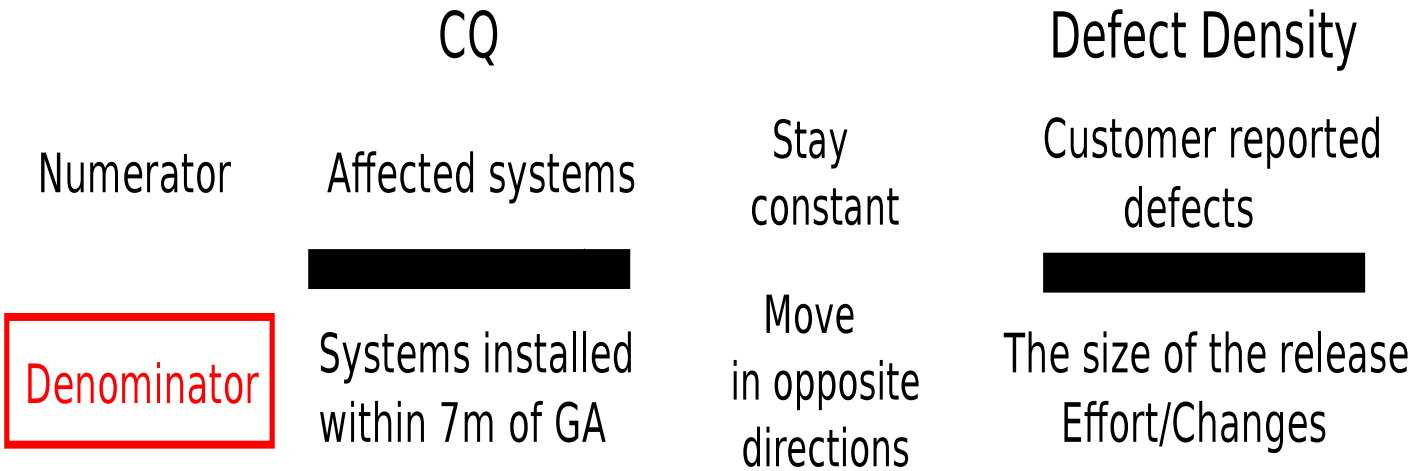
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

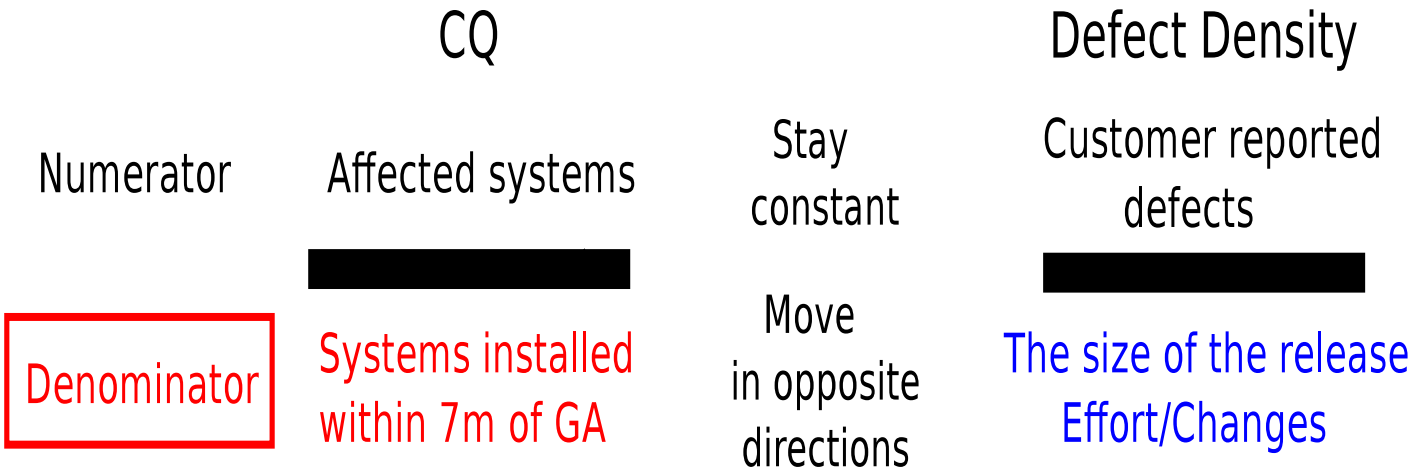
Formulation

`Minor releases have high defect density but low chances that any given customer will observe a defect`

Definition

Major Releases Have More Code Change

Mechanism



Law of Minor Release

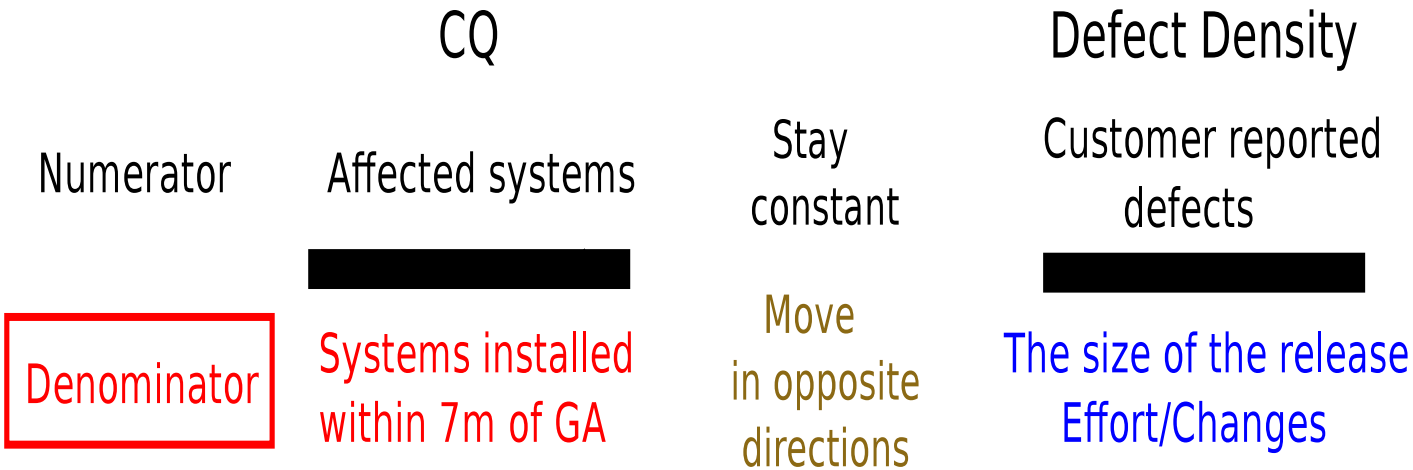
Formulation

Minor releases have high defect density but low chances that any given customer will observe a defect

Definition

Major Releases Have More Code Change

Mechanism



Discussion

- There exist Laws of Software Evolution, but
  - Focus on short-term, repeating relationships with a clear mechanism
  - Look outside SW cave to observe them
  - Chose practical questions
- Practice hints
  - Development process view does not represent customer views
  - Maintenance — the most important quality improvement activity

References I

Audris Mockus, David M. Weiss, and Ping Zhang.

Understanding and predicting effort in software projects.

In *2003 International Conference on Software Engineering*, pages 274–284, Portland, Oregon, May 3-10 2003. ACM Press.

Audris Mockus, Ping Zhang, and Paul Li.

Drivers for customer perceived software quality.

In *ICSE 2005*, pages 225–233, St Louis, Missouri, May 2005. ACM Press.

Abstract I

The traditional view of software quality focuses on counting bugs — issues that are observed and reported by users and implemented as changes to the source code. Fewer bugs intuitively (and obviously) imply higher software quality. This hasty conclusion, however, ignores complex equilibrium resulting from actions of different groups of participants in software production: developers, users, support, and sales. For example, users improve software quality by discovering and reporting defects that are too costly to be discovered otherwise. As new functionality is delivered in major releases, quality conscious users often stay on the sidelines until a second minor release delivers properly working features, bug fixes, and stability improvements. The major releases, being of lower quality, have fewer users and, consequently, fewer bugs. I will discuss several fundamental laws of software production system that explain this paradox in a quantitative manner. Each law has a clear mechanism of action, is grounded in resource and physical constraints, and is empirically validated. The laws provide guidelines on how to measure, understand, and improve quality of software.

Audris Mockus
Avaya Labs Research
233 Mt. Airy Road
Basking Ridge, NJ 07920
ph: +1 908 696 5608, fax:+1 908 696 5402
http://mockus.org, mailto:audris@mockus.org

Audris Mockus is interested in quantifying, modeling, and improving software development. He designs data mining methods to summarize and augment software change data, interactive visualization techniques to inspect, present, and control the development process, and statistical models and optimization techniques to understand the relationships among people, organizations, and characteristics of a software product. Audris Mockus received B.S. and M.S. in Applied Mathematics from Moscow Institute of Physics and Technology in 1988. In 1991 he received M.S. and in 1994 he received Ph.D. in Statistics from Carnegie Mellon University. He works at Avaya Labs Research. Previously he worked in the Software Production Research Department of Bell Labs.