CS 4530 & CS 5500 Software Engineering

Lesson 12.2: Metrics in Software Engineering

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

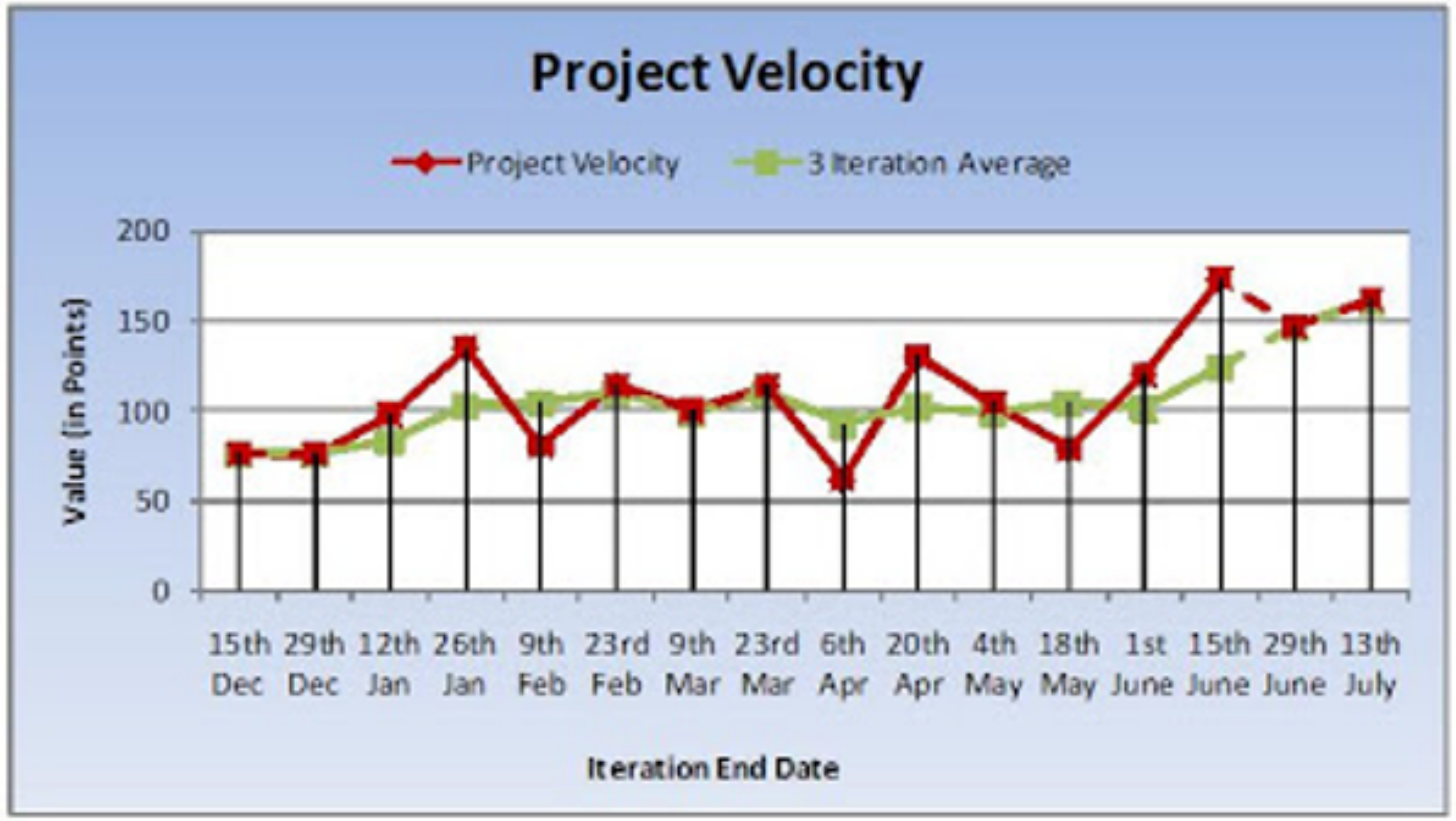
By the end of this lesson, you should be able to…

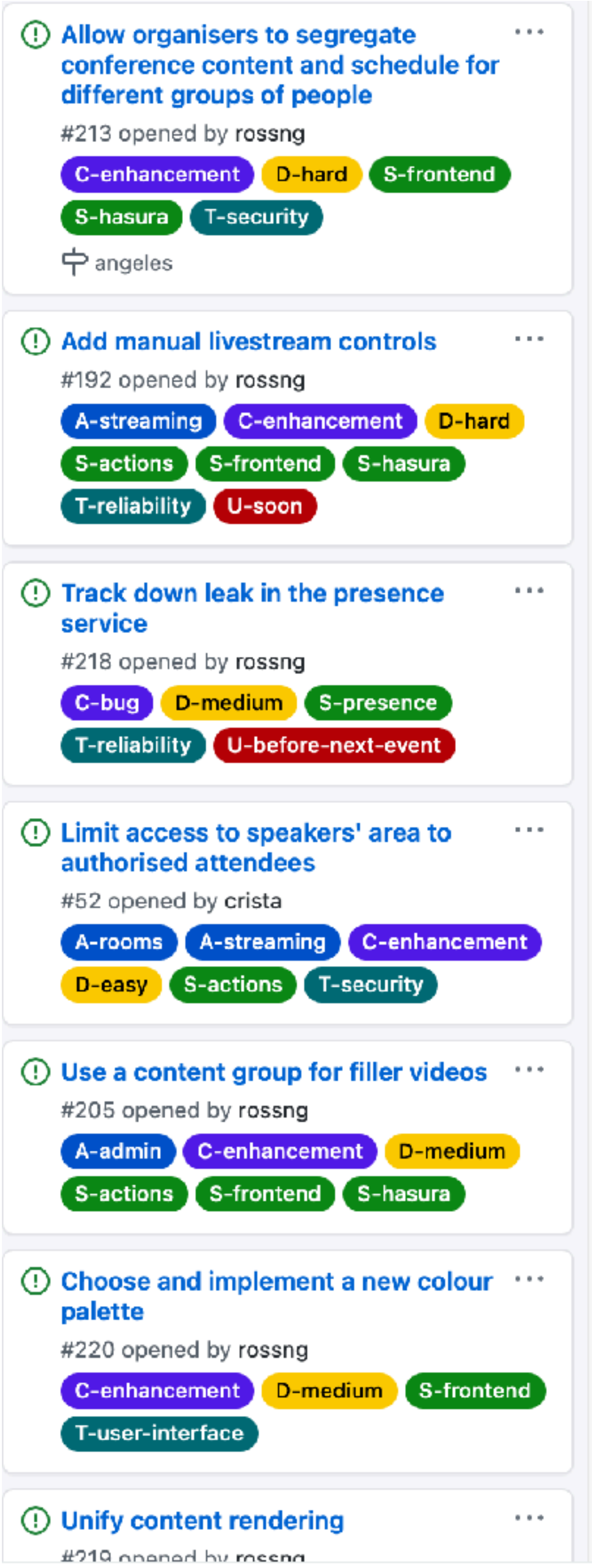
- Recognize commonly used metrics in software development
- Understand limitations and dangers of making development decisions based on metrics

Tracking Development Progress

Sprint Velocity

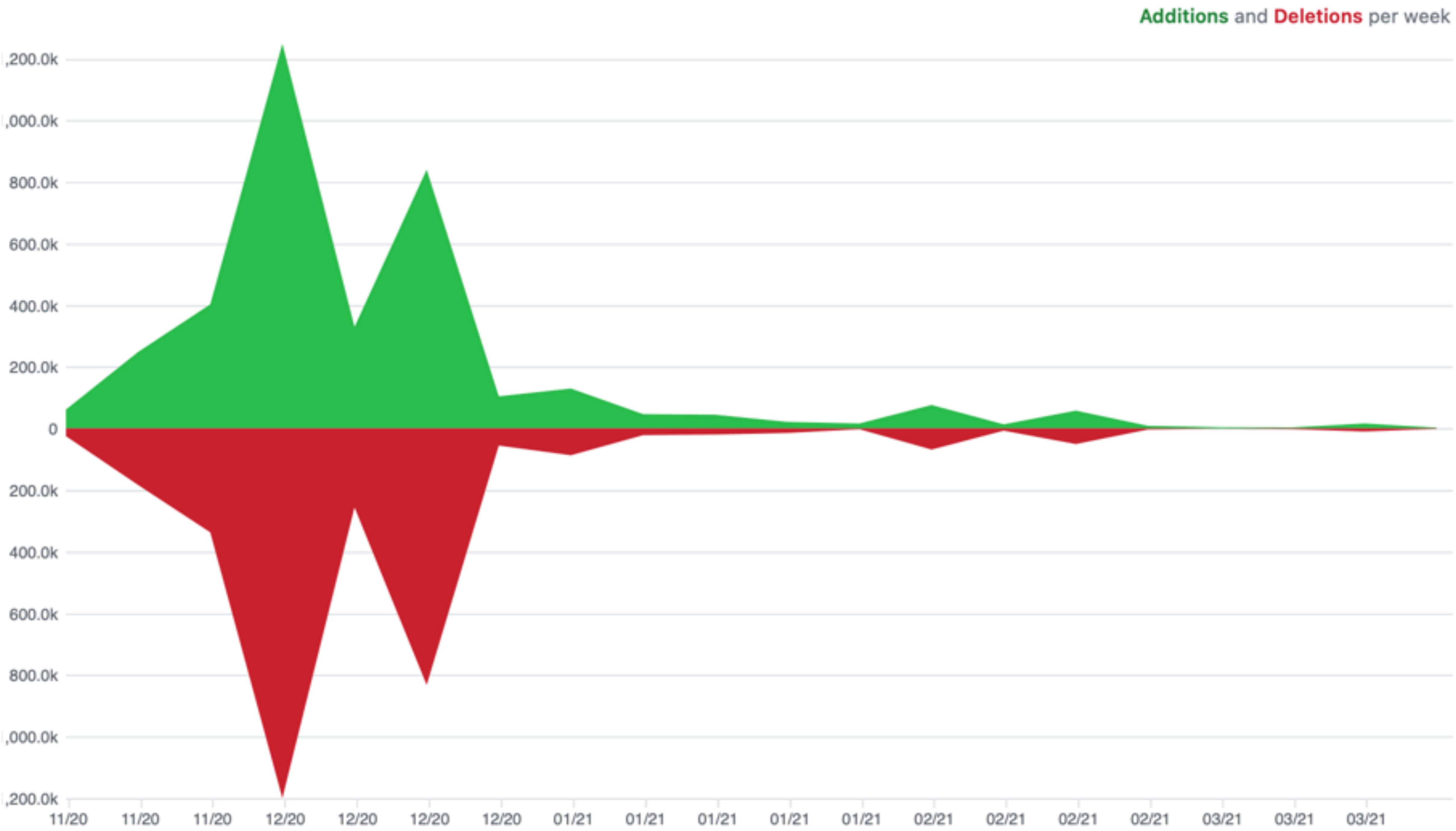
$$\text{Sprint Velocity} = \frac{\sum \text{Story Points Completed}}{\sum \text{Story Points Planned}}$$





Tracking Development Progress

Metric: Lines of Code



Tracking Development Progress

Metrics: Bugs open/closed, tests passing/failing

148 Active Issues

39	109
Closed Issues	New Issues

Status	Pipeline	Triggerer	Commit	Stages
passed	#277240683 latest		jna-getCall… 0ea5afd1 Tentative fix for #165	
passed	#271493675 latest		master a7df9e1e Add our own snapshots re...	
failed	#271413002		master 2e57a6f2 Clean up class loading in i...	
passed	#253308611		master e5e105be Merge remote-tracking br...	
passed	#248929992		master 498ee364 #byetravis	
passed	#248928953 latest		gitlab-setup d3494eb3 .	
failed	#248927900		gitlab-setup 4226cc2c .	
failed	#248927010		gitlab-setup 3169b89f .	
failed	#248924955		gitlab-setup dba4818f .	
failed	#248923860		gitlab-setup 39f8d395 .	

McNamara Fallacy

Reflecting on Vietnam decision making

- Measure whatever can be easily measured
- Disregard that which cannot be measured easily
- Presume that which cannot be measured easily is not important
- Presume that which cannot be measured easily does not exist

Software Metric: McCabe Cyclomatic Complexity

Is this code complex to understand?

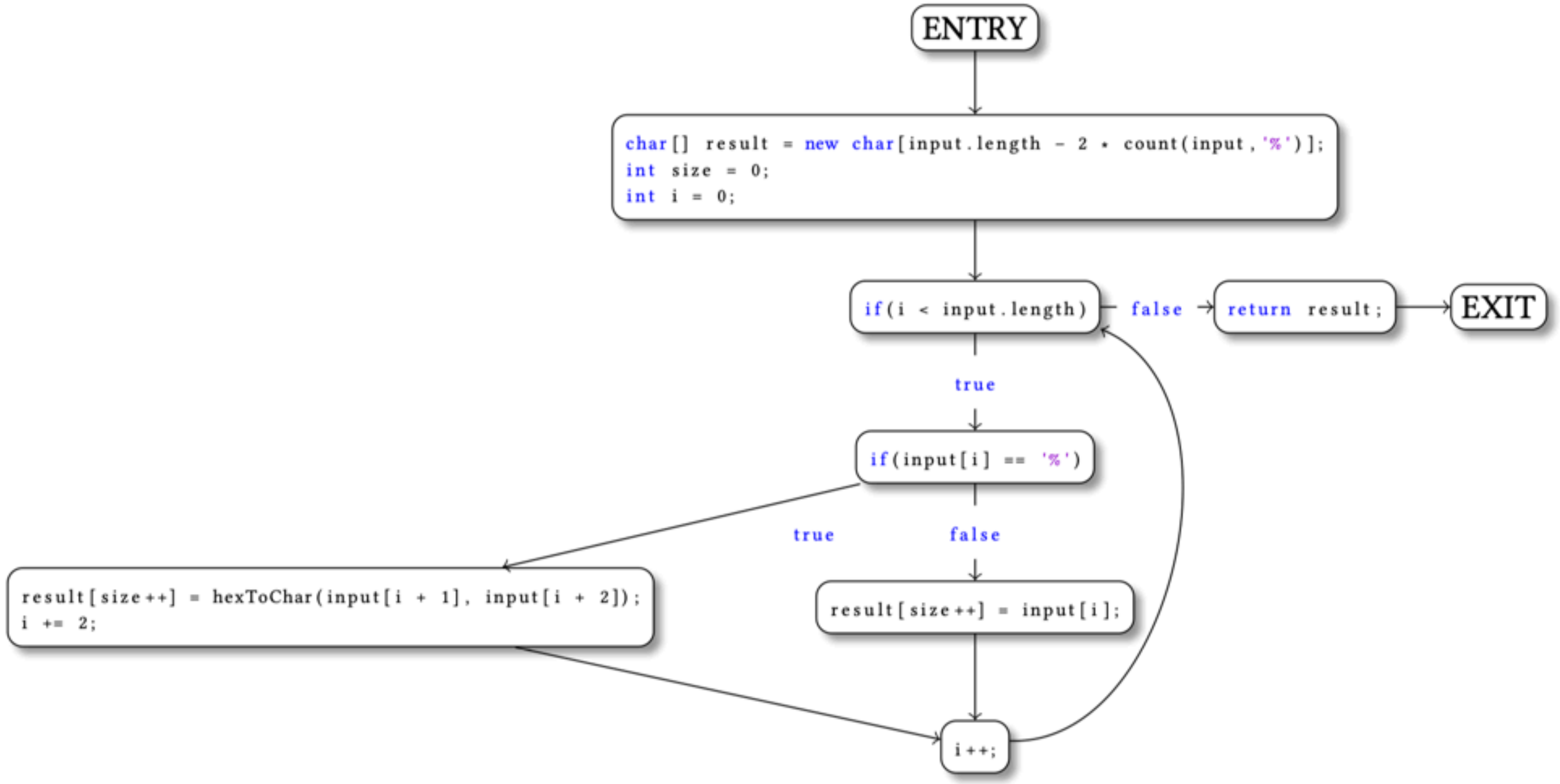
```
public static char[] percentDecode(char[] input) {
  char[] result = new char[input.length - 2 * count(input, '%')];
  int size = 0;
  for(int i = 0; i < input.length; i++) {
    if(input[i] == '%') {
      result[size++] = hexToChar(input[i + 1], input[i + 2]);
      i += 2;
    } else {
      result[size++] = input[i];
    }
  }
  return result;
}
```

Input: “`Hello%20World`”

Output: “`Hello World`”

Software Metric: McCabe Cyclomatic Complexity

Is this code complex to understand?



M = E − N + 2P

M = 10 − 9 + 2•1

M = 3

Is this good?

Software Metric: McCabe Cyclomatic Complexity

What does this value mean?

A critique of cyclomatic complexity as a software metric

by Martin Shepperd

McCabe's cyclomatic complexity metric is widely cited as a useful predictor of various software attributes such as reliability and

1 Introduction

The need for some objective measurement of software

Table 1 Empirical validations of cyclomatic complexity

Researchers	LOC	Errors *density*	Errors *absolute*	Programming effort	Bug location	Program recall	Design effort
Basili (Ref. 38)	r^2=0.94	r is –ve					
Basili (Ref. 39)				R=0.48	R=0.21		
Bowen (Ref. 40)			r^2=0.47		r= –0.09		
Curtis (Ref. 41)	r^2=0.41,0.81,0.79					r= –0.35*	
Curtis (Ref. 42)	r^2=0.81,0.66				r^2=0.4,0.42		
Davis (Ref. 43)						r is –ve, +ve	
Feuer (Ref. 44)	r^2=0.90***						
Gaffney (Ref. 45)				r^2=0.60			
Henry (Ref. 46)	r^2=0.84***		r^2=0.92****				
Kitchenham (Ref. 47)	r^2=0.86,0.88		r^2=0.46,0.49,0.21****				
Paige (Ref. 48)	r^2=0.90						
Schneiderman (Ref. 49)	r^2=0.61*****		r^2=0.32*****				
Shen (Ref. 50)			r^2=0.78***				
Sheppard (Ref. 51)	r^2=0.79			r^2=0.38		r=0.35	
Sunohara (Ref. 52)				r^2=0.4,0.38			r^2=0.72,0.7
Wang (Ref. 53)	r^2=0.62			r^2=0.59			
Woodfield (Ref. 54)				r^2=0.26,R=0.39			
Woodward (Ref. 22)	r^2=0.90						

r^2 = Pearson moment R = Rank Spearman

* r was 'improved' when modified for potentially 'aberrant' results

** correlated with N (i.e. Halstead's token count)

*** a simple decision count (i.e. $v(G)$–1)

**** indirect error count (i.e. version count), or program change count

***** using log-log transformations

2016 IEEE International Conference on Software Quality, Reliability and Security

Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density

Kazuhiro Yamashita*, Changyun Huang*, Meiyappan Nagappan†, Yasutaka Kamei*, Audris Mockus§, Ahmed E. Hassan†, and Naoyasu Ubayashi*
* Kyushu University, Japan; {yamashita, huang}@posl.ait.kyushu-u.ac.jp, {kamei, ubayashi}@ait.kyushu-u.ac.jp
† Queen's University, Canada; ahmed@cs.queensu.ca
‡ Rochester Institute of Technology, USA; mei@se.rit.edu
§ University of Tennessee-Knoxville, USA; audris@utk.edu

***Abstract*—Practical guidelines on what code has better quality are in great demand. For example, it is reasonable to expect the most complex code to be buggy. Structuring code into reasonably sized files and classes also appears to be prudent. Many attempts to determine (or declare) risk thresholds for various code metrics have been made. In this paper we want to examine the applicability of such thresholds. Hence, we replicate a recently published technique for calculating metric thresholds to determine high-risk files based on code size (LOC and number of methods), and complexity (cyclomatic complexity and module interface coupling) using a very large set of open and closed source projects written primarily in Java. We relate the threshold-derived risk to (a) the probability that a file would have a defect, and (b) the defect density of the files in the high-risk group. We find that the probability of a file having a defect is higher in the very high-risk group with a few exceptions. This is particularly pronounced when using size thresholds. Surprisingly, the defect density was uniformly lower in the very high-risk group of files. Our results suggest that, as expected, less code is associated with fewer defects. However, the same amount of code in large and complex files was associated with fewer defects than when located in smaller and less complex files. Hence we conclude that risk thresholds for size and complexity metrics have to be used with caution if at all. Our findings have immediate practical implications: the redistribution of Java code into smaller and less complex files may be counterproductive.**

***Index Terms*—Software metrics; Thresholds; Defect models;**

I. Introduction

There has been a considerable amount of research in the field of software defect prediction, especially in the last decade. Over 100 papers have been published just from 2000 - 2011 in this area of Empirical Software Engineering (ESE) research alone [16, 27]. The primary goal of such research is to be able to provide guidelines to practitioners on what kind

metrics themselves [11, 14], error models [6, 10, 26], cluster techniques [22, 34], and metric distributions [31, 32].

Almost all of these approaches treat extreme values of file metrics as detrimental. Defect prediction approaches that use a linear statistical model, conclude that an extreme value of the metrics implies poor quality of code, by virtue of choosing such a model. For example, 'larger files will have more defects' is a common refrain in ESE research [15]. Alternatively the "Goldilocks Principle" suggests that extreme values of a metric are a sign of poor quality code [17, 18]. Fenton and Neil provide a more comprehensive list of research studies with respect to metrics based defect prediction [12]. They find that the literature has contradictory evidence regarding the relationship between software defects and software metrics like size and complexity. It is, thus, unclear if metric thresholds should be used to identify source code files that are at high risk.

Consequently, we aim to observe *if a consistent relationship between metric thresholds and software quality* is present in OSS and industrial projects. Therefore we conduct a case study on three OSS and four industrial projects. The metrics that we choose to evaluate are size based (Total LOC in a file, and Module interface size in a file), and complexity based (cyclomatic complexity and module inward coupling). We evaluate software quality using two criteria - (a) defect proneness (probability of a file having a defect), and (b) defect density (the number of defects/LOC).

We replicate the most recently published state-of-the-art technique (proposed by Alves *et al.* [3]) to determine the thresholds for these metrics, and found them to be very close to ones reported earlier. In order to use this approach, we need a set of projects to calculate the thresholds from. In

Goodhart’s Law

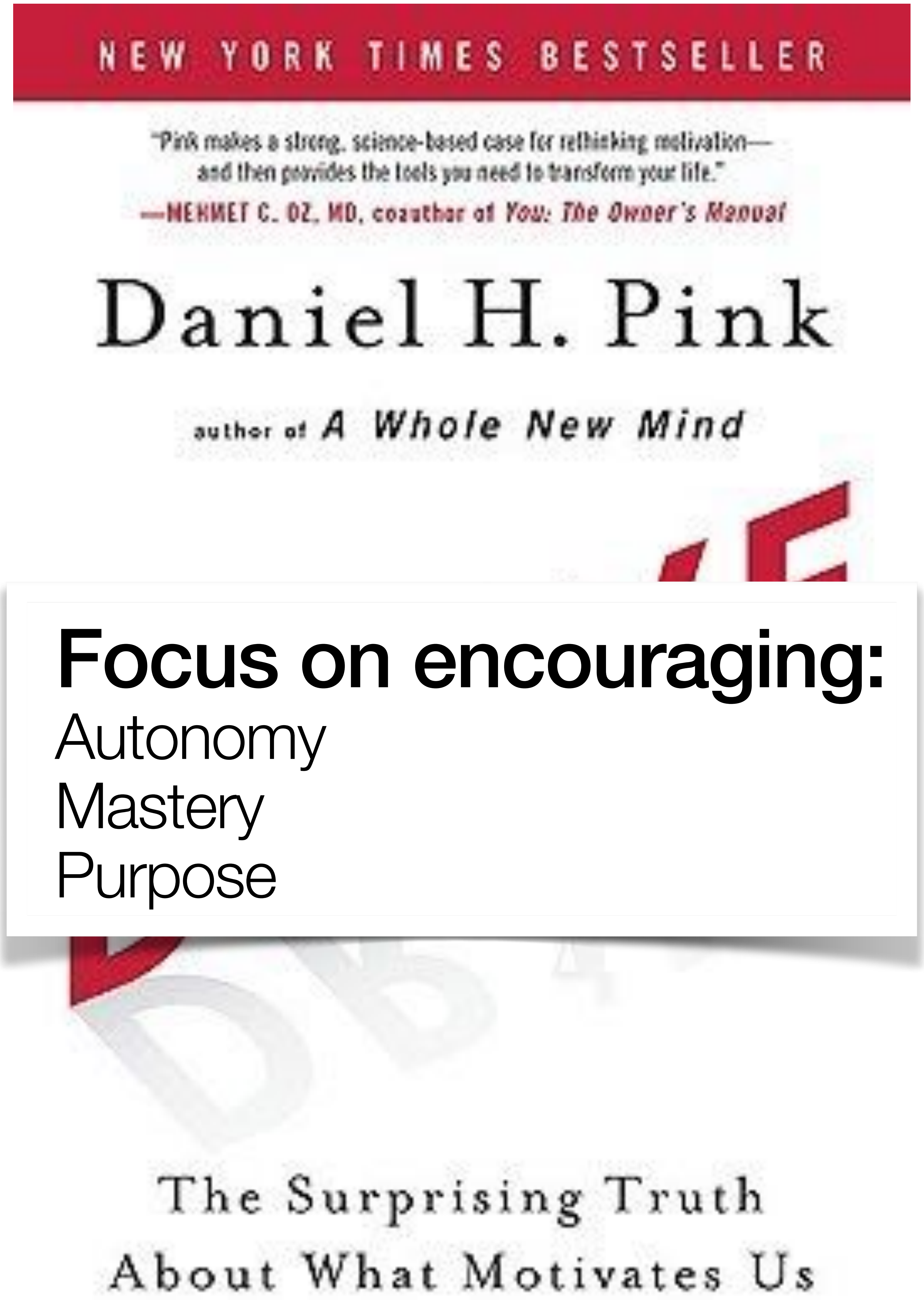
When a measure becomes a target, it ceases to be a good measure

Productivity Metrics

Intrinsic & Extrinsic Motivations

Extrinsic rewards:

Can extinguish intrinsic motivation
Can diminish performance
Can crush creativity
Can crowd out good behavior
Can encourage cheating, shortcuts, and unethical behavior
Can become addictive
Can foster short-term thinking



Focus on encouraging:

Autonomy
Mastery
Purpose