Learning Early Lifecycle IV&V Quality Indicators

Tim Menzies, Justin S. Di Stefano, Mike Chapman
Lane Department of Computer Science, West Virginia University
PO Box 6109, Morgantown, WV, 26506-6109, USA
tim@menzies.com, justin@lostportal.net,Robert.M.Chapman@ivv.nasa.gov

Abstract

Traditional methods of generating quality code indicators (e.g. linear regression, decision tree induction) can be demonstrated to be inappropriate for IV&V purposes. IV&V is a unique aspect of the software lifecycle, and different methods are necessary to produce quick and accurate results. If quality code indicators could be produced on a per-project basis, then IV&V could proceed in a more straight-forward fashion, saving time and money. This article presents one case study on just such a project, showing that by using the proper metrics and machine learning algorithms, quality indicators can be found as early as 3 months into the IV&V process.

1 Introduction

IV&V refers to an external entity which is responsible for the verification and validation of software. Often, IV&V is something which developers scorn; they ask "Why can't testing be done in house? We developed the code, can't we test it?". Despite these objections to the IV&V process, the available evidence suggests that an independent review of software can find more errors than developers [1]. It allows a fresh set of eyes to go over a piece of software, looking for possible bugs and/or logical errors. One of the great problems with IV&V, however, is its' resource drain, in both time and money. By the time a product gets sent to an IV&V facility, the budget is fixed, and IV&V practitioners must struggle to add value to a project within tight budgetary constraints. To facilitate this procedure, some general guidelines have evolved which dictate how to quickly identify problematic parts of a system. We are concerned in this paper with the rapid generation of effective identifiers using static code metrics.

Static code metrics are measurements of the features of source code. That is, they are ways of summarizing a segment of code by certain identifying characteristics. Figure 1 is a short list of some of the most commonly collected metrics[1]. There are many different views on which metrics actually perform well as error-predictors; some say that a cyclomatic complexity over 10 is a good indicator, while others will argue that essential complexity over 4 is better [14, 9]. Still others insist that the best overall metric is the simplistic LOC (Lines of Code) [18]. Whatever the argument, though, everyone freely admits that these are merely guidelines, which may or may not apply to a particular project. In the ideal case, we would seek indicators that are specifically applicable to our particular project; a tailor-made flag for error-prone (or error-free) code.

Metric Type	Metric	Definiton
McCabe	v(G)	Cyclomatic Complexity
	ev(G)	Essential Complexity
	iv(G)	Design Complexity
	LOC	Lines of Code
Halstead	N	Length
	V	Volume
	L	Level
	D	Difficulty
	I	Intelligent Content
	E	Effort
	B	Error Estimate
	T	Programming Time
Line Count	LOCode	Lines of Code
	LOComment	Lines of Comment
	LOBlank	Lines of Blank
	LOCodeAndComment	Lines of Code and Comment
Operator/Operand	UniqOp	Unique Operators
	UniqOpnd	Unique Operands
	TotalOp	Total Operators
	TotalOpnd	Total Operands
Branch	BranchCount	Total Branch Count
Object-Oriented	sum v(G)	Sum of Cyclomatic Complexities
	avg v(G)	Average of Cyclomatic Complexities
	max v(G)	Maximum Cyclomatic Complexity
	max ev(G)	Maximum Essential Complexity
	NOC	Number of Children
	Depth	Depth within class heirarchy
	RFC	Response for a Class
	WMC	Weighted Methods per Class
	CBO	Coupling Between Objects
	LOCM	Lack of Cohesion in Methods
	Fan-In	Number of derivation classes
	Dep on Child (Boolean)	Dependence on Child

Figure 1. Metric Groups.

[1]For a more complete discussion of some of the various metrics used, please refer to Appendix A

However, there's a catch; if indicators are to be tailored to a particular project, then data must first be collected from that project. When software arrives at IV&V, it often arrives without any accompanying error data[2]; IV&V facilitators must start from scratch in their search for errors. Therefore, it is important that the appropriate error indicator be identified as early as possible. This problem, rapid and early identification of effective error indicators based on source code metrics, is what we will study here.

In the course of our investigation, we hypothesized that two things would occur:

- Procedural metrics will outperform object oriented metrics (because procedural metrics provide more data than their object-oriented counterparts; more procedures than classes)
- Less complex learners will stabilize earlier than their more complex counterparts (simpler learners are less distracted by irrelevant details)

The remainder of this paper is structured as follows: §2 is a general overview of the NASA IV&V facility and mission. §3 is a description of the project which was used during this study, and §4 is a summary of the various learners that were used. §5 is a summary of the results of our tests, §6 is a discussion of those results, and §7 are the conclusions drawn from them.

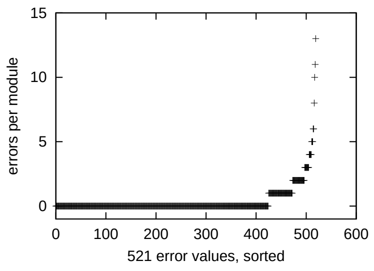

Figure 2. Distribution of errors: Most modules have no errors.

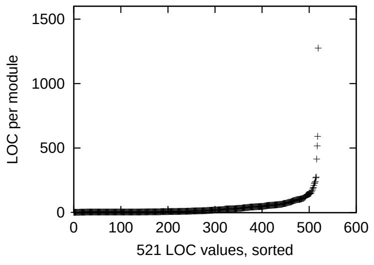

Figure 3. Distribution of LOC: Most modules are very short.

2 Background Of NASA IV&V

The NASA Independent Verification and Validation (IV&V) Facility in Fairmont, West Virginia is responsible for verifying that software developed or acquired to support NASA missions complies with the stated requirements. Additionally, the Facility validates that the software is suitable for its intended use. In short, the Facility ensures that the software is being developed properly, and that the right software is being developed or acquired.

Due to cost constraints, IV&V is generally applied to software modules which are determined to be most critical to mission success. While the Facility must always fully address those mission critical modules, there is a need for a quick and easy way to identify other modules which are not as critical, but may be more (or equally) error prone.

As the sole entity with the responsibility for IV&V of all NASA mission software, the IV&V Facility is in a unique position to create and maintain a master repository of software metrics (the MDP repository). Under this charter, the IV&V Facility reviews requirements, code, and test results from NASA's most critical projects; hence, many of the required metrics are collected as a matter of course. No other organization has insight into such a broad range of NASA projects. This affords the IV&V Facility an unequalled opportunity to research not only the early life cycle indicators of software quality, but other topics as well. Many large corporations have similar software metrics repositories; however, it is not always in their best interest to release data or results to the public. In the case of the IV&V Facility, the objective is to improve NASA's mission software regardless of the source. Sanitized data would be made available to NASA, industry, and academia to support software development and research by other organizations. This is consistent with the IV&V Facilities research vision of "See more, learn more, tell more."

3 The Project

The rest of this paper is dedicated to a case study on one NASA project (drawn from the MDP respository), which will be referred to using the moniker KC2. The goal of the case study is to determine how early in the IV&V process effective error indicators can be learnt from source code.

KC2 is a C++ program which contains over 3000 modules.[3] 521 modules are of interest to us since these modules

[2]The NASA experience is that error data is often only systematically collected as a result of the IV&V effort

[3]A module, for the purposes of our tests, is the equivalent of a C func-

were built by NASA developers. The remaining 2500 or so modules are COTS[4] software.

Of those 521 modules, 106 were found to have various numbers of errors, ranging from 1 to 13. A graphical representation of the errors per module and lines of code per module are presented in Figure 2 and Figure 3, respectively.

The KC2 project arrived at IV&V after 4 years of work by the developers. Since the project was being integrated into a larger, active system, IV&V work has continued on it since its' arrival. During the course of debugging and testing, various stages of error-data were accumulated. For this study, we chose to look at error data from 3, 6, 9, and 12 months into the IV&V process. In addition, we also evaluated the absolute latest set of error data, entered into the system in June 2002. Both procedural and object oriented metrics were collected on the KC2 data set, and both have been used in the analysis.

4 Methods for learning error indicators

The strengths of the various metrics as error indicators are assessed by passing the collected data through a *machine learner*; a piece of software that attempts to link *attribute ranges* with *classes*. An attribute range is a range of input values, for instance, $v(G) > 10$. A class is the specific instance with which an attribute range is linked; for instance, in this study, the two classes were "Error" or "No Error". For example, a typical machine leaner might reveal that the object oriented metric (attribute) "Dep on Child = TRUE" is always associated with an error (class); this, then, would be a very good indicator to use when checking for error-prone code. More likely, however, a range of values will be associated with a specific class, i.e. $v(G) > 10$ leads to errors.

The typical method of assessing the value of a discovered relationship (i.e., $v(G) > 10$ yields error), is a method called 10-way cross validation [6, 2]. In this process, the data is divided up into 10 evenly distributed buckets, and the machine learner is trained on 9 of the buckets. After training, the learner is then tested on the remaining bucket. This process is repeated 9 more times, changing the bucket which is left out each time. This ensures that the results of the learner are valid.

In addition to the standard 10-way cross validation test, we added another assessment criteria that is vital to our study; specifically, that of *early stability*. *Early stability* is the ability of a learner to quickly find an attribute range which will continue to be useful throughout the IV&V life of a project. For example, if 3 months of error data has been collected, and a machine learner predicts that $v(G) > 10$ is a worthwhile indicator, then this conclusion will be *stable* if this learner continues to find this a useful predictor after more error data is added. In these particular tests, we will only define a learner as having reached a stable conclusion if it predicts the *same* metric as being useful during every succeeding time increment of testing.

The following learners were the ones used during our tests.

4.1 Linear Regression & M5Prime

M5Prime is a linear regression, numerical prediction decision tree learner; it requires that the class attribute of a data set be a number, instead of a nominal value. It then performs a piecewise linear regression analysis, learning one linear equation for each piece, and produces a decision tree to pick which piece is relevant.

Linear regression has been used for learning software development estimators by Jeffries, et.al. [7]. We believe that this is the first application of M5Prime to learning software quality indicators.

4.2 OneR

OneR is a classification learner, which means that it attempts to *classify* examples according to a known classification scheme (i.e., error or no-error). OneR is an extremely simplistic classification leaner; it attempts to find the one attribute which can most successfully predict for the class [5]. To the best of our knowledge, OneR has not been used previously for learning software quality indicators. The pseudo-code for OneR is provided below.

```
For each attribute,
 For each value of the attribute, make a rule as follows:
  count how often each class appears
  find the most frequent class
  make the rule assign that class to this attribute-value
 Calculate the error rate of the rules
Choose the rules with the smallest error rate
```

An example of the output from OneR follows:

```
V:
< 523.905 => zero
< 609.865 => nonzero
< 937.885 => zero
>= 937.885 => nonzero
```

In order to read this tree correctly, first look at the top line; it tells you which attribute the tree refers to, in this case *V*(the halstead volume metric). Next, read each subsequent line with reference to the attribute, i.e. the second line means: *If V is less than 523.905 in your example, then classify it as "zero"*.

tion.

[4]COTS is an acronym for Commercial Off The Shelf

```
ORIGINAL:
```

outlook	*temp(°F)*	*humidity*	*windy?*	*class*
sunny	*85*	*86*	*false*	*none*
sunny	*80*	*90*	*true*	*none*
sunny	*72*	*95*	*false*	*none*
rain	*65*	*70*	*true*	*none*
rain	*71*	*96*	*true*	*none*
rain	*70*	*96*	*false*	*some*
rain	*68*	*80*	*false*	*some*
rain	*75*	*80*	*false*	*some*
sunny	*69*	*70*	*false*	*lots*
sunny	*75*	*70*	*true*	*lots*
overcast	*83*	*88*	*false*	*lots*
overcast	*64*	*65*	*true*	*lots*
overcast	*72*	*90*	*true*	*lots*
overcast	*81*	*75*	*false*	*lots*

```
SELECT class FROM original        SELECT class FROM original
WHERE outlook = 'overcast'        WHERE humidity >= 90

lots                              none
lots                              none
lots                              none
lots                              some
                                  lots
```

Figure 4. Attributes that select for golf playing behavior.

4.3 TAR2

TAR2 assumes the *small treatment effect*: that within a model, a very small number of variables control most of the other variables. This small treatment effect has been reported in many domains, albeit under different names. So much so that Menzies and Cukic [12, 11] and Menzies and Singh [13] speculated that small treatments are an emergent property that will appear in most models.

In models with small treatment sizes, a very fast and simple sensitivity analysis can be performed by TAR2 as follows. Firstly, use whatever data is available[5] and score each run (via some automatic oracle). Secondly, rank each attribute value by comparing their frequency in high scoring runs to their frequency in low scoring runs. In models with small treatments, a small number of attribute values should get outstandingly large rankings in this step. Thirdly, build treatments by combining a small number of attribute ranges with outstandingly large rankings (a "treatment" is a conjunction of restrictions on the input values that are intended to improve the results of subsequent model outputs). Fourthly, test those treatments by applying them as fresh constraints to a new run of a simulator. The treatments "work" if these constraints do improve the output of simulator. TAR2 automates steps one, two, and three.

It is worth noting that TAR2 does not restrict the analyzer to predicting for just one class outcome; by modifying the command line, or a configuration file, you can use TAR2 to select for any possible class. As an example of TAR2 in action, Figure 4 is a log of golf playing behavior, and Figure 5 are the learnt treatments for predicting for playing both *lots* of golf and *none*. An example output from TAR2 is show below:

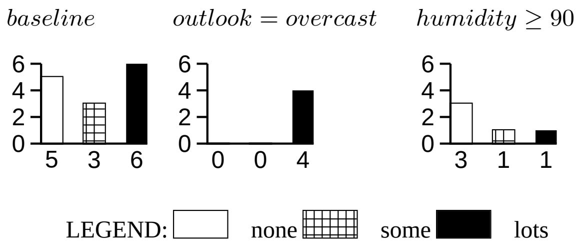

Figure 5. Changes to golf playing behavior from the baseline.

```
Worth=1.600000
Granularity=4 Promising=1.000000 Useful=1.200000 nChanges=1
Treatment:[outlook=overcast]
       none:                                [     0 -   0%]
       some:                                [     0 -   0%]
       lots:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  [     4 - 100%]
```

Given this example, which is a selection from a larger output, you can see that the learnt treatment is *outlook=overcast*, and that it selects for *4* cases in which *lots* of golf was played.

The newest increment of TAR2 (TAR2.2) contains two different discretization policies; where the results of these policies differ, two different treatments can be garnered. TAR2 has been used to compare Halstead & McCabe metrics in predicting for error-prone code before; see [14].

4.4 J4.8

Decision tree learners attempt to find paths which arrive at a specific class instance. J4.8, specifically, uses a method called decision tree induction, which is used by many researchers for generating quality code indicators. Figure 6 is one such decision tree.

In decision tree induction, data is split using a standard recursive splitting technique, which produces a decision tree whose leaf nodes contain training examples of one class [4]. This means that the output of the J4.8 algorithm is a "decision tree", garnered from the provided data, which suggests a path that can be taken in order to arrive at a specific class instance, i.e. "zero". J4.8 is a java port of the C4.5 algorithm [16].

C4.5 uses a heuristic *entropy* measure of information content to build its trees. The attribute that offers the largest *information gain* is selected as the root of a decision tree.

[5]If none is available, use monte carlo methods to generate it

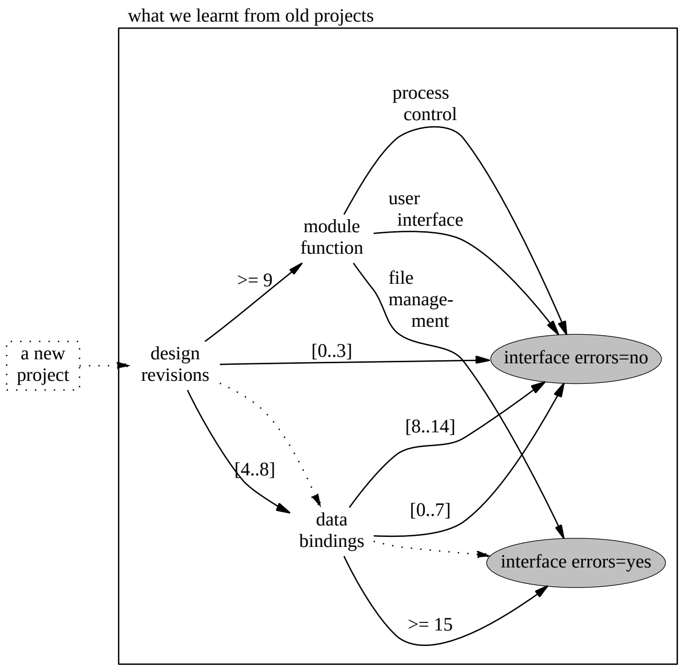

Figure 6. This decision tree was automatically learnt using machine learning techniques [15].

The example set is then divided up according to which examples do/do not satisfy the test in the root. For each divided example set, the process is then repeated recursively.

The information gain of each attribute is calculated as follows. A tree C contains p examples of some class and n examples of other classes. The *information required* for the tree C is as follows:

$$I(p,n) = -\left(\frac{p}{p+n}\right) log_2 \left(\frac{p}{p+n}\right) - \left(\frac{n}{p+n}\right) log_2 \left(\frac{n}{p+n}\right)$$

Say that some attribute A has values $A_1, A_2, ...A_v$. If we select A_i as the root of a new sub-tree within C, this will add a sub-tree C_i containing those objects in C that have A_i. We can then define the expected value of the information required for that tree as the weighted average:

$$E(A) = \sum_{i=1}^{v} \left(\frac{p_i + n_i}{p+n}\right) I(p_i, n_i)$$

The information gain of branching on A is therefore:

$$gain(A) = I(p,n) - E(A)$$

An example output from J4.8 is shown below:

```
LOBlank <= 7
| v(G) <= 6 => zero
| v(G) > 6 => nonzero
LOBlank >7 => nonzero
```

Given an example, you would classify it by following the tree. For example, if your particular function has attribute ranges $LOBlank = 5$ and $v(G) = 7$, then you would proceed down the $LOBlank <= 7$ sub-section of the tree (as indicated by the leading vertical bar) until you reached the $v(G) > 6$ segment; there, you find that you should classify your example as nonzero (error-prone).

5 Study & Results

Prior to conducting our study, we hypothesized that several things would happen:

- Procedural metrics will outperform object oriented metrics (because procedural metrics provide more data than their object-oriented counterparts; more procedures than classes)
- Less complex learners, such as OneR and TAR2, will stabilize earlier than their more complex counterparts (simpler learners are less distracted by irrelevant details)

We shall see if these hypothesis held up under light of our results. The next couple of subsections present the results of our study, in order of increasingly complex learners. The column headings are fairly self-explanatory; one that may need elucidating is *success rate*. With the exception of TAR2, success rate refers to the *average* success rate of a 10-way cross validation experiment. In the case of TAR2, success rate is *just* the success rate of TAR2 when run on all the data; the success rate under 10-way cross validation is provided in a separate column.

We classified our data with a binary classification; if the example had any errors whatsoever, then it was classified as nonzero; otherwise, its' classification would be zero.

5.1 OneR Results

Figure 7 and Figure 8 summarize the results of the OneR tests. Although OneR was unable to stabilize on the procedural metrics, it *did* produce a stable conclusion for the object-oriented ones, stabilizing in just 3 months from the projects inception at IV&V.

5.2 TAR2

Early in this article, we noted that TAR2 can predict for any of the provided classes; in this particular test, two classes existed: error, or no error. In addition, we also mentioned that TAR2 contains two different discretization policies. Where the results of these policies differed, one chart is shown for each result. Figure 9 through Figure 14 summarize the results of the TAR2 tests.

Period	Tree	Sucess Rate
Nov 1998	V: $< 523.905 \; => \; zero$ $< 609.865 \; => \; nonzero$ $< 937.885 \; => \; zero$ $\geq 937.885 \; => \; nonzero$	84%
Feb 1999	V: $< 545.345 \; => \; zero$ $< 609.865 \; => \; nonzero$ $< 739.745 \; => \; zero$ $< 929.6 \; => \; nonzero$ $< 1009.1 \; => \; zero$ $\geq 1009.1 \; => \; nonzero$	81%
May 1999	V: $< 545.345 \; => \; zero$ $< 609.865 \; => \; nonzero$ $< 710.45 \; => \; zero$ $< 912.6600000000001 \; => \; nonzero$ $< 983.26 \; => \; zero$ $\geq 983.26 \; => \; nonzero$	84%
Aug 1999	V: $< 545.345 \; => \; zero$ $< 609.865 \; => \; nonzero$ $< 721.165 \; => \; zero$ $< 912.6600000000001 \; => \; nonzero$ $< 983.26 \; => \; zero$ $\geq 983.26 \; => \; nonzero$	82%
Jun 2002	Total Opnd: $< 49.5 \; => \; zero$ $< 54.5 \; => \; nonzero$ $< 73.5 \; => \; zero$ $< 81.0 \; => \; nonzero$ $< 112.0 \; => \; zero$ $\geq 112.0 \; => \; nonzero$	86%

Figure 7. OneR Procedural Results
Conclusion: Not Stable

Period	Tree	Success Rate
Nov 1998	sum vg: $not\,? \; => \; nonzero$	74%
Feb 1999	sum vg: $< 41.5 \; => \; zero$ $\geq 41.5 \; => \; nonzero$	58%
May 1999	sum vg: $< 41.5 \; => \; zero$ $\geq 41.5 \; => \; nonzero$	58%
Aug 1999	sum vg: $< 41.5 \; => \; zero$ $\geq 41.5 \; => \; nonzero$	58%
Jun 2002	sum vg: $< 23.5 \; => \; zero$ $\geq 23.5 \; => \; nonzero$	75%

Figure 8. OneR Object Oriented Results
Conclusion: Stable after 3 months

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$34 \leq vG < 154$	100%	50%
Feb 1999	$5 \leq LOCodeandComment < 13$	100%	80%
May 1999	$100 \leq LOC < 1141$	90%	90%
Aug 1999	$99 \leq LOC < 1150$	88%	100%
Jun 2002	$118 \leq LOC < 1276$	83%	70%

Figure 9. Tar2 Procedural Results (Predicting for Errors)
Conclusion: Stable after 9 months

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$7 \leq evG \leq 101$	82%	80%
Feb 1999	$7 \leq evG \leq 103$	82%	80%
May 1999	$6 \leq evG \leq 104$	79%	90%
Aug 1999	$6 \leq evG \leq 104$	79%	80%
Jun 2002	$7 \leq evG \leq 125$	79%	100%

Figure 10. Tar2 Procedural Results (Predicting for Errors)
Conclusion: Stable after 3 months

TAR2 was able to achieve *several* stable conclusions for the procedural metrics, most of them within 3 months of the projects inception at IV&V. In addition, TAR2 was also capable of locating a stable conclusion for the object-oriented metrics.

5.3 J4.8

Figure 15 and Figure 16 summarize the results of the J4.8 tests. J4.8 was unable to stabilize on *either* the procedural or object-oriented metrics; in fact, it lacked even a basic root-node stabilization, producing theories which were for the most part widely diverse for each time period.

5.4 M5Prime

The M5Prime resultant trees were very large and added considerable bulk to this article. In addition, M5Prime failed to stabilize at any time during our tests, offering results that were just as scattered and diverse as those produced by J4.8. For these reasons, the M5Prime results have been omitted.

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$0.86 \leq L < 3$	100%	40%
Feb 1999	$0.43 \leq L < 3$	96%	100%
May 1999	$0.43 \leq L < 3$	96%	100%
Aug 1999	$0.43 \leq L < 3$	96%	100%
Jun 2002	$0.35 \leq L < 3$	97%	90%

Figure 11. Tar2 Procedural Results (Predicting for No Errors) Conclusion: Stable after 3 months

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$0 \leq LOBLank \leq 1$	96%	90%
Feb 1999	$0 \leq LOBlank \leq 1$	96%	100%
May 1999	$5 \leq LOC \leq 8$	98%	90%
Aug 1999	$4 \leq LOC \leq 8$	96%	60%
Jun 2002	$0 \leq T \leq 4.9$	97%	80%

Figure 12. Tar2 Procedural Results (Predicting for No Errors) Conclusion: Not Stable

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$56 \leq sumvG \leq 496$	100%	70%
Feb 1999	$8 \leq maxevG \leq 103$	100%	50%
May 1999	$12 \leq maxevG \leq 104$	100%	70%
Aug 1999	$12 \leq maxevG \leq 104$	100%	70%
Jun 2002	$8 \leq maxevG \leq 125$	100%	80%

Figure 13. Tar2 Object Oriented Results (Predicting for Errors) Conclusion: Stable after 6 months

Period	Treatments	Success Rate	10-way Cross
Nov 1998	$1 \leq max\ evG \leq 8$	45%	70%
Feb 1999	$2 \leq max\ vG \leq 12$	63%	70%
May 1999	$2 \leq max\ vG \leq 12$	63%	70%
Aug 1999	$2 \leq max\ vG \leq 12$	63%	70%
Jun 2002	$1 \leq max\ evG \leq 5$	78%	50%

Figure 14. Tar2 Object Oriented Results (Predicting for No Errors) Conclusion: Not Stable

Period	Tree	Success Rate
Nov 1998	$LOBlank \leq 7 \; => \; zero$ $LOBlank > 7 \; => \; nonzero$	83%
Feb 1999	$V \leq 543.7 \; => \; zero$ $V > 543.7$ $\| LOComment \leq 10$ $\| \| ivg \leq 6$ $\| \| \| evg \leq 1 \; => \; nonzero$ $\| \| \| evg > 1 \; => \; zero$ $\| \| ivg > 6 \; => \; nonzero$ $\| LOComment > 10 \; => \; nonzero$	83%
May 1999	$vg \leq 6 \; => \; zero$ $vg > 6 \; => \; nonzero$	84%
Aug 1999	$LOBlank \leq 7$ $\| vg \leq 6 \; => \; zero$ $\| vg > 6 \; => \; nonzero$ $LOBlank > 7 \; => \; nonzero$	85%
Jun 2002	$TotalOpnd \leq 49 \; => \; zero$ $TotalOpnd > 49$ $\| ev(G) \leq 4$ $\| \| iv(G) \leq 4 \; => \; nonzero$ $\| \| iv(G) > 4 \; => \; zero$ $\| ev(G) > 4 \; => \; nonzero$	84%

Figure 15. J4.8 Procedural Results Conclusion: Not Stable

Period	Tree	Success Rate
Nov 1998	$RFC \leq 8 \; => \; zero$ $RFC > 8 \; => \; nonzero$	74%
Feb 1999	$sumvg \leq 40 \; => \; zero$ $sumvg > 40 \; => \; nonzero$	67%
May 1999	$RFC \leq 8 \; => \; zero$ $RFC > 8$ $\| maxvg \leq 6 \; => \; zero$ $\| maxvg > 6 \; => \; nonzero$	61%
Aug 1999	$RFC \leq 8 \; => \; zero$ $RFC > 8$ $\| maxvg \leq 6 \; => \; zero$ $\| maxvg > 6 \; => \; nonzero$	62.5%
Jun 2002	$maxvg \leq 6 \; => \; zero$ $maxvg > 6$ $\| WMC \leq 7 \; => \; zero$ $\| WMC > 7 \; => \; nonzero$	75%

Figure 16. J4.8 Object Oriented Results Conclusion: Not Stable

6 Discussion

Figure 17 and Figure 18 display the successful results of our tests; that is, they display the learners for which stability was reached. The nomenclature for the TAR2 results is TAR2(E) refers to TAR2 when predicting for Errors, and TAR2(NE) refers to TAR2 when predicting for No Errors. In addition, the *Time Elapsed* field refers to the amount of time from the beginning of the IV&V process.

Learner	Time Elapsed	Metric
TAR2(E)	3 months	evG
	9 months	LOC
TAR2(NE)	3 months	L

Figure 17. Procedural Stabilization Results

Learner	Time Elapsed	Metric
OneR	3 months	sum vg
TAR2(E)	6 months	max evG

Figure 18. Object Oriented Stabilization Results

Earlier in this paper, we presented two hypotheses about this data. The first hypothesis was that the procedural metrics would outperform the object oriented ones; this hypothesis is not borne out by the results of the tests. Our second hypothesis was that the simpler learners would perform better than the more complex ones. This hypothesis *is* borne out by the results of the tests. The only two learners which stabilized were OneR and TAR2, our two simplest learners. It is interesting to note that TAR2 was able to stabilize in both tests, whereas OneR only stabilized for the object oriented segment. It would seem that when presented with a large mass of data, OneR, like its' more complex counterparts, still has trouble stabilizing over long time periods.

We found it extremely intriguing that both J4.8 (C4.5) and M5Prime failed to produce any stability over the various time periods. Both of these learning schemes are standard in most data mining approaches, and yet both proved to be unrevealing in these tests. This casts doubt on the error indicator generation methods proposed by other researchers, such as Selby[17] and Taghi[8], since simple learners can and, in this case, do outperform more complex and robust ones. We suggest that these complex learners, while excellent as data summary tools, are not as good at producing generalized conclusions as their less complex counterparts.

7 Conclusion

In conclusion, we have demonstrated several things:

- Stability can be found early in a project lifecycle
- Simpler learners are better for producing stable results
- Scarcity of data during early IV&V lifecycle does *not necessarily* present a large problem

In addition, we are able to present a recommended methodology for finding the most important metric for your project:

1. Gather metric data and error reports as early as possible
2. Use simple learners (i.e., OneR and TAR2) to analyze this data
3. Apply the results of those learners during the next cycle of debugging and testing
4. Check for stability at each cycle to ensure you are using the proper metric
5. Goto 1

Our conclusions *must* be taken with the proverbial grain of salt; while we recommend the methodology given above, we are not suggesting that you drop your other forms of error-prediction. It is extremely difficult to gather time series dumps of error data and metric collection, in addition to the success of these tests being a function of each project individually. Therefore, we suggest that you try our methodology *in addition* to your normal methods, and see which bring greater success to *your* projects.

Acknowledgements

We would like to thank Pat Callis, NASA IV&V Project Manager, for making available the resources for this study.

This research was conducted at West Virginia University under NASA contract NCC2-0979 and NCC5-685. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute nor imply its endorsement by the United States Government.

References

[1] J. D. Arthur, M. K. Groner, K. J. Hayhurst, and C. M. Holloway. Evaluating the effectiveness of independent verfication and validation. *IEEE Computer*, pages 79–83, October 1999.

[2] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transaction on Software Engineerining*, 25(4), July/August 1999.

[3] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[4] L. Holder. Intermediate decision trees, 1995. Available from `http://www-cse.uta.edu/~holder/pubs/ijcai95.ps`.

[5] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

[6] I.H.Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Academic Press, 2000.

[7] I. W. R. Jeffery and M. Ruhe. Using public domain metrics to estimate software development effort. In *Proceedings of the 7th International Software Metrics Symposium*, pages 16–27, April 2001.

[8] T. Khoshgoftaar and E. Allen. Model software quality with classification trees. In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*. World Scientific, 1999.

[9] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[10] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12), December 1989.

[11] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. Available from `http://tim.menzies.com/pdf/00ijait.pdf`.

[12] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://tim.menzies.com/pdf/00iesoft.pdf`.

[13] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In *2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February*, 2001. Available from `http://tim.menzies.com/pdf/00maybe.pdf`.

[14] T. Menzies, J. S. D. Stefano, M. Chapman, and K. Mcgill. Metrics that matter. In *Proceedings of the 27th Annual IEEE/NASA Software Engineering Workshop*, July 2002.

[15] A. Porter and R. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, pages 46–54, March 1990.

[16] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.

[17] R. Selby and A. Porter. Learning from examples: Generation and evalaution of decision trees for software resource analysis. *IEEE Trans. Software Eng.*, pages 1,743–1,757, December 1988.

[18] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, pages 30–36, March 1988.

Appendix A: Metrics

This section gives a brief overview of some of the software metrics which were collected in the course of this study. Inclusion in this section does not imply endorsement in any way by either NASA IV&V or WVU.

In order to facilitate the collection of various code metrics, many tools have evolved over the past few years. One of the most popular ones (and the one being used extensively at NASA IV&V) is the McCabe IQ© package. This package can evaluate Ada, C and C++ source code, and provides many different types of software metrics.

McCabe

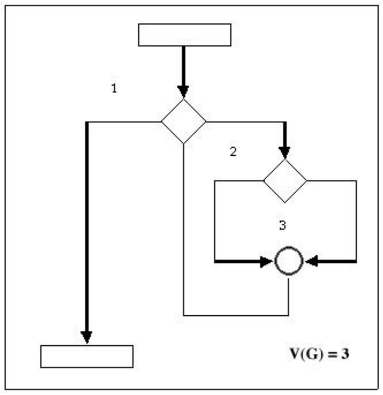

Figure 19. Example program flowgraph

The McCabe metrics are a collection of four software metrics: Essential Complexity, Cyclomatic Complexity, Design Complexity and LOC. Of these four, all but LOC are metrics which were developed by T. J. McCabe. McCabe & Associates claim that these complexity measurements provide insight into the reliability and maintainability of a module. For example, around NASA IV&V, a cyclomatic complexity of over 10 or an essential complexity of over 4 is flagged as a module that will be difficult to maintain and/or debug.

The following paragraphs present a short overview of the three complexity metrics mentioned previously.

Cyclomatic Complexity, or $v(G)$, actually measures the number of *linearly independent paths*[6] through a program's flowgraph[7]. $v(G)$ is calculated by:

$$v(G) = e - n + 2$$

[6] A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set

[7] A flowgraph is a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another

where G is a program's flowgraph, e is the number of arcs in the flowgraph, and n is the number of nodes in the flowgraph [3]. For example, Figure 19 is a simple flowgraph; it's cyclomatic complexity is 3, since the graph has 6 arcs and 5 nodes ($v(G) = 6 - 5 + 2 = 3$).

Essential Complexity, or $ev(G)$, is the extent to which a flowgraph can be "reduced" by decomposing all the subflowgraphs of G that are D-structured primes [8]. $ev(G)$ is calculated by:

$$ev(G) = v(G) - m$$

where m is the number of subflowgraphs of G that are D-structured primes. [3]

Design Complexity, or $iv(G)$, is the cyclomatic complexity of a module's reduced flowgraph. The flowgraph, G, of a module is reduced to eliminate any complexity which does not influence the interrelationship between design modules. This complexity measurement reflects the modules calling patterns to its immediate subordinate modules [10].

Halstead

Another commonly used collection of software metrics are the Halstead Metrics. They are named after their creator, Maurice H. Halstead. Halstead felt that software (or the writing of software) could be related to the themes which were being advanced at that time in the psychology literature. He created several metrics which are meant to encapsulate these properties; these metrics can be extracted by use of the McCabe IQ tool mentioned previously, and are discussed in detail below.

Halstead began by defining some basic measurements (these measurements are collected on a per module basis):

$$\begin{aligned} \mu_1 &= \text{number of unique operators} \\ \mu_2 &= \text{number of unique operands} \\ N_1 &= \text{total occurrences of operators} \\ N_2 &= \text{total occurrences of operands} \\ \mu_1^* &= \text{potential operator count} \\ \mu_2^* &= \text{potential operand count} \end{aligned}$$

Using these measurements, he then defined the *length* of a program P as:

$$N = N_1 + N_2$$

The vocabulary of P is:

$$\mu = \mu_1 + \mu_2$$

The *volume* of P, akin to the number of mental comparisons needed to write a program of length N, is:

$$V = N * log_2 \mu$$

V^* is the potential volume - the volume of the minimal size implementation of P.

$$V^* = (2 + {\mu_2}^*) log_2 (2 + {\mu_2}^*)$$

The *program level* of a program P with volume V is:

$$L = V^* / V$$

The inverse of level is *difficulty*:

$$D = 1/L$$

According to Halstead's theory, we can calculate an estimate $\hat{L}$ of L as:

$$\hat{L} = 1/D = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$$

The intelligence content of a program, I, is:

$$I = \hat{L} * V$$

The effort required to generate P is given by:

$$E = \frac{V}{\hat{L}} = \frac{\mu_1 N_2 N log_2 \mu}{2 \mu_2}$$

where the unit of measurement E is elementary mental discriminations needed to understand P.

The required programming time T for a program of effort E is:

$$T = E / 18 seconds$$

[8]D-structured primes are also sometimes referred to as "proper one-entry one-exit subflowgraphs". For a more thorough discussion of D-primes, see [3]