A Software Implementation Progress Model

Dwayne Towell[1] and Jason Denton[2]

[1] Abilene Christian University,
Abilene TX 79699, USA
dwayne.towell@acu.edu
[2] Texas Tech University,
Abilene TX 79602, USA
jason.denton@ttu.edu

Abstract. Software project managers use a variety of informal methods to track the progress of development and refine project schedules. Previous formal techniques have generally assumed a constant implementation pace. This is at odds with the experience and intuition of many project managers. We present a simple model for charting the pace of software development and helping managers understand the changing implementation pace of a project. The model was validated against data collected from the implementation of several large projects.

1 Introduction

Modern software development practices rely on periodically collected software metrics derived from a code base to provide management with feedback about the project and the process used to develop it [1, 2]. Well-defined and proven code metrics exist for some areas of software development [3, 4, 5], however the pace of implementation has no such established metrics based on code attributes. Several alternative, non-code-based progress metrics, such as function points [6] and earned value [1, 7], have been proposed and widely used. We believe it is possible to leverage existing size metrics to directly monitor progress in a code base, but to date such an approach has not been widely employed.

Here we propose implementation progress model based on development artifacts to interpret metrics and bridge the gap between concrete sampled data and expectations or beliefs about the underlaying process. On a small scale, this type of model may act as a predictor to set expectations over the next few data samples. This small-scale prediction helps provide timely feedback to management on the state of a project. Viewing a whole project, a model provides a portrait of the entire implementation process.

Without a formal, code-based implementation model management must rely on evidence other than implementation artifacts when making decisions about a project. In contrast, a formal implementation progress model based on implementation artifacts does not rely on external evidence yet establishes critical parameters and allows objectively evaluation based on inherent artifact attributes. Here we propose an implementation progress model based on implementation artifact metrics that matches our intuitive understanding of implementation progress.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 93–106, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The next section discusses supporting work. Section 3 describes the hypothesis, proposed implementation progress model, and research process. Results from actual projects are presented in Section 4. Finally, conclusions and directions for future study are in Section 5.

2 Related Work

Schneidewind uses time-series metrics to create a method for evaluating process stability [8]. Schneidewind asserts that metric trends are an indicator of the underlying process and that monitoring the trends can support managing the process. He suggests the shape of time-series data can be used to identify critical moments within a project. To further quantify project trends, an indirect metric based on time-series data is used. He defines a *change metric* as the difference between consecutive measurements of a primary metric. The model proposed here introduces a growth metric appropriate in the context of measuring project progress.

While discussing project progress, McConnell defines *code growth* for a project as the total size of project (source code) as a function of project time [2]. Code growth of traditional iterative development contains three distinct phases. In the first phase, architectural development and detailed design generate little code. The second phase provides staged deliveries and includes detailed design, coding, and unit testing. During this phase code growth is very high. Approaching initial delivery, the third phase, code growth slows to a crawl. Typical phase transitions occur at approximately 25% and 85% of the total implementation time for well-managed projects [2]. Specific details are not provided about metrics, but source lines of code, or a similar size metric, is assumed.

McConnell encourages the collection of time-series data to provide feedback supporting project management. Specifically, he recommends that collected data be viewed graphically since the *shape* can be used to diagnose project health. He graphically depicts the typical code growth pattern for a well-managed project, but acknowledges that its details varies to some degree. Our proposed progress model provides an empirical representation of the overall project shape and provides a specific interpretation of the three phases documented by McConnell.

3 Defining a Formal Progress Model

3.1 Informal Progress Models

Informal (non-mathematical) progress models already exist; as seen in project vocabulary and assumptions. Informal models are commonly used to answer project status queries, such as:

When will it be done, based on the current pace?

What was the size of the total effort for that project?

What fraction of the total effort has been spent?

Such informal progress models capture another key attribute of implementation progress. The informal model acknowledges that project speed is not constant throughout a project; projects "ramp up" and "slow down". These phrases refer to project speed and suggest the ability or desire to determine implementation velocity. As envisioned by experienced project managers, this velocity increases at the beginning and decreases near the end [2]. This is possibly an instance of the "S" shape progress or growth curse which is observed not only in projects but also in many other domains. A formal implementation progress model should be informed by this experience and capture the variations in velocity during implementation.

A formal implementation model should serve the same purpose as the informal model. The model must help answers questions about implementation speed and progress of current projects and provide a framework for making predictions about the future of the project. For example, changes in the rate of progress in an otherwise stable environment may indicate the project has transitioned to a new phase. This assumes the rate of progress is dependent on the project state.

3.2 Requirements for a Formal Progress Model

The interpretive power of an implementation progress model is important to consider. Interpretation of metric data relies on some understanding of our belief about the underlying process. In general, model parameters should be few in number, directly interpretable, and measured in existing units. These properties give the model parameters the most meaning and thus give the model the most explanatory power.

An implementation progress model should approximate actual project data collected. Figure 1 shows accumulated source lines of code sampled from the implementation phase of one project studied. The data in Figure 1 is very similar to the S-like curve described by McConnell [2].

This graph demonstrates the important characteristics of typical progress data. Overall progress is not linear with time; the fastest pace occurs during the middle of the project, while the ends are slower paced. The slope of the progress curve indicates the speed of progress.

3.3 Formal Implementation Progress Model

The primary goal of software implementation is creation of artifacts which contribute to delivering a working system. Implementation progress can be measured as change in an artifact. Progress over time can be measured as the sum of individual changes. We define implementation progress as the accumulated effort captured in code, which will eventually be delivered to the customer. For environments and development phases that emphasize code as the primary engineering delivery, we feel this is an appropriate definition.

Implementation metrics, traditionally used to measure size, can be employed to measure progress. Here we define a *growth metric* as the absolute difference between consecutive samples of a size metric, as shown in (1).

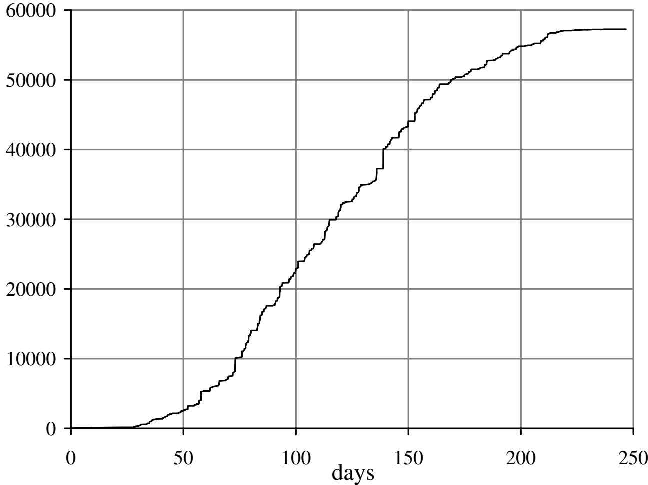

Fig. 1. Accumulated source lines of code changed for a sample project by day

$$\Delta_{m_t} = |m_t - m_{t-1}| \tag{1}$$

Where Δ_{m_t} is the growth in a metric m at time t.

Also useful is the idea of implementation velocity [9]; the rate at which progress is being made on a project. Evidence suggests implementation velocity begins and ends at zero while being at its highest in the middle. The simplest implementation velocity graph, consistent with experience, consists of three linear segments. Implementation velocity begins at zero. It increases linearly until the maximum sustainable velocity for implementation has been reached. The velocity remains constant until near the end of implementation when it begins to decrease. Then, implementation velocity constantly decreases until it reaches zero.

Figure 2 shows the idealized implementation velocity for a project as a function of time. The horizontal, center phase represents the steady, efficient development observed in the middle of the implementation phase. The positive slope at the left represents increasing velocity as implementation "gathers speed". The negative slope at the end of the graph shows the implementation phase decreasing speed as the end approaches.

The idealized graph shown here is symmetric; however, symmetry is not common in practice and is not required by the model presented. The idealized velocity as a function of time (v_t) can be described using three parameters.

$$v_t = \begin{cases} s\frac{t}{t_p}, & 0 \le t < t_p \\ s, & t_p \le t < t_q \\ s\frac{(t-t_f)}{t_q-t_f}, & t_q \le t \le t_f \end{cases} \tag{2}$$

In (2) the velocity is given as a function of time, where s is the maximum sustained velocity, t_p and t_q are the times of the phase transitions, and t_f is

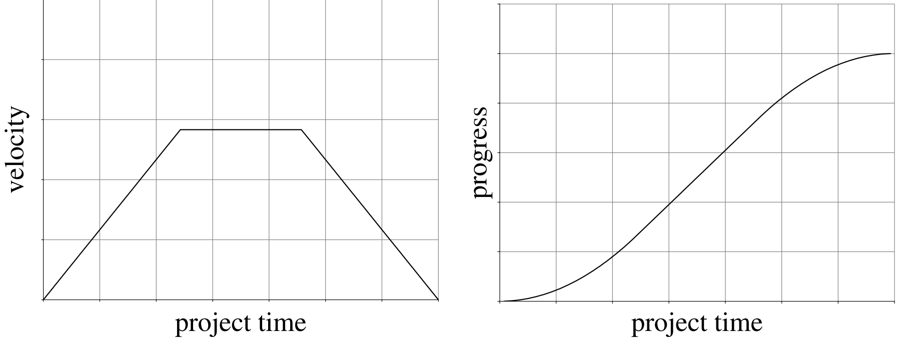

Fig. 2. Idealized implementation velocity as a function of time

the time at the end of implementation. Time may be measured in any real unit, such as days. Velocity is measured in size of metric change per time unit, such as lines of code per day.

Integration of the idealized velocity for a project produces idealized progress as a function of time (p_t).

$$p_t = \begin{cases} s\frac{t^2}{2t_p}, & 0 \leq t < t_p \\ st - \frac{1}{2}st_p, & t_p \leq t < t_q \\ s\frac{\left(t^2 - 2t_f t + t_q^2 + t_p t_f - t_p t_q\right)}{2(t_q - t_f)}, & t_q \leq t \leq t_f \end{cases} \tag{3}$$

The idealized implementation progress curve as a function of time is shown in (3). Progress is measured in accumulated metric growth to date, such as total lines of code changed.

4 Model Validation

We examined several size metrics as the basis for the growth metric used in our model [10]. *Source lines of code* (SLOC) is frequently used for estimating resources needed and should be readily available in most development environments [6, 11, 12, 13]. In this study, lines containing only white space and lines consisting of comment characters without any alphabetic characters were not counted. In addition, physical lines containing both code and comments were counted as two lines.

Two variations on the SLOC metric were considered. The simplest form counts the *SLOC change* (SLOCC) for each file. SLOCC is the absolute difference in SLOC between source files consecutively committed to the project repository; it counts SLOC added or deleted from the previous version. This assumes the correct removal of code artifacts is equivalent in terms of effort as correctly adding to the code base. We realize this may not strictly be the case, but it is difficult to determine what an appropriate weighting factor should be. To avoid introducing a weighting factor for this study, we assume all changes represent equal effort.

The second form measures the number of lines actually changed between submissions by comparing the files. This second measure is sometimes referred to as *code churn* (CHURN) [14]. CHURN is the count of source lines inserted, deleted, or changed between consecutively committed source files. It is probably a better change metric than SLOCC since CHURN captures more effort.

4.1 Alternative Models

Parameterized models provide an approximation of the sampled data for a particular data set. The model curve which most closely fits the data is considered the best; it introduces the least error. Model fit can be measured using the squared residual after subtracting the model curve from the sample data. To allow comparisons between models the average squared residual error ($\overline{R^2}$) is used. The model with the lowest $\overline{R^2}$ for a particular data set provides the closest approximation.

In addition to the proposed implementation progress model, three alternative models were chosen to provide a context for evaluating the fit of the proposed model.

The first model was a linear approximation. The linear model curve is given by (4). Linear approximation, with only two parameters, represents a practical lower-bound on the number of model parameters and the model with the highest expected $\overline{R^2}$.

$$linear_t = at + b \tag{4}$$

The second alternative model chosen was a multiphase, piecewise parabolic approximation. It contains eleven parameters; its model curve is shown in (5). This model was chosen to represent a practical lower-bound on $\overline{R^2}$.

$$multiphase_t = \begin{cases} at^2 + bt + c, & 0 \le t < t_p \\ dt^2 + et + f, & t_p \le t < t_q \\ gt^2 + ht + i, & t_q \le t < t_f \end{cases} \tag{5}$$

The multiphase model was chosen to provide an highly data-conforming model. The proposed model is a special case of (5).

The third model was a third-degree polynomial approximation, with four parameters as shown in (6). A third-degree polynomial approximation provides just enough flexibility to model the S-curve observed. It also provides a model of approximately the same number of parameters as the proposed model.

$$polynomial_t = at^3 + bt^2 + ct + d \tag{6}$$

4.2 Experimental Data

Seventeen projects from a single company were studied. All projects were developed using the same iterative process. They were six weeks to eighteen months in length and involved one to eight engineers. All projects produced entertainment and education oriented software designed to be marketed to consumers for use

with Microsoft® Windows® and on Macintosh® personal computers between 1995 and 2002. In this environment, before the prevalence of the Internet, once this type of consumer product was released to manufacturing, no maintenance changes were possible due to economic considerations. Manufacturing and distribution costs meant the projects had clear delivery dates after which no work was to be done. This is unlike other environments, where software is delivered in near real-time or deployed, and implementation evolves into a continuous cycle of maintenance. The progress model studied is expected to be meaningful when applied to each release of on-going projects, however additional studies will be needed to establish this. We expect results from this homogeneous group of projects will apply to initial development efforts of iteratively developed projects and to projects without maintenance phases.

4.3 Model Fitting Results

Evaluations of both metrics for each project were performed. The three alternative models described above and the proposed model were used. A numerical fitting routine was used to find parameter values that minimized $\overline{R^2}$.

Figure 3 shows progress measured via accumulated SLOCC and model curves for a project. As expected, the linear model provides a poor fit for the data and the multiphase model fits the data very accurately. Both the polynomial and proposed models provide fits between the linear and multiphase models.

The polynomial model exhibits wild "swings" near the ends. These swings are typical of polynomial curves which tend to favor data points near the center rather than the ends. In this case, the polynomial model suggests a "negative" amount of accumulated work had been accomplished until about day forty of the project. Similarly, it indicates reverse-progress begins to occur around day 220. In almost all cases, these polynomial model swings suggest negative progress occurs at the beginning and end of the project.

The multiphase model includes discontinuities, occurring on day 72 and 138. These discontinuities represent an instantaneous change in speed, which is inconsistent with an intuitive understanding of the process. In general, small changes in a data set may radically change the location and size of the discontinuities, which suggests the model does not accurately represent the implementation process.

The average squared residual error ($\overline{R^2}$) for each model is given in the legend of Figure 3. The values agree with a visual assessment of the fit except in the case of the polynomial model. While the lower $\overline{R^2}$ for the polynomial model is more desirable, the polynomial fit suffers extensively from undesirable swings near the ends. These swings violate a basic expectation of accumulated progress, that is, it should be monotonically increasing. While the proposed model has a larger $\overline{R^2}$, it is monotonically increasing and behaves as expected. This behavior appears to support in-project predictions better than the polynomial model.

In this project, most of the proposed model $\overline{R^2}$ can be seen to occur in the first third of the project. Each of the other three metrics show similar results; this may indicate early efforts are not as efficiently captured by the metrics as later

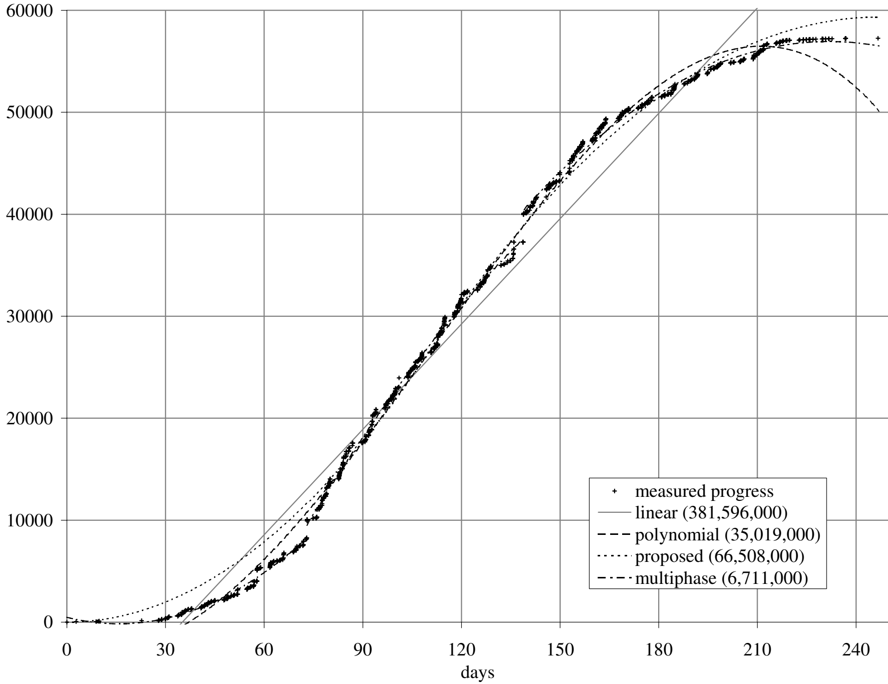

Fig. 3. Progress measured via accumulated source lines of code change (SLOCC) for project nine and progress model curves (with $\overline{R^2}$)

efforts. It could also indicate that during the later part of the project, the pace was unpredictably high (in violation of the implied model). Without additional information about the project, or its context, a determination cannot be made.

4.4 Data Analysis

The $\overline{R^2}$ for each metric is given in Tables 1 and 2. Figures 4 and 5 show $\overline{R^2}$ *relative* to the linear model $\overline{R^2}$ for each project. To improve viewing, projects are ordered by polynomial model relative $\overline{R^2}$.

In all cases the proposed model reduces $\overline{R^2}$ compared with the linear model, as expected, since the proposed model has an additional parameter. In many cases the proposed model *substantially* reduces $\overline{R^2}$ when compared with a linear model. In a few of these cases the reduction in $\overline{R^2}$ is almost to the level achieved using the multiphase model. In these conforming cases the proposed model provides a meaningful interpretation of the data.

Consider the proposed model $\overline{R^2}$ compared with the linear model $\overline{R^2}$. In cases where the proposed model substantially reduces $\overline{R^2}$, the model gave improved results with the addition of a single parameter. This substantial improvement suggests the data conforms to the model and the results may be relied upon to

Table 1. $\overline{R^2}$ measuring *source lines of code changed* (SLOCC)

		Model $\overline{R^2}$ (in millions)			
Project	samples	linear	polynomial	proposed	multiphase
1a	129	0.0667	0.0482	0.0434	0.0094
1b	1408	1.8269	1.6747	1.1458	0.4171
3	406	3.8133	0.9391	0.6371	0.0762
4	1394	1.0997	0.5958	0.6843	0.1176
5	90	0.0316	0.0108	0.0094	0.0021
6	1455	8.6590	2.0236	2.4551	0.2863
7	2204	12.3603	3.0171	4.0937	0.7440
8	138	0.0915	0.0245	0.0239	0.0028
9	1555	11.3106	1.1201	2.1672	0.2127
10	481	0.4575	0.0738	0.0386	0.0147
11	164	0.0111	0.0043	0.0037	0.0017
13	1274	2.4112	0.8349	0.8790	0.1589
14	715	0.4516	0.1139	0.1331	0.0181
15	723	0.1256	0.1215	0.1074	0.0210
17	827	2.9719	1.1158	1.2222	0.2215
19	967	6.3210	2.6441	1.7401	0.2475
20	1214	1.9231	0.3513	0.1566	0.0414

Table 2. $\overline{R^2}$ measuring *code churn* (CHURN)

		Model $\overline{R^2}$ (in millions)			
Project	samples	linear	polynomial	proposed	multiphase
1a	129	0.3485	0.2205	0.1601	0.0289
1b	1408	22.0988	8.2642	15.6206	1.7608
3	406	19.6692	4.9855	3.4455	0.2821
4	1394	2.6183	1.7401	2.0264	0.5516
5	90	0.2008	0.0530	0.0590	0.0119
6	1455	19.7174	5.3214	5.7359	0.8605
7	2204	36.1142	9.8627	12.7864	1.9951
8	138	0.2487	0.0674	0.0549	0.0087
9	1555	49.1829	4.7964	9.3587	0.8686
10	481	8.9162	2.8376	3.0023	0.0517
11	164	0.0362	0.0203	0.0155	0.0077
13	1274	9.4001	3.9973	3.2209	0.7597
14	715	1.7914	0.4591	0.5131	0.0593
15	723	0.4844	0.4684	0.4064	0.0814
17	827	11.0583	3.1825	3.3451	0.7040
19	967	20.5449	8.5899	5.9072	0.5758
20	1214	7.8927	1.7450	0.8902	0.1993

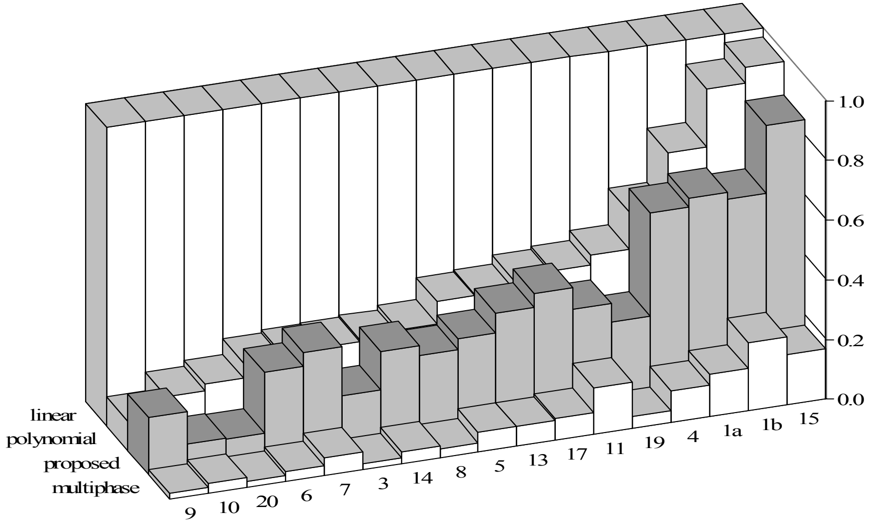

Fig. 4. *Source lines of code change* (SLOCC) average squared residual error ($\overline{R^2}$) relative to linear $\overline{R^2}$

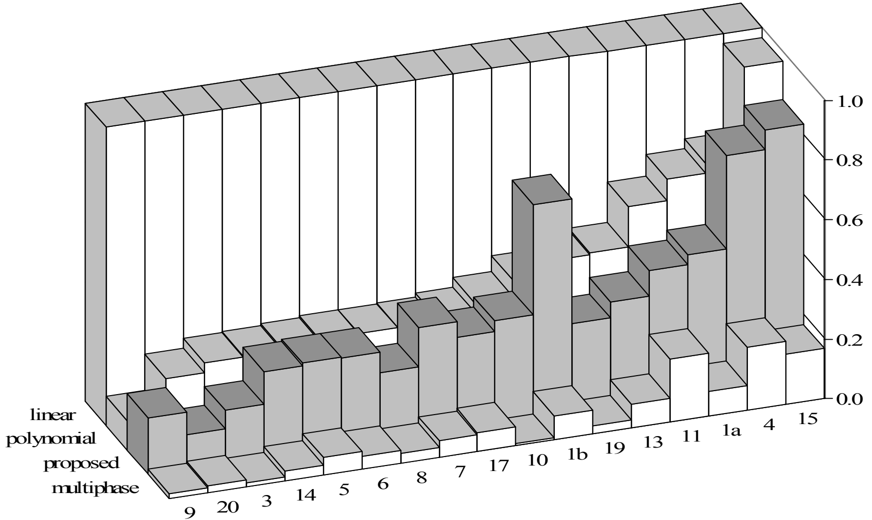

Fig. 5. *Code churn* (CHURN) average squared residual error ($\overline{R^2}$) relative to linear $\overline{R^2}$

correctly interpret the data. In the non-conforming cases, where the reduction is less significant, the model may not be appropriate and the results should only be used judiciously. Based on the available projects, we suggest the proposed model may be relied upon when the $\overline{R^2}$ is at most half that of the linear model.

Projects may fail to conform to the model for a number of reasons. Almost all projects exhibit “pauses” corresponding with weekends when developers do no work. Some projects also include larger periods when no apparent progress is made during a holiday break. Both of these phenomena can be seen clearly in projects 1a and 15. Using work days instead of calendar days would eliminate a major cause of time-related noise. Several projects include substantial, sudden, and anomalous progress. In all cases where these events were examined closely, the anomaly has proven to be the result of an unfortunate side-effect of the specific data collection procedure used. For example, a renamed file was detected as a combination of a substantial deletion and a subsequent addition. A commitment to collect the needed data during the project could reduce noise by allowing anomalies to be detected and corrected while any additional required information is still available.

Tables 3 and 4 show the model parameters for each project and metric, ordered by $\overline{R^2}$ relative to linear $\overline{R^2}$. With only seventeen projects and no independent data available, few definite conclusions can be reached, however several items are worth noting.

In about half of all cases, the model indicates t_p and t_q are essentially the same. In these cases, the model $t_q - t_p$ is close to zero, suggesting steady progress did not occur; implementation was either accelerating or decelerating. This could indicate development under a tight schedule or a process that could be improved. It is also interesting to note that in these cases the model was able to substantially reduce $\overline{R^2}$ while effectively using only two parameters.

Table 3. *Source lines of code change* (SLOCC) progress model parameters and $\overline{R^2}$ relative to linear $\overline{R^2}$

		Model parameters			
Project	relative $\overline{R^2}$	s	t_p	t_q	t_f
20	0.081	289.8	64.2	64.3	187.1
10	0.084	156.9	30.6	30.7	132.0
3	0.167	376.0	48.9	49.0	120.3
9	0.192	480.4	109.8	109.8	246.7
8	0.262	42.5	17.2	17.5	128.9
19	0.275	501.7	108.1	108.4	164.8
6	0.284	361.9	57.5	126.1	248.0
14	0.295	91.8	21.3	97.8	238.2
5	0.297	92.0	14.9	14.9	48.8
7	0.331	183.9	93.4	288.9	555.2
11	0.336	48.9	27.5	62.4	70.9
13	0.365	250.5	93.8	213.9	220.8
17	0.411	268.4	6.3	110.4	181.0
4	0.622	215.6	24.1	138.5	201.7
1b	0.627	303.1	28.9	220.2	248.3
1a	0.650	183.8	14.0	14.1	44.9
15	0.855	92.3	4.5	140.9	163.9

Table 4. *Code churn* (CHURN) progress model parameters and $\overline{R^2}$ relative to linear $\overline{R^2}$

		Model parameters			
Project	relative $\overline{R^2}$	s	t_p	t_q	t_f
20	0.113	560.6	68.2	68.3	187.1
3	0.175	820.9	45.7	45.7	120.3
9	0.190	1028.0	110.6	110.7	246.7
8	0.221	75.9	21.5	21.8	128.9
14	0.286	170.6	23.2	92.4	238.2
19	0.288	882.7	111.8	112.0	164.8
6	0.291	631.7	43.4	136.4	248.0
5	0.294	211.3	17.5	21.4	48.8
17	0.302	538.3	10.7	109.3	181.0
10	0.337	412.6	16.7	16.8	132.0
13	0.343	548.8	87.9	211.6	220.8
7	0.354	325.9	85.2	298.6	555.2
11	0.427	100.9	20.4	59.2	70.9
1a	0.459	365.8	14.5	14.5	44.9
1b	0.707	667.8	54.1	238.3	248.3
4	0.774	386.7	17.5	148.2	201.7
15	0.839	181.7	0.5	139.7	163.9

In conforming cases where $t_q - t_p$ is much larger than zero, the model indicates steady, sustained implementation occurred between t_p and t_q. In these cases, the implementation velocity (s) can be stated with great confidence. Velocity is a surrogate for productivity in the dimension measured by the specific metric. For example, Table 3 shows project six averaged over 360 lines of new code per calendar day between project days 58 and 126.

In the projects studied, implementation velocity (s) varies by more than an order of magnitude. While part of this variation is due to the number of engineers assigned to the project, likely some is due to proficiency. This is consistent with studies showing individual programmer productivity varies by as much as an order of magnitude [15].

5 Conclusions

Interpreting implementation progress measurements is difficult. A simple model is needed to provide a framework to help interpret the data. We have developed a piecewise approximation based on a three-phase model of linear implementation velocity. The model corresponds well to our intuition of how project progress occurs. It identifies project phase boundaries as well as the velocity of implementation during each phase. Furthermore, the progress model allows comparisons of project velocity between projects and easily supports estimating.

The progress model fits the available sample data better than a linear model. With only one additional parameter, the model produces fits with approximately

two-thirds less error than a linear fit. When compared with a polynomial fit, the progress model performs at least as well as a polynomial model which has one additional parameter.

Any model is only as good as the data on which it is based. Errors were discovered in both dimensions of the sample data. Spurious data entries were occasionally introduced due to the check-in process used. Similarly, using project work days, instead of calendar days, could have improved the quality of data in the time dimension.

5.1 Future Work

This work provides a sound basis for further study in this area. The progress model presented here only considers non-maintenance implementation. Projects with clear delivery dates, after which continuing development is not planned, fall into this category. Projects in maintenance or under continuous development may not exhibit phases similar to projects with firm end dates and deserve to be investigated, although this would require further work.

The stability of the model suggests it could be used to make predictions. Estimating project parameters such as final size, delivery date, development pace, etc. during implementation should be investigated. Similarly, comparisons of teams or projects based on model parameters could be studied.

Investigation of other metrics as a basis for measuring progress should be undertaken. If a size metric for object-oriented software were developed, investigating its use as a basis for a growth metric would be very valuable. Variations of existing metrics better tuned to capture change should be studied. One example of this type of metric is the sum of cyclomatic complexity of all changed functions, rather than simply the change in cyclomatic complexity of a source artifact.

References

1. Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)
2. McConnell, S.: Software Project Survival Guide. Microsoft Press, Redmond, WA (1998)
3. Fenton, N.E.: Software Metrics: A Rigorous Approach. Chapman and Hall, London (1991)
4. Kafura, D., Canning, J.: A validation of software metrics using many metrics and two resources. In: Proceedings of the 8th International Conference on Software Engineering. (1985) 378–385
5. Fenton, N.E., Neil, M.: Software metrics: roadmap. In: Proceedings of the conference on The future of Software Engineering, ACM Press (2000) 357–370
6. Albrecht, A.J., John E. Gaffney, J.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transactions on Software Engineering **9**(6) (1983) 639–648
7. Humphrey, W.S.: A Discipline for Software Engineering. Addison-Wesley (1994)
8. Schneidewind, N.F.: Measuring and evaluating maintenance process using reliability, risk, and test metrics. IEEE Transactions on Software Engineering **25**(6) (1999) 761–781

9. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
10. Towell, D.: An implementation progress model. Master's thesis, Texas Tech University (2004)
11. DeMarco, T.: Controlling Software Projects - Management, Measurement and Estimation. Yourdon Press, Inglewood Cliffs, NJ (1982)
12. Lind, R.K., Vairavan, K.: An experiemental investigation of software metrics and their relationship to software development effort. IEEE Transactions on Software Engineering **15**(5) (1989) 649–653
13. Jorgensen, M.: Experience with the accuracy of software maintenance task effort prediction models. IEEE Transactions on Software Engineering **21**(8) (1995) 674–681
14. El-Eman, K.: A methodology for validating software product metrics. Technical Report NRC/ERB-1076 44142, National Research Council Canada, Institute for Information Technology (2000)
15. H. Sackman, W. J. Erikson, E.E.G.: Exploratory experimentation studies comparing online and offline programming performance. Communications of the ACM **1**(1) (1968) 3–11