Chapter XXII
Motives and Methods for Quantitative FLOSS Research

Megan Conklin
Elon University, USA

ABSTRACT

This chapter explores the motivations and methods for mining (collecting, aggregating, distributing, and analyzing) data about free/libre open source software (FLOSS) projects. It first explores why there is a need for this type of data. Then the chapter outlines the current state-of-the art in collecting and using quantitative data about FLOSS project, focusing especially on the three main types of FLOSS data that have been gathered to date: data from large forges, data from small project sets, and survey data. Finally, the chapter will describe some possible areas for improvement and recommendations for the future of FLOSS data collection.

INTRODUCTION

Numbers, statistics, and quantitative measures underpin most studies of free/libre open source (FLOSS) software development. Studies of FLOSS development usually require the researchers to have answered questions like: How many FLOSS projects are there? How many developers? How many users? Which projects are dead, which are flourishing? What languages are popular for development? How large are development teams, and how are these teams structured?

These questions are fun to answer in the context of FLOSS development because project teams are self-organized, widely-distributed geographically, and use many different programming languages and software development methodologies. Teams are organized in an ad hoc, decentralized fashion. Projects can be very hard to track, and changes can be difficult to follow. Developers primarily use the Internet for communication, and teams are organized around the idea that anyone can contribute. Since the organization of the teams is done via the Internet and since the source code is open for anyone to view, it may seem as though data about these projects is as open as the projects themselves.

This is in direct contrast to the way proprietary projects are most often structured, and consequently, data about proprietary projects are collected and analyzed in a different way. Empirical software engineering researchers have, in the past,

typically used metrics from a single company or a single proprietary project. This data was collected systematically and distributed in a tightly controlled manner, consistent with the proprietary nature of the software being developed. Whereas data analysis about proprietary software practices was primarily a problem of scarcity (getting access and permissions to use the data), collecting and analyzing FLOSS data becomes a problem of abundance and reliability (storage, sharing, aggregation, and filtering of the data).

Thus, this chapter will explore the motivations and methods surrounding the mining of FLOSS data, specifically how and why the collection, aggregation, distribution, and analysis of this data takes place. We will first discuss motives: why does software engineering research rely on metrics at all, and why do we need FLOSS metrics in particular? We will then study methods: what is the current state-of-the-art in FLOSS data mining? Finally, we note some possible future trends, and propose some general recommendations for measuring FLOSS projects quantitatively.

BACKGROUND

Importance of Metrics to Software Engineering

The collection and aggregation of real-world and historical data points are critical to the task of measurement in software engineering. Quantitative and empirical approaches to software engineering require real-world data; for example, the branch of software engineering concerned with estimation will use empirical or historical data to seed the estimate calculation. More generally, the four reasons for measuring software creation processes are commonly listed as a characterization, evaluation, prediction, or improvement on these processes (Park, Goethert, & Florac, 1996). All of these goals require useful data (measurements) in order to be carried out effectively. Interesting measures of the software process can vary depending on the goals of the research, but they could include things like the number of errors in a particular module, the number of developers working in a particular language or development environment, or the length of time spent fixing a particular code defect (Yourdon, 1993). The collection domain of a research project will differ as well; measures can be collected for a group of products, a group of developers, a single software product, a single release of a software project, or even for a single developer.

The empirical software engineering literature is replete with examples of how gathering metrics about projects can lead to important insights. Software engineering metrics can be used to avoid costly disasters, efficiently allocate human and financial capital, and to understand and improve business processes. One famous example of a software error that caused significant financial and property damage was the European Ariane 5 flight 501 disaster of 1996 (Jezequel & Meyer, 1997). The European rocket crashed 40 seconds after liftoff, reportedly due to an error in the way software components were reused within the system. This was a US$500 million software engineering error. In 1975, Fred Brooks made famous another software engineering debacle: the management of the IBM OS/360 project (Brooks, 1975). His conclusions about the inefficiencies in the way programmers were added to the development team became known as *Brooks' Law*, and this remains one of the tenets of software engineering practice to this day. Using metrics about team composition, communication, and productivity, Brooks concluded that work done by a set of programmers will increase linearly as programmers are added to a project, but communication and coordination costs will rise exponentially. Brooks' Law is most often remembered as: "adding manpower to a late project makes it later."

There are hundreds of these examples in the software engineering literature about metrics in proprietary projects, but where are the metrics