Proc. COMPSAC 88

The Software Process Model

David A Gustafson*, Austin C Melton*, Ying-Chi Chen*, Albert L Baker**, and James M Bieman**

*Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas 66506, (913) 532-6350 and **Department of Computer Science, Iowa State University, Ames, Iowa 50011

Managing software development is a difficult process. The waterfall model/approach and the iterative development model/approach have been used to both visualize the process and to control the development. A model that is independent of the development approach is needed. Such a model should also help in understanding software development in general.

A practical approach to modeling software development is to model the evolution of the full set of documents produced in a software project. We define such a software process model, **SPM**. The **SPM** provides a framework for measuring, analyzing, and understanding the software development process. It is a general model that should be usable for software development using any development approach. We use the model to formally characterize research in software measures and metrics. Data obtained from two software projects are presented in the **SPM** format.

Introduction

Ultimately, the concern of software engineers must be for the product of the software development process. And the product is more than the executable implementation. The product is a set of documents that may include a requirements document, specifications documents, design documents, implementation documents, testing documents, and other forms of external documentation. We would like to understand the actual, current processes used and the processes that can be used to produce these products.

A realistic and general model is essential for understanding the software development process. Two prevelant models are the waterfall model [9] and the incremental refinement model [11] (which might also be called the "iterative refinement" model or the "step-wise refinement" model). While both of these models have contributed to our understanding of the software development process, they possess common and general deficiencies:

1) The models prescribe a sequential order of document development.

2) The models focus on final documents.

3) They do not model the evolution of all the documents produced.

We define a product based model of the software development process that we call the Software Project Model, **SPM**. This model does not have the deficiencies listed above and accounts for the beneficial features of the existing models.

Another key ingredient to understanding the software development process is the ability to measure both the process and product. The need for measures to manage the process and product is clearly evident [8]. The results to-date in the area of software measures have hardly satiated this need. While there are numerous reasons why measures research has not produced more usable results, one major reason is that there has been no general framework for the definition and evaluation of proposed measures. We find that the **SPM** provides such a framework. It accounts for measures of the process, which we refer to as "process measures" or "process data", and measures of the product, which we refer to as "software measures". For instance, the framework provided by the **SPM** leads to a clear and precise distinction between a software measure and a software metric (this issue is discussed in more detail later).

The **SPM** has a wide range of applicability in software development environments, especially in areas that relate distinct documents or relate documents at different points in their evolution. Most expert systems for software engineering fit into these categories [2,4,10,12]. For instance, a sophisticated environment of the future might help maintain the relationship "is correct with respect to" between a specification document and an implementation document.

This paper includes a demonstration of the **SPM**'s effectiveness as a framework for software measures.

In the next section we analyze the two prevalent models and carefully define the SPM. We use the SPM in the third section as a framework for measures of the software process and product.

The Software Project Model

While there are a number of distinctly different paradigms for software development [5], the waterfall and incremental refinement models mentioned in the first section are representative of views held by many software engineers. The waterfall model [9], depicted simply in Figure1, added to professional understanding of the software development process by

1) making explicit that the software product includes distinct documents, and

2) suggesting a logical order of activities related to producing each document (we ought not produce a detailed design document after having completed the implementation)

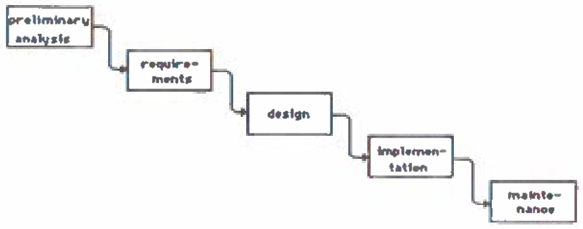

Figure 1 : A Simple View of the Waterfall Model

But there is now general consensus that software development is not quite such a simplistic, sequential process. There is, in practice, a high degree of parallelism in the production of software documents involving both formal and informal communication. In addition, developers use information gleaned from early efforts on a document as they work toward a "final" version of that document. The waterfall model does not account for this type of information. It only reflects that, for instance, the information in the specification document is used to produce the detailed design.

The other model mentioned in the first section addresses the second deficiency of the waterfall model. The incremental refinement model [11] is based on the PLEASE programming environment in which successive versions of a program from the first formal specification through the final production version are recorded in the same document.

The early specifications often use operators (eg, \exists -- ``there exists") that are usually not efficient enough to be used in the final production system. This model clearly supports a stronger relationship between documents as they are developed than does the waterfall model. However, the incremental refinement model suffers from
other deficiencies:

1) The model does not encompass all the documents produced. It fails to account for those documents that are expressed informally, e.g., initial requirements and test documents.

2) The model still implies an inherently sequential approach to software development.

The waterfall model has one additional characteristic which we believe detracts from its generality. It is focused primarily on the process of software development. The iterative refinement model suffers less from this drawback, but the **SPM** model presented in the last part of this section is developed even more from the perspective of the software product. While we acknowledge that this is a question of perspective and it is the entire system of process and product that we wish to model, the product perspective offers greater possiblities for management and understanding.

The **SPM** is an abstraction of the activities and products of actual software development. The model focuses on evolving products of a software project and the relationships between products. Each of the products can be described as a document.

We assume that development can take place in parallel; therefore several of the documents may be under development simultaneously. The **SPM** does not prescribe an order of activities and the products need not be named. Thus, the **SPM** may be applied to a variety of lifecycle schemes.

The development of a software system is the process of transforming representations of a system. Each "phase" of the software development process produces a distinct representation of the system. A representation is a document and each document provides a unique view of the system. It follows that these alternative views are of different levels of abstraction and precision, and the documents that represent each view should be identifiably distinct. Using the model, each document is either written in a different language or the documents are identifiable in some other manner. Thus, a requirements document can be distinguished from a specification document; a specification can be distinguished from a design; a design can be distinguished from an implementation; and so on.

We view software development as the process of refining a set of documents. The process is time dependent and therefore a real-time clock is a critical component of the model. The following definitions formally characterize the **SPM**.

Definition : The **Software Project Model (SPM)** is a set of document histories, **SPM** = { H_1, H_2, ..., H_n}, where each H_i is a history of the different versions of one document.

Definition : A **document history** H_i is a tree whose nodes are versions of documents, and $H_i = (V_i, E_i, r_i)$ where

V_i is a set of document versions of type **i**,

E_i is a set of ordered pairs of the form **(a,b)**, **a, b** $\in V_i$, which represent transitions from one version in V_i to another, and

r_i is the root of the H_i tree or the initial version.

A tree is used to represent a document history to allow the modeling of the development of alternative final versions. Consider the development of a system that is to run alternatively on hardware produced by several different vendors. Such a system may require alternative specifications, designs, and implementations for each machine. The tree form of a document history allows us to model such alternatives.

Definition : A **document version** $d_i(j) \in V_i$ is an ordered pair $d_i(j) = \langle d'_i(j), td_i(j) \rangle$, where $d'_i(j)$ consists of the text of the document version, and $td_i(j)$ is a real time stamp which represents the completion date/time for $d'_i(j)$.

The time stamp can be augmented with additional management data such as the publication date, the time/date that the document version was released to teams working on other documents, or the number of person/hours used to develop the document version. This allows the calculation of a wide range of process measures such as the the number of person-hours used to develop the document version.

The granularity of a document history is arbitrary and depends upon the needs of a particular project. A new version may be created when teams report completion of a version or after a formal version approval process. Or one may consider the document to be a new version each time the online version is edited.

The use of **SPM** to describe a possible software project is illustrated in figure 2. This depicts an idealized development that went through two passes on the documentation.

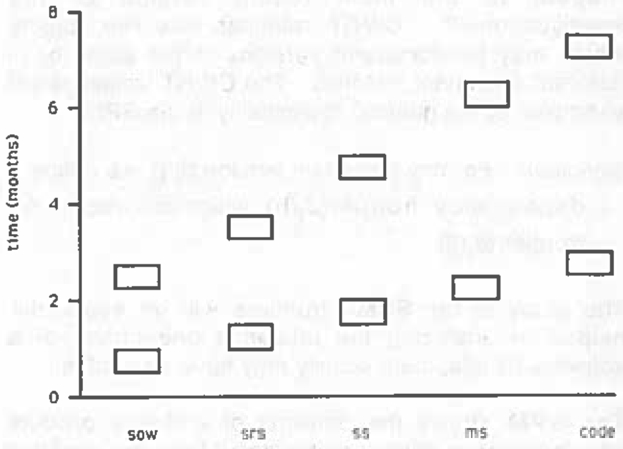

Figure 2 : The Software Project Model (SPM) for an idealized development

Because of the inclusion of a real-time stamp, we can use the model for temporal analyses, including an analysis of the possible dependencies between documents. For instance, at each time **t** in the development of a project we define a **frontier(t)** to be the set of the most recent versions of documents in the **SPM** with completion prior to time **t**.

Definition : Given a **SPM**, the **frontier(t)** is the set of document versions in the **SPM** determined by

1) deleting from the **SPM** all document versions $d_i(j)$ and edges into these versions, where $td_i(j) > t$, and

2) including the leaf nodes from the resulting **SPM** in the **frontier(t)**.

Definition : For any document version $d_i(j)$ we define a **dependency frontier($d_i(j)$)** which consists of the **frontier($td_i(j)$)**.

The study of the **SPM** frontiers will be especially helpful in analyzing the influence one phase of a software development activity may have on another.

The **SPM** shows the patterns of software product development in much greater detail than the waterfall model and establishes the same correspondence between specifications and implementation as the iterative refinement model. In addition, the **SPM** models parallel activities and provides a complete project history.

The **SPM** can form the basis for describing a variety of relations between products. One such useful family of relations are the "correct with respect to" (**CWRT**) relations. A boolean result is expected from a **CWRT** relation which answers a question like "is the most recent version of the implementation correct with respect to the most recent version of the specifications?" **CWRT** relations take two objects which may be document versions in the same or in different document histories. The **CWRT** relations are examples of the general applicability of the **SPM**.

Definition : For any document version $d_i(j)$ we define a **dependency frontier($d_i(j)$)** which consists of the **frontier($td_i(j)$)**.

The study of the **SPM** frontiers will be especially helpful in analyzing the influence one phase of a software development activity may have on another.

The **SPM** shows the patterns of software product development in much greater detail than the waterfall model and establishes the same correspondence between specifications and implementation as the iterative refinement model. In addition, the **SPM** models parallel activities and provides a complete project history.

The **SPM** can form the basis for describing a variety of relations between products. One such useful family of relations are the "correct with respect to" (**CWRT**) relations. A boolean result is expected from a **CWRT** relation which answers a question like "is the most recent version of the implementation correct with respect to the most recent version of the specifications?" **CWRT** relations take two objects which may be document versions in the same or in different document histories. The **CWRT** relations are examples of the general applicability of the **SPM**.

Examples of Software Process Models

Below are two examples (figure 3 and figure 4) of SPM diagrams drawn for actual projects. The projects were student projects written in a software engineering course.

The documents included in this project were a statement of work (SOW), a data flow diagram (DFD), an entity-relation-attribute specifcation of the data flow (ERA), a specification of the first phase implementation (P0), a specification of inputs for project walkthroughs(W/I), a hierarchy diagram (HD), a textual description of the module interconnections (HS) , a test plan (TEST) and the source code (CODE). The documents were required at particular times but were also required to be revised whenever errors or inconsistencies were discovered. The dates indicate when the documents were turned in for evaluation.

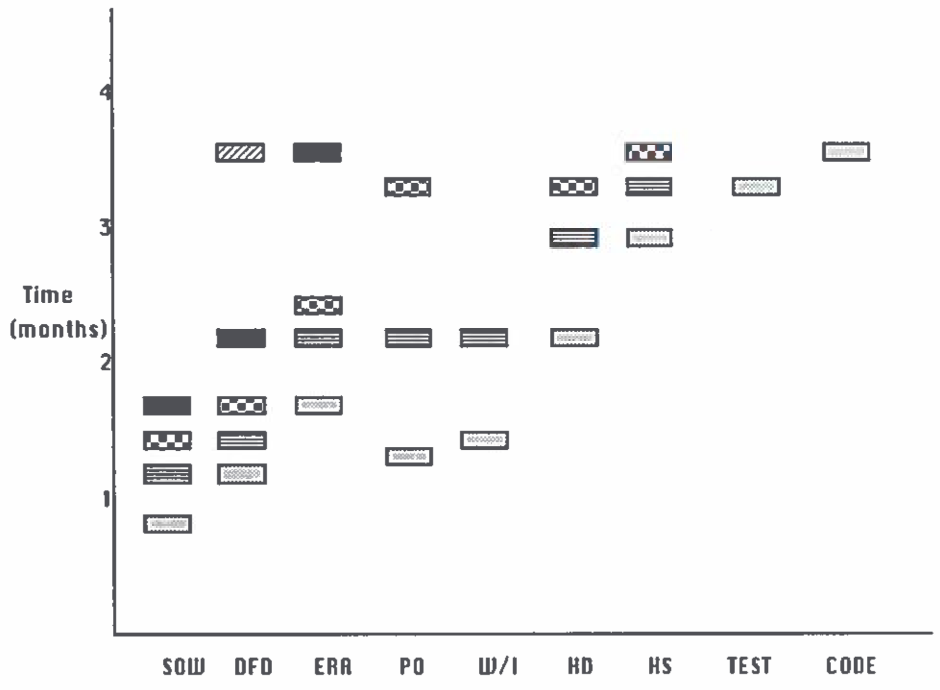

Figure 3 : SPM diagram for development A

The development process depicted in figure 3 was orderly. The students revised prior documents to ensure consistency. Although there appears to be many versions of documents, the development was relatively stable. Figure 4 shows another development effort.

The development effort depicted in figure 4 shows fewer documents and fewer versions of the documents.

A current research effort is to develop quantitative measures and metrics to assess the development process. Although there are many possible numbers that can be calculated from the data on these diagrams, it is important to develop a good foundation [1, 3] for any such measure or metric. The next section describes a framework for these measures and metrics.

A Framework for Software Measurements

The SPM provides a framework for quantifying the software process and software products. The model helps clarify the distinction between the products of software development and the development process. Each node in the SPM represents the text of a document version, the completion time, and possibly additional process data. We find that the published approaches to quantifying source code structure can be described in terms of the SPM. McCabe's cyclomatic number [7] and Halstead's software science effort [6] are examples of measures of a version of an implementation document -- one node in a document history.

In an effort to make software engineering terminology match terminology from the physical sciences and mathematics, we distinguish between a measure and a metric. A metric traditionally refers to a distance function and determines how two objects differ in the measurement of a specified property. To be consistent with the traditional meaning of "metric", we will use the term "metric" to be a function with two arguments that produces a real-number result. The term "measure" will be used to describe a function with one argument. We first give the mathematical definition of a metric.

definition : Let X be a set, $\Re$ be the set of real numbers, and let f be a function where $f : X \times X \rightarrow \Re$. The function f is a metric on X, and X is a metric space if f satisfies the following properties, where $x \in X$, $y \in X$, and $z \in X$:

1) $f(x,x) = 0$
2) $f(x,y) = 0$ if and only if $x = y$
3) $f(x,y) = f(y,x)$
4) $f(x,z) \le f(x,y) + f(y,z)$

Definition : The function $f : X \times X \rightarrow \Re$ is a pseudo-metric if f satisfies properties 1, 3, and 4.

Thus, one may view a metric $f : X \times X \rightarrow \Re$ as a function giving the difference or distance between two points. For example, if X is a set of programs, $a \in X$, and $b \in X$, then f(a,b) may represent the difference with respect to some property of the two programs a and b.

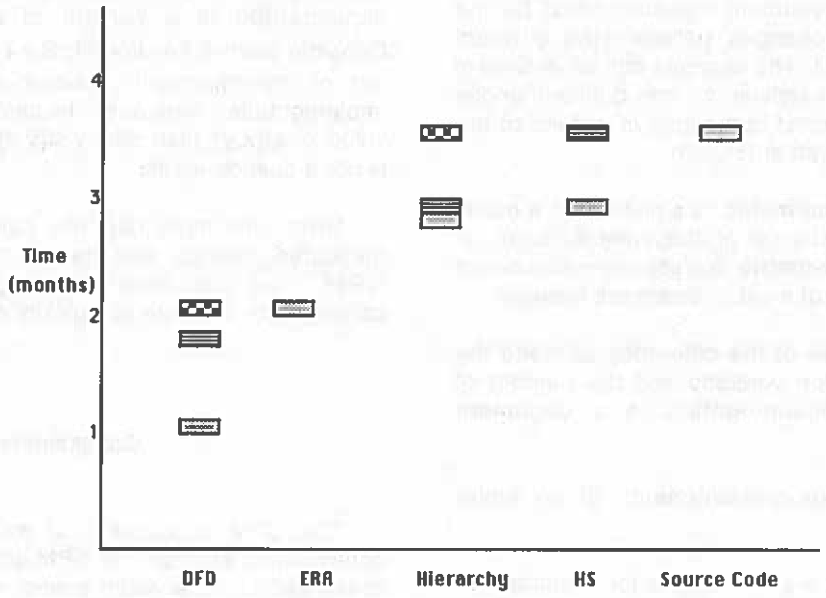

Figure 4 : SPM diagram for development B

Most defined software measures, including the software science values and McCabe's cyclomatic number, do not measure a difference between two programs. Thus we prefer to use the term "software measures" for such quantifiers which take only one argument.

Definition : Let **V** be a set of document versions. A function **f** where **f : V -> ℜ** is a **software measure.**

Definition : A **software metric** is a metric with a metric space consisting of a set of document versions. A software **pseudo-metric** is a pseudo-metric over a domain consisting of a set of document versions.

A simple example of a software pseudo-metric is the absolute value of the difference in the number of statements in two versions of a source program.

Consider a function **m** that compares the number of predicates in two source programs **a** and **b**. If version **a** and **b** have the same number of predicates $m(a,b)=0$ even when $a \neq b$. Therefore **m** is a software pseudo-metric. In fact, all published software measures can be used to define analogous pseudo-metrics.

We can also define measures whose inputs are sets of documents or entire document histories.

Definition : Let **H** be a specified set of document histories. A **document measure** is a function **f : H -> ℜ** .

For example, a document measure could be the average number of changes between the different versions of a document. The changes can be defined in various ways. As an example, we can define changes in terms of the number of operations in a delta of the source code control system (SCCS).

Definition : A **document metric** is a metric with a metric space consisting of a set of document histories. A **document pseudo-metric** is a pseudo-metric over a domain consisting of a set of document histories.

The absolute value of the difference between the number of specification versions and the number of versions of the implementation is a document pseudo-metric.

We can also take measurements of an entire project.

Definition : Let **P** be a set of **SPMs** for a number of projects. A **project measure** is a function **f : P -> ℜ**.

An example of a project measure would be average length of time between document versions. It would map the entire **SPM** to one number that represents how often the documents were updated.

Definition : A **project metric** is a metric with a metric space consisting of a set of **SPMs**. A **project pseudo-metric** is a pseudo-metric over a domain consisting of a set of SPMs.

An example of a project pseudo-metric would be the difference between the average length of time between document versions for project A and for project B.

As mentioned previously, if **m** is a measure with domain **D** we can define a pseudo-metric $pm(d_1,d_2) = |m(d_1) - m(d_2)|$, where $d_1 \varepsilon D$ and $d_2 \varepsilon D$. However, a metric can also be defined independently of a measure. One example is the previously mentioned number of changes between two document versions.

Some quantitative comparisons between pairs of objects are not metrics or pseudo-metrics. Consider the following two examples:

1) Often we want to measure the direction of change. For example, we might measure the growth in the size of a document. A pseudo-metric would only show the magnitude of the change and not distinguish between an increase or a decrease in the size. We might describe a "metric" that can show direction as a vector.

2) We may want to determine how many abstract operations in a version of a specification are correctly implemented in a version of an implementation. Consider such a function **si: S x I ->ℜ**, where **S** is the set of specification versions and **I** is the set of implementation versions. In this case, if we have a value for **si(x,y)**, then clearly **si(y,x)** is not defined and **si** is not a pseudo-metric.

Given this more carefully defined environment of measures, metrics, and pseudo-metrics based on the **SPM**, we can extend this rigor into the typical paradigm for software complexity measures research.

Conclusions

The SPM is capable of modeling all software development efforts. The SPM allows visual evaluation of the effort. This visual aspect will help in analyzing software development.

The proposed model also provides the framework to allow the development of measures and metrics. Further, it is clear that this model with its vast possibilities for various measures and metrics opens a whole new perspective on the areas of software complexity measures and metrics. In addition, the SPM provides a formal framework for controlling and analyzing these measures and metrics. These measures and metrics will help to compare different development efforts in terms of stability, rate of progress and possibly quality. We feel that this approach has tremendous possibilities as a software management tool.

bibliography

[1] Albert Baker, James Bieman, David Gustafson, and Austin Melton, "Modeling and Measuring the Software Development Process", **Proc HICCS** 1987.

[2] Robert Balzer, " A 15 year perspective on automatic programming", **IEEE Trans. Software Engineering**, SE-11 (11) , pp1257-1268, November 1985.

[3] James Bieman, Albert Baker, Paul Clites, David Gustafson, and Austin Melton, "A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification", **Journal of Systems and Software** , 8(1) pp13-37 (Jan1988).

[4] Stephen Fickas, "Automating the transformational development of software", **IEEE Trans. Software Engineering**, SE-11 (11), pp1268-1277, November 1985.

[5] Brent Hailpern, "Multiparadigm languages and environments", **IEEE Software**, 3 (1), pp6-9, January 1986.

[6] M. H. Halstead, **Elements of Software Science**, Elsevier, New York, 1977.

[7] T. J. McCabe, "A complexity measure", **IEEE Trans. Software Engineering**, SE-2 (4), pp308-320, 1976.

[8] C. V. Ramamoorthy, Atul Prakash, and Wei-Tek Tsai, "Software engineering", **Computer**, 17(10), pp191-210, October 1984.

[9] Martin L. Shooman, **Software Engineering: Design, Reliability and Management**, McGraw-Hill Book Company, 1983.

[10] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold, "Research on knowledge-based software environments at kestrel institute", **IEEE Trans. Software Engineering**, SE-11 (11), pp1278-1295, November 1985.

[11] Robert B. Terwilliger and Roy H. Campbell, "Please: predicate logic based executable specifications", **Proceedings of the ACM Computer Science Conference**, ACM, February 1986.

[12] Richard C. Waters, "The programmer's apprentice: a session with kbemacs", **IEEE Trans. Software Engineering**, SE-11 (11), pp1296-1320, November 1985.