Bug Prediction

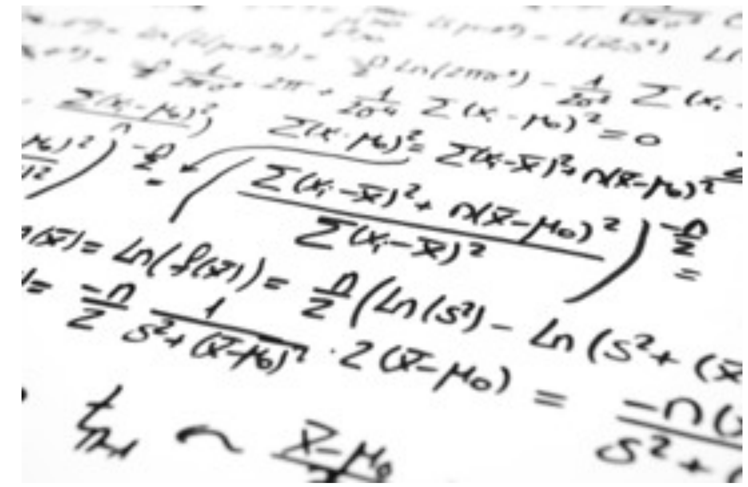
SW-Wartung & Evolution

Emanuel Giger

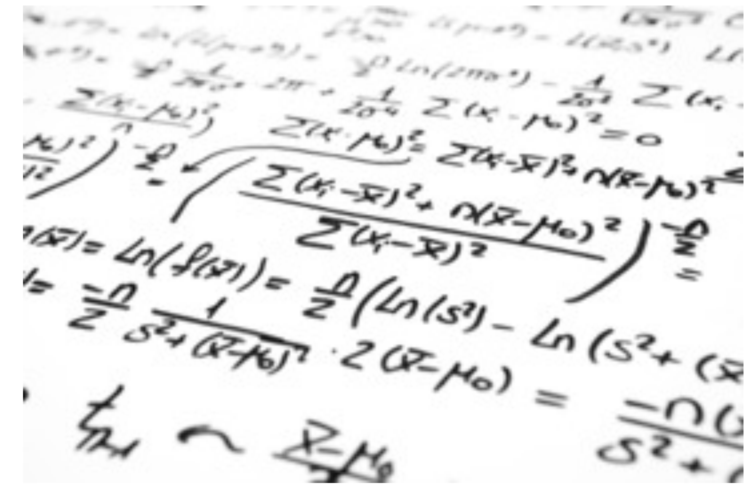
University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

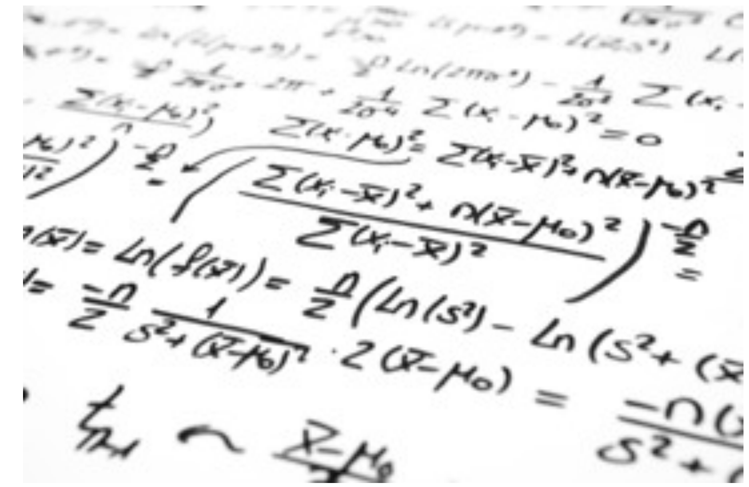
Software has Bugs!

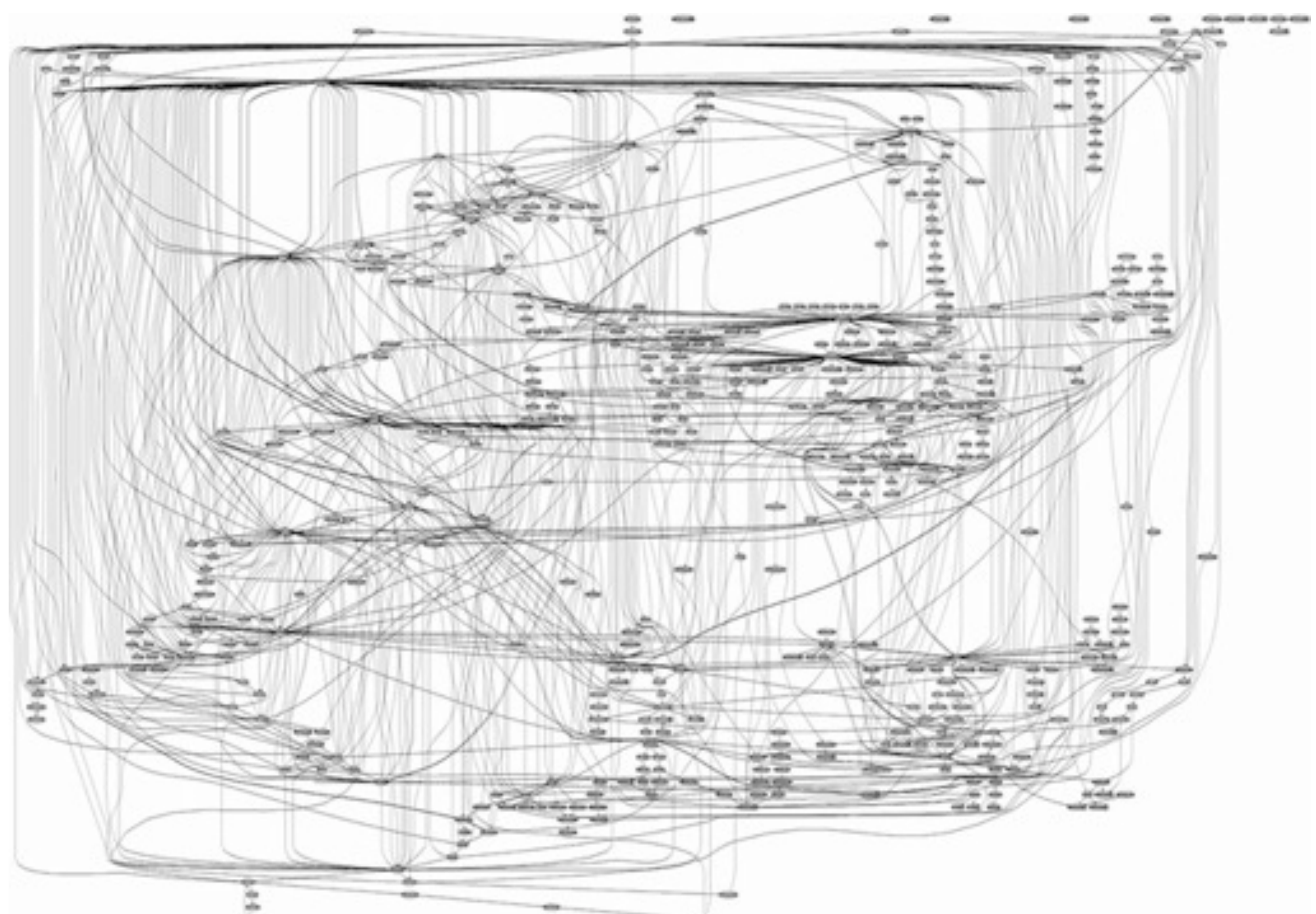


Software has Bugs!

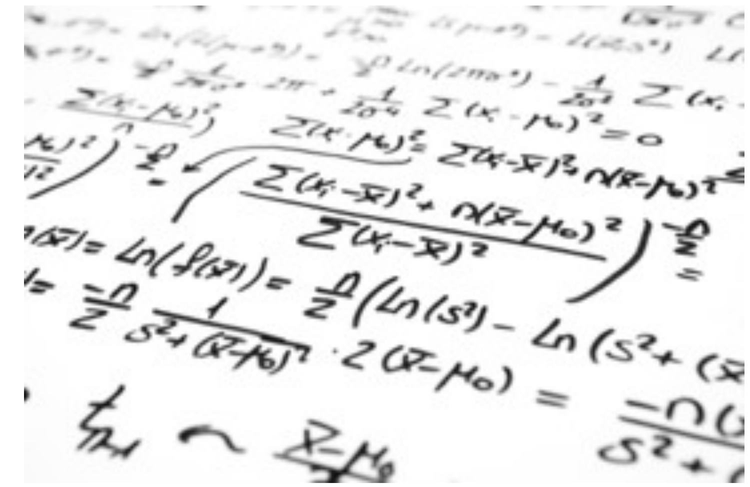


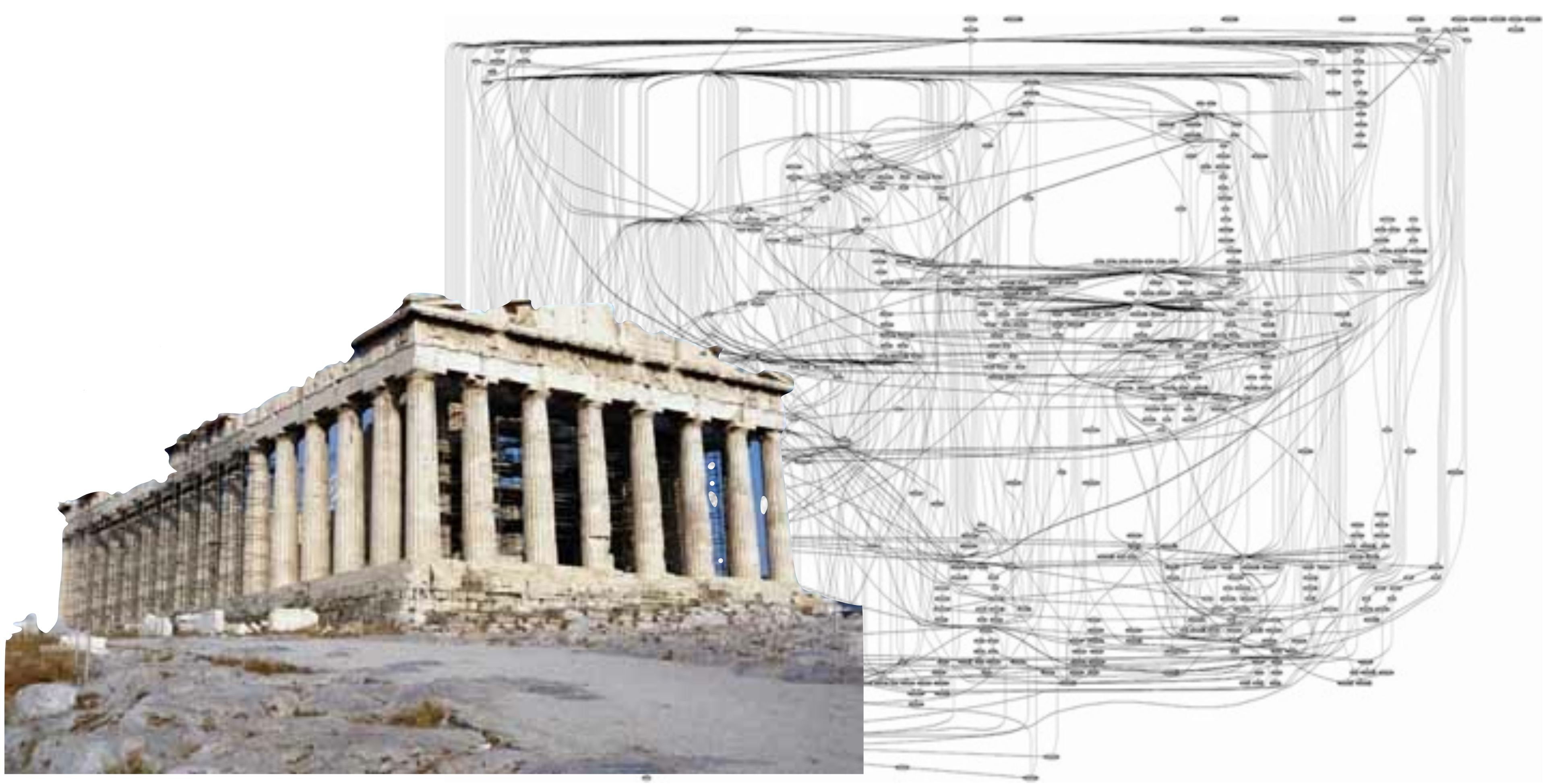
Software has Bugs!



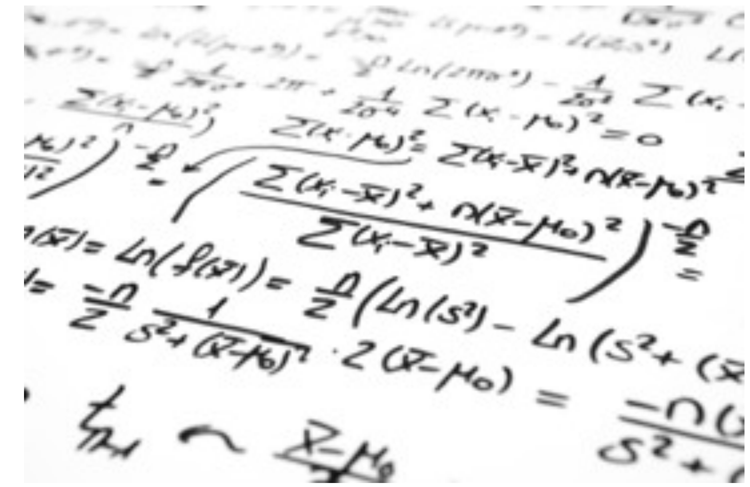


Software has Bugs!



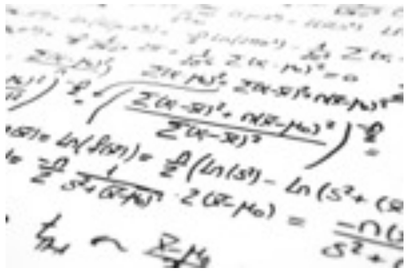
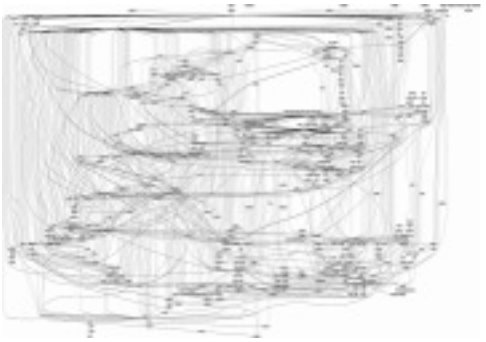


Software has Bugs!

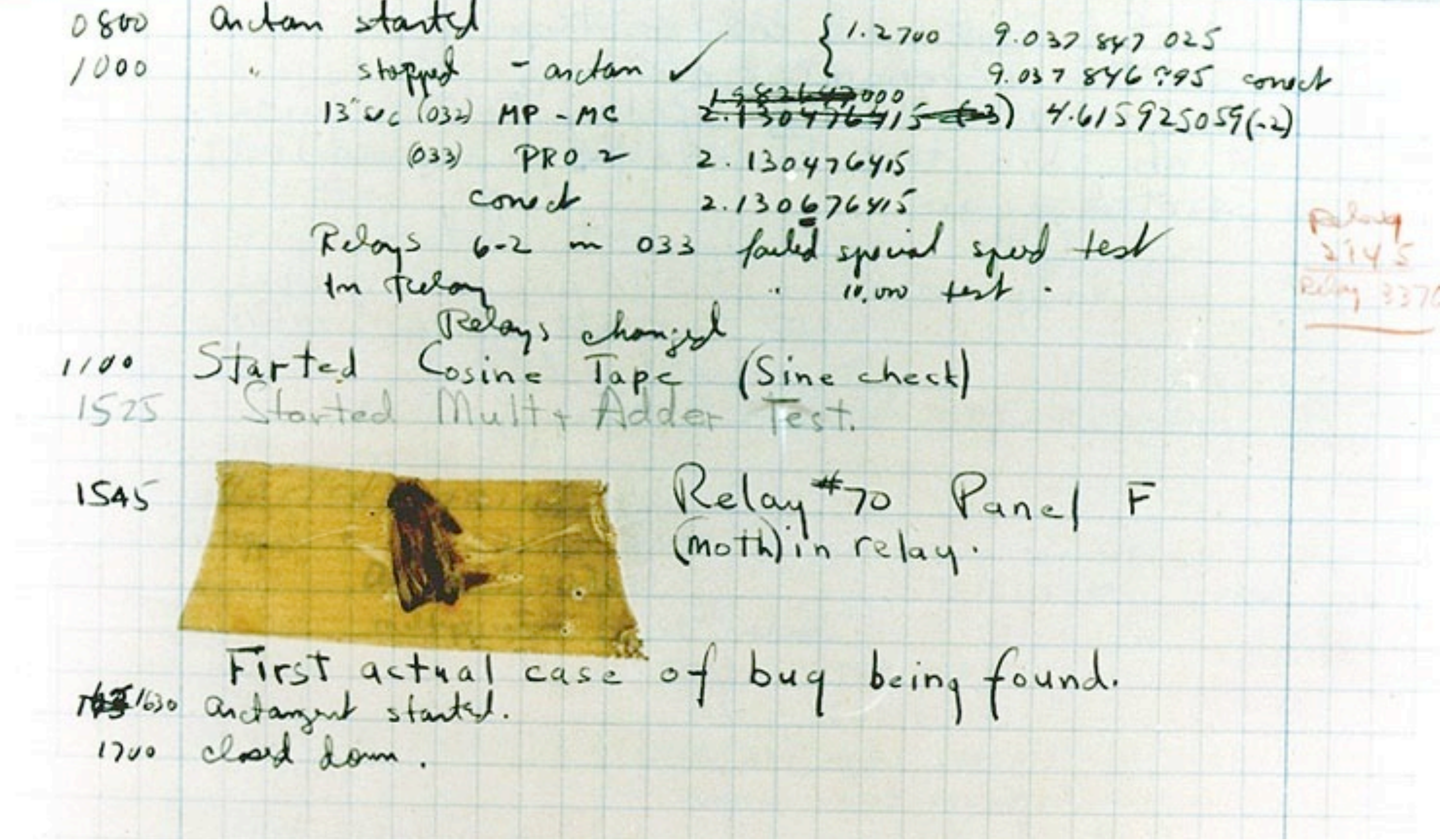


Software has Bugs!



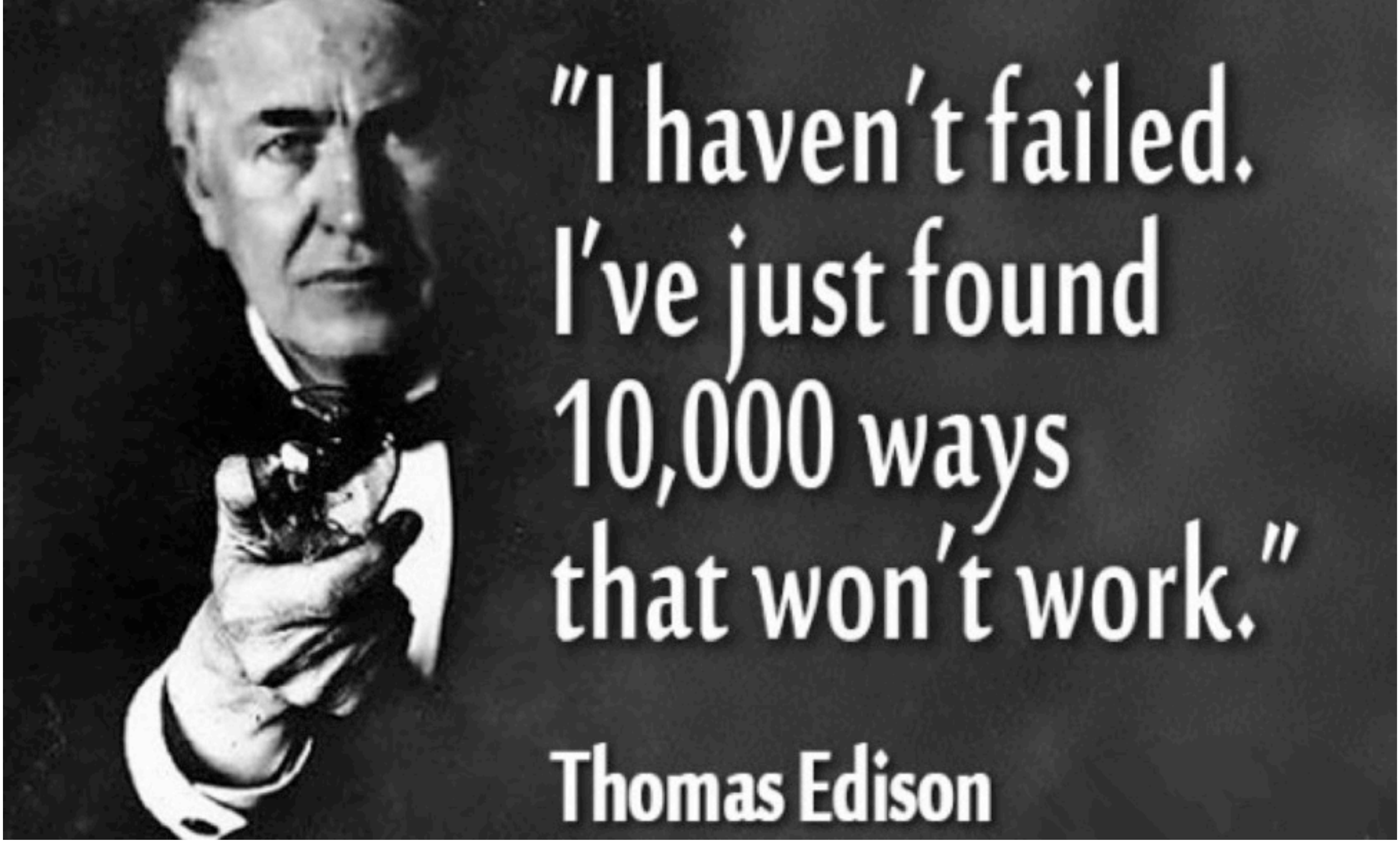




Bugs! Bugs! Bugs! Bugs! Bugs!



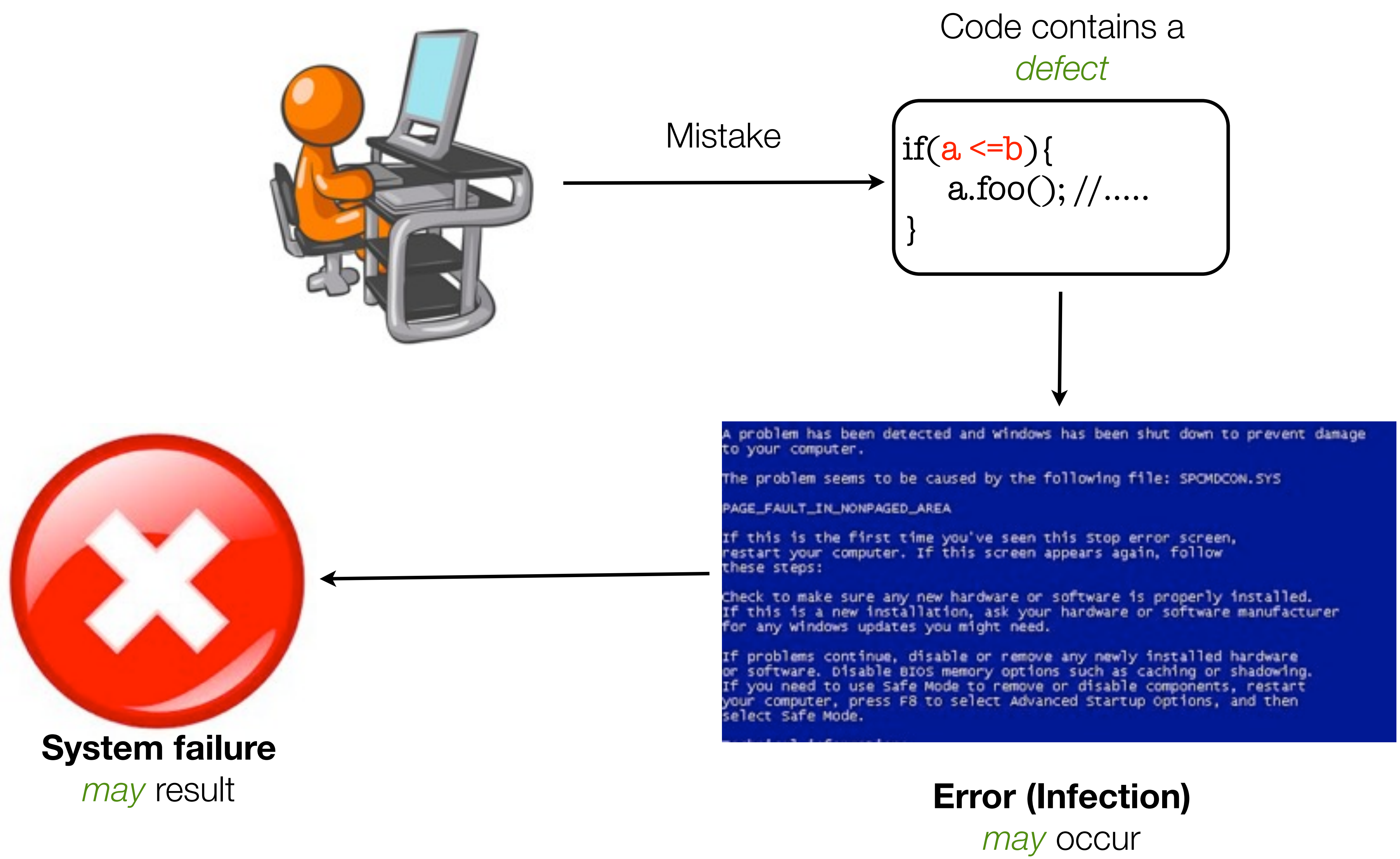
First case of a bug

Anecdotal story from 1947 related to the Mark II computer



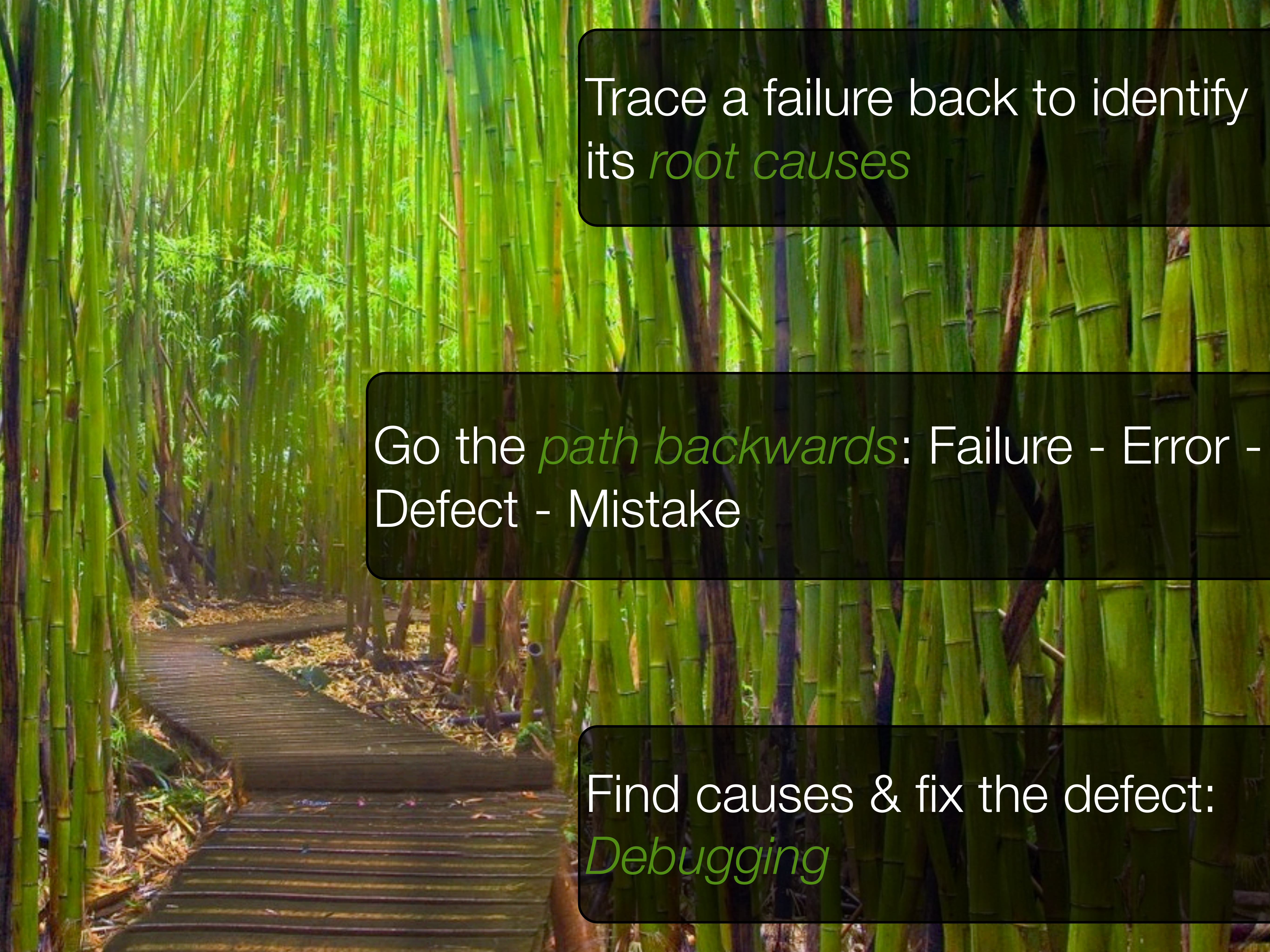
"...then that 'Bugs' - as such little faults and difficulties are called - show themselves..."

Noise in communication infrastructure



Why are bugs in our software?

The Path of a Bug



Trace a failure back to identify its *root causes*

Go the *path backwards*: Failure - Error - Defect - Mistake

Find causes & fix the defect: *Debugging*

Stages of Debugging

- Locate cause
- Find a solution to fix it
- Implement to solution
- Execute tests to verify the correctness of the fix

Bug Facts

- *"Software Errors Cost U.S. Economy $59.5 Billion Annually"*[1]
- ~36% of the IT-Budget is spend on bug fixing[1]
- Massive power blackout in North-East US: *Race Condition*
- Therac-25 Medical Accelerator: *Race Condition*
- Ariane 5 Explosion: *Erroneous floating point conversion*

[1]2002, US National Institute of Standards & technology

[2]iX Studie 01/2006, Software-Testmanagement


Quality control: **Find** defects as early as possible
Prevent defects from being shipped to their productive environment

Quality Assurance (QA)...

...is limited by time and money

Quality Assurance (QA)...

...is limited by time and money

Spend resources with maximum efficiency!
Focus on the components that fail the most!

Defect Prediction

Identify those components of your system that are *most critical* with respect to defects

Build forecast (prediction) models to identify bug-prone parts *in advance*

Defect Prediction

Combines methods & techniques of *data mining*, *machine learning*, *statistics*

Defect Prediction

Decision Trees, Support Vector Machines,
Neural Network, Bayesian Network, ...

Crime Fighting, Richmond, VA

- 2005, Massive amount of crime data
- Data mining to connect various data sources
- Input: Crime reports, weather, traffic, sports events and paydays for large employers
- Analyzed 3 times per day
- Output: Forecast where crime was most likely to occur, crime pikes, crime patterns
- Deploy police forces efficiently in advance

Defect Prediction

Problem: Garbage In - Garbage Out

Defect Prediction Research:

What is *the best input* to build the most efficient defect prediction models?

Defect Prediction

Defect Prediction Research:

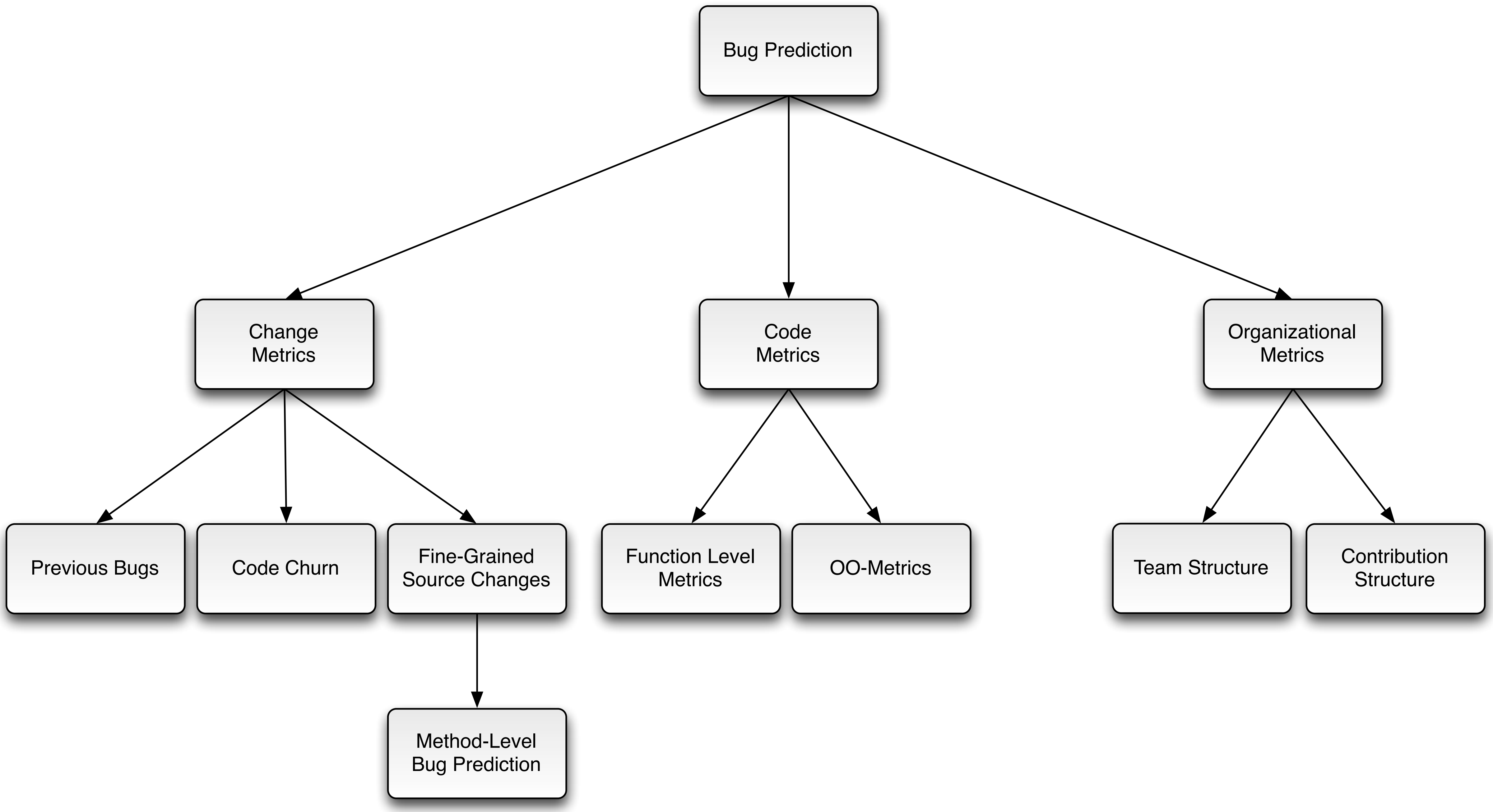
How can we *minimize* the amount of required *input data* but still get *accurate* prediction models?

Defect Prediction

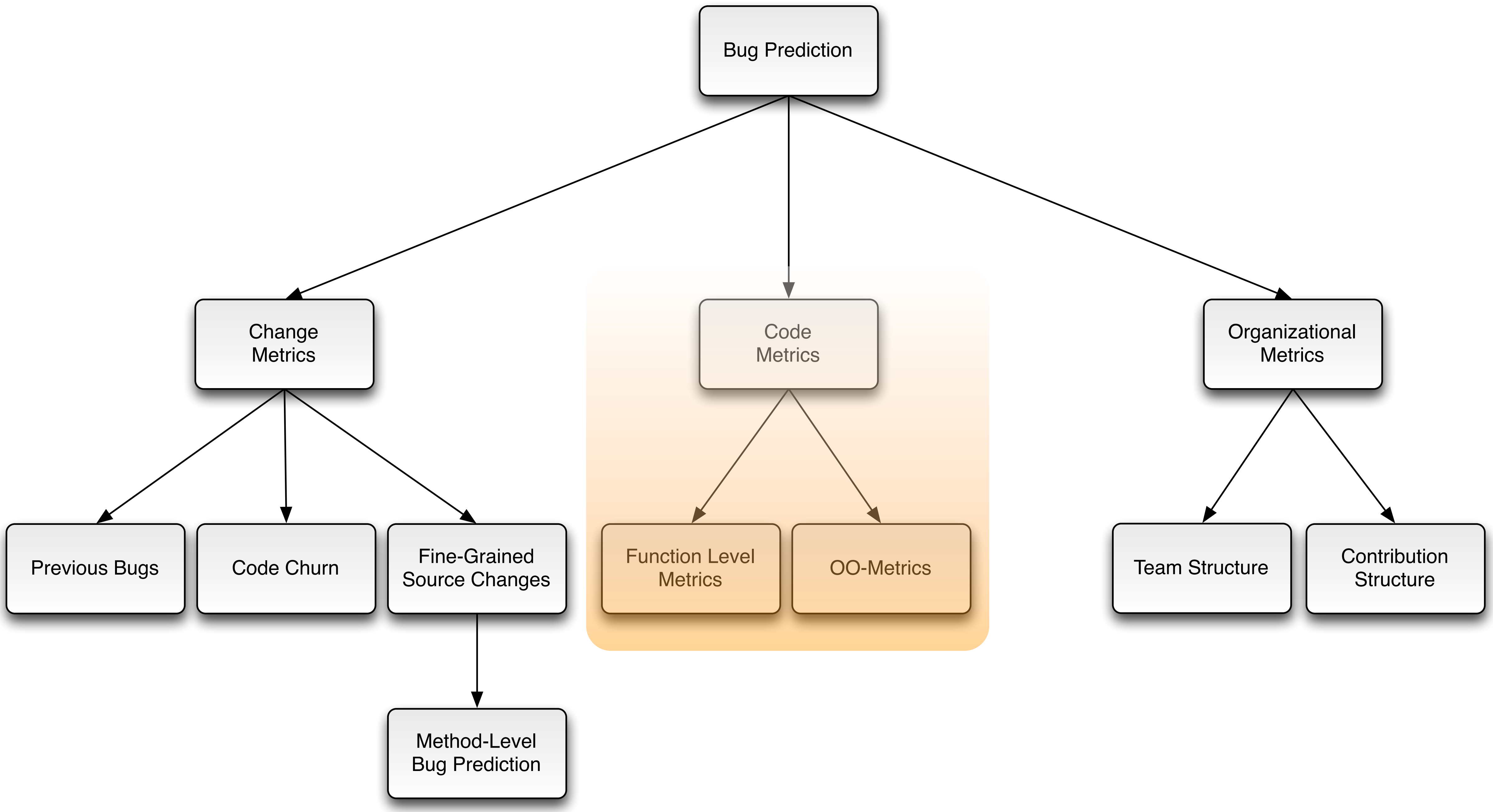
Defect Prediction Research:

How can we turn prediction models into *actionable tools* for practitioners?

Bug Prediction Models



Bug Prediction Models



Code Metrics

Directly calculated on the code itself

Different metrics to measure various aspects of the *size* and *complexity*

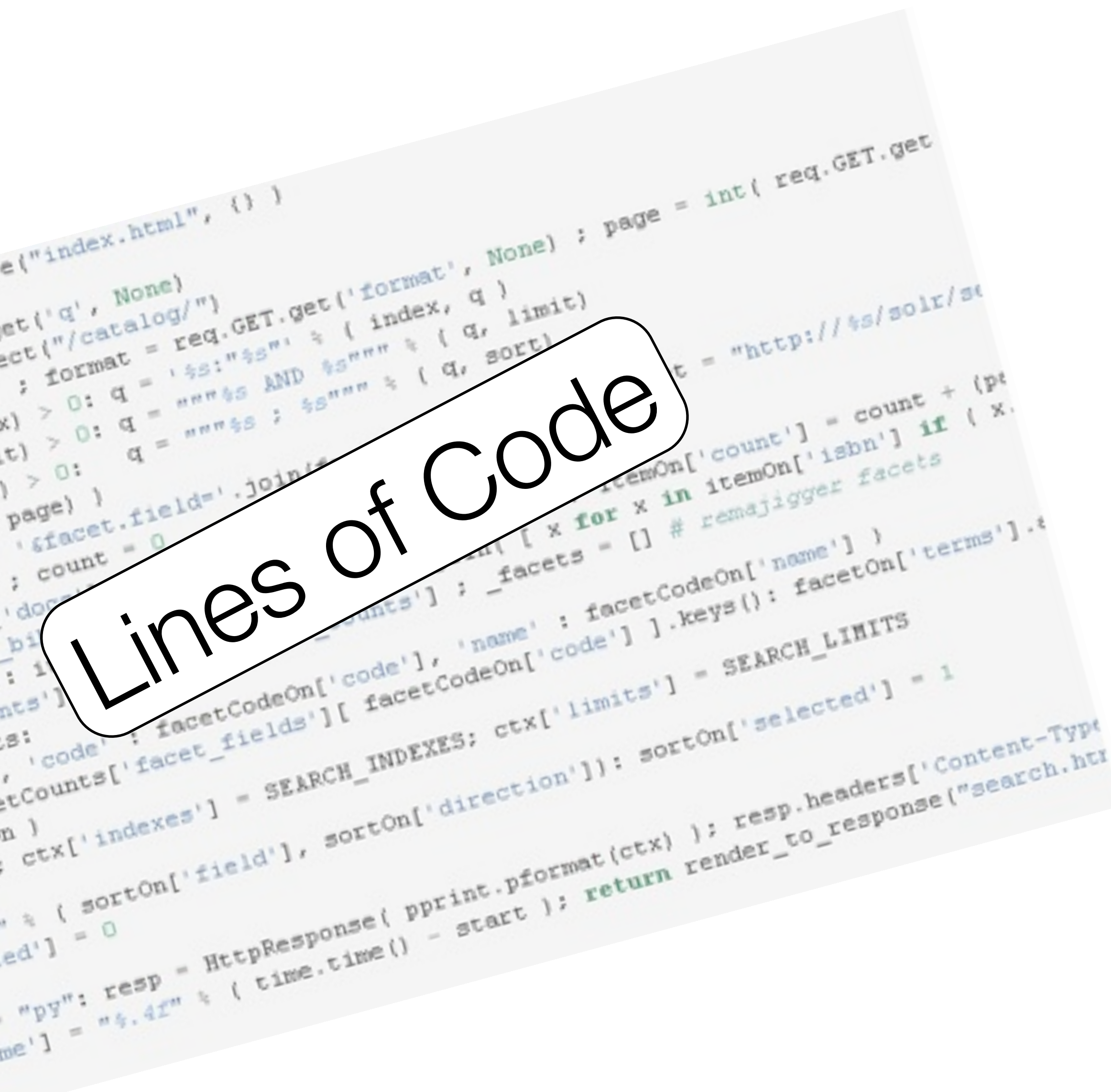
Larger and more complex modules are harder to understand and change

Code Metrics

Directly calculated on the code itself

Different metrics to measure various aspects of the *size* and *complexity*

Larger and more complex modules are harder to understand and change

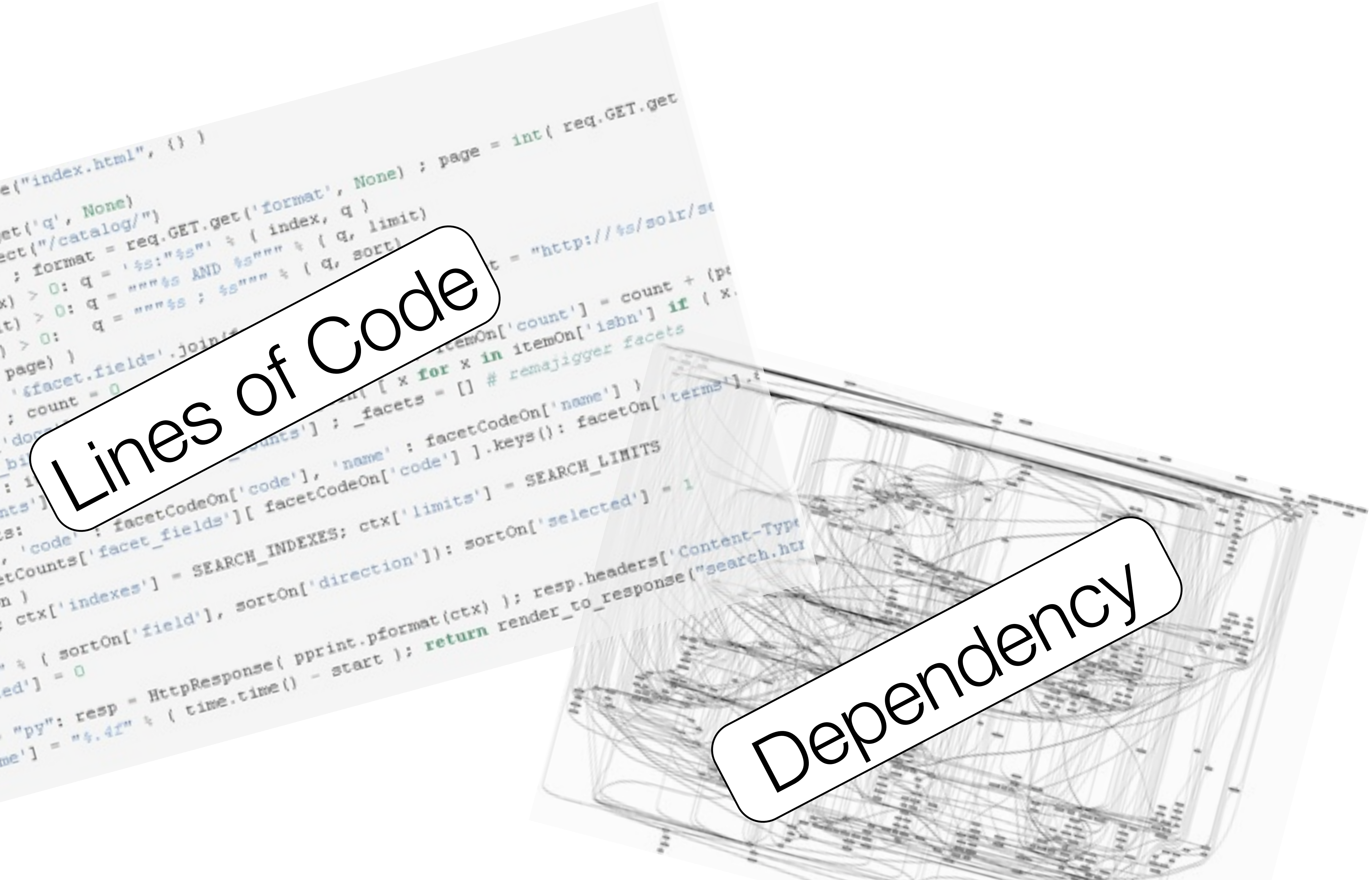


Code Metrics

Directly calculated on the code itself

Different metrics to measure various aspects of the *size* and *complexity*

Larger and more complex modules are harder to understand and change

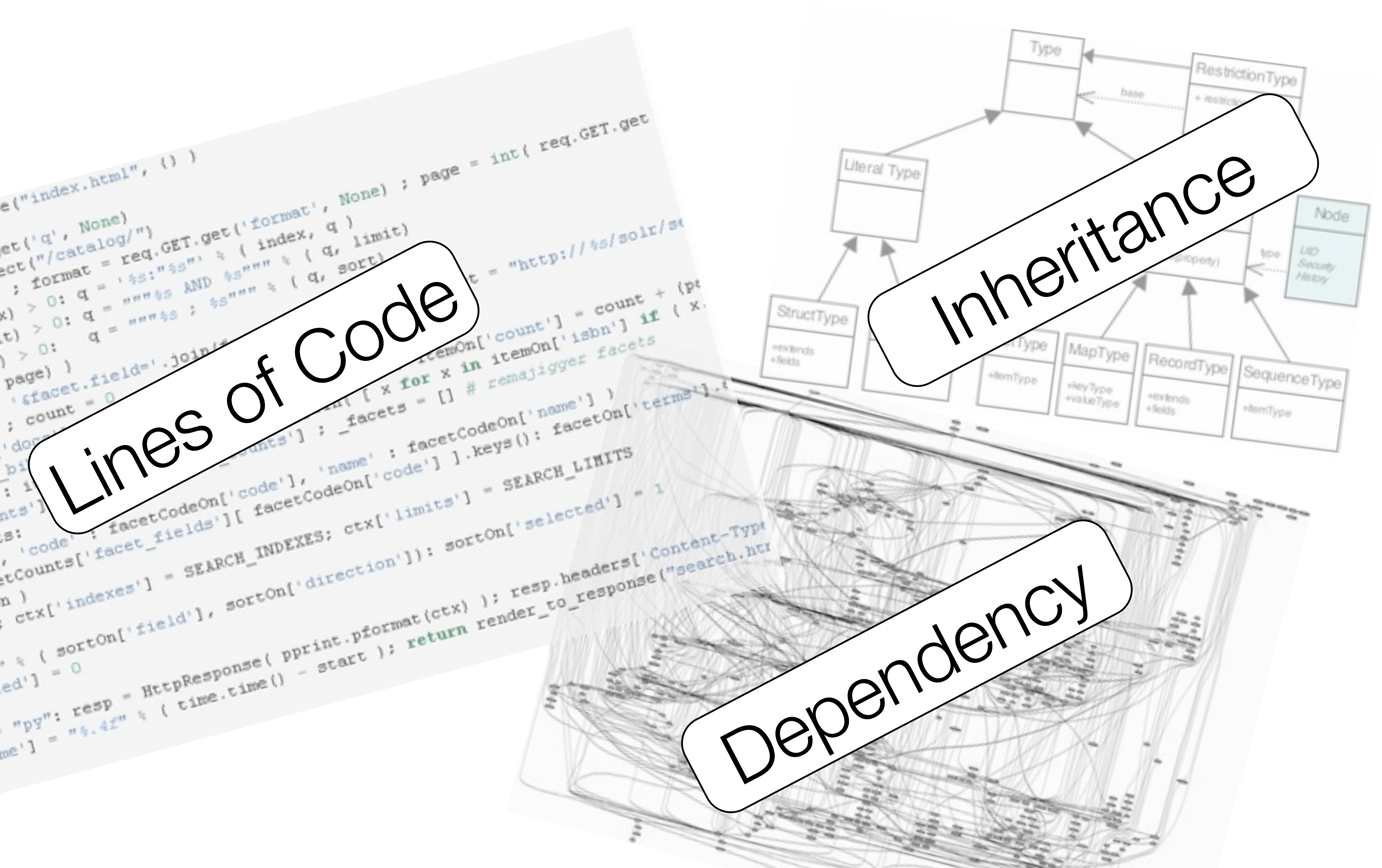


Code Metrics

Directly calculated on the code itself

Different metrics to measure various aspects of the *size* and *complexity*

Larger and more complex modules are harder to understand and change

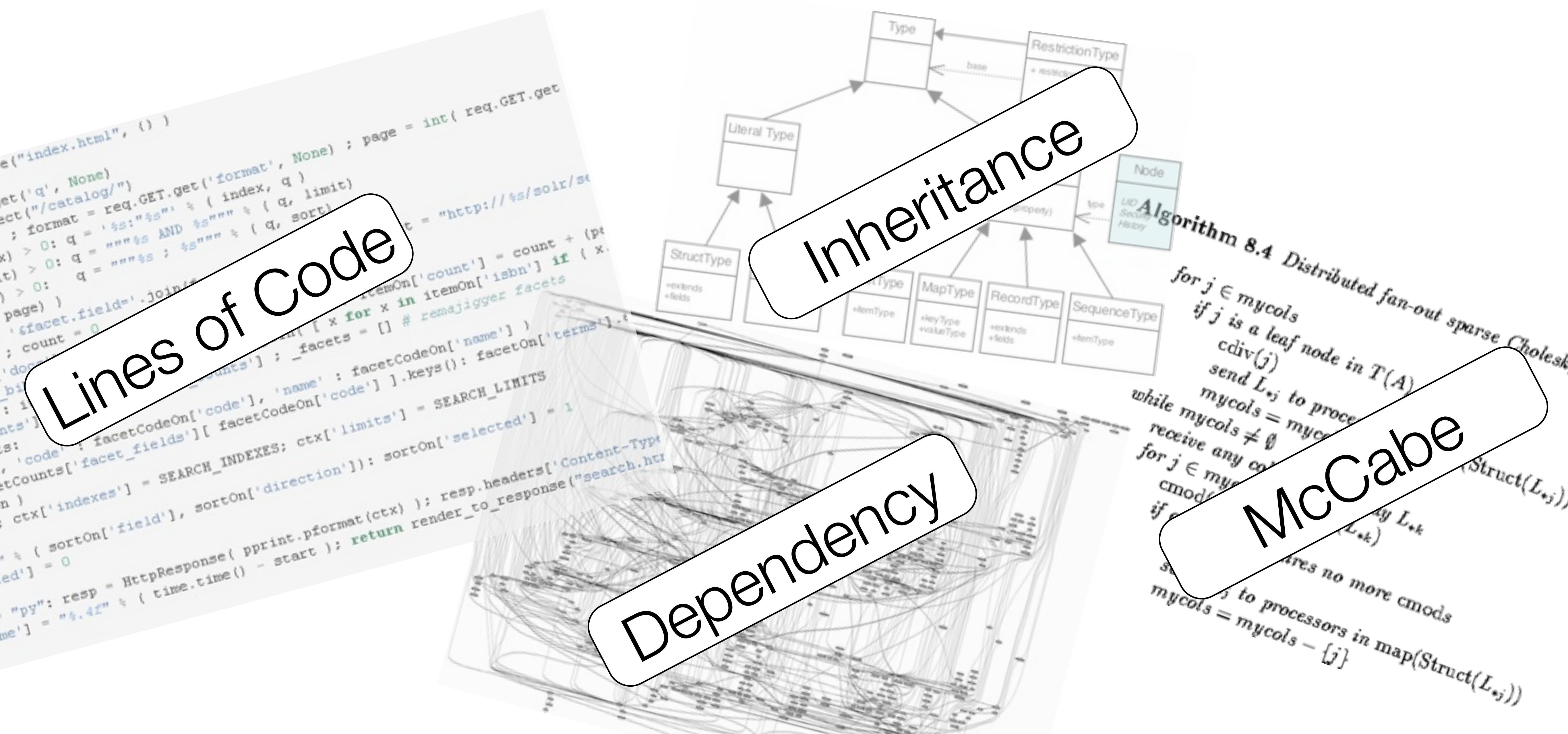


Code Metrics

Directly calculated on the code itself

Different metrics to measure various aspects of the *size* and *complexity*

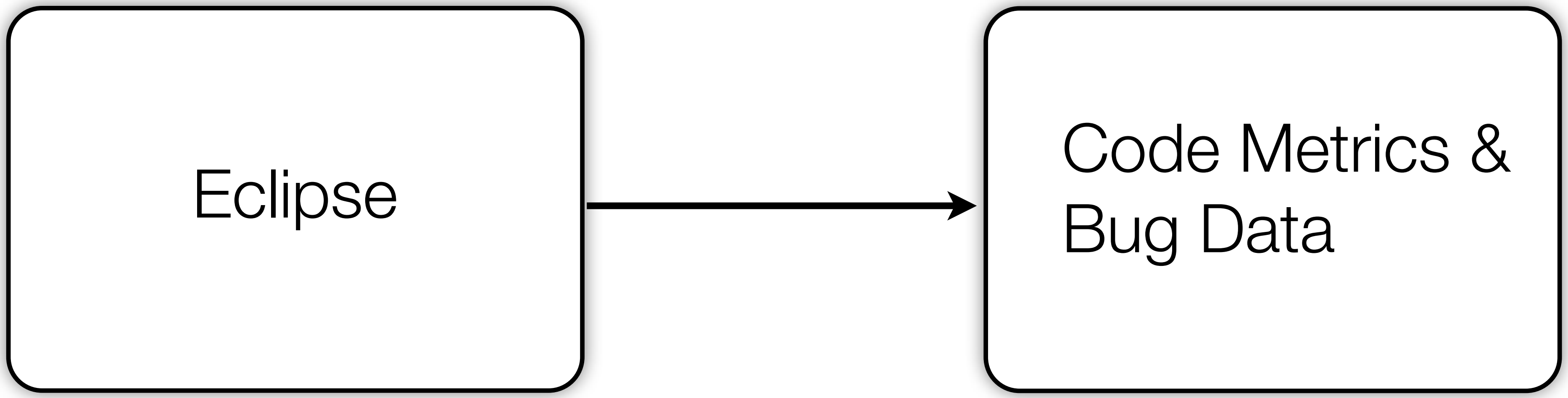
Larger and more complex modules are harder to understand and change



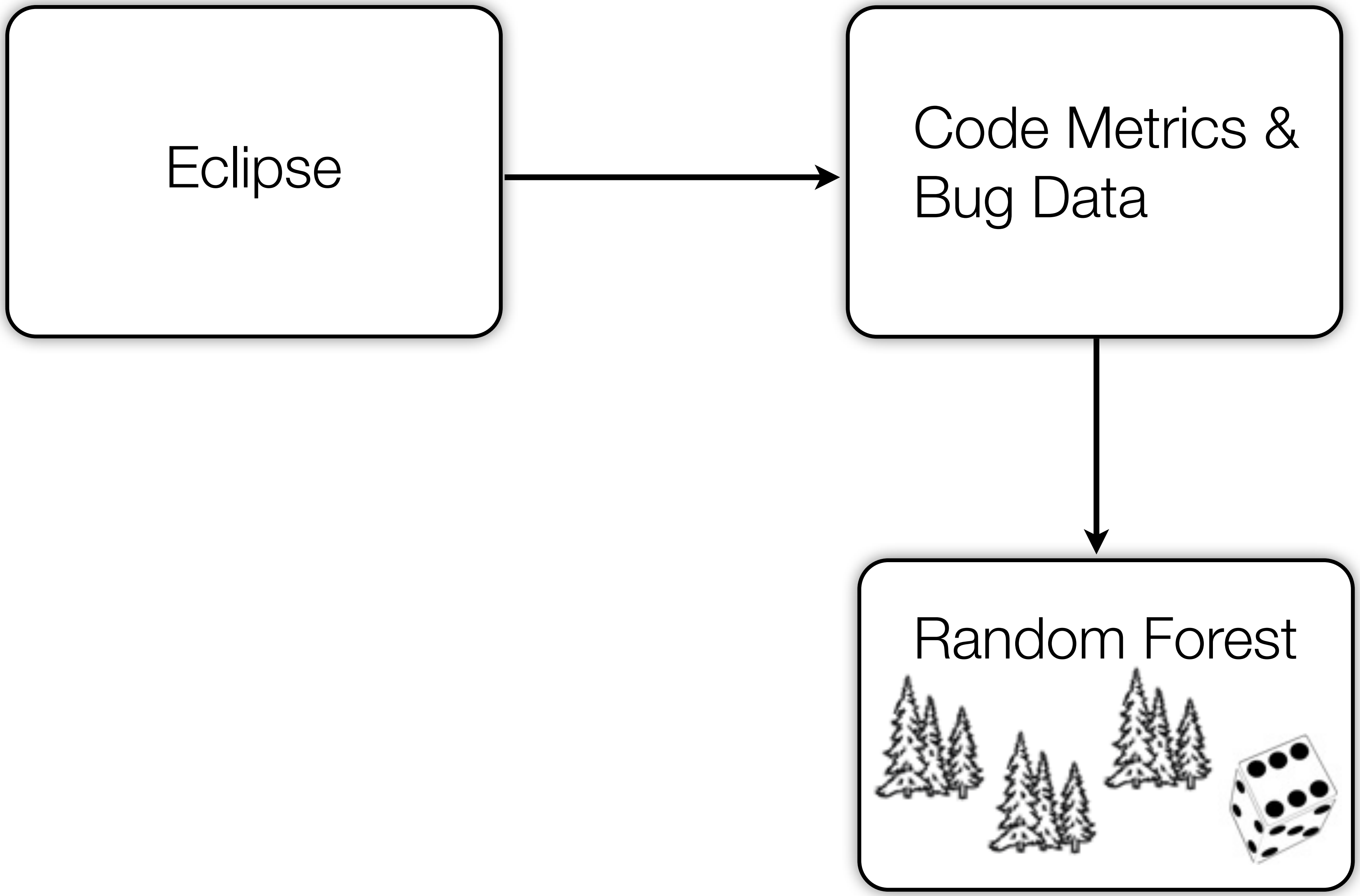
Bug Prediction Setup

Eclipse

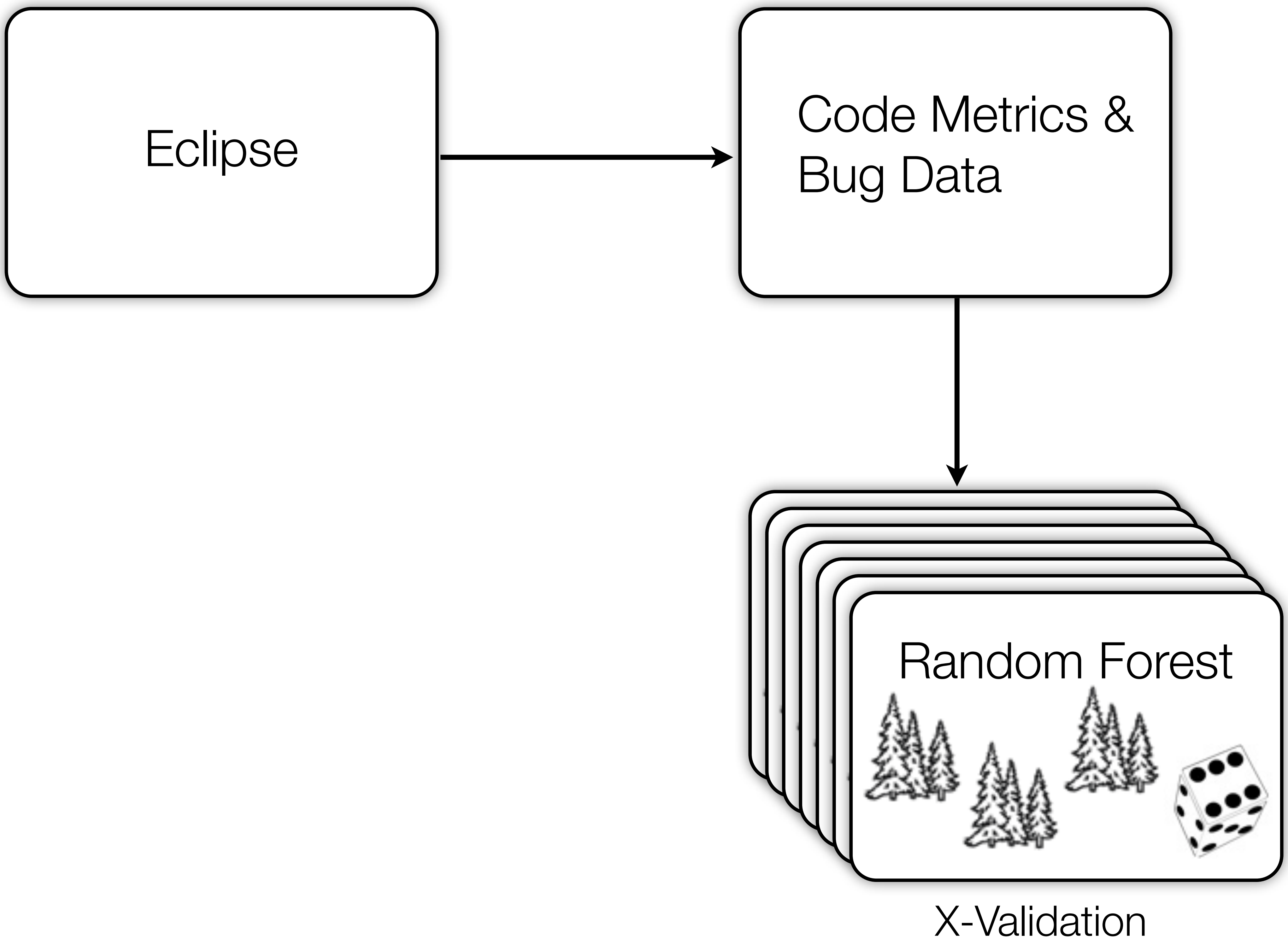
Bug Prediction Setup



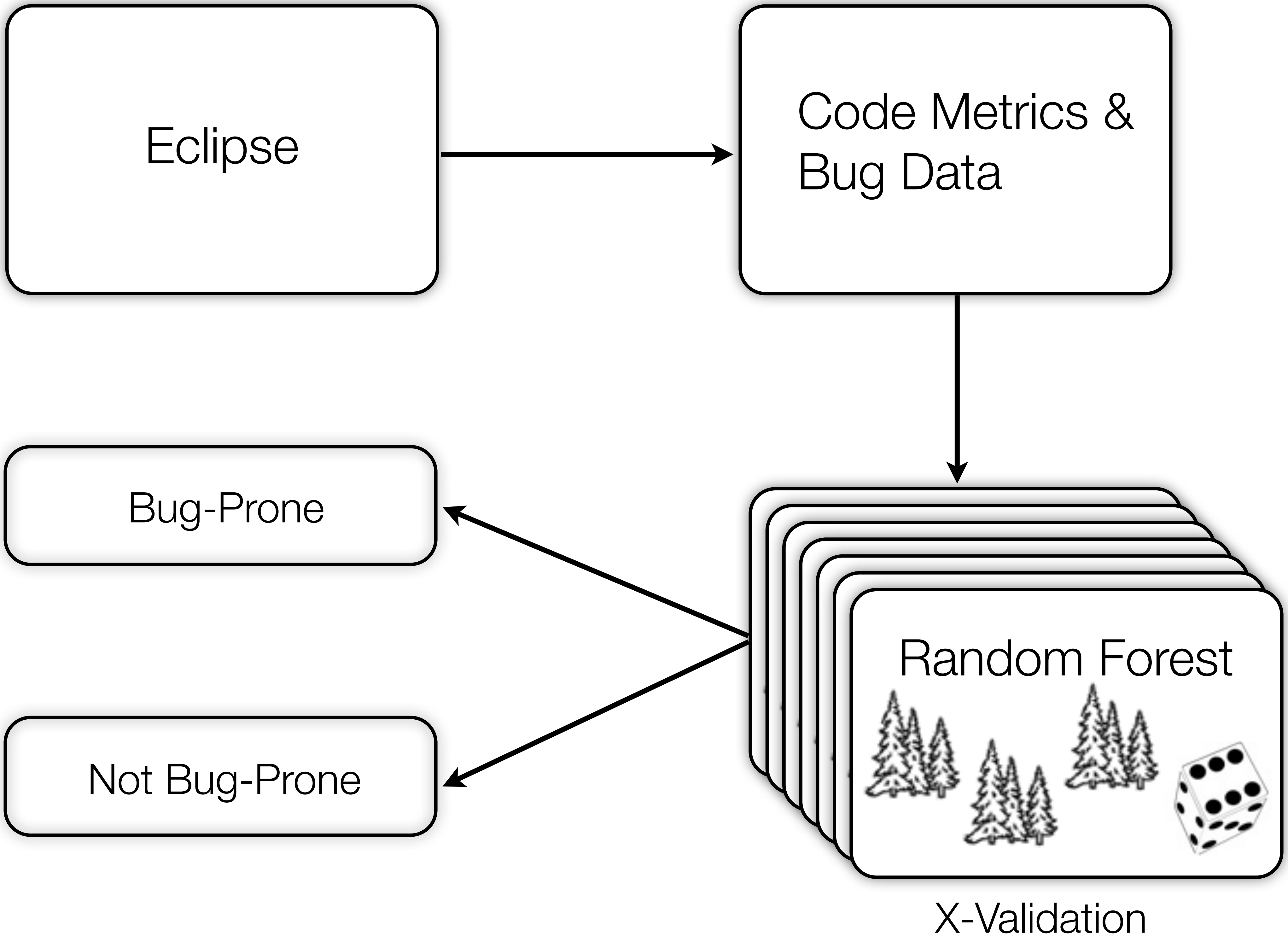
Bug Prediction Setup



Bug Prediction Setup



Bug Prediction Setup



m = Mccabe		$v(g)$	cyclomatic_complexity
		$iv(G)$	design_complexity
		$ev(G)$	essential_complexity
locs	loc		loc_total (one line = one count
	loc(other)		loc_blank
			loc_code_and_comment
			loc_comments
			loc_executable
			number_of_lines (opening to closing brackets)
Halstead	h	N_1	num_operators
		N_2	num_operands
		μ_1	num_unique_operators
		μ_2	num_unique_operands
	H	N	length: $N = N_1 + N_2$
		V	volume: $V = N * log_2 \mu$
		L	level: $L = V^*/V$ where $V^* = (2 + {\mu_2}^*) log_2 (2 + {\mu_2}^*)$
		D	difficulty: $D = 1/L$
		I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
		E	effort: $E = V/\hat{L}$
		B	error_est
		T	prog_time: $T = E/18$ seconds
misc = Miscellaneous			branch_count
			call_pairs
			condition_count
			decision_count
			decision_density
			design_density
			edge_count
			global_data_complexity
			global_data_density
			maintenance_severity
			modified_condition_count
			multiple_condition_count
			node_count
			normalized_cyclomatic_complexity
			parameter_count
			pathological_complexity
			percent_comments

Data Mining Static Code Attributes to Learn Defect Predictors

Tim Menzies, *Member*, *IEEE*, Jeremy Greenwald, and Art Frank

Abstract—The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes versus Halstead versus lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.

Index Terms—Data mining detect prediction, McCabe, Halstead, artifical intelligence, empirical, naive Bayes.

1 Introduction

Given recent research in artificial intelligence, it is now practical to use *data miners* to automatically learn predictors for software quality. When budget does not allow for complete testing of an entire system, software managers can use such predictors to focus the testing on parts of the system that seem defect-prone. These potential defect-prone trouble spots can then be examined in more detail by, say, model checking, intensive testing, etc.

The value of static code attributes as defect predictors has been widely debated. Some researchers endorse them ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]) while others vehemently oppose them ([21], [22]).

Prior studies may have reached different conclusions because they were based on different data. This potential conflation can now be removed since it is now possible to define a *baseline experiment* using public-domain data sets[1] which different researchers can use to compare their techniques.

This paper defines and motivates such a baseline. The baseline *definition* draws from standard practices in the data mining community [23], [24]. To *motivate* others to use our definition of a baseline experiment, we must demonstrate that it can yield interesting results. The baseline experiment of this article shows that the rule-based or decision-tree learning methods used in prior work [4], [13], [15], [16], [25] are clearly outperformed by a *naive Bayes* data miner with a *log-filtering* preprocessor on the numeric data (the terms in italics are defined later in this paper).

Further, the experiment can explain *why* our preferred Bayesian method performs best. That explanation is quite technical and comes from information theory. In this introduction, we need only say that the space of "best" predictors is "brittle," i.e., minor changes in the data (such as a slightly different sample used to learn a predictor) can make different attributes appear most useful for defect prediction.

This brittleness result offers a new insight on prior work. Prior results about defect predictors were so contradictory since they were drawn from a large space of competing conclusions with similar but distinct properties. Different studies could conclude that, say, lines of code are a better/worse predictor for defects than the McCabes complexity attribute, just because of small variations to the data. Bayesian methods smooth over the brittleness problem by polling numerous Gaussian approximations to the numerics distributions. Hence, Bayesian methods do not get confused by minor details about candidate predictors.

Our conclusion is that, contrary to prior pessimism [21], [22], data mining static code attributes to learn defect predictors is useful. Given our new results on naive Bayes and log-filtering, these predictors are much better than previously demonstrated. Also, prior contradictory results on the merits of defect predictors can be explained in terms of the brittleness of the space of "best" predictors. Further, our baseline experiment clearly shows that it is a misdirected discussion to debate, e.g., "lines of code versus McCabe" for predicting defects. As we shall see, *the choice of learning method* is far more important than *which subset of the available data* is used for learning.

2 Background

For this study, we learn defect predictors from static code attributes defined by McCabe [2] and Halstead [1]. McCabe and Halstead are "module"-based metrics, where a module

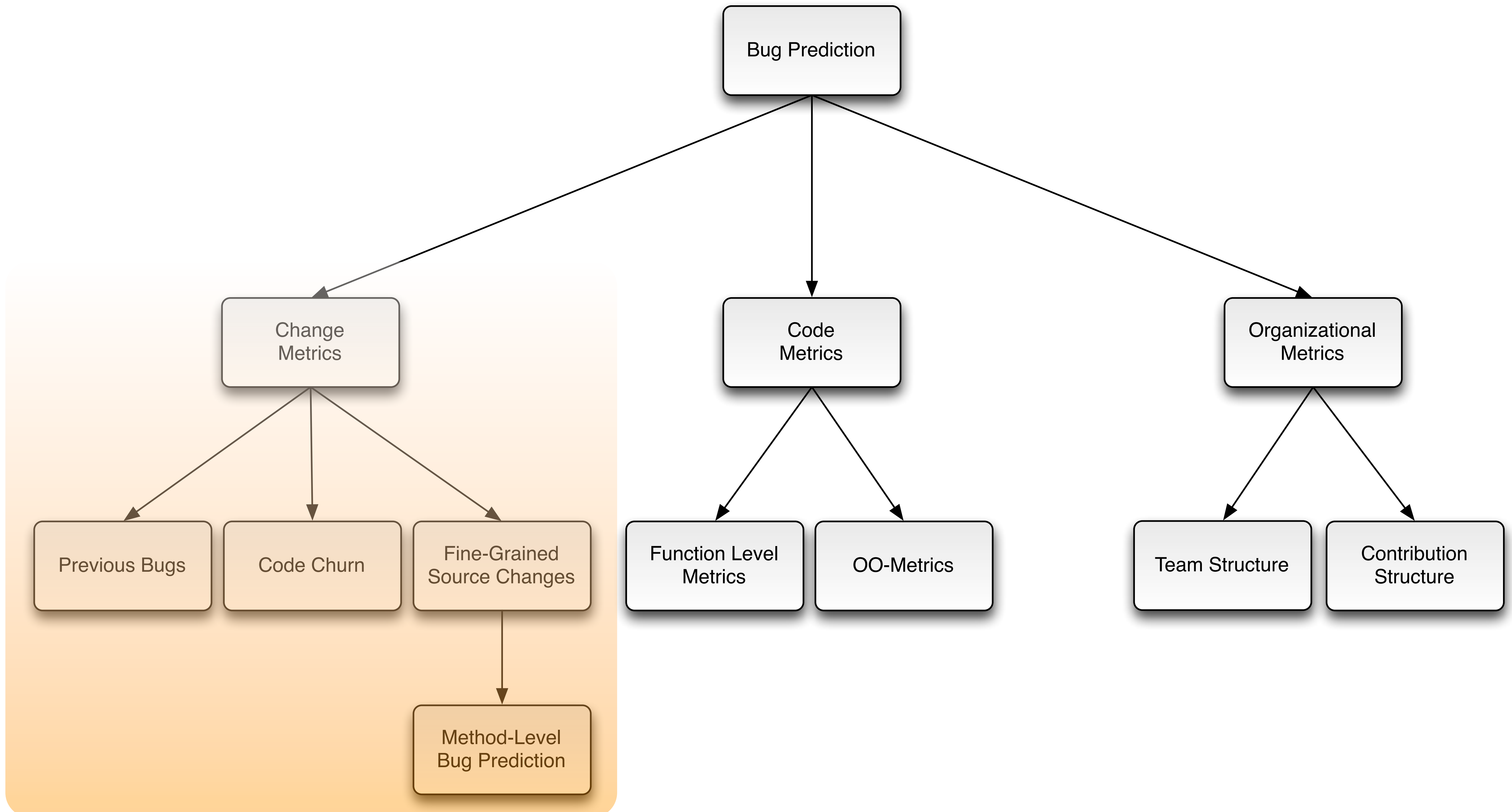
1. http://mdp.ivv.nasa.gov and http://promise.site.uottawa.ca/SERepository.

- *T. Menzies is with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-610. E-mail: tim@menzies.us.*
- *J. Greenwald and A. Frank are with the Department of Computer Science, Portland State University, PO Box 751, Portland, OR 97207-0751. E-mail: jegreen@cecs.pdx.edu, arf@cs.pdx.edu.*

Manuscript received 2 Jan. 2006; revised 9 Aug. 2006; accepted 13 Sept. 2006; published online 30 Nov. 2006.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0001-0106.

Size and complexity are indicators of defects

Bug Prediction Models



Change Metrics

- Process Metrics
- Reflect the development activities
- Basic assumptions: *The modules with many defects in the past will most likely be defect-prone in the future as well.*
- *Modules that change often have inherently a higher chance to be affected by defects.*

Code Changes

Revisions

Commits to version control systems

Coarse-grained

Files are the units of change

Revisions

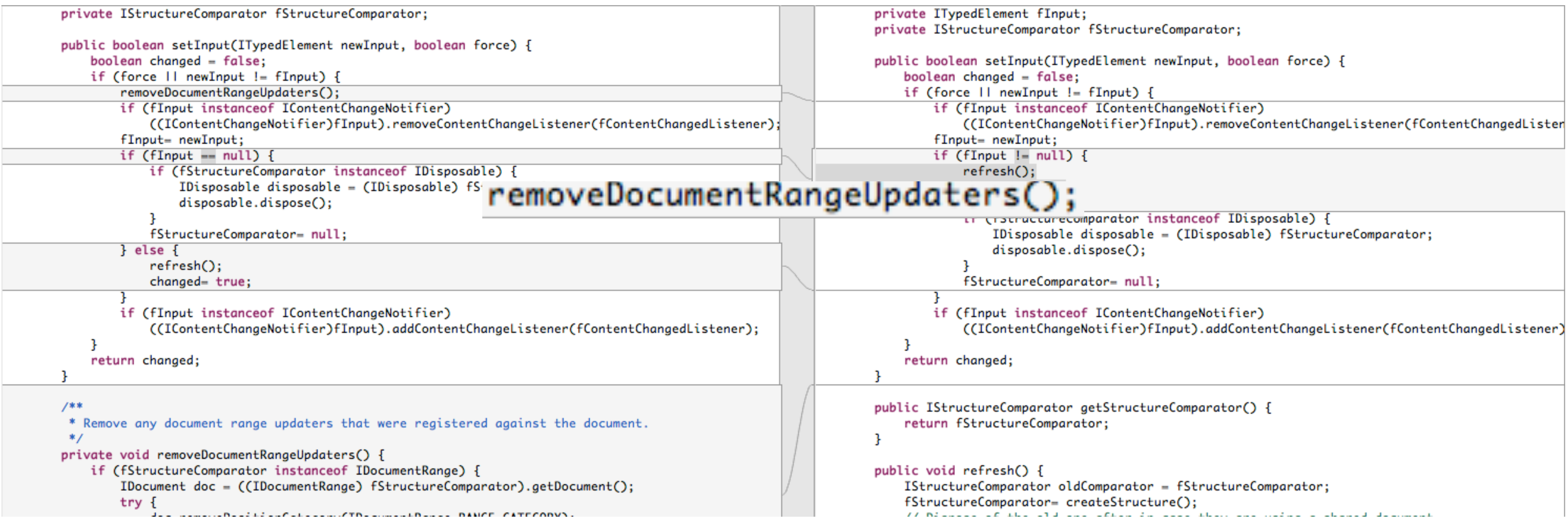
There is more than just a file revision

```java
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        removeDocumentRangeUpdaters();
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedListener);
        fInput= newInput;
        if (fInput == null) {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        } else {
            refresh();
            changed= true;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener);
    }
    return changed;
}

/**
 * Remove any document range updaters that were registered against the document.
 */
private void removeDocumentRangeUpdaters() {
    if (fStructureComparator instanceof IDocumentRange) {
        IDocument doc = ((IDocumentRange) fStructureComparator).getDocument();
        try {
```

```java
private ITypedElement fInput;
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedLister
        fInput= newInput;
        if (fInput != null) {
            refresh();
            changed= true;
        } else {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener)
    }
    return changed;
}

public IStructureComparator getStructureComparator() {
    return fStructureComparator;
}

public void refresh() {
    IStructureComparator oldComparator = fStructureComparator;
    fStructureComparator= createStructure();
```

Revisions

There is more than just a file revision



Revisions

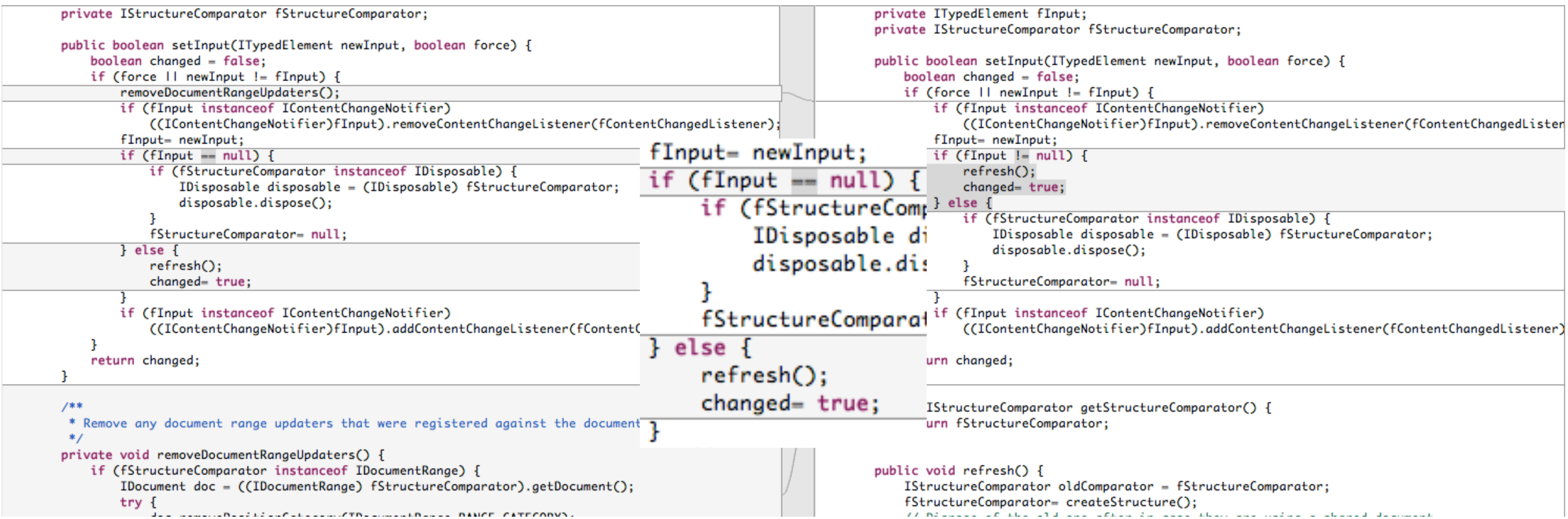
There is more than just a file revision

```java
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        removeDocumentRangeUpdaters();
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedListener);
        fInput= newInput;
        if (fInput == null) {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        } else {
            refresh();
            changed= true;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener);
    }
    return changed;
}

/**
 * Remove any document range updaters that were registered against the document.
 */
private void removeDocumentRangeUpdaters() {
    if (fStructureComparator instanceof IDocumentRange) {
        IDocument doc = ((IDocumentRange) fStructureComparator).getDocument();
        try {
```

```java
private ITypedElement fInput;
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedLister
        fInput= newInput;
        if (fInput != null) {
            refresh();
            changed= true;
        } else {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener)
    }
    return changed;
}

public IStructureComparator getStructureComparator() {
    return fStructureComparator;
}

public void refresh() {
    IStructureComparator oldComparator = fStructureComparator;
    fStructureComparator= createStructure();
```

Revisions

There is more than just a file revision



Revisions

There is more than just a file revision

```java
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        removeDocumentRangeUpdaters();
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedListener);
        fInput= newInput;
        if (fInput == null) {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        } else {
            refresh();
            changed= true;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener);
    }
    return changed;
}

/**
 * Remove any document range updaters that were registered against the document.
 */
private void removeDocumentRangeUpdaters() {
    if (fStructureComparator instanceof IDocumentRange) {
        IDocument doc = ((IDocumentRange) fStructureComparator).getDocument();
        try {
```

```java
private ITypedElement fInput;
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedLister
        fInput= newInput;
        if (fInput != null) {
            refresh();
            changed= true;
        } else {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener)
    }
    return changed;
}

public IStructureComparator getStructureComparator() {
    return fStructureComparator;
}

public void refresh() {
    IStructureComparator oldComparator = fStructureComparator;
    fStructureComparator= createStructure();
```

Revisions

There is more than just a file revision

```java
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        removeDocumentRangeUpdaters();
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedListener);
        fInput= newInput;
        if (fInput == null) {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        } else {
            refresh();
            changed= true;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener);
    }
    return changed;
}

/**
 * Remove any document range updaters that were registered against the document.
 */
private void removeDocumentRangeUpdaters() {
    if (fStructureCompa
        IDocument doc =
        try {
```

```java
private ITypedElement fInput;
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedLister
        fInput= newInput;
        if (fInput != null) {
            refresh();
            changed= true;
        } else {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener)
    }
    return changed;
}

public IStructureComparator getStructureComparator() {
    return fStructureComparator;
}
```

```java
private void removeDocumentRangeUpdaters() {
    if (fStructureComparator instanceof IDocumentRange) {
        IDocument doc = ((IDocumentRange) fStructureComparator).getDocument();
```

Revisions

There is more than just a file revision

```java
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        removeDocumentRangeUpdaters();
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedListener);
        fInput= newInput;
        if (fInput == null) {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        } else {
            refresh();
            changed= true;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener);
    }
    return changed;
}

/**
 * Remove any document range updaters that were registered against the document.
 */
private void removeDocumentRangeUpdaters() {
    if (fStructureComparator instanceof IDocumentRange) {
        IDocument doc = ((IDocumentRange) fStructureComparator).getDocument();
        try {
```

```java
private ITypedElement fInput;
private IStructureComparator fStructureComparator;

public boolean setInput(ITypedElement newInput, boolean force) {
    boolean changed = false;
    if (force || newInput != fInput) {
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).removeContentChangeListener(fContentChangedLister
        fInput= newInput;
        if (fInput != null) {
            refresh();
            changed= true;
        } else {
            if (fStructureComparator instanceof IDisposable) {
                IDisposable disposable = (IDisposable) fStructureComparator;
                disposable.dispose();
            }
            fStructureComparator= null;
        }
        if (fInput instanceof IContentChangeNotifier)
            ((IContentChangeNotifier)fInput).addContentChangeListener(fContentChangedListener)
    }
    return changed;
}

public IStructureComparator getStructureComparator() {
    return fStructureComparator;
}

public void refresh() {
    IStructureComparator oldComparator = fStructureComparator;
    fStructureComparator= createStructure();
```

Code Changes

Revisions

Commits to version control systems

Coarse-grained

Files are the units of change

Code Churn

Textual UnixDiff between 2 File Versions

Ignores the structure of code

No change type information

Includes textual changes

Code Churn

Does not reflect the type and the semantics of source code changes

```
/*******************************************************************************
 * Copyright (c) 2000, 2004 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *     IBM Corporation - initial API and implementation
 *******************************************************************************/
package org.eclipse.compare.structuremergeviewer;

import org.eclipse.swt.events.DisposeEvent;
import org.eclipse.swt.widgets.*;
import org.eclipse.jface.util.PropertyChangeEvent;

import org.eclipse.compare.*;
import org.eclipse.compare.internal.*;


/**
```

```
/*******************************************************************************
 * Copyright (c) 2000, 2004 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Common Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/cpl-v10.html
 *
 * Contributors:
 *     IBM Corporation - initial API and implementation
 *******************************************************************************
package org.eclipse.compare.structuremergeviewer;

import org.eclipse.swt.events.DisposeEvent;
import org.eclipse.swt.widgets.*;
import org.eclipse.jface.util.PropertyChangeEvent;

import org.eclipse.compare.*;
import org.eclipse.compare.internal.*;


/**
```

Code Changes

Revisions

Commits to version control systems

Coarse-grained

Files are the units of change

Code Churn

Textual UnixDiff between 2 File Versions

Ignores the structure of code

No change type information

Includes textual changes

Fine-Grained Changes[1]

Compares 2 versions of the AST of source code

Very fine-grained

Change type information

Captures all changes

Code Changes

Revisions

Commits to version control systems

Coarse-grained

Files are the units of change

Code Churn

Textual UnixDiff between 2 File Versions

Ignores the structure of code

No change type information

Includes textual changes

Fine-Grained Changes[1]

Compares 2 versions of the AST of source code

Very fine-grained

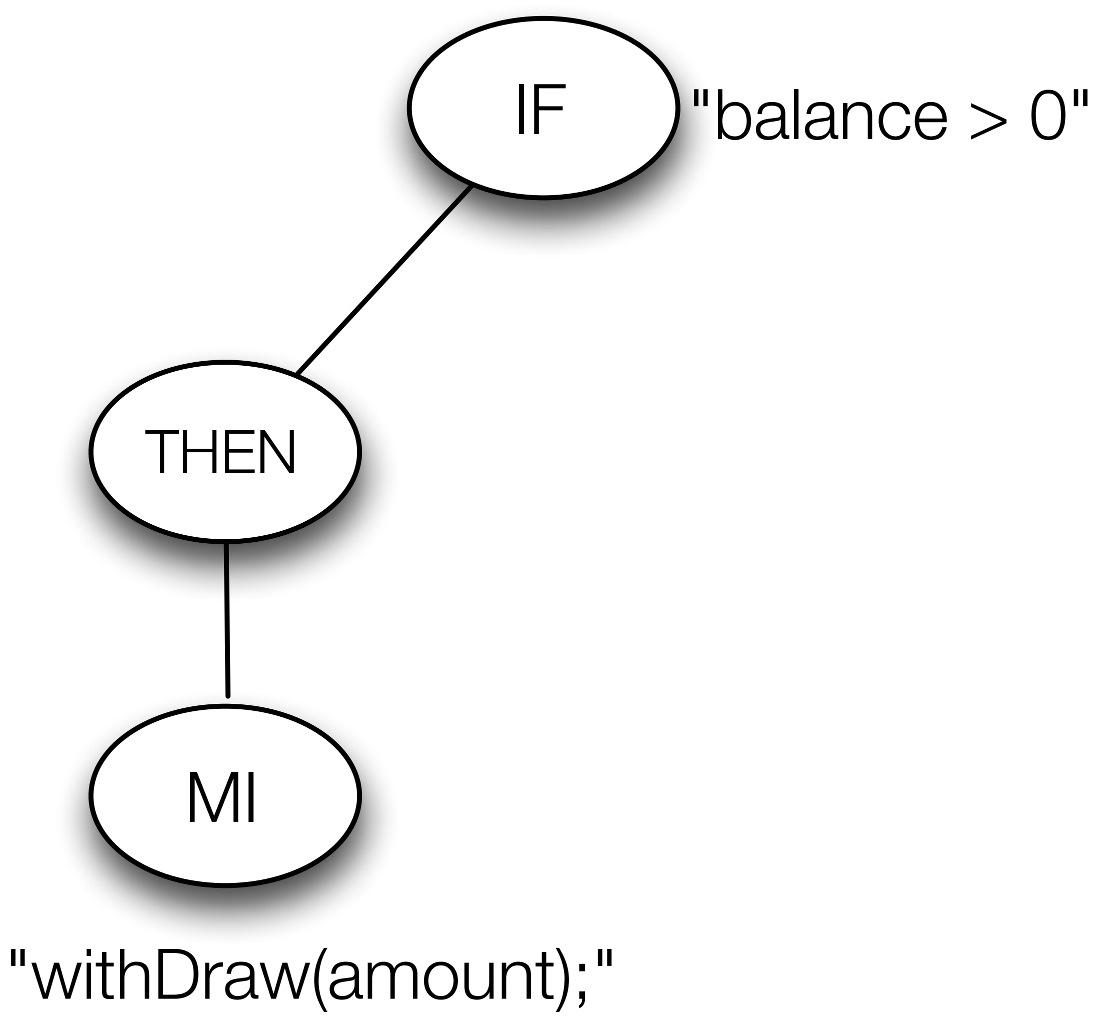
Change type information

Captures all changes

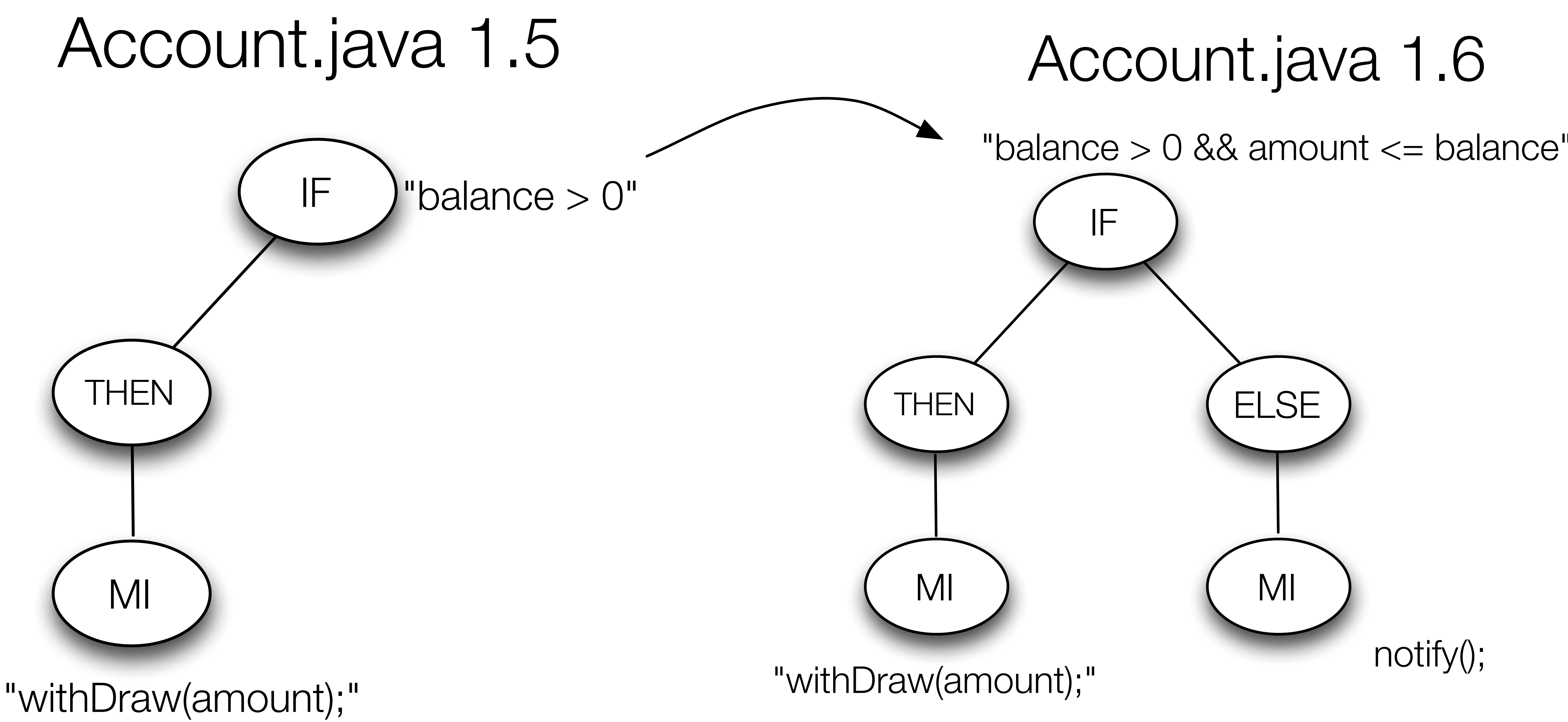
[1] [Fluri et al. 2007, TSE]

Fine-grained Changes

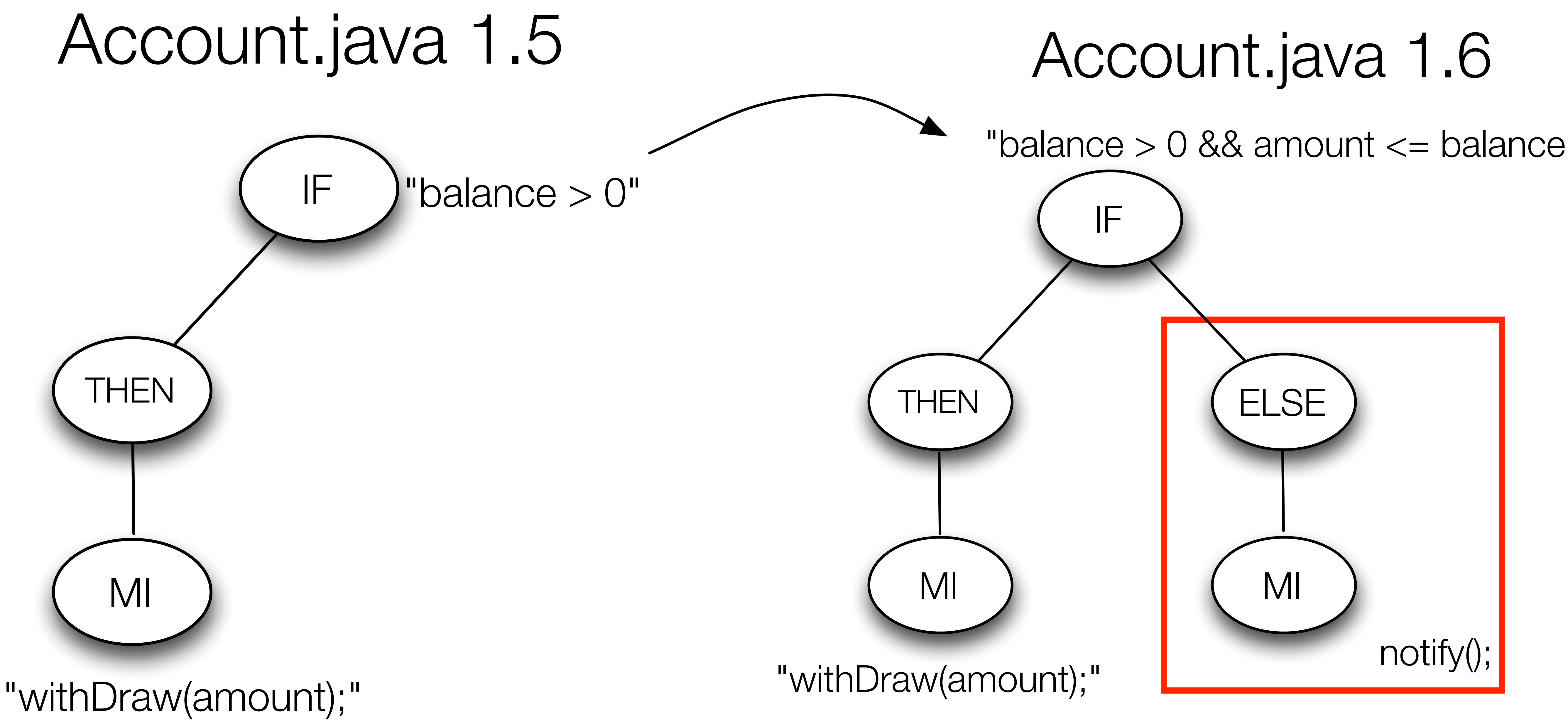
Account.java 1.5



Fine-grained Changes

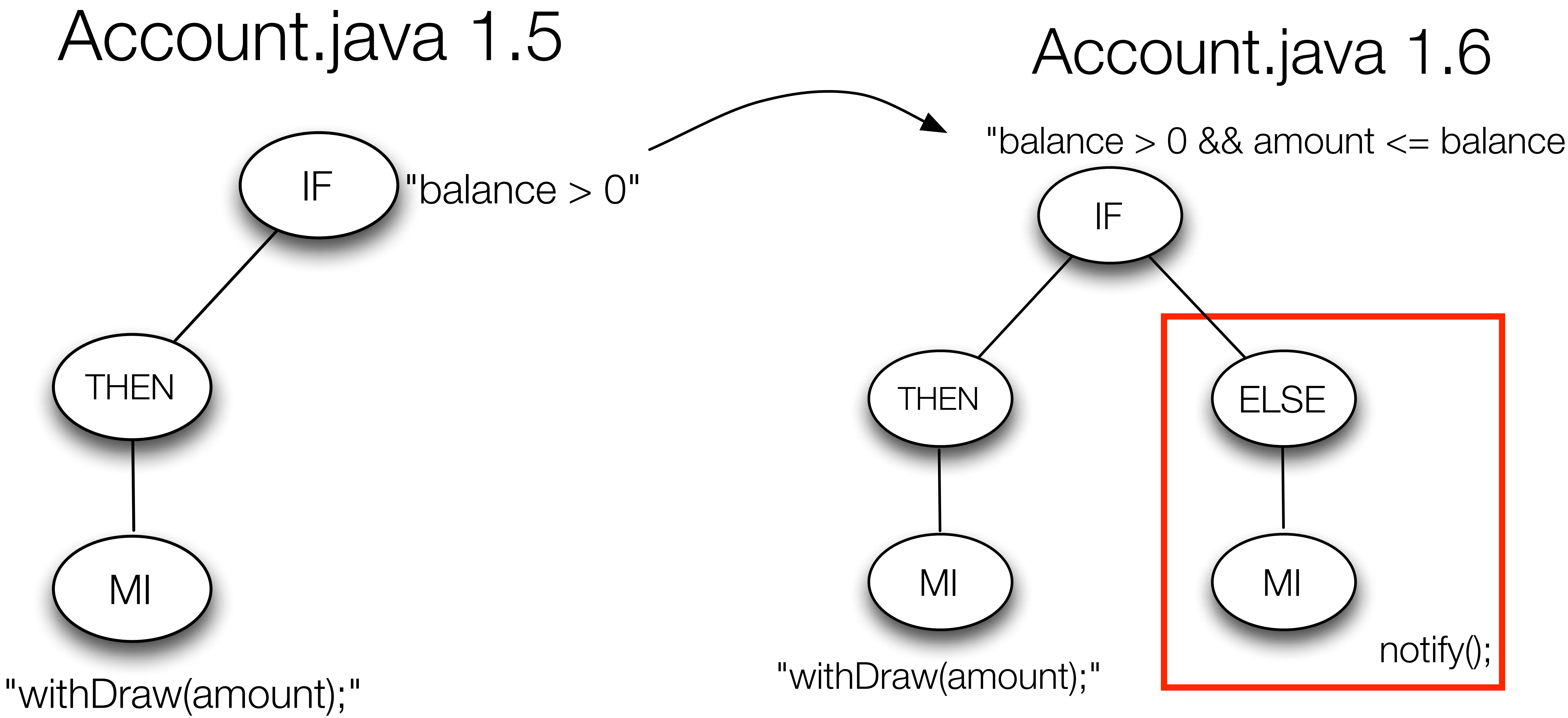


Fine-grained Changes



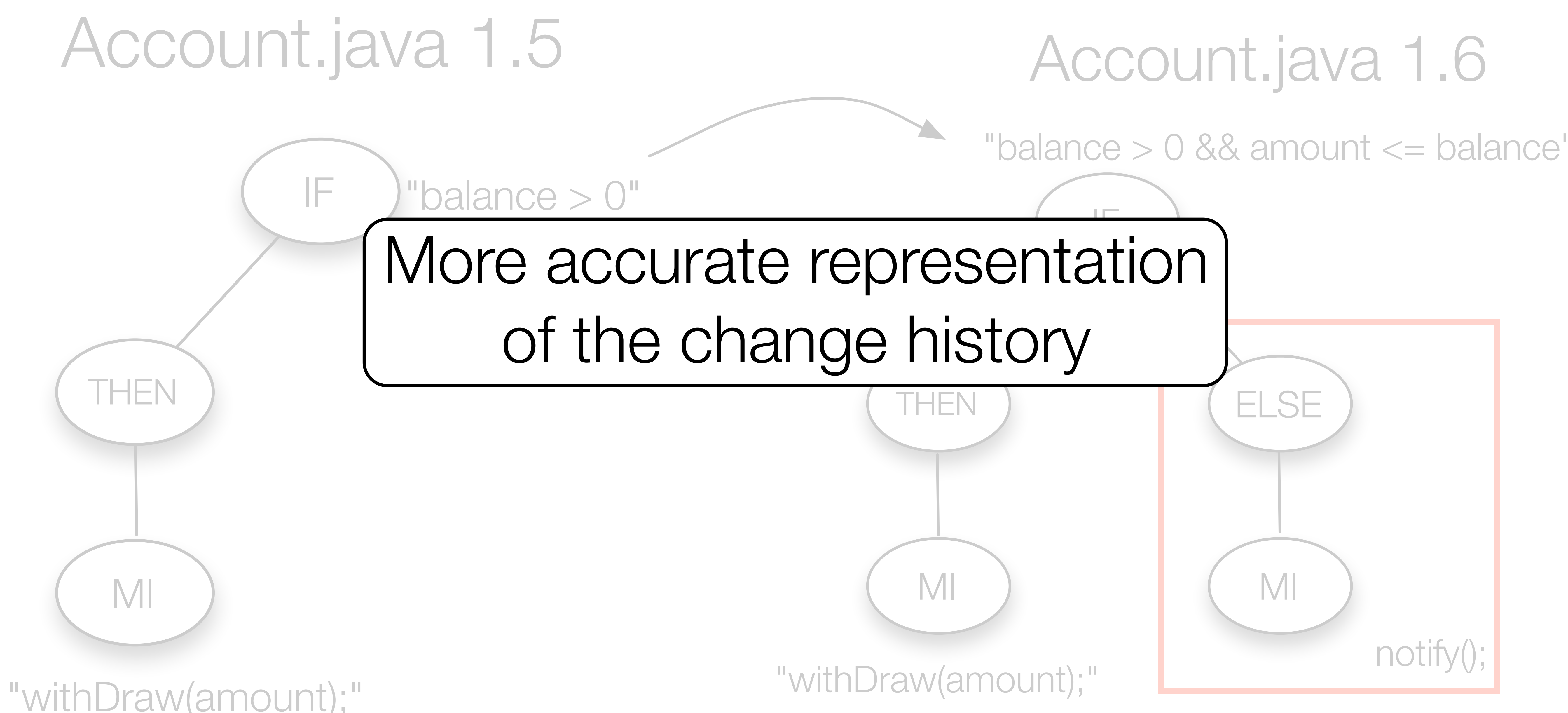
1x condition change, 1x else-part insert, 1x invocation statement insert

Fine-grained Changes



1x condition change, 1x else-part insert, 1x invocation statement insert

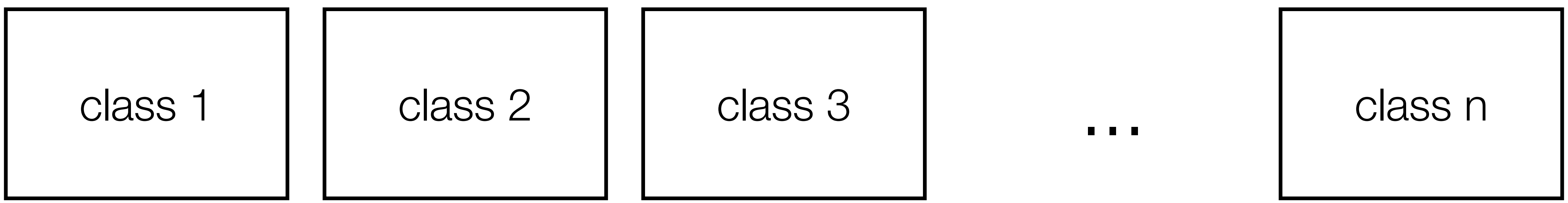
Fine-grained Changes



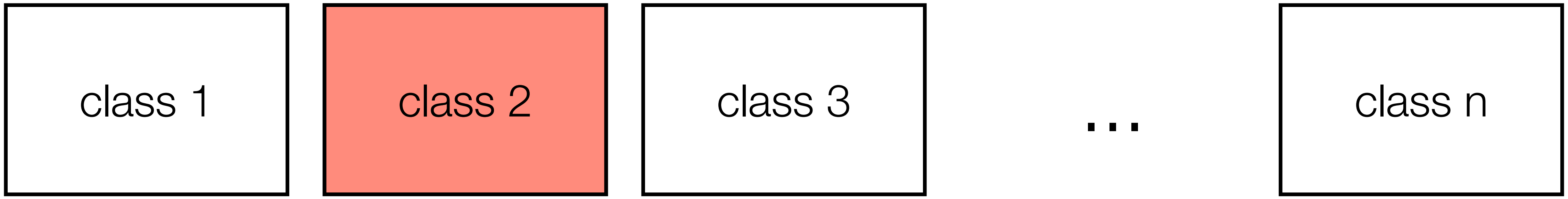
More accurate representation of the change history

1x condition change, 1x else-part insert, 1x invocation statement insert

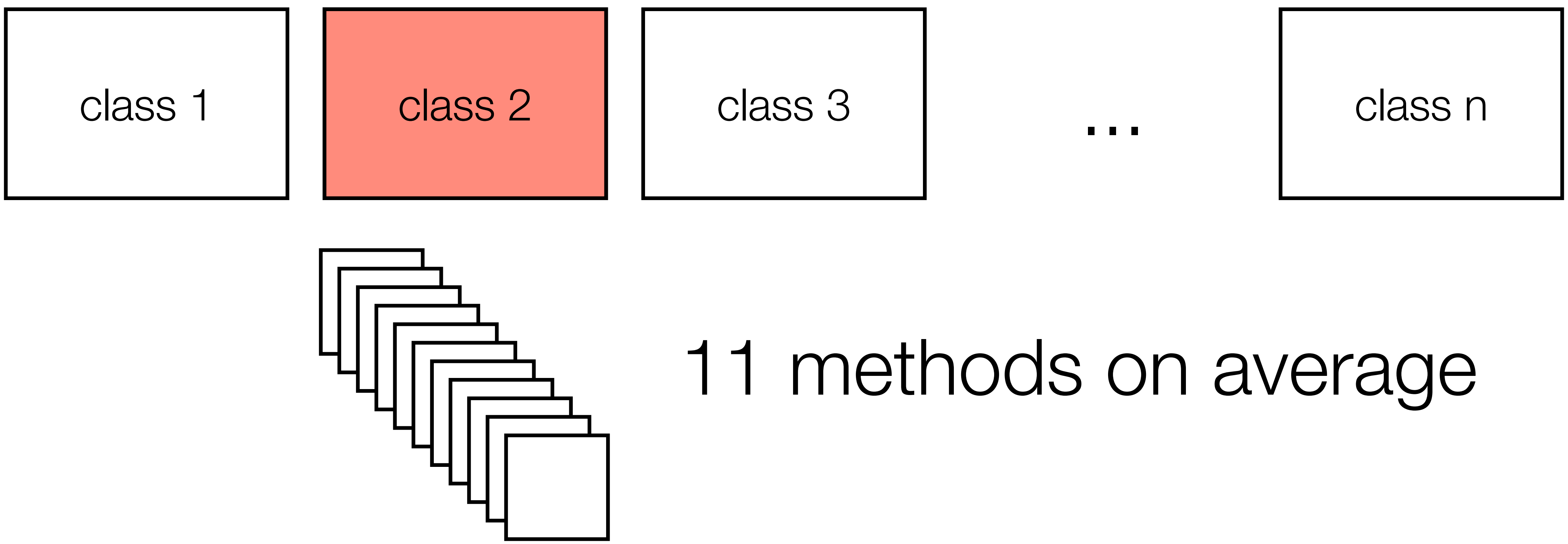
Method-Level Bug Prediction



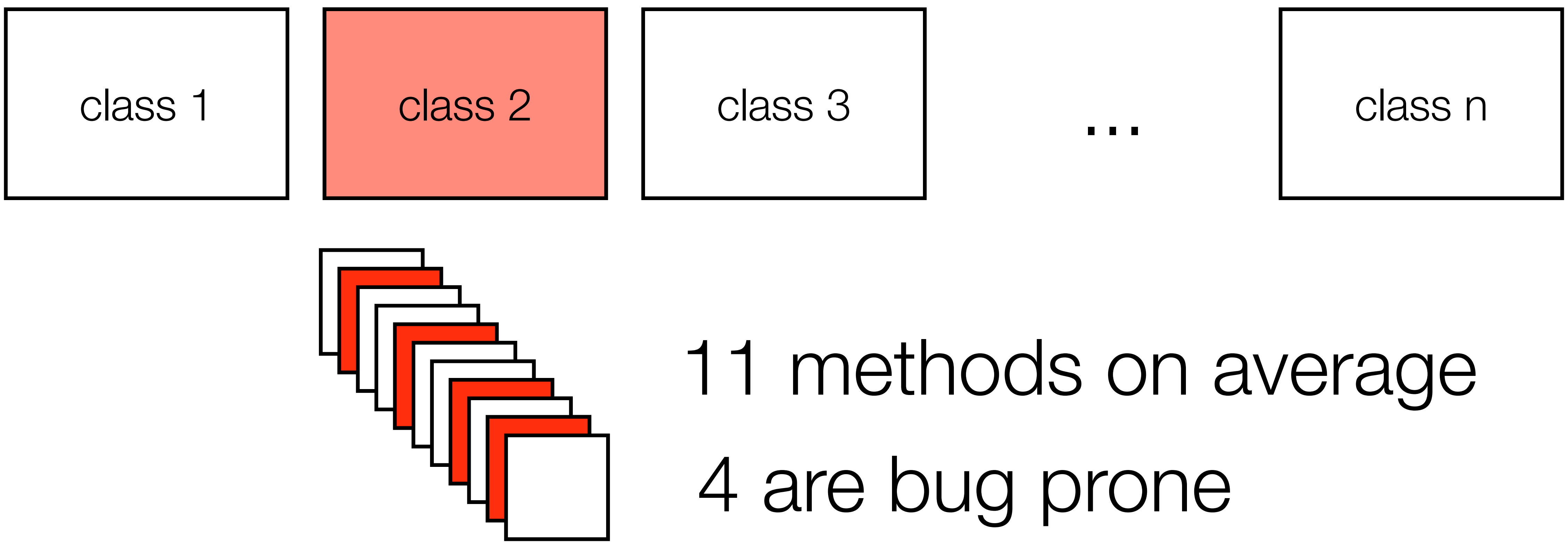
Method-Level Bug Prediction



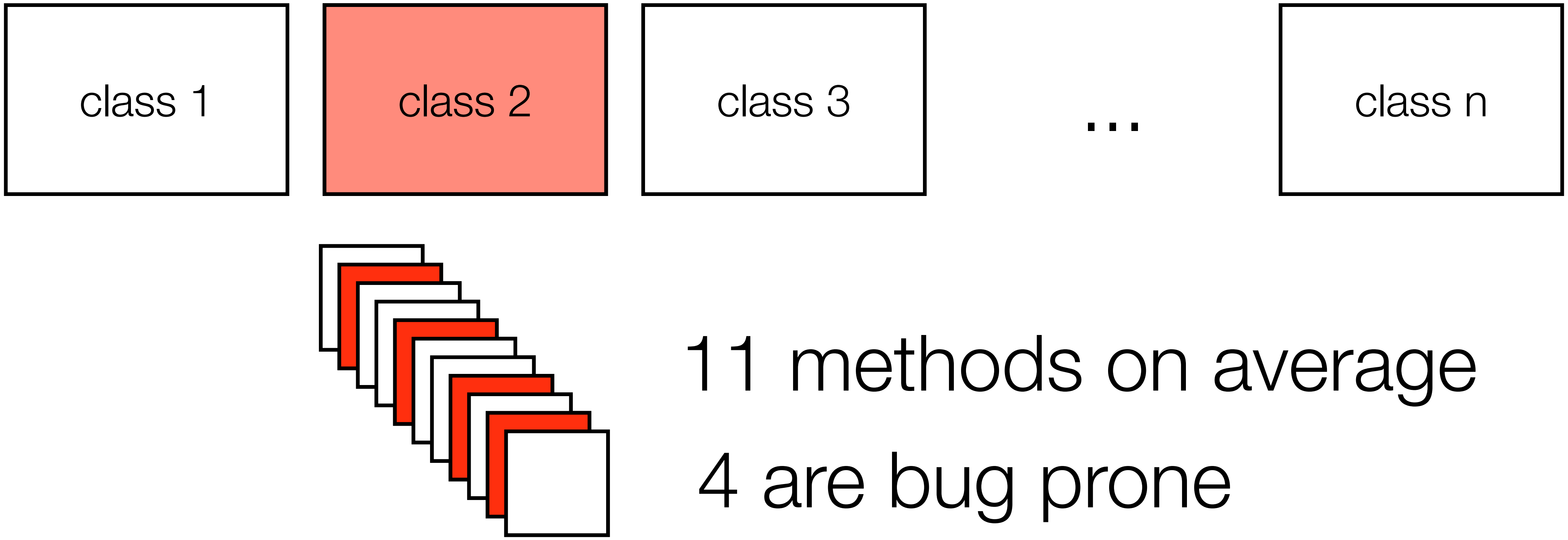
Method-Level Bug Prediction



Method-Level Bug Prediction



Method-Level Bug Prediction

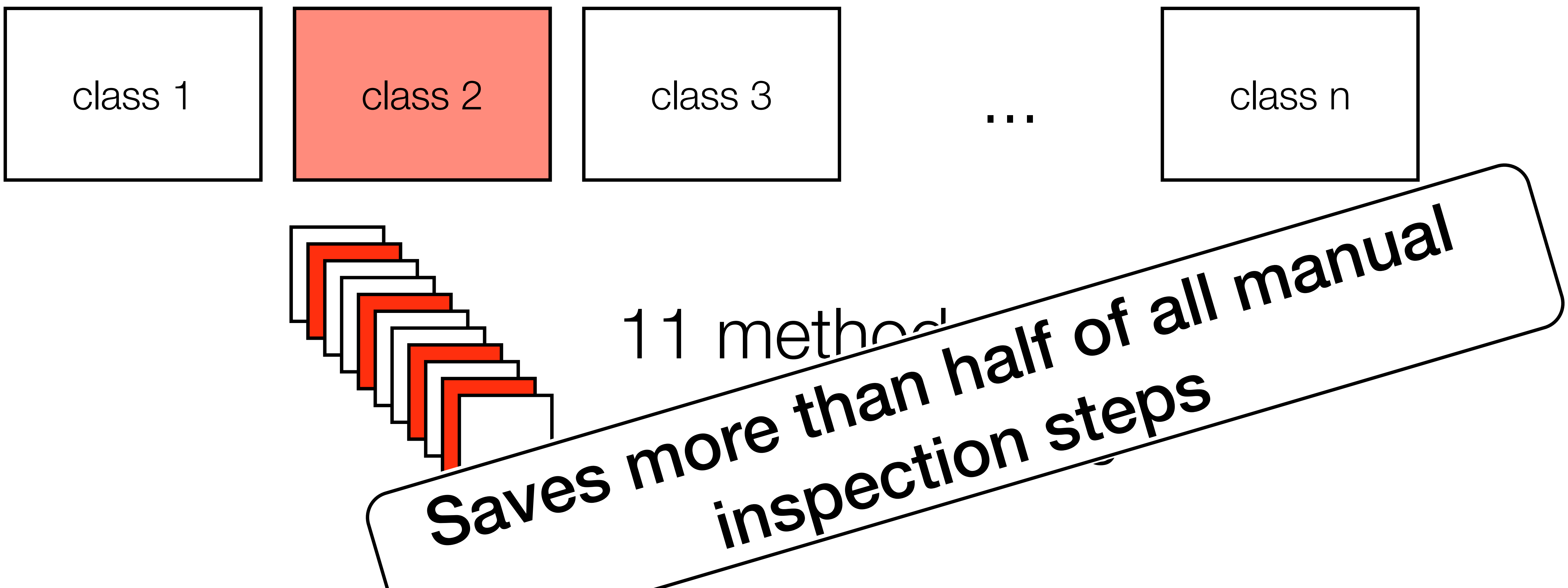


11 methods on average

4 are bug prone

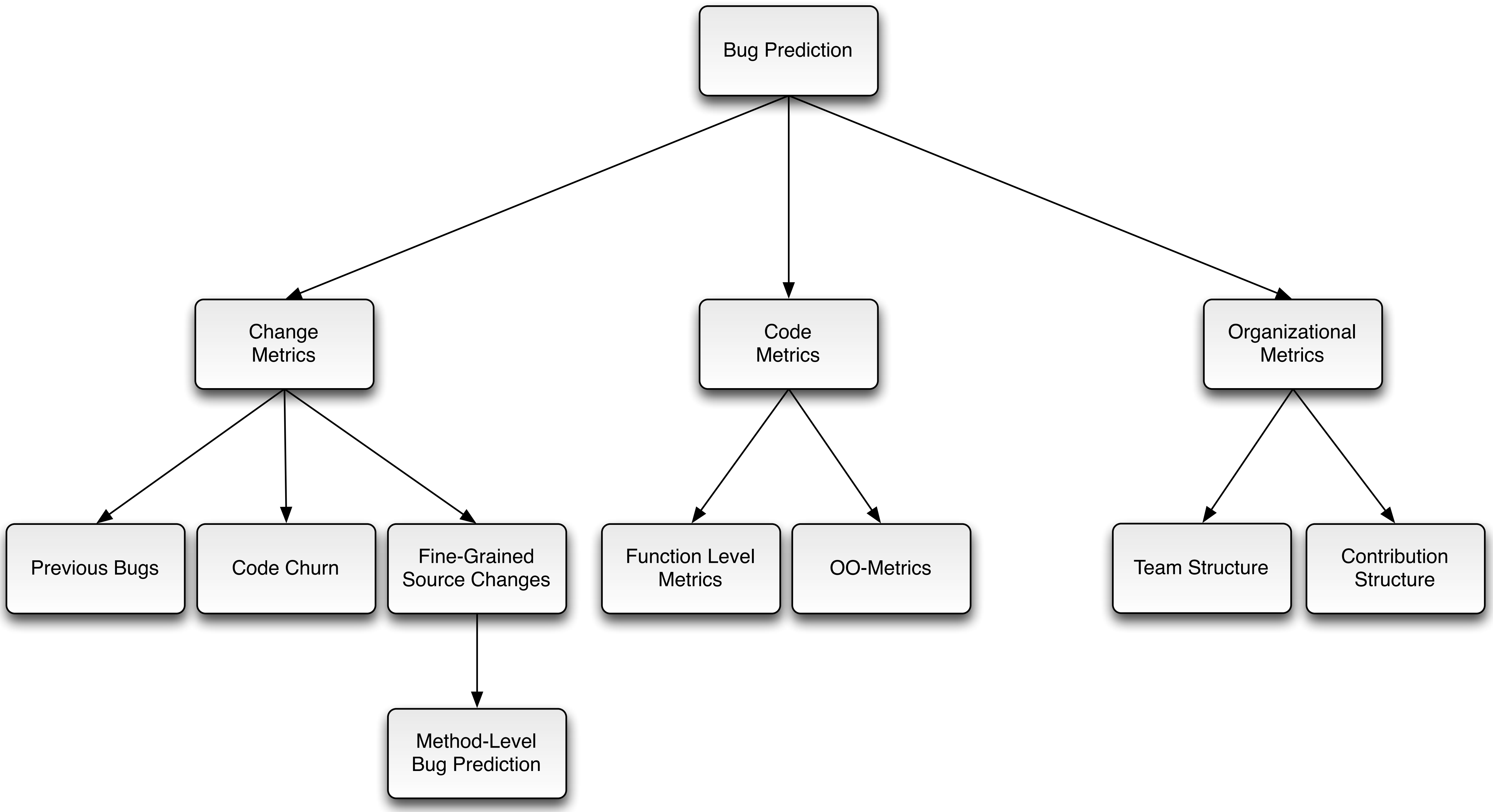
Retrieving bug-prone methods saves manual inspection steps and improves testing effort allocation

Method-Level Bug Prediction

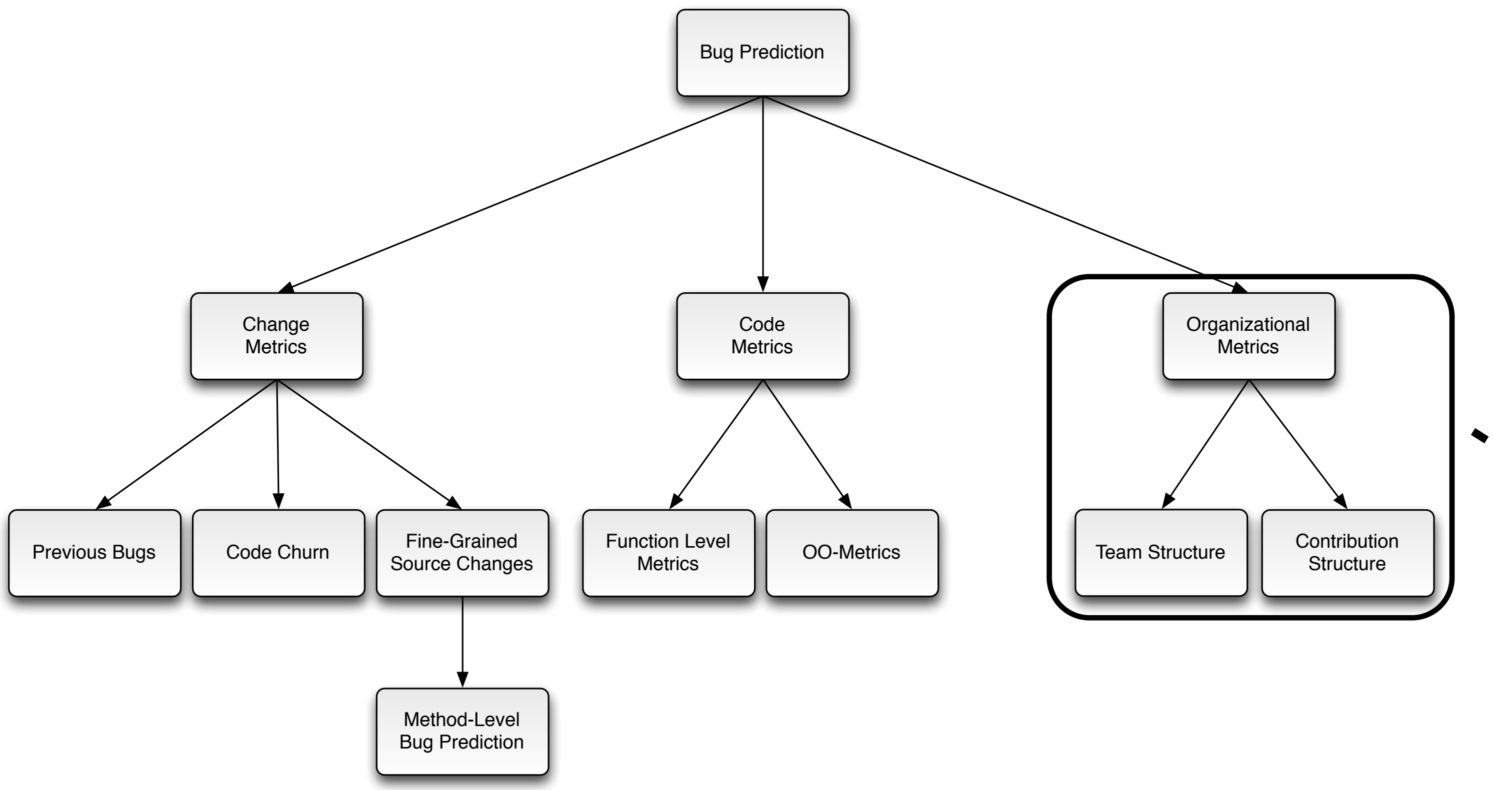


Retrieving bug-prone methods saves manual inspection steps and improves testing effort allocation

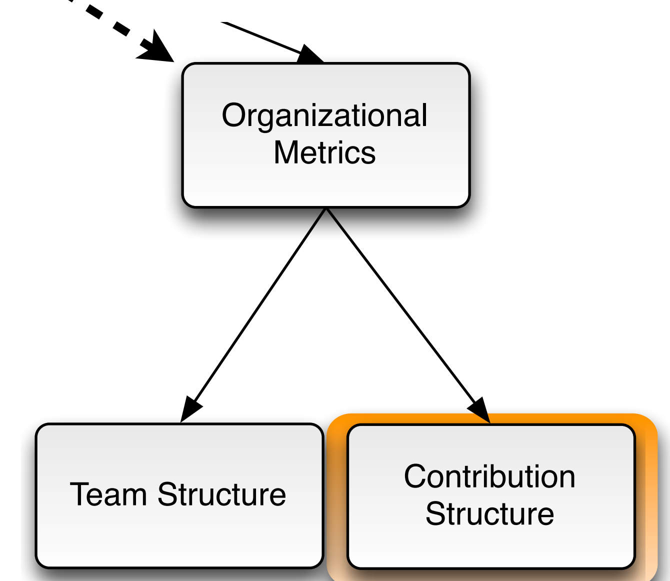
Bug Prediction Models



Bug Prediction Models



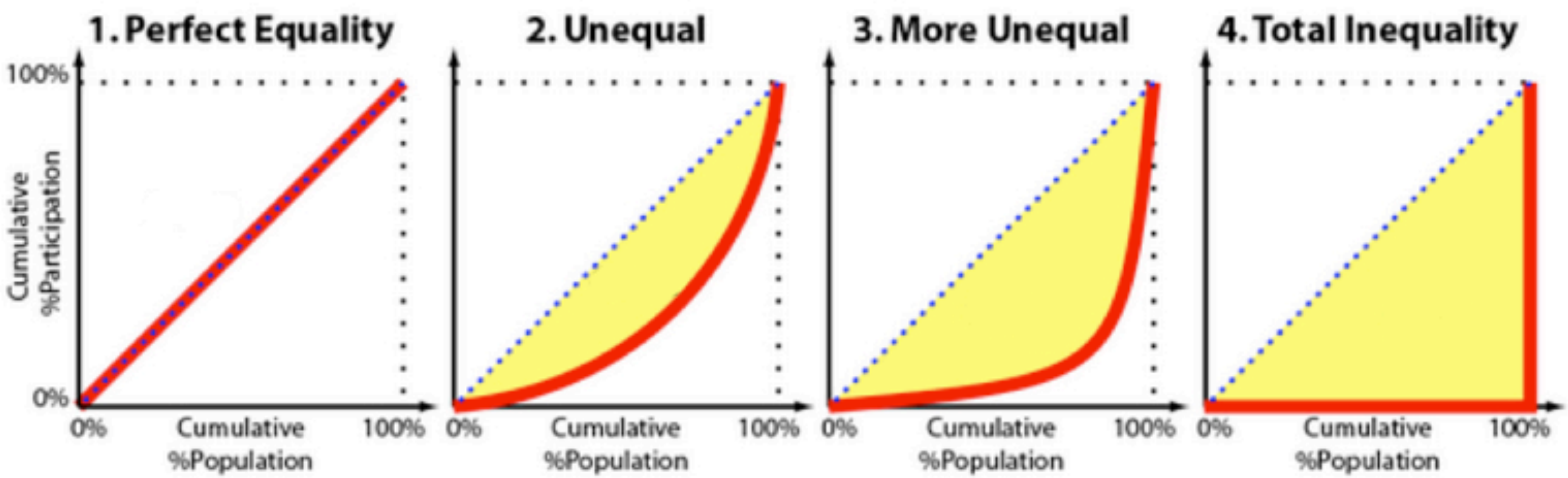
Using the Gini Coefficient for Bug Prediction



Organizational Metrics

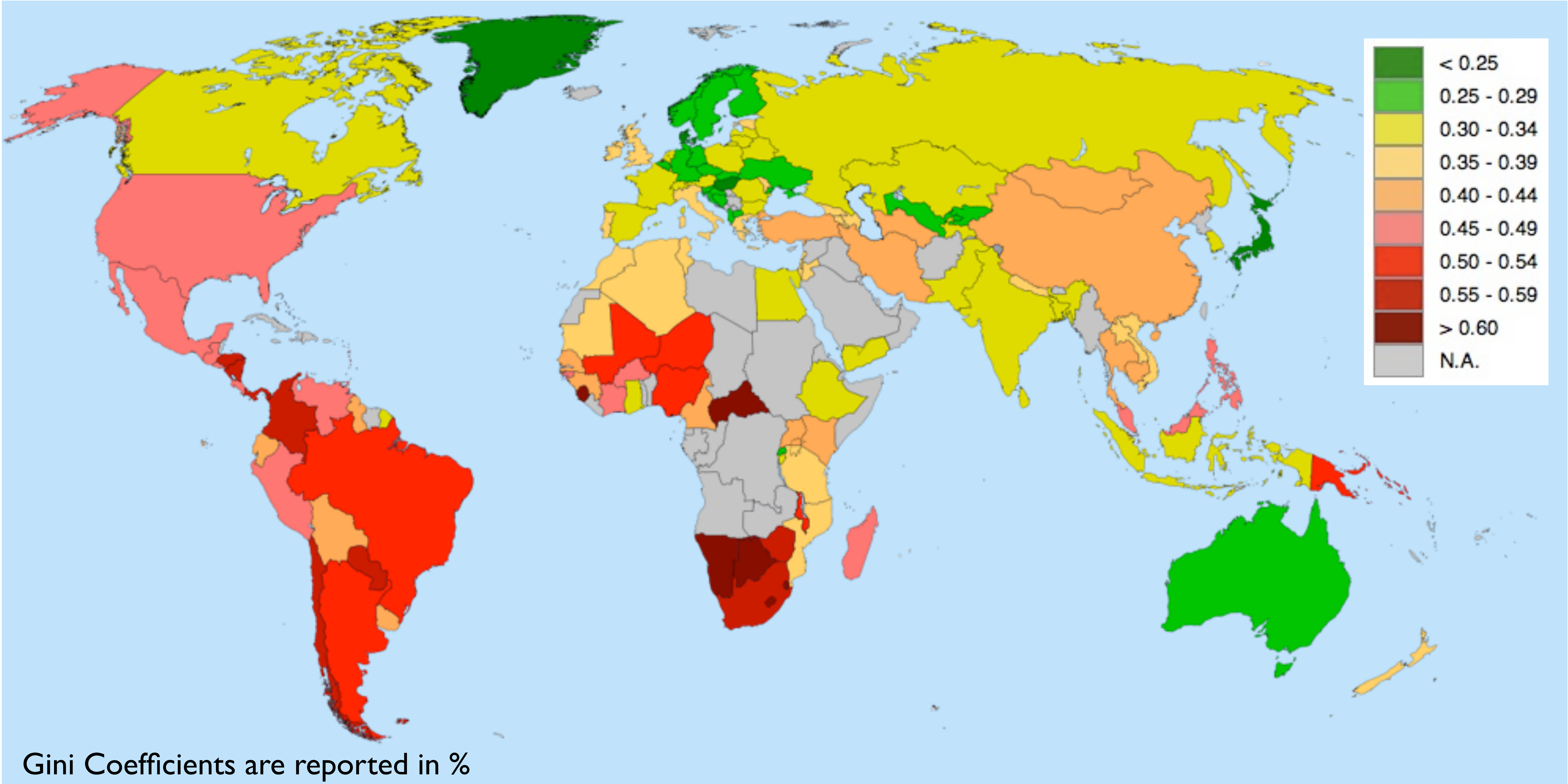
Basic Assumption: Organizational structure and regulations influence the quality of a software system.

Gini Coefficient



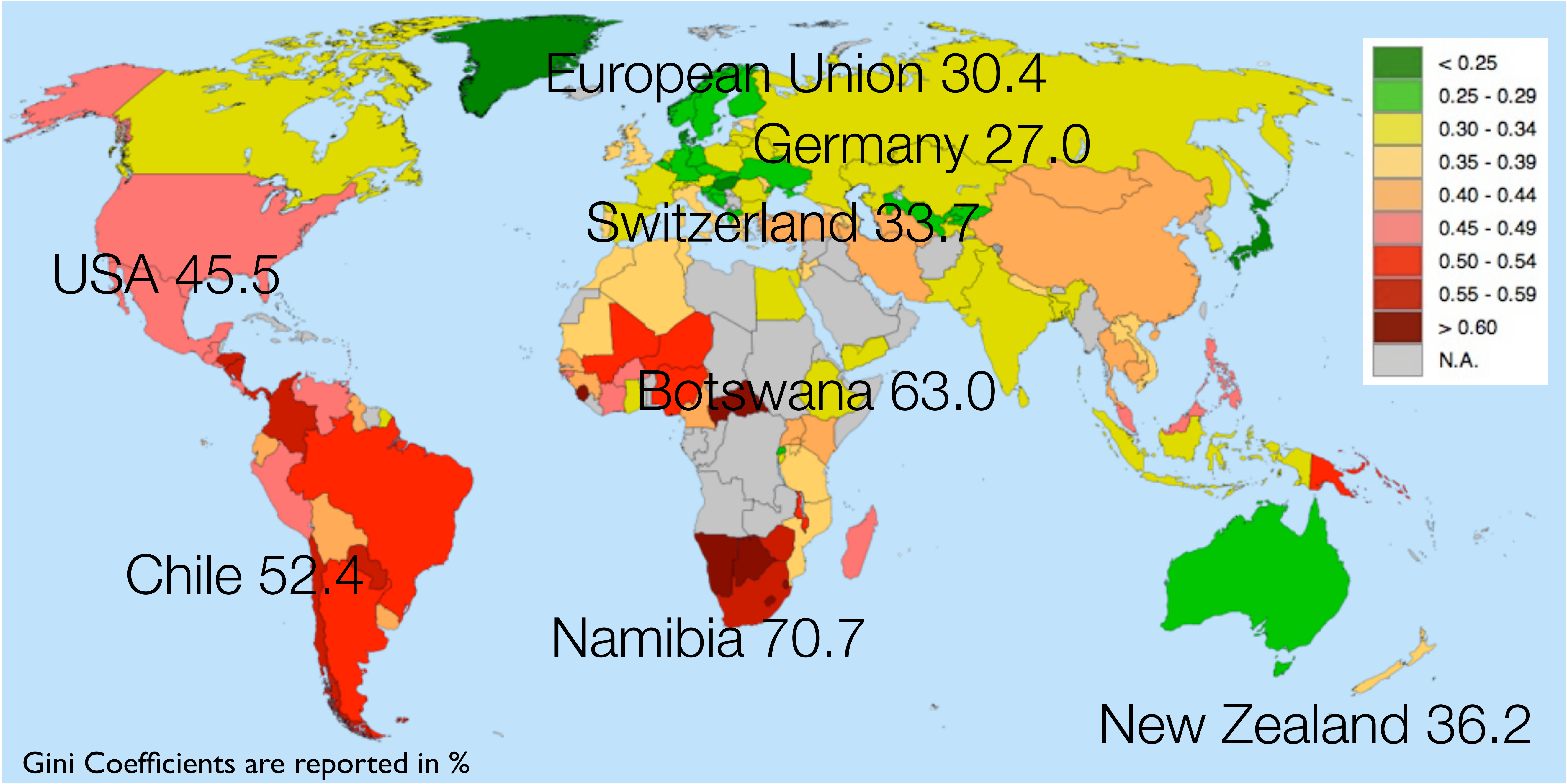
- The Lorenz curve plots the cumulative % of the total participation against the cumulative % of the population
- Gini Coefficient summarizes the curve in a number

Income Distribution



[1]CIA - The World Factbook, **DISTRIBUTION OF FAMILY INCOME - GINI INDEX**, *https://www.cia.gov/library/publications/the-world-factbook/rankorder/2172rank.html*

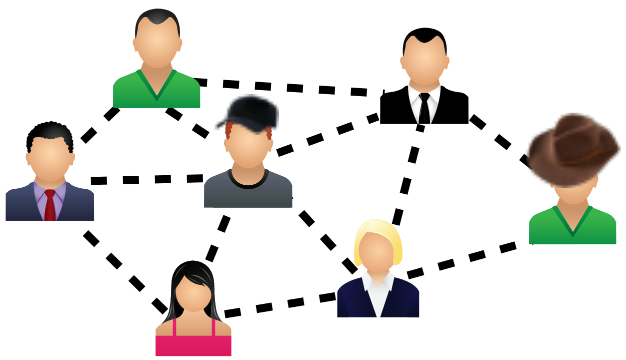
Income Distribution



[1]CIA - The World Factbook, **DISTRIBUTION OF FAMILY INCOME - GINI INDEX**, *https://www.cia.gov/library/publications/the-world-factbook/rankorder/2172rank.html*

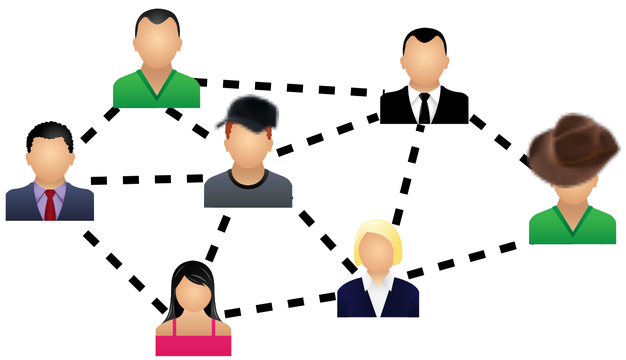
What about Software?

What about Software?



Developers = Population

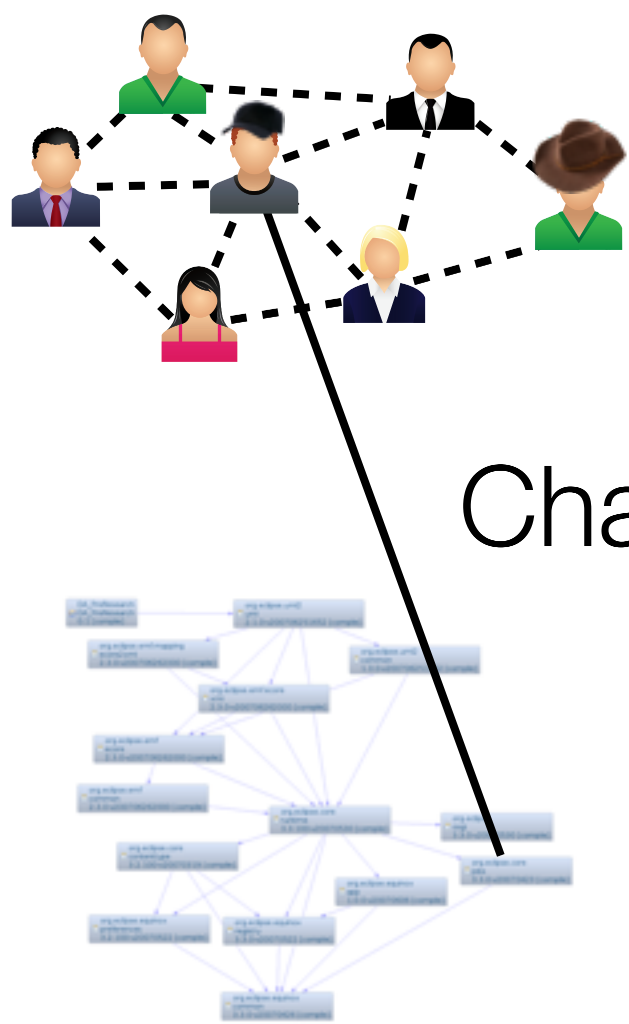
What about Software?



Developers = Population

Files = Assets

What about Software?

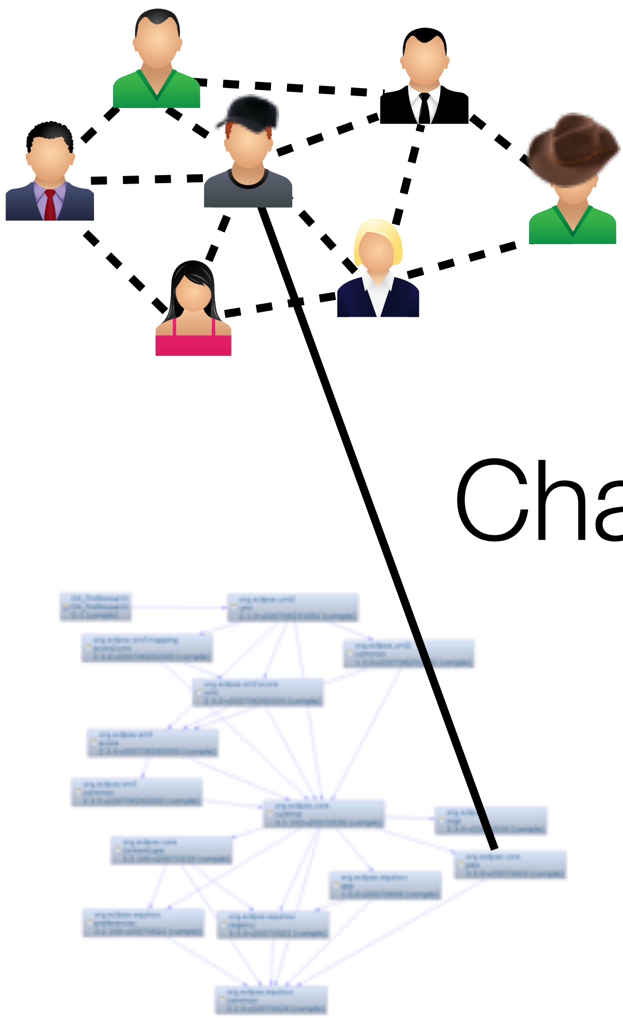


Developers = Population

Changing a file = “being owner”

Files = Assets

What about Software?



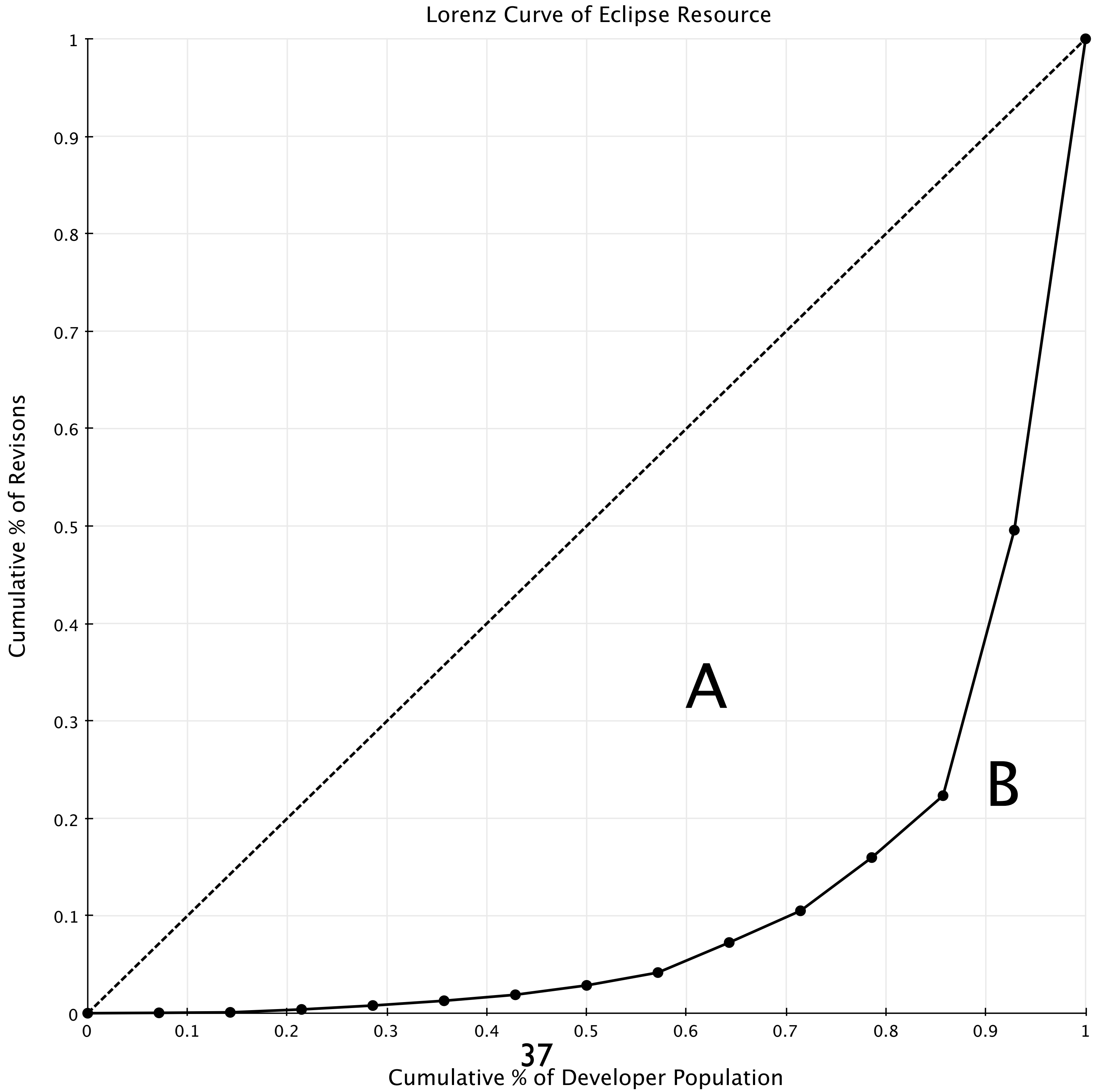
Developers = Population

Changing a file = “being owner”

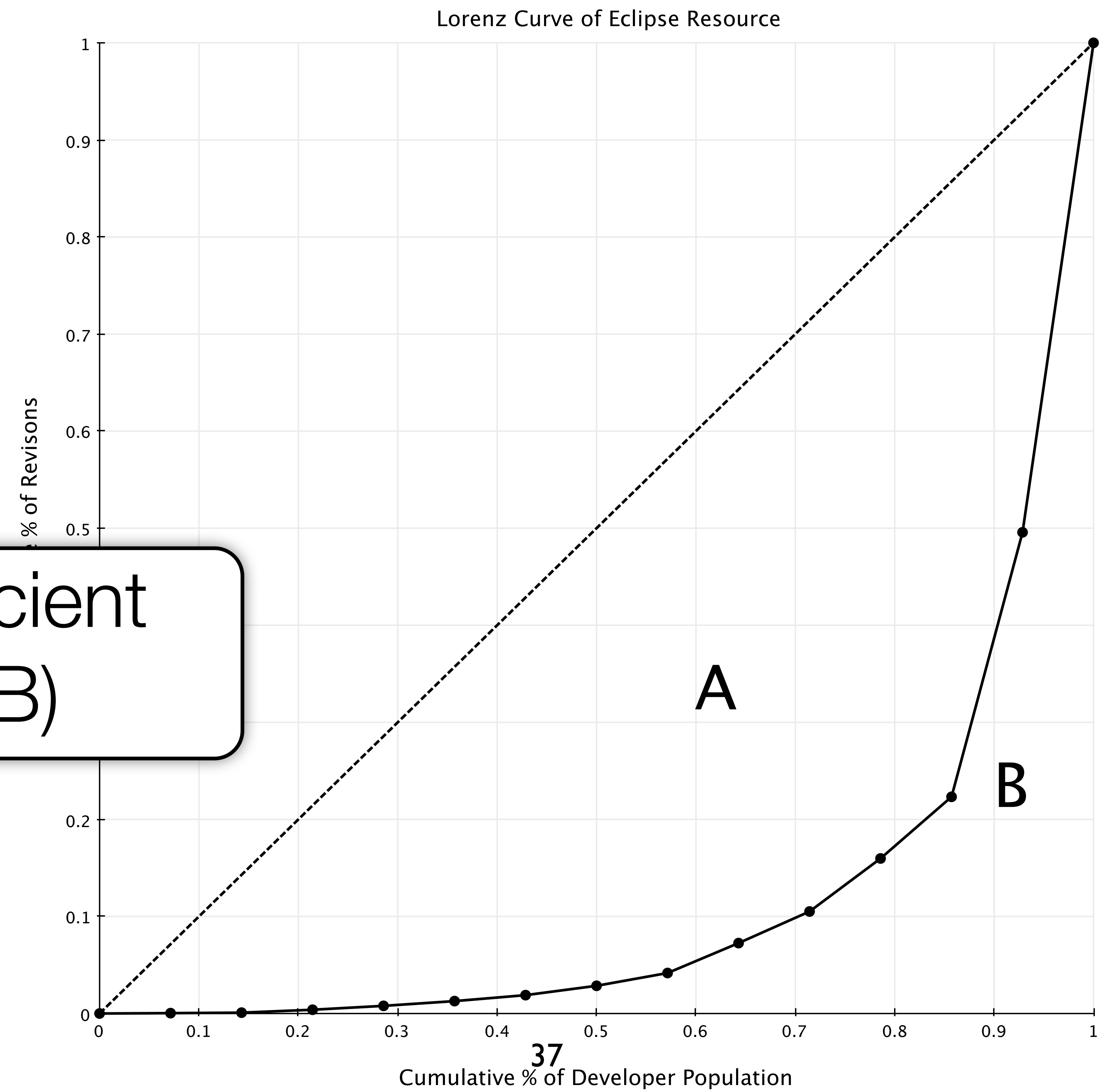
Files = Assets

How are changes of a file distributed among the developers and how does this relate to bugs?

Eclipse Resource



Eclipse Resource



Gini Coefficient
= A / (A + B)

Study

- Eclipse Dataset
- Avg. Gini coefficient is 0.9
- Namibia has a coefficient of 0.7
- Negative Correlation of ~-0.55
- Can be used to identify bug-prone files

Study

- Eclipse Dataset
- Avg. Gini coefficient is 0.9
- Namibia has a coefficient of 0.7
- Negative Correlation of ~-0.55
- Can be used to identify bug-prone files

The more changes of a file are done by a few dedicated developers the less likely it will be bug-prone!

Economic Phenomena

- Economic phenomena of code ownership
- Economies of Scale (Skaleneffekte)
- I'm an expert (in-depth knowledge)
- Profit from knowledge

Economic Phenomena

- Economic phenomena of code ownership
- Economies of Scale (Skaleneffekte)
- I'm an expert (in-depth knowledge)
- Profit from knowledge

Costs to acquire knowledge can be split, e.g., among several releases if you stay with a certain component

Diseconomies of Scale

- Negative of effect of code ownership?
- Loss of direction and co-ordination
- Are we working for the same product?

Another Phenomena

- Economies of Scope (Verbundseffekte)
- Profiting from breadth-knowledge
- Knowledge of different components helps in co-ordinating
- Danger of bottlenecks!

Implications & Conclusions

- How much code ownership & expertise?
- What is your bus number?
- What is better? In-depth- or breadth-knowledge?
- What' is the optimal team size?

Promises & Perils of Defect Prediction

- There are many excellent approaches that reliably locate defects
- Deepens our understanding how certain properties of software are (statistically) related to defects
- X-project defect prediction is an open issue
- Much of it is pure number crunching, i.e., correlation != causality
- Assess practical relevance of defect prediction approaches

Cross-Project Defect Prediction

- Use a prediction model to predict defect in other software projects
- Study with open source systems (e.g. Eclipse, Tomcat) and MS product (e.g., Win-Kernel, Direct X, IE)
- Results: Only limited success
- Another example of how difficult it is in SE to find generally valid models

Cross-project Defect Prediction

A Large Scale Experiment on Data vs. Domain vs. Process

Thomas Zimmermann
Microsoft Research
tzimmer@microsoft.com

Nachiappan Nagappan
Microsoft Research
nachin@microsoft.com

Harald Gall
University of Zurich
gall@ifi.uzh.ch

Emanuel Giger
University of Zurich
giger@ifi.uzh.ch

Brendan Murphy
Microsoft Research
bmurphy@microsoft.com

ABSTRACT

Prediction of software defects works well within projects as long as there is a sufficient amount of data available to train any models. However, this is rarely the case for new software projects and for many companies. So far, only a few have studies focused on transferring prediction models from one project to another. In this paper, we study cross-project defect prediction models on a large scale. For 12 real-world applications, we ran 622 cross-project predictions. Our results indicate that cross-project prediction is a serious challenge, i.e., simply using models from projects in the same domain or with the same process does not lead to accurate predictions. To help software engineers choose models wisely, we identified factors that do influence the success of cross-project predictions. We also derived decision trees that can provide early estimates for precision, recall, and accuracy before a prediction is attempted.

Categories and Subject Descriptors. D.2.8 [Software Engineering]: Metrics—*Performance measures, Process metrics, Product metrics.* D.2.9 [Software Engineering]: Management—*Software quality assurance (SQA)*

General Terms. Management, Measurement, Reliability.

1. INTRODUCTION

Defect prediction works well if models are trained with a sufficiently large amount of data and applied to a single software project [26]. In practice, however, training data is often not available, either because a company is too small or it is the first release of a product, for which no past data exists. Making automated predictions is impossible in these situations. In effort estimation when no or little data is available, engineers often use data from other projects or companies [16]. Ideally the same scenario would be possible for defect prediction as well and engineers would take a model from another project to successfully predict defects in their own project; we call this *cross-project defect prediction*. However, there has been only little evidence that defect prediction works across projects [32]—in this paper, we will systematically investigate when cross-project defect prediction does work.

ESEC/FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

The specific questions that we address are:

1. To what extent can we use cross-project data to predict post-release defects for a software system?
2. What kinds of software systems are good cross-project predictors—projects of the same domain, or with the same process, or with similar code structure, or of the same company?

Considering that within companies, the process is often similar or even the same, we seek conclusions about which characteristics facilitate cross-project predictions better—is it the same domain or the same process?

To test our hypotheses we conducted a large scale experiment on several versions of open source systems from Apache Tomcat, Apache Derby, Eclipse, Firefox as well as seven commercial systems from Microsoft, namely Direct-X, IIS, Printing, Windows Clustering, Windows File system, SQL Server 2005 and Windows Kernel. For each system we collected code measures, domain and process metrics, and defects and built a defect prediction model based on logistic regression. Next we ran 622 cross-projects experiments and recorded the outcome of the predictions, which we then correlated with similarities between the projects. To describe similarities we used 40 characteristics: code metrics, ranging from churn [23] (i.e., added, deleted, and changed lines) to complexity; domain metrics ranging from operational domain, same company, etc; process metrics spanning distributed development, the use of static analysis tools, etc. Finally, we analyzed the effect of the various characteristics on prediction quality with decision trees.

1.1 Contributions

The main contributions of our paper are threefold:

1. Evidence that it is not obvious which cross-prediction models work. Using projects in the same domain does not help build accurate prediction models. Process, code data and domain need to be quantified, understood and evaluated before prediction models are built and used.
2. An approach to highlight significant predictors and the factors that aid building cross-project predictors, validated in a study of 12 commercial and open source projects.
3. A list of factors that software engineers should evaluate before selecting the projects that they use to build cross-project predictors.

Promises & Perils of Defect Prediction

- There are many excellent approaches that reliably locate defects
- Deepens our understanding how certain properties of software are (statistically) related to defects
- Cross-project prediction is an open issue
- Much of it is pure number crunching, i.e., correlation != causality
- Assessment of the practical relevance of defect prediction approaches