Automated monitoring of software quality*

by J. A. CLAPP and J. E. SULLIVAN

The Mitre Corporation
Bedford, Massachusetts

THE PROBLEM

Widespread acceptance of computer systems for commercial and military applications has led to widespread dissatisfaction with the software components of these systems. Major criticisms have focused on the high cost of development and maintenance, the slippages in delivery dates, and the poor quality of software. Those of us who produce software readily admit to these complaints since we ourselves have to rely on software to help us do our job and have experienced the same problems. We are equally ready to acknowledge that the solution to these problems lies in better management of software development, but we quickly claim that software development is harder to manage than other kinds of production processes.

We know from experience that it is notoriously difficult to determine, prior to a software development project, just what product and product quality to expect with given resources, or vice versa, what resources will be needed for a given product. Even while a project is in progress, we do not know how to determine just what kind of product is emerging, how far along it is, and therefore how to project the (perhaps all too visible) resource expenditures that have already occurred to the final cost and time, so that plans can be altered and adjustments made. The result is that it is easy to be overambitious, and not even realize it until it is too late to do anything but sacrifice quality or commit still more time and money, and hope for the best.

Not surprisingly, such cost and schedule overruns—in conjunction with the poor product that is the usual result of such a badly controlled process—come to be regarded by customers as exemplifying the “high cost of software.” But the notion that the cost is high is really only a conjecture; all that is known for sure is that, for most large, complex software systems, we do not know how to estimate cost, the proper relation of cost to quality, or even what quality is, and are therefore frustrated in our attempts to manage.

The problem comes down to a lack of generally applicable measures and measure-relating theorems that are useful to the manager in planning and controlling software development. Hoare,1 in calling for a “software engineering discipline,” and Boehm,2 commenting on the factors of software costs, have said the same thing: we lack quantitative data and a systematic body of knowledge that will allow us to bring experience gained in one area to bear on a new situation in another area.

WHAT WE HOPE TO DO ABOUT IT

Our objective is to provide measures that give clear and current visibility to a software development project so that managers can know the real status of the software at all times and spot potential trouble in time to do something about it, and programmers can be made aware of the effects of their work. Making measurements implies the collection of data. It is often impractical or impossible to collect the necessary data manually because it is too costly or the data lacks the consistency and reliability which is required. However, the production of software has a distinct advantage over other kinds of production: much of the actual work takes place within a computer, and thus status information as well as data about the characteristics of the emerging product can be easily captured automatically and analyzed.

For these reasons, we decided to design and implement an automated software implementation monitor, called Simon, which extracts measures continuously throughout a software development project, stores data in a centralized data base, and provides analyses of the data to managers and programmers. In short, Simon's direct purpose is to provide project visibility in a practical, that is to say automatic way.

In the long run, over many projects, the data gathered by Simon should become a part of our experience base, to support research into software quality and cost, and the causes thereof. Simon's contribution is to automate, control, and standardize the process of data collection and analysis. Equipped with observations and measures, we can relate pre- and post-implementation costs and final product quality to controllable variables during development.

WHAT IS SIMON?

Overview

Simon is an automated aid that is integrated into a conventional programming environment in the sense that it runs under the Operating System and invokes existing tools...
such as a text editor, compiler, debugging tool, etc. In general, any transactions between the software builders and the computer are filtered through Simon to provide the following advantages: (1) Simon can monitor what is being done, by whom, and when; (2) standard procedures can be enforced; and (3) additional information can be requested of a user which would not be required by the operating system or other facility services. The data gathered before, during, and after a transaction is recorded in a data base which is structured to permit analyses to be performed and reports generated about any aspects of the data which are of interest.

Monitors are now in use which collect data about software automatically but these tend to serve specialized purposes, such as analyzing software performance characteristics, or hardware resource usage. In a few cases, monitors maintain a data base about the characteristics of the software under development. For example, the Automated Software Evaluation System collects such data but limits its analyses to supporting software evaluation and validation. Still other software packages analyze and report on cost data to show managers where estimates and actual values deviate. All of these capabilities and others would also be incorporated into Simon in the belief that the whole can be greater than the sum of its parts.

The design of Simon is open-ended so that it can be adapted to different programming environments and extended as we gain experience in its use and as new techniques for software development emerge.

At the time of this writing, only a general design of Simon has been completed. A refinement of this design (in top-down fashion, naturally) will lead to the development of a version of Simon tailored to our particular programming environment and the measures of current interest to us. The description of Simon which follows is, of necessity, quite general. It is intended to convey the important concepts in its design and the kinds of capabilities it might offer, in order to provide a framework for future implementations by ourselves or others.

The specific method of implementing Simon will naturally be dependent on what data is desired and can be captured from existing sources, and what data must be generated by Simon. Figure 1 illustrates a typical Simon configuration of four modules: a pre-compiler, a post-compiler, a module testing tool and a monitor. The information flow among these units, the units native to the operating system, and the managers and programmers is also outlined. This configuration will be used as the basis for the descriptions which follow.

Functions

Simon has four principal functions, viz:

1. Edit
2. Compile (Static Module Analysis)
3. Test (Dynamic Module Analysis)
4. Status Report

Editing is the direct entry by project personnel of Simon file data, including documentation and certain status data such as estimates, schedules and costs, as well as source code. As a side-effect of editing, in the case of significant products such as documents and program modules, simple measures are extracted to project whether and when the change process will converge.

Static module analysis is the extraction of information derivable from the source (or object) code without actually running it or simulating a run. This is the kind of information that is available at compile time (or at edit time, but as a practical consideration many compilers already provide much of the information we want). Such information includes: file and variable set/used lists (concordances), inter-module reference graphs, the several measures of module complexity and data element complexity, independent path analysis (for testing), and static resource requirements (e.g., module memory size).

Dynamic module analysis is the extraction of information normally available only when the module is running—i.e., an instrumented test. This includes paths tested and success/failure data (for reliability estimation) and dynamic resource use (e.g., execution time).

Status reporting, the heart of the matter, is the systematic dissemination of summaries of the information collected, together with projections based on that information (including estimates and budgets). This implies that all pertinent information about a project is maintained in central files. There are in fact three basic kinds of files in Simon: (1) entity files, (2) relational files, and (3) event files.

Data base

Entity files track the current status of all entities thought to have a significant bearing on the project. An entity is any item that has duration in time, and properties that change over time without disturbing the basic unity of the entity. There are eight entity classes:

1. the project itself
2. people (programmers and managers)
3. program modules
4. documents
5. system tests
6. macros
7. files and external variables
8. programming errors

The current values of properties are recorded for each instance of an entity. "Project" properties are those not specifically related to other entities individually or in small combinations—e.g., the overall schedule, and overhead budget and cost entries. The properties of people are generally those that have relevance to a given hypothesis under study—e.g., year of entry into the computing profession, productivity, or current number of simultaneous projects. The most extensive data will be kept on module properties: the source code itself, a short description and interface specification, estimated and current length and complexity, current computed reliability and so forth. For documents, the text and some simple measures such as number of outlined vs. completed sections are maintained. System tests have specifications that go through stages of development; thus their properties are the status of this specification and also the status of the system with respect to passing or failing the test. Macros are centralized source language fragments or forms; depending on the environment, COMPOOL entries or manually inserted COMMON statements serve the same purpose. Macros, files and external variables are usually characterized only by their definition and brief description. Finally, programming errors are "entities" of interest in software research. To go into some detail in this one case, the following properties are typically kept for each error:

1. time found (symptoms appeared)
2. time debugged
3. time fixed
4. classification by type(s)
5. method of discovery (code reading, diagnostic, test, etc.)
6. mental level of error (motor, memory, logic)
7. when made (original code, change, adding instrumentation, etc.)
8. direct costs (man-hours, machine time) of this error.

Relational files are lists that keep track of interrelationships between entities; in effect, they are extensions of the properties of the entities involved. For example, there is a file relating each module to the errors that occurred within it, and another file that keeps track of intermodule references (who-calls-whom).

Event files contain the time and complete details of all transactions that affect the status of the other two files. An event is thus any particular instance of one of the first three major functions—i.e., edit, compile or test. The purpose of the file is mainly historical, to be able to reconstruct a prior state, and also to permit computation of any current state descriptor not originally defined—in other words, to support hindsight research. For practical reasons, very little of this file is kept online.

**How Simon operates**

Corresponding to each Simon function and each type of transaction used in developing software, a dialog takes place between the user and the system to supply inputs to the data base and to cause requested actions to occur. To illustrate how combinations of data are collected to add to or modify different files during a compilation request, the following sequence of actions might take place:

1. The user is requested to specify a reason for the compilation, e.g., to add instrumentation, or to fix an error. In the case of an error, he may be requested to supply data about the error if it has not been previously recorded, or to cite the error(s) if already entered.
2. A preprocessor scans the source data and adds code for system macros referenced by the module. Other alterations might be made to the code to introduce standard run-time instrumentation. If so directed, the preprocessor also performs analyses of the source code and records the results in the module entity file, e.g., the complexity measure might be calculated, or forbidden forms or references detected.
3. The relational files are updated to show what macros are referenced, and what modules are called by this module.
4. The compiler is called with either a standard set of options or user-specified variations. The object code is directed to the data base. The compiler output is scanned to obtain, for entry into the entity and relational files, the object size, a list of external references set and used, and data about compiler-generated error messages, such as the number and severity of errors or the number of statements flagged.
5. The cost of compilation and current date and time are used to update budget and schedule information in the module entity file.
6. The fact of compilation, and reason therefor, and the time are entered into the online part of the event file; other details (e.g., the compiler-generated listing) are retained in the "paper" part of the event file.

**USES OF SIMON**

**Overview**

Research into management techniques is essentially the formulation and validation of hypotheses relating what can be done or seen at one stage of a project to the effects thereof at a later stage. Practical management is largely a matter of applying validated hypotheses, or theorems, to a particular case—although good managers rarely refrain from doing a bit of "research," formal or informal, while they're at it.

In the case of software, we have difficulty formulating hypotheses in the first instance because even the terms are
agree that modularity is a good thing, and that the lack of
and quality. Secondly, even widely-held hypotheses are
understood only vaguely. For example, everyone seems to
agree that modularity is a good thing, and that the lack of
quality is, and so it is not surprising that we cannot relate
cost to quality. Many hypotheses, some of them
widely publicized, seem to be in this category at present.
For example:

1. Structured Programming leads to greater compre-
hensibility and reliability.
2. Complexity (the inverse of comprehensibility) and the
cost of debugging are strongly covariant.
3. The programmer “inherent skill factor,” sometimes
said to have a 1-to-26 or even higher range, (is/is not)
really that important when properly separated from
other variables, (for example, learned work habits),
and (does/does not) manifest itself in comprehensi-
ability measures which can be taken before reliability
can be estimated.
4. Chief-programmer techniques lead to vanishing error
rates and lower development costs.

The list is potentially endless. And when it finally comes
down to validating any of these hypotheses, even when
properly understood, we are up against the same classic
problems faced by social scientists: (1) truly controlled experi-
ments of representative scale are impractical, and (2) even
just plain observations are lacking or not consistently based.
Simon should overcome this last problem, at least, bringing
us closer to the goal of useful measures and theorems in three
ways:

1. by hypotheses validation, as mentioned;
2. by consistent data collection, toward the formation
of better hypotheses; and
3. to the extent that current hypotheses are correct, by
providing a direct visibility aid to the managers and
programmers.

Direct use of Simon

As a simple example of how such a system can be a direct
aid to the manager, consider what can be learned from the
error data described earlier. At any given point, there are a
certain number of errors that have been observed, and some
smaller number that have been fixed. It is a simple matter
to fit a line through the points expressing total number vs.
time and another line through the number of corrected
errors vs. time and observe whether the lines cross (i.e., all
known errors become fixed) before the project deadline.
Such a method has been described by Coutinho. 4 Of course,
the validity of such a projection will depend on the assumed
model of the way in which errors are discovered and removed.
The model is an hypothesis that has to be proved, which
brings us back to the main purpose of Simon from our stand-
point, namely that of serving as a vehicle for measurement
definition, hypothesis validation and formation.

SIMON AS A TOOL FOR EXPERIMENT

An experiment

The same data on the number of errors vs. time used in
the prior example might be used to evaluate a testing tool
or technique by contrasting the plot for modules in which
one method of testing was employed with the plot for
modules in which it was not used. This is one of the techniques
which was used to evaluate FADEBUG-1. 4 All of the quanti-
tative measures of FADEBUG-1 could be derived easily
from the Simon data base. In this case, the hypothesis was
that a tool such as FADEBUG-1 would reduce the cost of
validating software.

Another experiment

We close with a description of an experiment that we plan
to conduct with Simon, and that will perhaps summarize its
purpose in most concrete terms. The hypothesis to be
examined is a refinement of the second one on our list, viz:
(a) length of code is roughly proportional to errors of clerical
level only; (b) complexity of code (in terms of independent
paths, a concept described in Sullivan 4) is roughly propor-
tional to errors of higher level; and (c) the latter account
for so much of the effort expended in debugging that com-
plicity is an approximate indicator of required debugging
effort, or when that effort is artificially constrained, un-
reliability.

1. A software project is selected which will be using the
programming environment in which Simon operates.
2. The data collection and analysis requirements for
Simon are determined based on the hypotheses. For
example, error data must distinguish clerical errors
from errors of a higher level, and the time spent de-
bugging an error must be collected from programmers.
3. Any analysis or collection routines not already avail-
able under Simon must be written.
4. The system is defined, designed and implemented in
the usual way, using Simon as the intermediary be-
tween programmers and the computer.
5. The hypothesis is evaluated: complexity does, or does
not, correlate well with the higher-level errors and
debugging costs (or projected costs to achieve 100
percent reliability) of the several modules. (Of course,
there is always the “gray area” of say, 80 percent
correlation—which leaves almost 40 percent of the
phenomenon to be explained.) Refinements or alter-
native hypotheses may be developed at this point,
fitted to the historical data and further evaluated on
other projects.

From the collection of the Computer History Museum (www.computerhistory.org)
CONCLUSIONS

Problems of software cost and quality can be solved by providing measures and measure-relating theorems which can be used by managers to plan and control software development. Currently, progress is hampered by the lack of data of the kind and amount needed to validate hypotheses about the definition of such measures, and their utility in controlling cost and quality. A software implementation monitor can aid in the validation of hypotheses and in the development of new insights by making it possible to collect and analyze current data about activities, costs, and product characteristics and their interrelationships.

REFERENCES
