MODELING SOFTWARE ARTIFACT COUNT ATTRIBUTE WITH S-CURVES

A Dissertation

by

NORMAN K. MA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2007

Major Subject: Computer Science

MODELING SOFTWARE ARTIFACT COUNT ATTRIBUTE WITH S-CURVES

A Dissertation

by

NORMAN K. MA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Co-Chairs of Committee, Dick B. Simmons
                                              William M. Lively
Committee Members, Robert J. Hall
                                              Udo W. Pooch
Head of Department, Valerie E. Taylor

December 2007

Major Subject: Computer Science

ABSTRACT

Modeling Software Artifact Count Attribute with S-Curves.

(December 2007)

Norman K. Ma, B.S., University of Illinois at Urbana-Champaign;

M.S., University of Tennessee at Knoxville;

M.B.A., Southern Methodist University

Co-Chairs of Advisory Committee: Dr. Dick B. Simmons
Dr. William M. Lively

The estimation of software project attributes, such as size, is important for software project resource planning and process control. However, research regarding software attribute modeling, such as size, effort, and cost, are high-level and static in nature. This research defines a new operation-level software project attribute that describes the operational characteristic of a software project. The result is a measurement based on the s-curve parameter that can be used as a control variable for software project management. This result is derived from modeling the count of artifact instances created by the software engineering process, which are stored by software tools. Because of the orthogonal origin of this attribute in regard to traditional static estimators, this s-curve based software attribute can function as an additional indicator of software project activities and also as a quantitative metric for assessing development team capability.

DEDICATION

to my grandparents and my parents, David and Ching.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE   Page

CHAPTER I

INTRODUCTION

Estimation of software project attributes (such as size) is important for project resource

planning and process control. However, size, effort, and cost, do not show the dynamic

nature of the software engineering process. While concepts like 'software project' are

generally understood, they are not often understood in detail. Object Management

Group's Four-layer Metamodel Hierarchy [36] utilizes a framework in order to account

for various elements of a software project before proceeding to count artifact instances.

In addition, Appendix A contains a glossary of terminology that can provide grounding

for ambiguous terms. The Four-layer Metamodel Hierarchy is graphically displayed in

Figure 1 below:
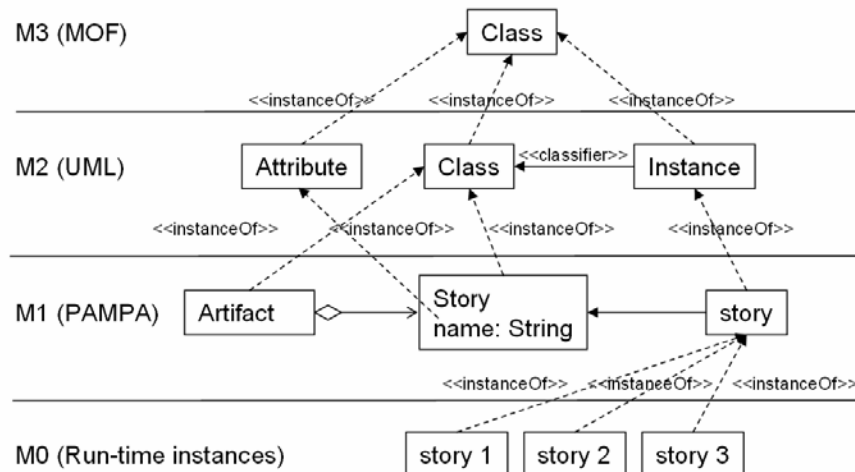
Fig. 1. Example of OMG's four-layer metamodel hierarchy [36]

Each layer of the hierarchy defines a language that can be instantiated at the lower layer. The most well known layers are M2 and M1, where the M2 layer defines the Unified Modeling Language (UML) ; at layer M1, users use UML to create a particular system called a user model. In other words, the user's model is an instantiation of UML. Finally, when the user model is running, instances of the elements of user model come into existence at layer M0. It is the counting of instances at layer M0 of the PAMPA software project user model that is the focus of this dissertation.

The UML is a de facto graphic-based modeling language for describing the logical, process, physical, development views of a system [35]. The PAMPA knowledge base model describes the various parts of a software project and is constructed using the UML at layer M1. During a software project, instances of PAMPA elements are instantiated at layer M0. From here, a 'class attribute' describes a characteristic of a class, and an 'instance attribute' describes a characteristic of an instance. Of those described, the model collects attributes that are measurable or countable, the definitions of which are described in Figure 2:
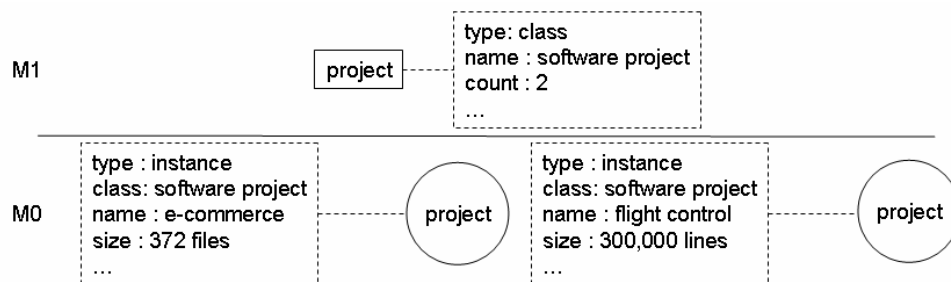
Fig. 2. Two instances of the project class

The figure contains the 'software project' class represented by a rectangle and two instances of the software project class, whereas the dotted rectangle contains the attributes of the corresponding object. For the software project class, the value of the 'type' attribute is 'class', the value of the 'name' attribute is 'software project', and the 'count' attribute is set at a value of 2, indicating that two instances of the software project class exist; the latter represented as circles, located just below the class. Each instance has multiple, corresponding attributes, such as type, class, name, size, etc. For instance, the object with the name 'e-commerce' has a 'size' attribute that contains the number 372; furthermore, this size attribute is also measured, in units of 'files'. The second instance has its own set of attribute-value pairs and are listed as follows: (type - 'instance'), (class - 'software project'), (name - 'flight control'), (size - 300,000 lines of code). We note that the 'flight control' instance's size attribute is measured in units of 'lines of code', with a quantity of 300,000. Moreover, the software project class has a class attribute named 'count'. That attribute has a value of 2 and is measured in the unit of 'software project instance'. The next section describes artifact. The focus of the above discussion is the project class. If the discussion is about a composition [18] such as the PAMPA project list, i.e. ProjectList, class, then two instances of the project list class can be represented as in Figure 3:
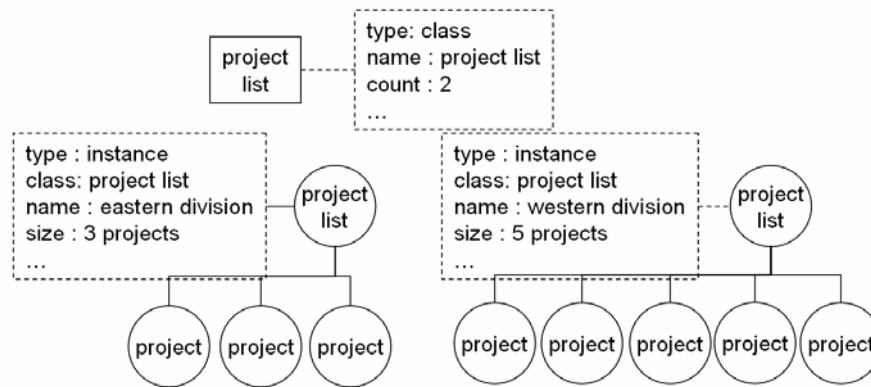
Fig. 3. Two instances of the project list class

Figure 4 is the PAMPA knowledge base, which contains 35 classes. Both the project list class and the project class are part of this PAMPA class diagram.
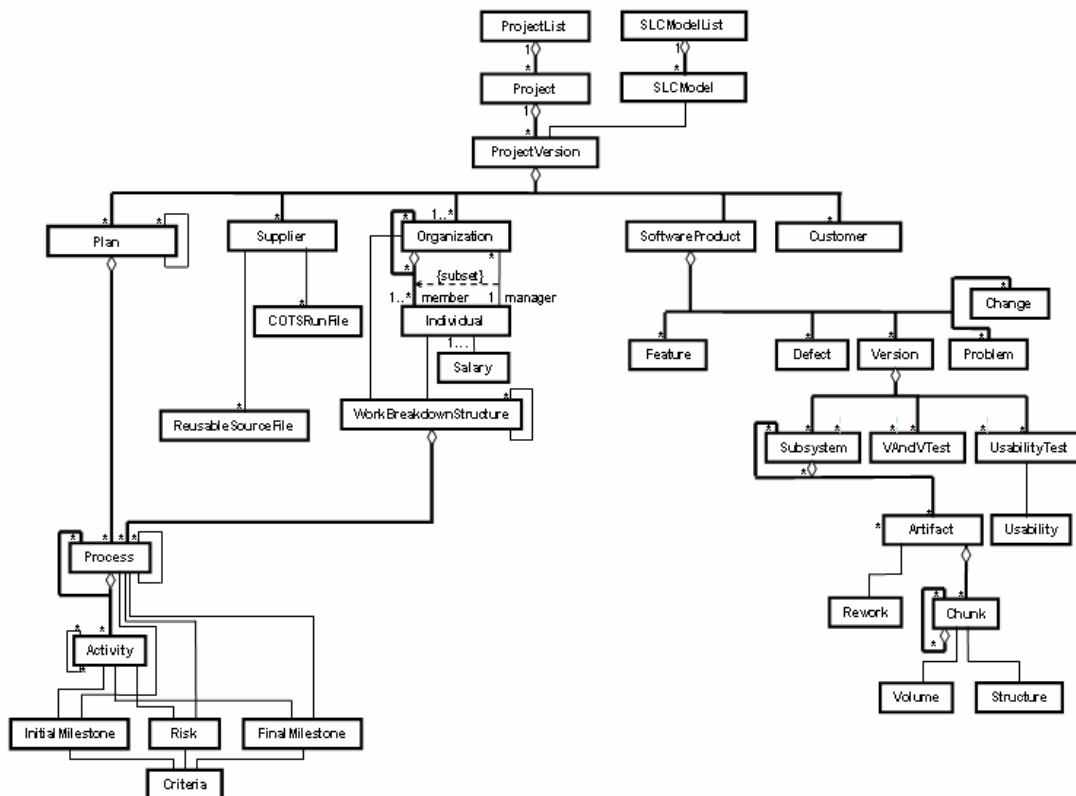
Fig. 4. PAMPA classes

We are interested in the artifact class at the lower right hand side of the graph because artifacts are tangible results created by software engineering activities, where Artifact is marked as a component of Subsystem, and Artifact is composed of Chunk. In this model, artifacts are defined as project objects that are created and stored by software tools. In this case, the PAMPA artifact class is an abstract class composed of instantiateable classes:
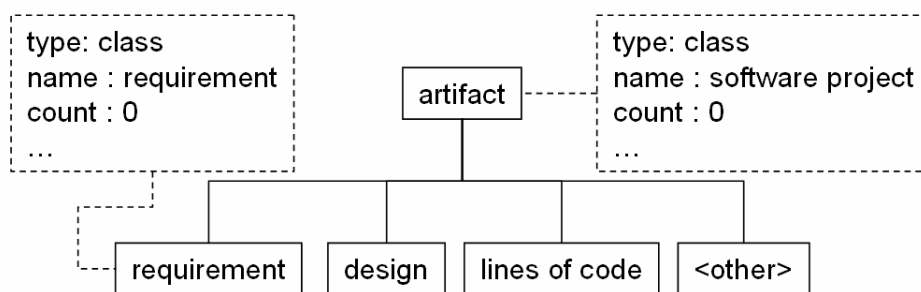
Fig. 5. Type of artifact classes

Figure 5 illustrates further detail within the artifact's three sub-classes: requirement, design, lines of code, and attributes of the requirement class; the count class attribute within the requirement class is zero because there are no instantiation of objects from that class. Further detail is necessary when software projects generate requirements, such as use cases or stories, and other types of artifacts during subsequent phases of the software development cycle.
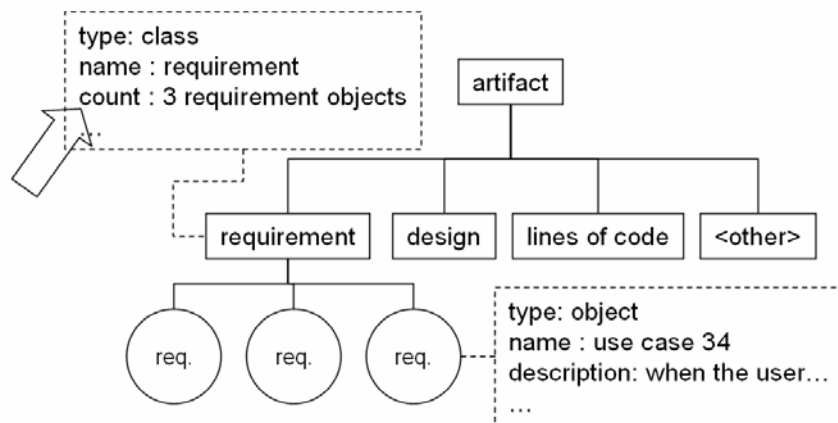
Fig. 6. Instantiated artifact objects

The model is interested in measuring and recording the change of the count class attribute of the requirement class, which is graphically identified in Figure 6 with an arrow. In this particular scenario, there are 3 use case instances, and we say:

"There are 3 requirement instances in the system. We are particularly interested in the 'count' class attribute of the requirement class. The count class attribute is measured by counting the number of requirement instances. At this time, the count class attribute has a quantity of 3 with unit of 'requirement instances'."

and we represent the above statement mathematically through the following formula:

$requirement\_record = ((t_n,3),(t_{n+1},6),(t_{n+2},37))$, which indicates that there are 3 requirement instances at time *n,* 6 at time *n+1,* and 37 at time *n+2.*

The focus of this research is on the trend of the 'count' class attribute of artifacts. For each artifact record, both a straight line and an s-curve are used to model the dynamic change of the count variable. To accomplish this, a format was created and will

be introduced later in the paper; its function is to focus on the growth and readiness of the artifact count. These are two important operational-level measurements that can help in managing software on a day-to-day basis. Finally, a procedure to manage the process and creation of software artifacts is introduced.

In creating and introducing these measures, this dissertation has contributed to the field by identifying artifact counts as a grounded software project management activity measurement [29], thus creating a way to use them to both measure and to control the artifact generation process which in turn provides detailed in-process indicator of software engineering processes to help better managing the day-to-day activities of software engineering projects. These has been significant increase in the quantity of software code that are being created, both due to improving in software technology and increase in overall software engineers. However managing software development are still mostly at the requirement level where the day-to-day activities are not being measured. However, the availability of s-curve parameters presented in this dissertation can be a starting point in the more scientific management of the software development process.

A. Software Artifact Attribute Magnitude

Software Engineering is a result-oriented endeavor executed through disciplined processes, whereas software artifacts are essential results of software engineering. The latter is defined as a measurable item, retrievable with computer aided software engineering (CASE) tools; since a successful software project produces software

artifacts that meet Requirements [11], examining artifact changes during the software life cycle can improve the production processes By which they are created. The specific variable that this study examines is the *count* of artifacts; the artifact instances are collected from all software life cycle phases, gathered by software tools.

To date, a significant amount of software research has been focused on the Point Estimation of project attributes, such as size, defect count [56], and cost of software products [1, 4, 7, 24, 27]; only general, loose research has been conducted regarding production goals and estimations, especially those that are predictive, using information artifacts at early phases of the software life cycle, such as use cases, lines of code, object points, functional points [28], etc. A successful predictive model would make software cost estimates more accurate, and project resource allocation more proactive.

While software engineering tools' function is the transformation of artifacts from high-level human minds down to structured machine code that conforms to the Software Engineering Transformation Axis (SETA). At the more detailed end of the software engineering activity spectrum is executable code and source files; from these, researchers can generate detailed artifact visualizations [49, 51] retrieved from software tools, such as a configuration management system. As an example, a succinct mid-level software project perspective in SETA is provided by the PAMPA (Project Attribute Monitoring and Prediction Associate) software project template [42, 52]. However, neither the top-level attribute estimations, nor the low-level visualization techniques yield a perspective that's detailed enough to understand artifact generation activities throughout the software life cycle.

Software engineering is result-oriented and, in order to achieve results, efficient artifact creation is necessary [47]. Since this process is usually team-oriented, each team member's choices are important factors in determining efficiency and efficacy. To date, no other research has sufficiently examined these choices - choices that result in determining a project's direction. Moreover, the path of each team member on a decision tree splits quickly because there are so many choices and variables along the way; these commonly include size, defect, and cost.

We assert that continual storage and measurements of artifact values during software development can provide standardized [21, 22], quantitative values that help guide a detailed understanding of software artifact creation activities [2]. This dissertation achieves this by making a departure from tradition thought, in order to present a behavior of artifact magnitudes graphed and described using both s-curves [15] and straight lines from liner regressions. S-curves, traditionally found to be useful in describing technology adoption behaviors [12], are also useful when describing the magnitude of software projects; this is confirmed by our independent research. Using data from an experiment the researchers compared the S-curve against a linear graph approach and found the former to be superior.

B. Organization of the Dissertation

The rest of the dissertation is organized as follows. In Chapter II, operation-level software engineering activities are defined. Chapter III presents the experiment, where the collection of software project data is described. Chapter IV contains a description of

the measured data. Chapter V concludes the discussion by summarizing the findings and

suggesting future work.

CHAPTER II

OPERATIONAL LEVEL SOFTWARE ENGINEERING ACTIVITIES

A. Introduction

According to the Cone of Uncertainty software project description [8], the uncertainty of the possible cost, size, and features of a software product progressively decreases along the software construction phases, namely: initial concept, product definition, marketing requirement, technical requirement, design, test cases, and development. Within this process, different participants are interested in different objectives [33]. For instance, producers are interested in profit, software engineers are interested in building a quality product, software managers are interested in productivity and budgets [26], and users care about the value that the software system brings to their lives [9]. Many of these questions hinge on the estimation of software size and cost [19]. From the accounting perspective, one asks questions such as which account to charge for "time spent on talking with the customer" or question of fixed cost allocation. These individual accounting decisions affect the eventually profitability of a software project. However, existing research [5] has not addressed the lack of detailed association between software accounts and software artifacts created during the software life cycle; tracking artifacts is important to cost estimation because it can use the life cycle to breakdown software costs [34, 48]. Unfortunately though, only high-level accounting information are available to project management in most software projects. This research presents a way

to more easily track *Artifacts* - defined as any object that is stored by software tools. The goal of software projects is to create executable code that satisfies requirements derived from the original concept. The process of creating software involves many steps; to accurately estimate costs, a model needs to individually examine these steps.

 A program starts with an object code, created by the *assembler* software tool; this tool then translates assembly code to specific machine code. The source of assembly code is run through language compilers, which then translates source code into assembly code. Moving up the software translation axis, source code are generated either by software engineers or by automatic program generators; these then automatic program generators can create source code based on design document that are used by software engineers.

Moving further up the translation axis, design documentation and specification are created by human from requirement documents. We define all the intermediate items that represent the original software product concept as *Artifacts*, including the final machine code. We also note that these progressively more specific artifacts are created by software tools and humans. Each type of artifact is associated with a number and a unit, for example, a use-case type artifact might have a *value* of '7 Use Case Count' and a machine code type artifact might have a *value* of '5,783 Byte Count'. Specifically, the types of artifact that have been collected are shown in Table 1:

Table 1. Artifact and Artifact Unit

Artifact Type	Artifact Unit	Example Artifact Value
Issue	Issue Count	'12 Issue Count'
Source File	Source File Count	'3,541 Source File Count'
Line of Code	Line of Code Count	'7,758 Line of Code Count'
Design Object	Design Object Count	'14 Design Object Count'
Test Case Yes	Test Case Yes Count	'29 Test Case Yes Count'
Requirement	Requirement Count	'74 Requirement Count'
Database Table	Database Table Count	'7 Database Table Count'
Operand	Operand Count	'6,622 Operand Count'
Operator	Operator Count	'3,940 Operator Count'
Unique Operand	Unique Operand Count	'1,158 Unique Operand Count'
Unique Operator	Unique Operator Count	'23 Unique Operator Count'

We have described the various types of software artifacts and software tools that generate those artifacts. PAMPA (Project Attribute Monitoring and Prediction Associate) provides a perspective on the relationship between software project objects and a framework for the application of software processes [49]. The PAMPA perspective is shown below:
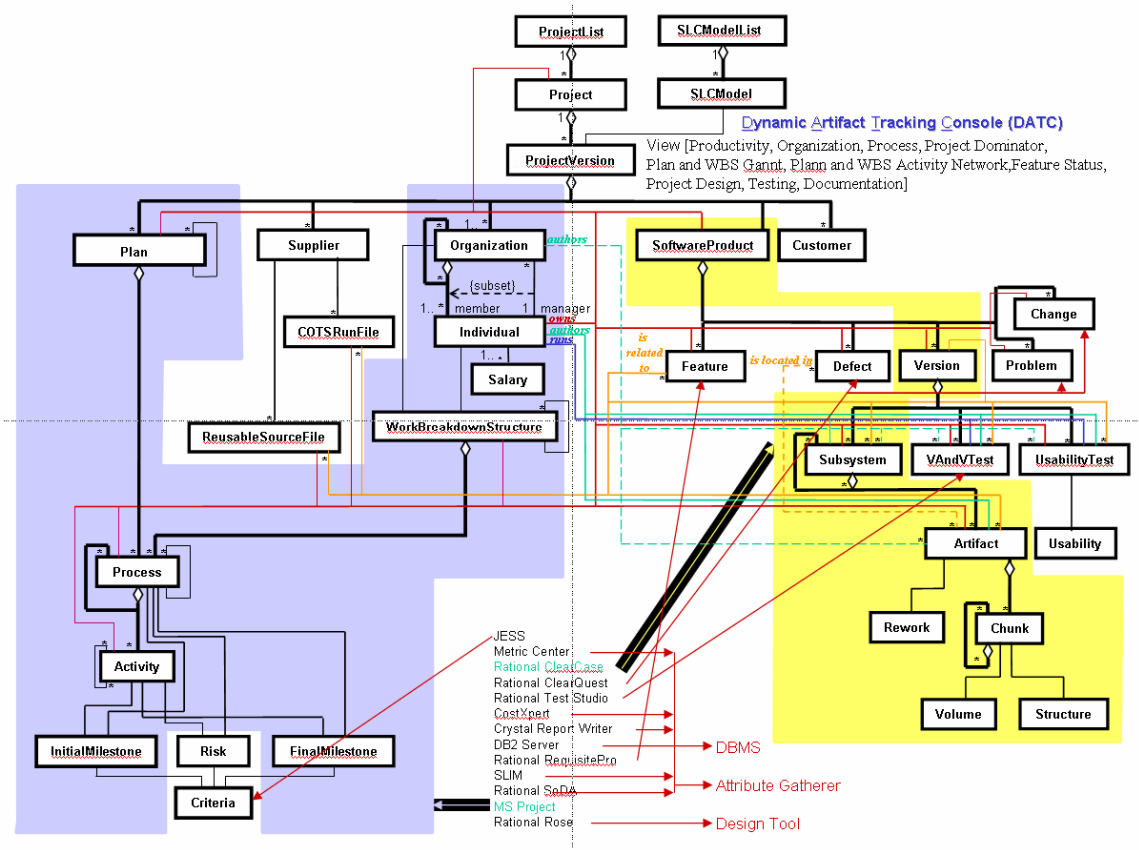
Fig. 7. Project Attribute Monitoring and Prediction Associate (PAMPA)

The software tools at the lower-center part of Figure 7 both help the technical personnel to create the software system and provide important project measurements for software project management. This research focuses on the operational-level activities of generating artifacts. Artifacts - the essential result of a software project, also include very structured software Executables and Source Code. For example, a high-level software engineering artifact could be a story (contains 31 words) such as:

> Write a short program to verify the successful creation of a development environment for developing C language applications, verify the editor, compiler, and the integrated development environment have been installed correctly.

and a partial listing of the complete transformation of the above story along the same conceptual transformation axis to the final low-level Intel processor-based machine code (containing 8,873 bytes):

```
0016540 000000 002401 000000 000570 000000 000004 000000 000002
0016560 000000 000000 002427 000000 050000 000100 237777 000000
0016600 000002 002447 000000 053537 067151 060515 067151 051103
0016660 061537 072162 000060 057537 074543 073547 067151 061537
```

The above example lists the extreme possibilities of an artifact.

Software projects can be defined in three-levels: Strategic, Operational, and Tactical. The former can include deciding what to build and placement of the software product in the market place, Tactical activities can include locating defects or building of an executable, while the majority of the software engineering process involve Operational activities. These include processes like artifacts translation, from general artifacts to more specific types, then finally to executable code. The most interesting aspect to both researchers and practitioners is the translation from a high-level concept to concrete machine code. This involves operation-level perspective of daily activities of generating software artifacts. These activities are usually facilitated by software tools, such as configuration management, graphical design, issue tracking, requirement management, etc. Recent software engineering environments, involving concurrent wide-geographic development, Agile development [3, 6], commercial off-the-shelf (COTS), and component engineering, also highlight the utility of software engineering tools as a binder that unites the software engineering processes. Software engineering tools provide situation awareness of the present software project [14, 45]; that is,

providing a managerial-level perspective of the life cycle of software artifacts through out the requirement, testing, design, development, and maintenance phases of a software project.

In this dissertation, a software project's artifacts have been collected using software engineering tools. Eleven project artifacts have been collected as part of the Canonical Attribute Project Set (CAPS). They are Requirement Count, Lines of Code, File Count, Issue Count, Design Objects, Test Cases, Unique Operator Count, Unique Operand Count, Operator Count, Operand Count, and Database Table Count.

B. Software Artifact Description

Requirement Count is the number of requirements derived from Extreme Programming stories. An Extreme Programming (XP) [37, 41] practice story gives a description of desired system behavior. Larger more vague stories can be broken down into sub-requirements or Use Cases. A Use Case is a specific description of a functionality provide by the system to the user.

Design Object Count is the number of design-related artifacts generated from the requirement. Design objects based on the Unified Modeling Language (UML) include Use Cases, Object Diagrams, Class Diagram, Relational Database Model, Sequence Diagram, etc.

Database Table Count is the number of database tables created to meet the requirement. Usually each table represent a Class in the object-oriented representation.

In addition, database tables can also represent Business Processes, Conceptual Ideas, and any other items that need to be processed by computing systems.

Lines of Code (LOC) are the lines of source code that were created by the develop team to satisfy the requirements. In this particular study, JavaScript and JSP are the main types of source code, which are instantiated by an Apache web server when accessed by a web client.

File Count is the number of files that were created by the development team to satisfy the requirements. Both external files and team-created files are involved in many software projects. The source of external files stem mainly from the user interface, database, and web services platforms. These include graphical user interface builders and help files, database source files and interface files, and web server source code and interface files.

Unique Operator Count is the number of unique operators in the source code. Operators transformation of numbers and numerical calculations. In addition, operator can transform strings and software objects.

Unique Operand Count is the number of unique operands in the source code. These are mainly variables that representing numbers, text, and codified conceptual objects.

Operator Count is the total number of operators in the source code to meet the requirements. Operators indicate the size and variety of transformation that the software project uses to satisfy the requirement.

Operand Count is the total number of operands in the source code created to meet the requirements. Operand count indicates in general the scope and size of facts that need to be represented, managed, and transformed by the software product to satisfy the requirements.

Issue Count is the number of report that shows deviation from requirements or expected software behavior that significantly affect the efficiency of the interaction between the software system and the user. In addition, issues also describe software development situations that affect the effective operation of the development process. An example of development process-related issues include: development environment readiness and efficiency. While readiness and efficiency are not quantitative, reports of these situations are countable.

C. Conclusion

In this chapter, we have described the activities within the software engineering process that generate artifacts; in addition, artifacts from the software life cycle phases during the experiment were described and defined. Following, terms were introduced in order to set the context of the software project experiment. A brief description of the concept and purpose of software engineering tools were also given, namely, the translation of software engineering artifacts into progressively more specific artifacts.

CHAPTER III

COLLECTING PROJECT DATA

A. Introduction

Artifacts were collected using software engineering tools from a single semester graduate-level software engineering course. The course lab structure was based on industry software development organization and structure. The project followed the Extreme Programming practice (XP) and a successful electronic commerce web site named Purchase Tracker was created. Software engineering tools were used to facilitate the construction of the electronic commerce web site and were also used for collecting software artifacts [54].

B. Experiment Description

Eighteen graduate students participated in this software engineering project, which mirrored an industrial software development project. The project's goal as part of a graduate-level software engineering course was to create an electronic commerce web-site named Purchase Tracker. In addition, part of the class formed a separate team whose role was to collect project artifacts that were being created by the Application Team. The Application Team followed the Extreme Programming (XP) practice throughout the project period (100 days) and gave five demos, one each at project day 24, 52, 80, 94,

and 100. The application was successfully completed by satisfying 29 out of the 30 test cases.

The structure of the two teams is shown in Figure 8:
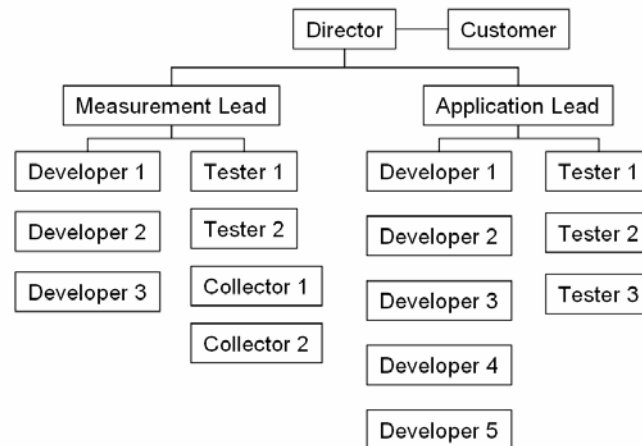
Fig. 8. Experiment team organization

The customer role provided the team with stories for both the application team and the measurement team. The main function of the Customer is to clarify requirement and provide feed back to the project. The Director is the over-all coordinator of activities. Each team has a Project Lead, charged with carrying out the Developer's role. Each of the team members are assigned three stories through mutual agreement.

The developers of the application team focused on the electronic commerce application and carried out the extreme programming practice with parallel requirement gathering, design, development, and testing. Fire application demonstrations were carried out; these were important milestones in moving the artifact magnitudes towards

the final value. The main function of the testers was to author the Test Cases that validate the Stories. The final test cases were agreed upon by the developers, and the testers carried out the testing throughout the development process. Two Collectors were responsible for the task of collecting artifacts. Their roles were specifically created to assure focus and consistency of the collecting process.

Weekly reports were written and entered into the SSIP (Shared Software Infrastructure Program) web site. The team members were able to view each other's weekly report to enhance communication. The team members were recommended to spend around 9 hours per week on the project; thus, over 2,340 labor hours were spent on the Application and the Measurement project.

C. Application Description

The application is a three-tier electronic application with a web-based user interface, an application server (hosting Java server page code), and a database server.

D. Software Tools Description

Many software tools were used to collect artifact information: RequisitePro was used to track the requirements; Rational Software Architect (RSA) was used to construct design phase artifacts; Eclipse was used as the Integrated Software Development platform; ClearQuest was used to track the issues that were generated by the team during the development process. In addition, a configuration management system was used to store artifacts, and operating system shell scripts were used to collect artifacts. The Dynamic

Artifact Tracking Console (DATC) was also used to assist in the collection of artifact

from the tools.

E. Process Description

The quality of crafted software depends strongly on the ability of individual

programmer; this differs from engineering software, wherein predictable applications are

possible with a various range of programmers with different skill sets and experiences.

This is due to the software engineering processes, where each participant is responsible

for one or more software processes. Below is a table, Table 2, of the processes that were

followed for the project experiment:

Table 2. Software Engineering Experiment Team Processes

Measurement process	1.1 Form the team and assign role
	
	1.2 Requirement Gathering Process
	    Define initial software project measurements for collection
	
	1.3 Understand the operation of existing measurement tools
	    Understand requirement collection tool operation
	    Understand configuration collection tool operation
	    Understand problem report collection tool operation
	1.4 Create database for storing measurements
	1.5 While not end of Application Project
	    1.5.1 Collect measurements daily
	    1.5. 2 Store measurement
	    1.5.3 Display measurement
	    1.5.4 If project demonstration time
	        Demonstrate project
	    End If
	End while

Table 2. Continued

Application Development process	2.1 Form the team and assign role
	2.2 Requirement Gathering Process
	Generate Use Cases
	Talk to project director and customer
	2.3 While Application is not done
	2.3.1 Design Process
	2.3.2 Implementation Process
	Understand design
	If source file not created
	Create source file in configuration system
	Check out source file
	Edit source file
	Unit test
	Check in source file
	2.3.3 Testing Process
	If there is a problem
	Do problem report process
	2.3.4 Build Process
	Check out source code
	Build application
	Report build result
	2.3.5 If project demonstration time
	Demonstration project
	End If
	End While

Configuration Management Process	3.1 Set up initial software development tools Set up requirement management tool Set up source code configuration tool Set up problem report tool 3.2 Build Application Project daily
Problem Report Process	4.1 Generate problem report
Problem Resolution Process	5.1 View Problems 5.2 Assign priority 5.3 Assign problem to appropriate role
Consultation Process	6.1 Identify uncertainty and form question

Table 2. Continued

Consultation Process	6.2 Ask consultant question 6.3 Listen to answer and resolve uncertainty
Course Administration Process	7.1 Enter weekly report at the HUB web site by Saturday. 7.2 Attend weekly laboratory coordination 7.3 Read and respond to cpsc606 emails 7.4 Contribute to project discussion

F. Software Artifact Source

Software artifacts were collected during the experiment to give the stakeholders a higher

and more abstract level of project situation assessment. We focus on software artifacts

because they are tangible and measurable. The necessary product for this particular

project is binary, Intel Corporation code that moves from hard disk storage to random

access memory, then to processing unit registers and cache memory. This code, in the

processing unit, receives environmental human inputs mediated through the likely path

of a remote client computer transferred through the network protocol stack. The received

signals are processed according to the operators defined by the machine processor;

output signals are then emitted from the processor to the user through a similar path. In

addition to the response to the human user, some of the output signals might be targeted

toward the manipulation of an external environment not directly related to the human

user. Such signals might impart storage and retrieval of information and data.

While the result of the software engineering process is binary code, it takes software tools to achieve this. In the earliest days of programming, binary code was directly entered into the computing machine. In those earlier days of software engineering, when viewing the phases of software life cycle and possible artifacts, most of the indicators were displayed in unstructured, text form. In the requirement phase, the developer would formulate the environment and possibly transcribe them in the laboratory notebook. At the same time, he would start to design the software and also have the option of writing down any design on the laboratory notebook. The development and phases would possibly involve detailed step-by-step instruction where "unit test" is carried out after the actual bits of a register has been loaded into the core memory. Expectation of the result can be written down in a structured way or might be kept in the 'developer' mind. The artifacts from this earliest stage of software development is the laboratory notebook. The machine executables were not part of the artifact because the code were not stored.

The first historic significant tool in the programming paradigm is the assembler. This is a software tool that translates written human language mnemonics into binary code. Assembly language is human readable and bridges between the conceptual level of software engineering to the binary software product. The ensuing development of software tools takes a higher concept object along this conceptual axis and translates it down to a more concrete object. We call this the Software Engineering Translation Axis (SETA). The instantiation of the highest level of abstraction along this axis are human thoughts, the next lower level, or more concrete level, are natural written and spoken

words, and the levels below that are more structured thoughts that are characterized by sequence of patterns. Continuing to the specific, the next level SETA objects are structured, written and spoken words, such as an 'instructional manual' or 'medical doctor patient oral report protocol', while the lowest level of SETA are physical binary code instantiations that control binary gates of a computing machinery processor.

Within the SETA context, we define a SETA object, *s*, as an object with a structure property, ' *struct*(·) '. The upper limit of the structure property of a SETA object is human thought; for the purpose of software engineering artifact measurement, we define a human thought unit as a record of time lapse of three-dimensional electrical activities, *T*. It is interesting to note that while we are describing the translation of abstract and unstructured thoughts down to definitive instructions on a machine processor, the fact is that these SETA objects, which are at the extreme ends of the axis, are both instantiated as time lapsed electrical activities. Excruciating amounts of resources are currently spent on translating between these two sets of electrical patterns, mostly from the less-structured end to the more-structured end for software engineering processes. More specifically, this research focuses on a particular type of human thought - conceptual thoughts that contain patterns. This research does not concern brain firing patterns, essentially anything within the brain that spurs action outside of it (such as moving limbs or other body parts); this is because of the causal relationship between the cause and effect of these thoughts.

For example, this research would model patterns from a pilot's thoughts during landing, or something as abstract as a pattern representing a rocket's launch to Mars. To

point out the expansiveness and immediacy of human thought - a set might include a stone carver chipping away on a large round stone disk, another could be an English publication that interprets the carvings on the stone disk. As we have illustrated from these examples, the present capabilities of computing machineries are certainly not capable of physically instantiate all high-level SETA objects. We focus on preferred sequence of patterns because these high-level SETA object can have interesting software engineering (ISE) consequences; they are transferable to executable code that can run on state-of-the-art computing machines. We represent these ISE conceptual thoughts as $\{o_T \mid ISE(o_T) > \mu\}$, where *mu* is a threshold of interestingness that is based on the present capability of the software engineering processes. That is, ISE objects afford an opportunity for software engineering processes to translate these patterns down to binary code (which we represent it by $o_{executable}$) that controls machine processors to instantiate the high-level ISE pattern. This has been difficult because of possibility of a wide gap between the representation capabilities of SETA objects at the ends of the spectrum.

At one end of the axis, patterns that represent different time periods, wide geographic locations, and range of details can all co-exist at the same time in a mind; at the other exist computing machines that, at the time of this research, are still not capable of representing the same patterns as the mind. However, that assertion has not been calculated. Moreover, it is not the purpose software engineering to duplicate the pattern of thoughts on computing machineries. The practice of software engineering is to instantiate high-level SETA objects to machine instructions that has an impact in the real-world. Taking the earlier landing gear example, it is mischievous to instantiate

computer code that displays pictures of the landing process. It is prudent and correct if

the software engineering process created an embedded landing gear control system that

was represented by the original high-level SETA object. At this point, we simply define

the software engineering processes as activities that translates a human thought SETA

object to a binary SETA object. That is, below is the transformation of a SETA object by

the software engineering processes

$$o_{begin} \rightarrow o_1 \rightarrow \cdots \rightarrow o_n \rightarrow o_{executable}$$

We define a Software Engineering Tool as a set of *translators* where each translator can

change the structure of SETA object. Let $O = \{o_i, o_j, \cdots, o_n\}$ be a set of SETA objects and

$t(O_a) \mapsto O_b$ be a translator that maps a set of software engineering objects to another set

of software engineering objects; thus, Software Engineering Tools transform one or

more objects into more objects with the main goal of eventually creating the binary

object, $T = \bigcup t$. For example, the traditional sequential software life cycle can be

represented as:

$$o_{begin} \xrightarrow{T_{requirement}} O_{requirement} \xrightarrow{T_{testing}, T_{design}} O_{testing} \wedge O_{design} \xrightarrow{T_{development}} O_{development} \wedge o_{executable}$$

In this sequence, a high-level object is the input for the requirement tool which resulted

in as set of requirements objects. These requirement objects are consumed by testing and

design tools to generate testing and design objects. Developers take testing and design

objects and create development objects, including the goal of the software engineering

process, the executables. We define software *artifacts* as SETA objects generated by

software engineering tools. Software artifacts are alternatively defined as SETA objects that have been created using software tools.

We take a closer examination of the relationship between SETA objects, tools, and the software engineering process by using a simple example. In this example, researchers want to generate a set of canon angles for accurate placement of projectiles. In this case, the high-level source object - the beginning of the software engineering process, includes a pattern of parabolic mathematical equations, images of canons, and understanding of wind, direction, weather conditions, and other factors that can affect the flight of trajectory. Another pattern in the beginning set of high-level objects is a soldier looking up a firing table and sets the canon according to the numbers printed in the table; they are very simple patterns that are easily understood at a high level. The other end of SETA spectrum is binary or executable codes that display values in the firing table. We examine the software engineering process with the following:

$$O_{begin} \xrightarrow{T_{notebook}} o_{design} \wedge o_{requirement} \wedge o_{development} \wedge o_{executable}$$

The original pattern object is on the left-hand side of the graph (above), and the final executable object at the right-hand side. The software engineering process involves the utilization of the software engineering tool that is an engineering notebook. In the notebook the requirement, design, development, and executable are all recorded in an orderly fashion, and artifacts (design, requirement, development, executable) can be obtained directly from the software engineering tool. We note that even at this very simple level, the software engineering tool function as an extended memory and

organizer of the human mind; it is a fact that selection of appropriate software tools can assure successful execution of software engineering process.

We now look at a more recent example of software engineering project, Web Services. Once again, the high-level SETA object is fairly straight forward. It contains some patters of ideas. In this web services case, the pattern would be multi-perspective but simple nevertheless due to its high-level. The *begin* object contains patterns of sales transaction, concept database, and value of information. There might be a storyboard-like sequence of a client computer automatically asking geo-location server its latest location; in the process, one pays the server computer a small sum for the information. The client computer then contacts a highway traffic server for the estimate congestion spots; it also pays the server a price for the information and, with the information, figures out the best route to the destination. This high-level SETA object would take longer to be translated to executables, and software tools and disciplined software engineering processes certainly would be necessary in this endeavor. We display the transformation below:

$$o_{begin} \xrightarrow{T_{requirement}} O_{requirement} \xrightarrow{T_{testing},T_{design},T_{development}} O_{testing} \wedge O_{design} \wedge O_{development} \wedge o_{executable}$$

The above is a description of the Extreme Programming practice, where testing, design, and development are executed in parallel. We note that in the description above, the required tools are used in order to generate requirement objects before the testing, design, and the development process. This order need not be followed strictly, since high-level requirement object would need to be translated into more structured objects,

in order to move towards the concrete executable along the Software Engineering Translation Axis.

G. Conclusion

The project is an Extreme Programming (XP) project carried out by a team with some inexperienced team members. Artifacts were collected to assess the operational activities of the team. In addition, the project collected data that indicates the generation of software attribute magnitudes during the project. This practice is 'developer and result-centric', wherein a small number of capable developers have a clear vision of the final product and are charged to produce a product with a high demonstration rate and low documentation activity [40]. The functionality of the final product is based on the personal activities carried out by the software engineers, thus it is paramount that the team members understand the expectation of the final product. In the Extreme Programming practice, documentation and testing activities are traded for rapid turn-around time and frequent Demonstrations. The artifacts exhibit evidence of the Extreme Programming activities that have been carried out in this software engineering experiment.

CHAPTER IV

MEASURE AND DATA DESCRIPTION

A. Introduction

Significant progress has been made concerning software engineering processes and project estimation of cost and size. As processes are broken down to activities, operational measurement becomes valuable to software team members, developers, leads, and managers because operation-level activities generate artifacts. Availability of measurement [55] is analog to a mirror, and can give a person visual feedback for improvement; this is in contrast to high-level measurements, such as cost. Understanding operational activities however, require visibility at the activity level [53]; thus, software tools that store the result of operational level activities (namely artifacts) can be used additionally as a tool for an operational-level activity assessor.

For example, imagine a software engineer sitting in from of a state-of-the-art machine displaying an Integrated Development Environment (IDE). How does the software engineer know the state of the software project, or even the progress of his own particular part of the project? Similarly, how does the software manager answer the same question? In the experiment using the Extreme Programming practice, the remedy is to have a Demonstration as often as possible. However, this only solves part of the problem, as a Demonstration is only a local illustration of a much larger software landscape. A chart that spans time can provide higher-level perspective that can benefit

equally higher-level actions such as resource planning [46]. This section gives

prescriptive direction regarding the processing of artifact information collected from

software engineering tools to a standard format called Normal Proportion Artifact Graph

(NPAG). NPAG enables the display of multiple artifacts in a single graph without Unit

Collision or Scaling Problem. From this format, we give a graphing procedures using the

s-curve and liner fitting because these can generate grounded quantitative measurements

and visualizations that serve to improve team members' understandings of the present

state of the software project. Lastly, we provide a procedure for using the graph

parameter values as an easy-to-use Control Variable for the operation-level, artifact

generation, software engineering processes.

B. Unit Description

The record of collected artifacts is instantiated as a sequence of pairs, where the first

item of the pair is a time-dependent value and the second item is an artifact-dependent

value. For this experiment, the unit of the time-dependent value is 'day' and the units of

the artifacts being collected are listed in Table 3:

Table 3. Description of Artifacts' Units

Artifact	Definition	Unit	Example
Unique Operator	Unique operators inside the source code.	"Unique Operator"	In this routine there are 3 Unique Operators: '+', "*", '-".
Unique Operand	Unique operands inside the source code.	"Unique Operand"	In this routine there are 2 unique operands: "count" and "max_count".
Operator	Number of operators inside the source	"Operator"	There are 5 operators in the routine: '+', '+', '+', '-', and

Table 3. Continued

Artifact	Definition	Unit	Example
	code.		'/'
Operand	Number of operands inside the source code.	"Operand"	There are 4 operands in this routine: "count", "count", "count", "max_count".
Database Table	Number of database tables used for the application	"Table"	There are 6 database used in the application.
Requirement	Number of requirements.	"Requirement Count"	There are 21 requirements that have been met in this phase of the development.
Yes Case	Number of test cases that are classified as Pass.	"Yes Case"	14 out of 50 test cases were assigned with a value of "Pass"
Design Object	Number of design objects.	"Design Artifact"	There are 30 design graphs created using the tool.
Lines of Code	Number of lines of code.	"Lines of Code"	There are over 20,000 lines of code in this directory.
File	Number of files.	"File"	There are 192 files in this directory.
Issue	Number of issues being tracked.	"Issue"	After 3 month of development, the we have over 40 issues in the issue tracking database.

C. Normal Proportion Artifact Graph (NPAG) Format

Figure 9 contains direct plots of the eleven artifacts collected during the experiment.

Viewing all the artifact data in a single display can provide a larger perspective in

understanding the software engineering operation process. However, Figure 9 is not an

appropriate display, due to unit collision and scaling problem of the vertical axis.
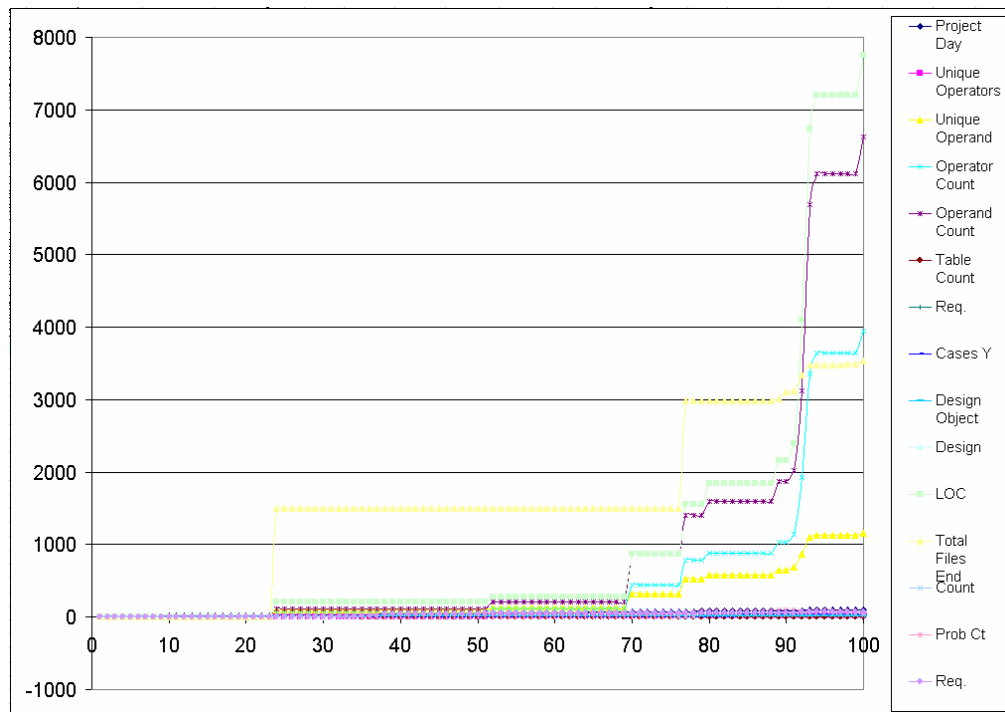
Fig. 9. Raw artifact values displayed in a single graph with unit collision and scaling problems

Unit collision occurs when attempting to display various units of artifacts on a single vertical axis, which cause confusion to the viewer of a graph; whereas Scaling Problem occurs when the simultaneous display of various artifact ranges cause smaller-range and smaller-sized artifacts to be overwhelmed by larger-range and sized attributes. We propose the Normal Proportion Artifact Graph (NPAG) format as a standard visualization format for the display of software engineering artifact data [23, 25]. The NPAG format focuses on the relationship between the artifact with respect to time, independent to the absolute magnitude or the unit of the artifact. This idea of using proportion to compare the quantity of different magnitude is analogous to using rate of return to represent return on investment. For example, the profits from three software

applications might be $5,000, $10,000, and $50,000 per year (from an absolute perspective), the profit from these software applications can be presented as 30 %, 60%, and 2%, respectively. The NPAG format eliminates both unit collision and the scaling problems with a single justifiable transformation; this is possible by dividing each value of an artifact's record data with its maximum value, including the artifact's Unit:

$$p_i = \frac{c_i}{c_{max}} \cdot 100$$

For example, if the maximum value of the artifact Requirement Count artifact is 57 ReqCount and at time 50 its value is 47 ReqCount, we carry out the transformation thus,

$$c_{50} = 47 \text{ ReqCount} \mapsto c'_{50} = \frac{47 \text{ ReqCount}}{57 \text{ ReqCount}} \cdot 100 \mapsto c'_{50} \frac{47}{57} \cdot 100 \mapsto c'_{50} = 82$$

and map the value of c_{50} from '47 ReqCount' to 82, which we can use justifiably as a proportional number 82. The result of the NPAG transformation is a sequence of artifact values in [0,100] that indicate the proportion of the artifact magnitude to the *maximum artifact value* along the project timeline. We point out that this operation is clearly different than dividing the artifact values by a unit-less number, such an operation would require justification in both why the particular number was used, and also why the division operation was carried out. On the other hand, we justify the operation (divide artifact values by the maximum artifact value) by stating the desire to view all artifacts in the same graph. The resultant proportion is a grounded experimental value and an creditable indicator to the percentage the magnitude of the artifact to its maximum value.

When viewed along the horizontal time line, the change of the artifact

magnitudes can indicate the generation behavior of the artifact. We note that this

operation is not an un-grounded transformation of the artifact values by an unjustified

parameter; it is a factual transformation of the artifact values to a scale that enables the

comparison of all artifact records at the same time. Figure 10 is the Normal Proportion

Artifact Graph (NPAG) for this experiment:
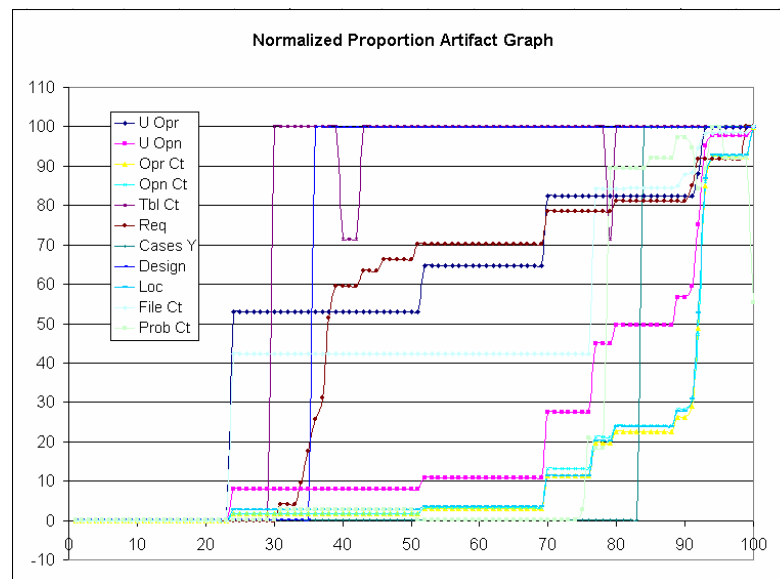
Fig. 10. Normalized artifact magnitudes sample 1

In the above graph, the horizontal axis ranges from 0 to 100; this denotes the start

and end time of the project. The vertical axis also spans from 0 to 100, noting the

proportion of the artifact's magnitude relative to its maximum size. Since the vertical

axis values are derived by dividing the original united value by that of the maximum

united value, the value is a proportion. An example of description of a sequence of

artifact values in a the NPAG format would be "At half-way through the project, the

Requirement (brown colored) artifact has reached close to 70 percent of its maximum

magnitude. The Unit of requirement is Requirement Count."

The set of experimental data are processed to a standard format for investigation.

The time span of the data is from the beginning of the software project to the completion

of the project. However, it is possible for the time span to be any reasonable segment of

time, which ends with a milestone. For the experimental project, the requirements were

met.

The collected artifact values are composed of a sequence of pairs, where the first

item is a time indicator and the second of the pair is the particular artifact's magnitude,

$((t_0, a_0), (t_1, a_1), \cdots, (t_n, a_n))$. For example, the *loc* (line of code) artifact contains a

sequence of pairs with units of *day* and *line of code*, and the requirement artifact is a

sequence of pairs with units *day* and *requirement count*. The sequence is automatically

created by software tools. During the project, software tools periodically measure the

size of a particular artifact and create a record of that fact and store it with its

corresponding time-related value. The time unit in this particular study is the number of

days that have passed since the start of the project; in future research, units can be: *built

number of the project*, *release of the project*, etc. While various progress indicator can

be used for the independent variable axis -- through the normalization process where

each measure is divided by the maximum measurement of the sequence --

$$t_i^{'} = \frac{t_i}{t_{last}}$$

the time indicator becomes a unit-less time indicator. The purpose of normalizing the

time indicator is to enable a possible comparison between projects of varying durations.

Through normalization, a project's time measurement becomes a universal time

indicator in [0,100]; that is, it indicates the percentage of time consumed before the

project stops.

Through the same process of dividing each of the artifacts in the recorded

sequence by the maximum artifact value, we transform the artifact from a particular

value (a numerical number and a unit) to a unit-less representation of *proportion*. Both

the time axis and the vertical axis become a proportion after the normalization process.

The purpose of the normalization process is to map all artifacts onto the same vertical

axis which indicates the progress of the artifact generation towards the final magnitude

at the end of the time segment. To provide a standard perspective, the vertical axis is

displayed at 2/3 of the length of the horizontal axis.
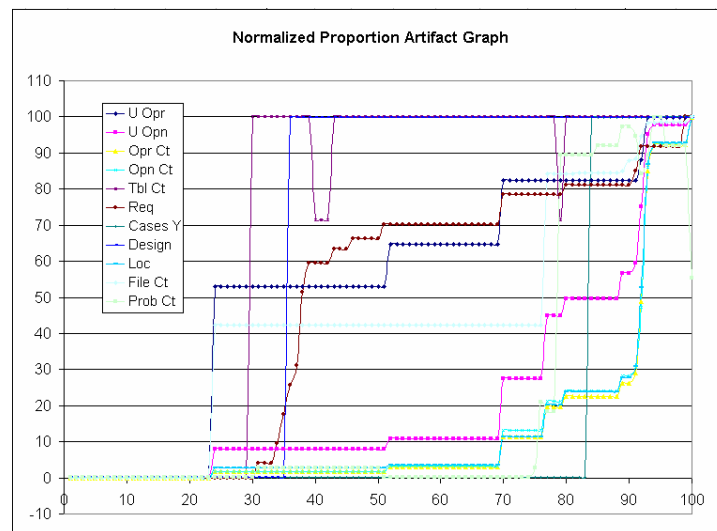
Fig. 11. Normalized artifact magnitudes sample 2

The Figure 11 NPAG shows Unique Operators, Unique Operands, Operator Count, Operand Count, Database Table Count, Requirement Count, Test Cases, Design Object Count, Line of Code, File Count, and Issue Count.

This section gives more specific description of the Normalized Proportion Artifact Graph (NPAG) format. In this format, the plot illustrates the generation of a particular, tracked from 0 percent of the final magnitude of 100 percent (the maximum attribute value during the project period).

As software tools are being used to collects various artifacts in a project, the pair (t_i, a_i) represents the time and artifact value. At the end of the collection period, an artifact record is a sequence of pairs $r = ((t_0, a_0), (t_1, a_1), \cdots, (t_n, a_n))$. Let t_{max} and a_{max} be the maximum value of the sequence. For instance, these values might be '23 release' or '9,450 lines of code'. We transform each of the values in the sequence thus

$t_i \mapsto t'_i = \dfrac{t_i}{t_{max}} \cdot 100$ and $a_i \mapsto a'_i = \dfrac{a_i}{a_{max}} \cdot 100$, and we define this sequence of transformed

artifact values as $r_{NPAG} = ((t'_0, a'_0), (t'_1, a'_1), \cdots, (t'_n, a'_n))$. A collection of artifact record in NPAG format is represent as $\{r_1, r_2, \cdots, r_n\}_{NPAG}$. Eleven artifact records were collected in the experiment that was carried out by the author, as shown in Figure 12 below:
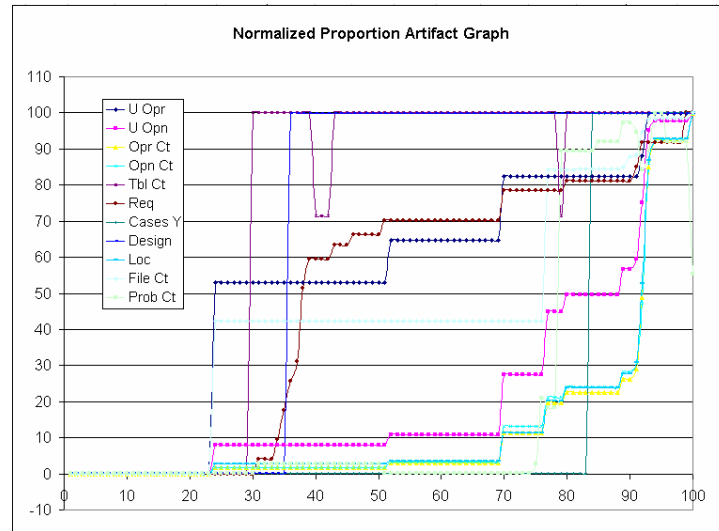
Fig. 12. NPAG data representation and graph

D. S-curve and Straight Line Description

The S-curve is an equation defined as $c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$, where t is the independent

variable, c is the dependent variable, g and r are parameters, and L is a constant. A

possible usage of the equation is to fit a sequence of t and c pairs to derive the g and r

parameters using the log-compression transformation, followed by the linear regression

fitting procedure to derive the *readiness* and *generation* parameters. Figure 13

summarizes the steps of the transformation.

$$t \mapsto t' = \frac{t}{t_n}$$

$$((t_0, c_0), (t_1, c_1), \cdots, (t_n, c_n)) \longrightarrow ((t'_0, m'_0), (t'_1, m'_1), \cdots, (t'_n, m'_n)) \longrightarrow \boxed{\begin{array}{l} \text{Liner} \\ \text{Regression} \end{array}}$$

$$c \mapsto m = \frac{c}{c_n} \mapsto m' = \ln(\frac{L}{m} - 1)$$

$$s \mapsto r = \exp(b)$$
$$b \mapsto g = -s$$

$$c = \frac{L}{1 + r \cdot \exp(-g \cdot t)} \longleftarrow \qquad \qquad m' = s \cdot t' + b$$
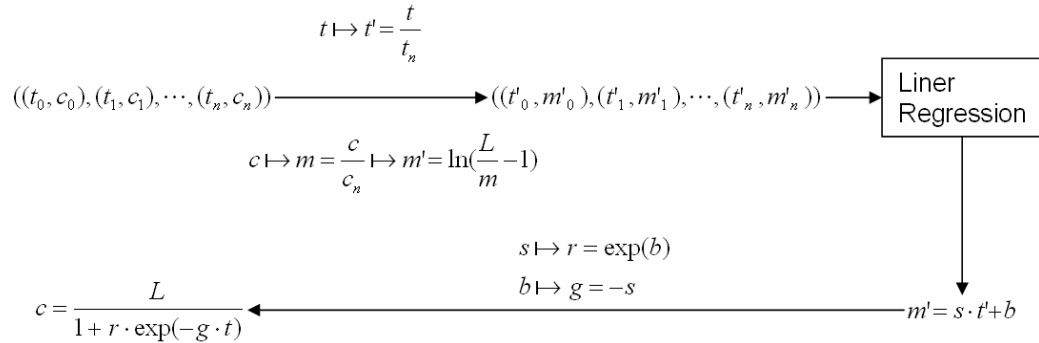
Fig. 13. Fitting an s-curve

S-curve can be used to describe adaptation of technology [17] or infection rate of malware such as a worm [54] with the passage of time, or other adaptation-related measurements. In our context, we use a s-curve to describe the evolution of a software artifact.

Two important parameters of an s-curve are those describing *readiness* and *generation* - characteristics of how the artifact was generated by the development team, Figure 14. A project team generates multiple artifacts; thus, each sequence of artifact counts result in a pair of s-curve parameters. These can be an indicator of a team's artifact generation capability. A project team generates multiple artifacts in a time period; thus, it is reasonable to characterize a team's artifact generation capability based on the team's artifact generation history, which is based on the ground parameters of the s-curves, $capability = f((r_0, g_0), (r_1, g_1), \cdots, (r_n, g_n))$. Generally, it is desirable to generate artifacts as early and as quickly as possible.
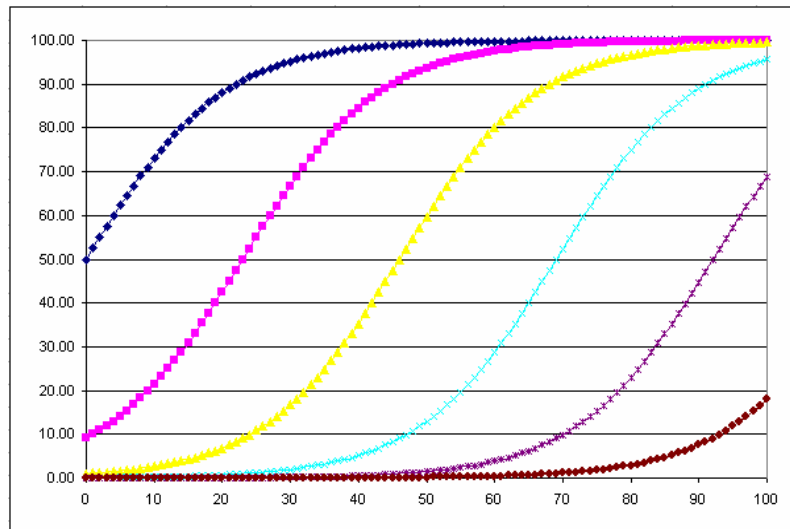
Fig. 14. S-curves with various readiness parameter values

The vertical axis indicates the percent of artifact being generated when compared to the final artifact size. *r* is the smallest for s-curves at the left of the graph and *r* value is large to the right. In other words, smaller *r* value indicates that the artifacts were being created earlier in the project. Similarly, we show the equivalent effect of the *readiness* parameter when graphing NPAG formatted graph use liner regression fit. Similarly, when we use a liner graph to describe the experimental data, the descriptive range of the lines can be characterized by the horizontal axis intercept parameter. Below are the linear fitted plots with *t-intercept* values of 0, 10, 30, 50, 70, and 90:

We note that the Churn (pink-colored curve in Figure 15) of s-curve behaves in an expected diminishing way for ready-to-release software [20].
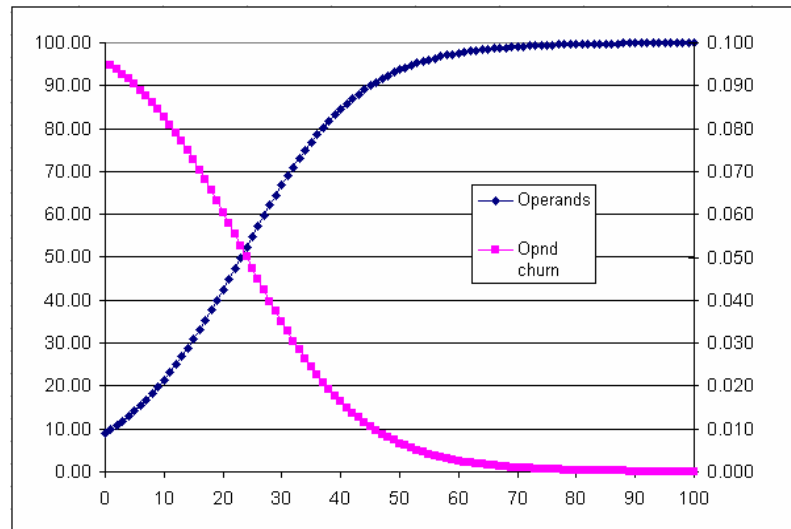
Fig. 15. S-curve and its diminishing churn

An s-curve is described by $c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$, where t is a time-related

variable and c indicates completion percentage, as measured from the ending artifact

size. The parameters r and g indicate the *readiness* and *generation* characteristic of the s-

curve.

1. The Readiness Parameter

The *readiness* parameter indicates when the software team begins to produce the artifact.

Figure 15 above shows 6 example curves, with *readiness* values 1, 10, 100, 1000, 10000,

and 100000.
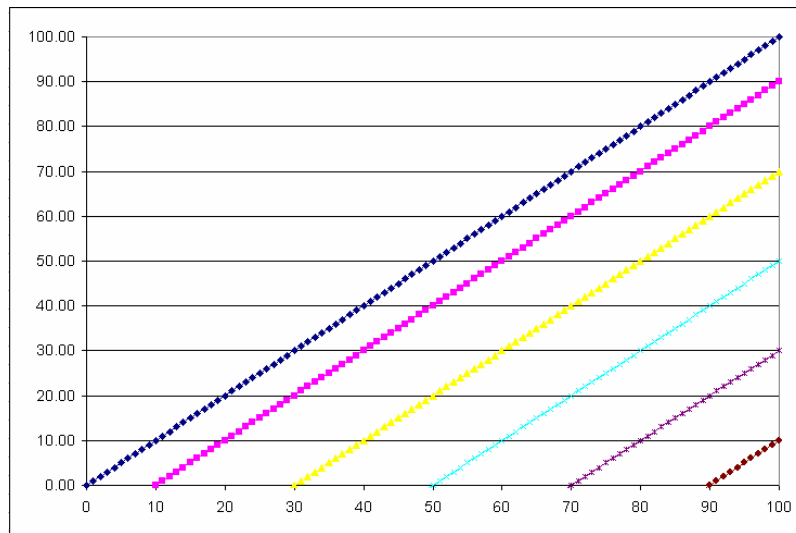
Fig. 16. Linear graph of various intercept parameter values

We note that the lines intercept the horizontal time axis at different locations,

Figure 16. This can be interpreted as the time when the artifact generation has begun.

For example, the pink line indicates that the creation of that artifact begins when at when

10 of the project has been completed, and reached 100 percent of the ending artifact

magnitude at the end of the project time (where the horizontal axis is at 100). On the

other hand, the purple line denotes the beginning of the artifact creation, when 70

percent of the project time has passed and the artifact value is at 10 percent of the final

magnitude of the artifact. This is an important point. We note that liner fitting of a line to

an NPAG formatted graph are not likely to end with the artifact magnitude at 100

percent of the artifact magnitude. This is a disadvantage of using liner fit on the NPAG.

2. The Generation Parameter

The *generation* parameter indicates how quickly the development team is able to generate artifacts. The Figure 17 shows lines with *generation* values of 0.5, 0.2, 0.15, 0.1, 0.05, and 0.02.
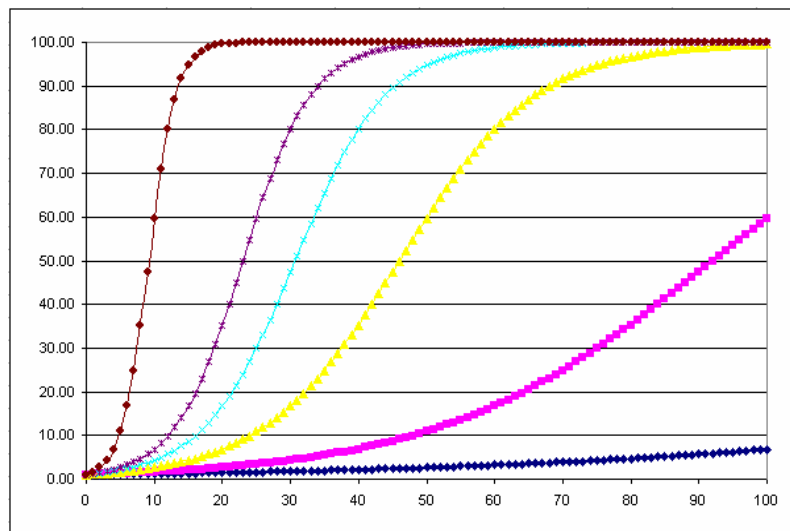
Fig. 17. S-curves with various generator parameter values

In this graph we see that all the artifact lines started at time 0 and most finished at the 100 mark at the end of the project time (except the pink and the blue line). The first line (brown) show that the artifact magnitude represents a quick rise to its final size at about 20 percent into the project, while the yellow line grows more slowly and finally reached the final artifact size close to the end of the project. Thus, the generation parameter of the s-curve describes the quickness in which the artifact magnitude grow to reach its project ending size.

In addition to using the s-curve, it is reasonable to use linear regression to analyze the artifact size. We investigate the representational range of that graph by using an equivalent of the s-curve generation parameter *slope*. Figure 18 below shows various plots with a constant t-intercept parameter value and with various generator values, 10, 5, 2, 1, and 0.2:
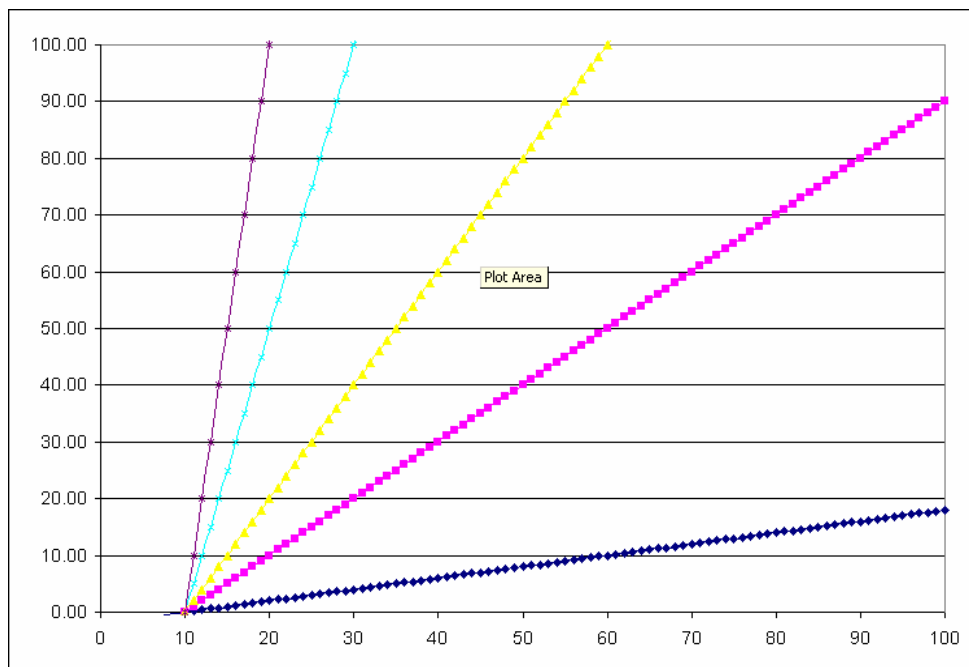
Fig. 18. Linear fit with various generator parameter values

We note that all the graph begins at 10 percent of the project time and grows at various rate. The blue plot indicates the final artifact magnitude, reached at a point close to 20 percent of the final project magnitude, while the pink plot reached 90 percent of the final artifact size. For artifact with faster generation, a larger *generator* value indicates faster artifact creation.

3. The S-curve Constant

We have described the *readiness* and the *generation* parameters of the s-curve,

$$c = \frac{L}{1 + r \cdot \exp(-g \cdot t)},$$ and the independent and the dependent variables. Lastly, we

describe the expected maximum value, *L*. Figure 19 shows plots of s-curve with *L* values

of 120, 100, 80, 40, 10, and 5.
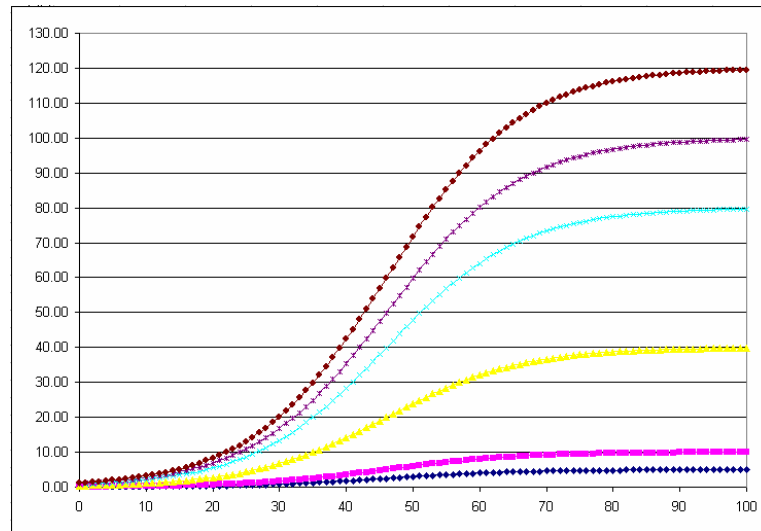
Fig. 19. S-curve with various expected maximum, L, values

We can see that the effect of the expected maximum value of the s-curve is the end result

of the cycle; its stabilization appoint is at the expected maximum value.

E. Fitting Data Using S-curves and Straight Lines

By analyzing collected artifacts through the s-curve perspective, each fitted artifact

record contains the *readiness* and *generation* parameters that describe two important

characteristics of how a software team generates software artifacts (thus towards successful project completion) during a software project, namely *readiness* and *how quick*. It would be interesting to investigate the relationship between the multiple *readiness* parameters amongst the artifacts, for instance. In addition, explaining the facts of a software project (the collected artifacts) through the s-curve perspective provides a more systematic and measurable foundation for software artifact tracking, measurement, and analysis.

Figure 20 displays the expected s-curves for a project following the waterfall development method.
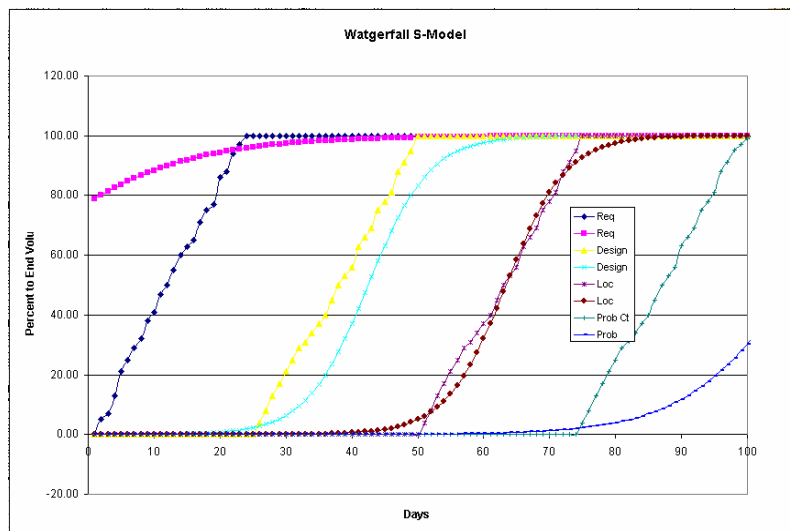
Fig. 20. S-curves fitted to idealized waterfall artifacts

We note the sequential placement of the S-curves along the timeline represents the requirement, design, development, and testing artifacts. The *readiness* and *generation* parameters for these idealized phases are listed in Table 4:

Table 4. S-curve Parameter Values for Waterfall Artifacts

	Requirement	Design	Development	Problem
readiness	0.29	9193	1127509	305845
generation	0.079	0.21	0.22	0.12

After putting the artifact data into the NPAG format, we investigate graphical methods

to represent the magnitude changes in the experiment. The first artifact we investigate is

the Lines of Code, shown in Figure 21. The final size is of the project is 7,758 lines,

generated by the 18-member team in 100 days. That amount does not include code from

components that were used to build the system. The NPAG formatted Line of Code

graph is shown with both linear and s-curve fit, while the pink-colored s-curve seems to

better track the Line Of Code magnitude (as compared to a liner fit). We note that the

factual artifact magnitude increases as a *step-function*, which we assert is partially driven

by project Demonstration milestones. The straight-line liner regression is based on the

minimization the squared of error of the magnitude points. However, the straight line fit

does not account for the final increase of Line of Code phenomena, which is quite

common. However the s-curve seems to fit the data better, with a slow rise at the

beginning and a faster generation following the Lines of Code count.
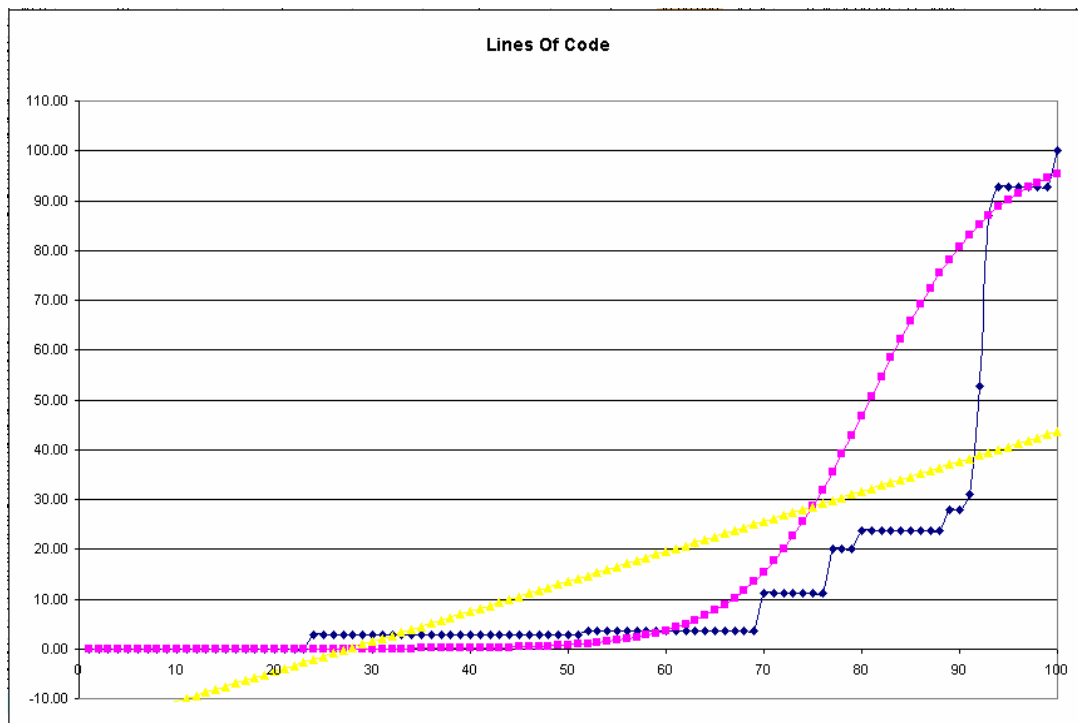
Fig. 21. NPAG format line of code with s-curve and linear fit

Figure 22 illustrates the characteristic step-function pattern common to File Count artifacts. It indicates the total number of the files needed to meet the application requirement. Once again, the pink s-curve seems to reflect the behavior the artifact's step-function behavior, while the straight line fit seems to indicate that there is continual generation of the number of files. From an experienced software engineering point of view, the s-curve definitely reflects the dynamics of the project activities that resulted in the step-wise file count record.
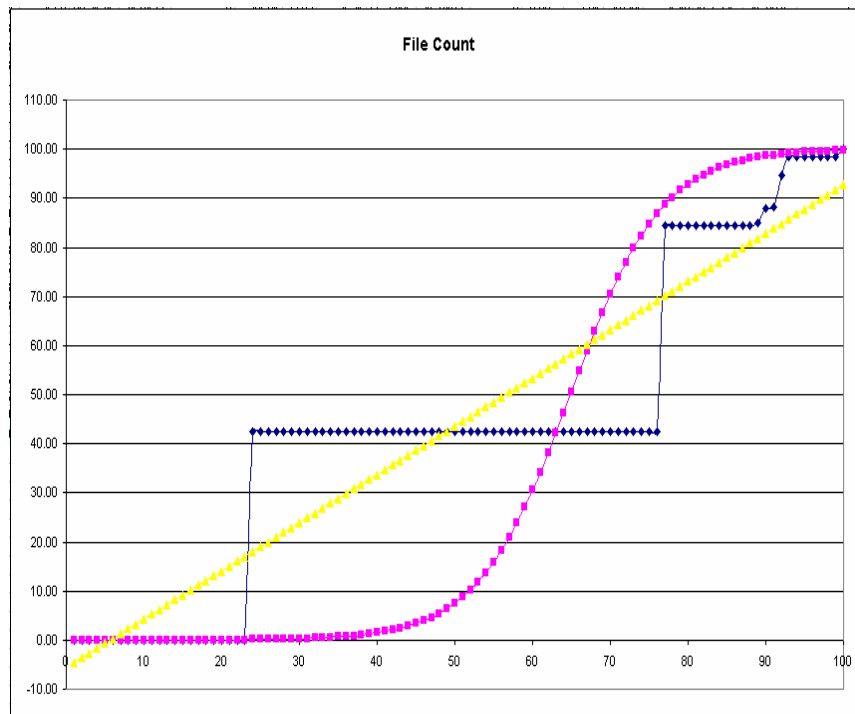
Fig. 22. File count with s-curve and linear fitting

Figure 24 is a record of the Issue Count, defined as a record of issues that arise during the development process; it forms a pattern typical of a team that generate most of their files as the end of a project nears, indicative of 'scrambling' to meet the final project demonstration milestone. Interestingly, the fitted, pink s-curve did not reach the 100 percent mark at the end of the project period. This seems to indicate that this artifact was not completed/fully mature at the end of the project; hence, more time was necessary for the s-curve to reach its project final value. Notable is the fact that the Issue Count is not a *monotonically increasing* curve, and the final issue count did drop to 50 percent of the maximum value. With this as a possible cause of the un-completed s-curve, it did not reach the 100 (maximum) artifact magnitude.
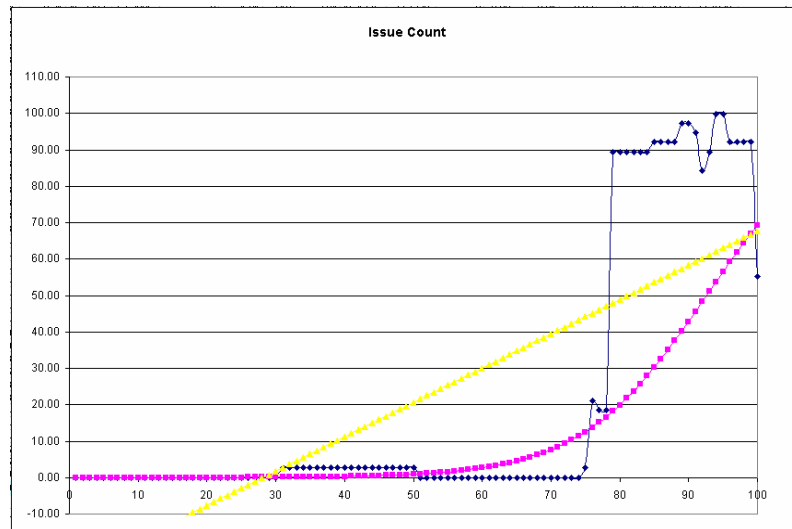
Fig. 23. Issue count with s-curve and linear fitting

Design Object, Figure 24, in this project indicates the number of UML-style (and any other) artifacts that are design-oriented [27]; an example is a graph of relational tables that is a common pre-curser to database table implementation. For a team using UML-styled graphs, Use Cases, Sequence Diagram, Activity Diagram, and Object Diagram, each is counted individually and added as a Design Object artifact. This two-step pattern was visible about one-third of the way into the project's timeline, seeming to indicate that the team took time to design, and that all design objects were created in a single session. This could be because the team was on a strict schedule that does not allow for the designing process to be completed throughout the project; however, this is a count of the number of Design objects, which is a more detailed investigation into the design objects that might give further indication of the detailed-dynamics of the design process. Of note again is the clear superiority of using s-curve to fit a step-function when compare to the straight line fit.
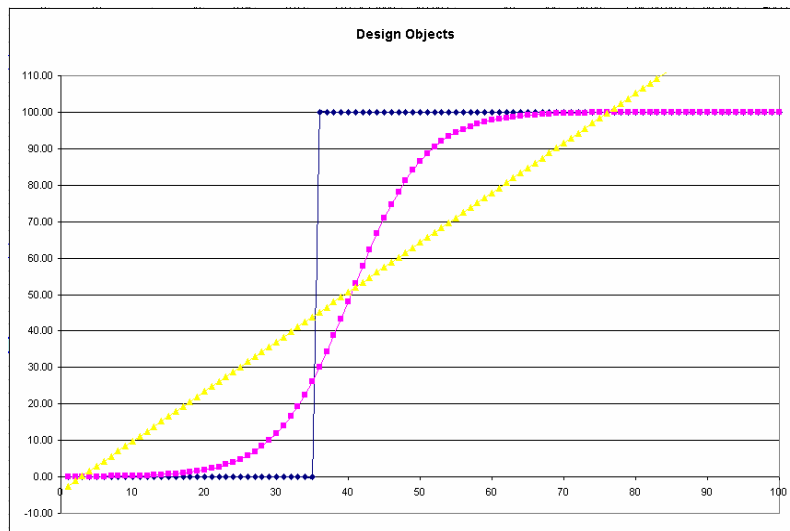
Fig. 24. Design object count with s-curve and linear fitting

Figure 25 shows the operand count -- programming variables that have been used in an application -- from the experiment. Variables are used to represent physical world objects or concepts. They can also be used to represent objects within the software system. For example, if an array is used to represent a sequence of transactions that have taken place in a single day. That array is used to represent external reality. A developer can also use an additional array to organize the details of the past week. In this case we have two operands: one represents an external item and another is used for internal organization. The graph shows generation in array magnitude at latter part of the project; the s-curve is clearly a better representation of this fact than a line derived form a regression.
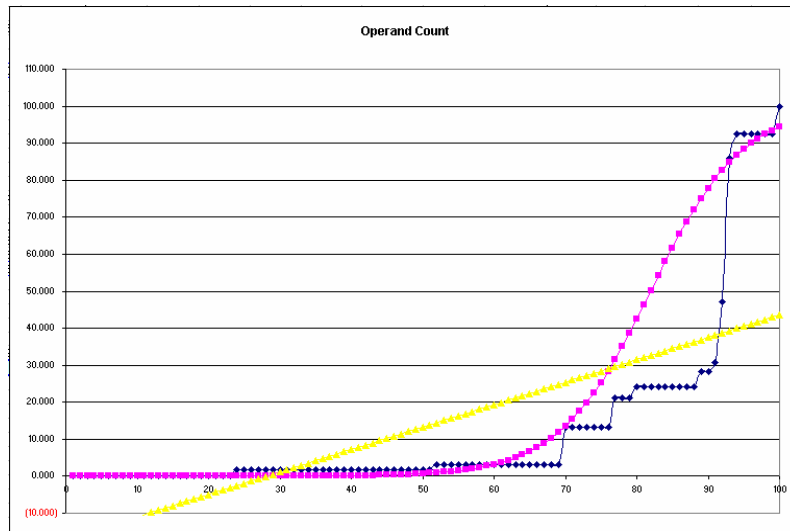
Fig. 25. Operand count with s-curve and linear fitting

Figure 26 shows the Operator Count artifact, where operators are used to manipulate data objects (operands). The number of operators indicate the extent of the data transformation in an application. However, this research focuses on the behavior of the operator magnitude by putting it into the standard NPAG format. We note a similar increase in magnitude towards the end of the project time. When considering the differences between the estimation and the actual completion as a measurement of fit, the s-curve fitting is a better fit than straight line when estimating Operator Count.
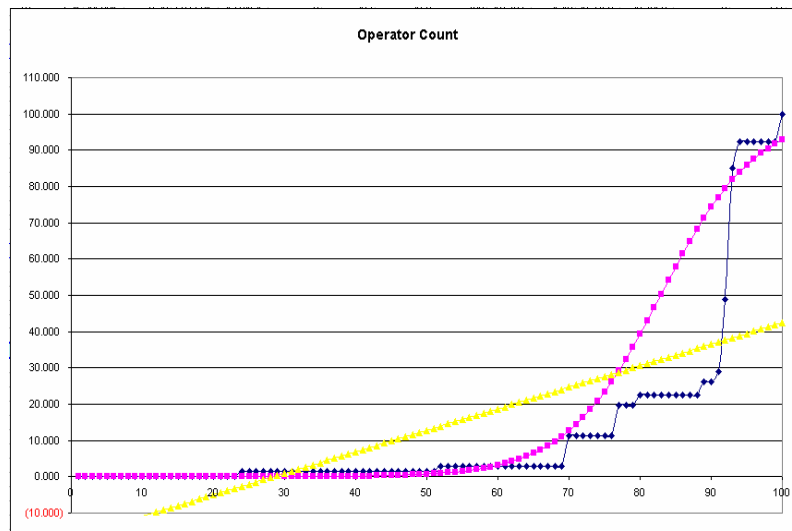
Fig. 26. File count with s-curve and linear fitting

'Requirement' is a higher conceptual object along the SETA (Software Engineering Transformation Axis), Figure 27. The graph below indicates a continual generation of the Requirement Count artifact along the time axis, although significant amount of requirements have been created (over 50 percent) at the project's half-way point. We note that the number of requirements continue to grow, in a step-wise function, as the project progresses. This is a reasonable phenomena, especially considering that higher concept Requirements need to be clarified during the development activity and clarification adds more structure and qualification (which necessarily implies the use of more words).

Fig. 27. Requirement count with s-curve and linear fitting

Figure 28 shows the database-related artifact count, referenced as Table Count. The experiment project created the database tables about one third of the way into the project; at this point, there seems to be a couple of incident results in the change of the number of database tables, but the overall size of them is stable throughout most of the project. In this artifact, the s-curve traces the step-wise increase of the database table; the increase and stabilization of the s-curve correspond to the artifact magnitude. Once again, the linear regression seems to indicate that the team was ready to create the database tables before the beginning of the project; this is a perfect example linear limitations – a line has difficulty in summarizing step-wise increments of artifact magnitude.
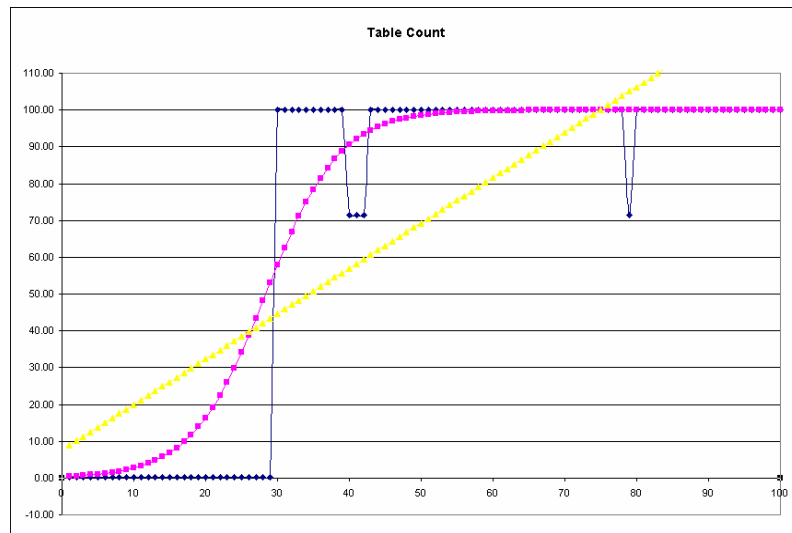
Fig. 28. Table count with s-curve and linear fitting

The Cases Passed count, Figure 29, indicates that the tests are not being done at the beginning of the project, but rather that they are completed at once, late in the project. This can be justified if the testing is system integration test. However, the end point of the fitted s-curve fit did not reach the 100 percent mark at the end of the project. Contrary to the Issue Count artifact, there isn't a decrease in the Test Case Passed count to explain the final low ending point; the conclusion drawn form this is "The end point of the s-curve of the Test Case Passed artifact did not reach the 100 percent mark indicates either the starting time of Test Case Passed is late, or alternatively, the project ended too early." Instead of actually starting date at day 84, Figure 30 shows the ending of the s-curve reached 100 percent at day 74.
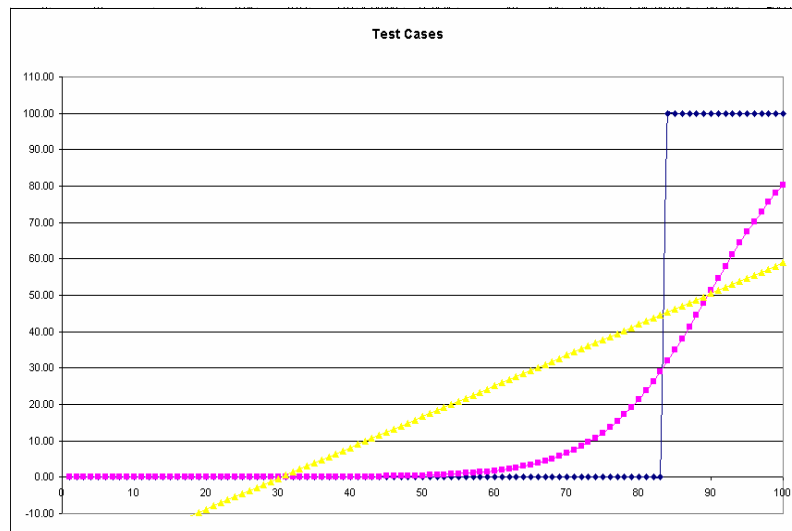
Fig. 29. Test cases passed count with s-curve and linear fitting
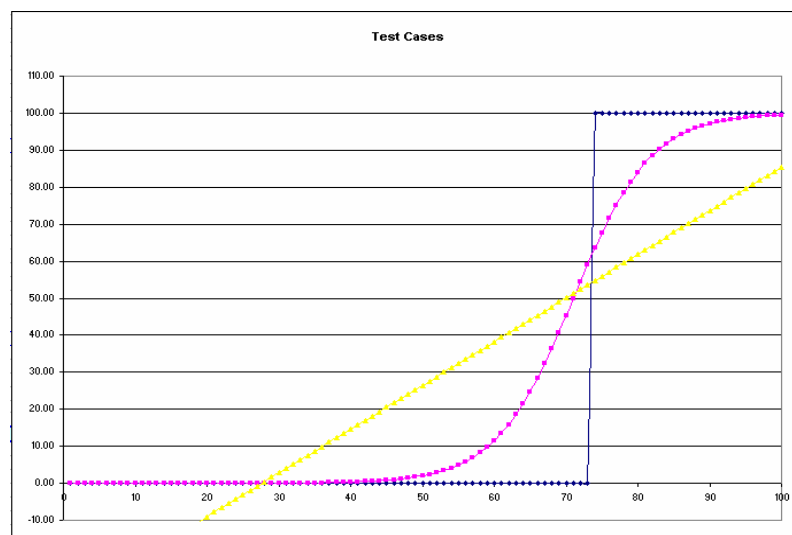
Fig. 30. Test cases passed count with hypothetical earlier starting date

The Unique Operands and Unique Operator graphs, Figures 31 and 32, show that s-curves are good representations of software artifact generation. Specifically, the figure indicates about 10 percent of the eventual operands were created around one third of

way into the project and new operands are continuously being created throughout the project. This can be an indicator that the scope of the project is continuously expanding to cover new requirements, or this can indicate that a project has high complexity and more operands are being created in order to represent the domain more clearly.
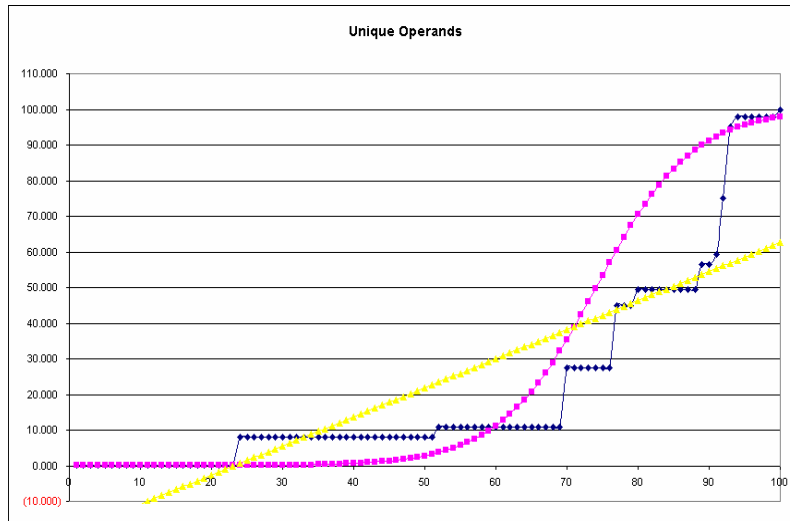
Fig. 31. Unique operands count with s-curve and linear fitting
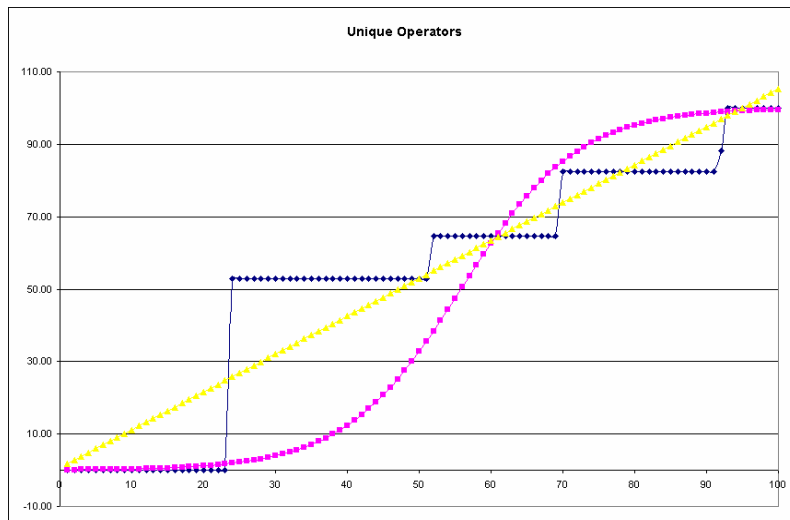
Fig. 32. Unique operators count with s-curve and linear fitting

F. Compare S-curve Fit to Straight Line Fit

Table 5 compares the estimated values to actual values for both s-curves and a liner fit.
The results (square root of the square the difference between the model and the actual
value) show that the s-curve fits better in 8 out of 11 cases. The deviation value is the
sum of the absolute daily differences between the fitted curve and the actual value.

Table 5. Compare of S-curve and Linear Performance

Measurement	S-curve deviation		Linear deviation	
Requirement Count	2310		1102	X
Design objects	823.2	X	2258	
Test Cases	994.2	X	2383	
Lines of Code	808.9	X	1328	
File Count	1862		1066	X
Issue Count	1262	X	2186	
Unique Operator Count	1634		915.8	X
Unique Operand Count	930.1	X	1180	
Operator Count	667.4	X	1340	
Operand Count	687.7	X	1321	
Database Table Count	777.5	X	2371	

G. Describing Experiment Data Parameters

The experiment NPAG (Normalized Proportion Artifact Graph) contains 10 artifact
magnitude records and has also been subjected to fitting methods. We propose that,
using this normalized format as a common foundation for the operational control and
also for the visual and analytical investigation of software project artifacts, the result is
like that displayed in Figure 12 on page 41. Two parameters that describe a straight line
fit is the slope and intercept $c = m \cdot t + b$, m and t in the equation respectively. The

intercept is where the equation cross the vertical axis. Since our focus in on the behavior

of artifacts along the time dimension of the graph, we focus on the horizontal

interception that is defined as $\dfrac{-b}{m}$ where $c = 0$. We interpret the slope as an indicator of

generation and change to the artifact and the t-intercept as an indicator of the beginning

of the artifact creation. We note that there is not a strong visual correspondence between

NPAG, Fig. 33, and Fig. 34, especially the indication of step-wise increment of the

software artifacts. However, the straight line from the linear regression does give a

factual perspective of the artifacts based on the *readiness* and the *generation* parameters.
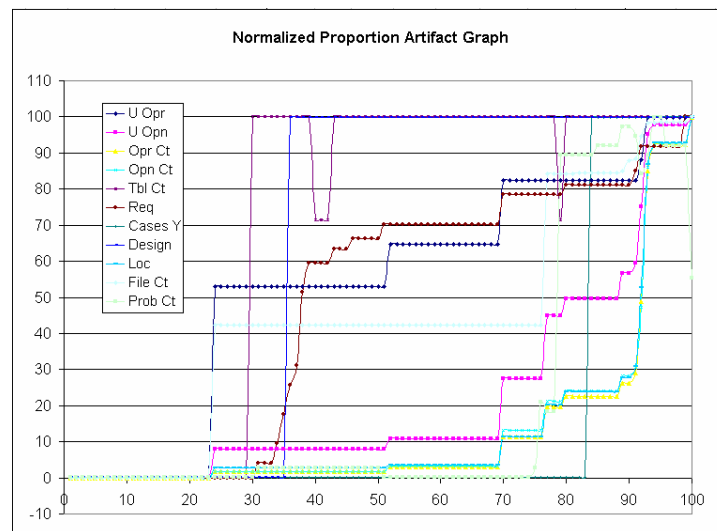
Fig. 33. Experimental result in NPAG format
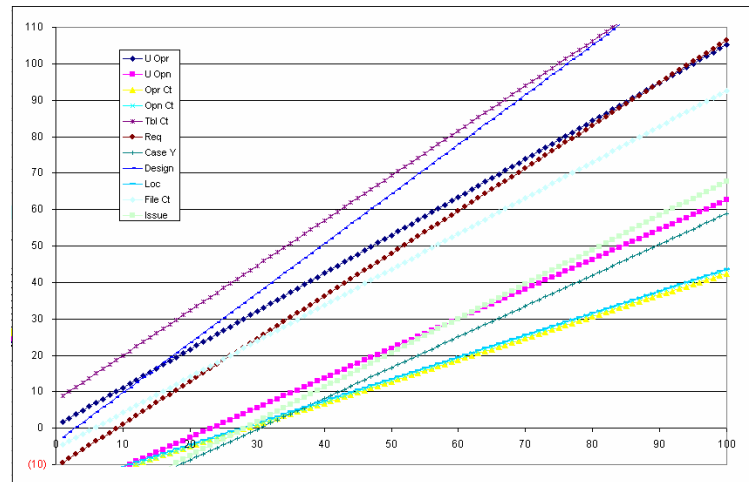
Fig. 34. Linear representation of artifact magnitude

In NPAG, the Database table artifact (purple color) and the Unique Operand (pink color) plots are significantly different from that displayed in Figure 17. The Database tables have been created at an early stage in the development period, while Unique Operand grows more slowly, only to explode at the end of the project timeline. This distinction is not immediately apparent when all artifacts are graphed using straight lines, as in Figure 34. However, upon closer inspection the straight (purple) Database Table Count is above the straight (pink) Unique Operand line; this confirms that Database Table Count artifact does start earlier than the Unique Operand line. This distinction is apparent by comparing the parameters values in Table 6.

Table 6. Linear Regression Parameters of Experiment Normalized Proportion
Attribute Graph (NPAG)

	Requirements	Case	Design	LOC	Files	Issues
Readiness	9.121	30.38	2.833	27.58	5.674	28.14
Generation	1.231	0.8459	1.363	0.6019	0.9823	0.9424

	Unique operators	Unique operands	Operators	Operands	Tables
Readiness	- 0.6571	23.23	28.47	28.20	- 6.155
generation	1.045	0.8154	0.5932	0.6065	1.234

The s-curves and straight lines are described by parameters of an equation that
quantitatively summarizes the data points being graphed. Each s-curve has *readiness* and
a *generation* parameters, and a straight line has *slope* and a *time-intercept* parameters.
Below are the experiment artifacts using liner fit and also using s-curve fit; it seems that
s-curves, as shown in Figure 35, give a more realistic graphical representation of when
and how the magnitude change along time than liner fitted lines.
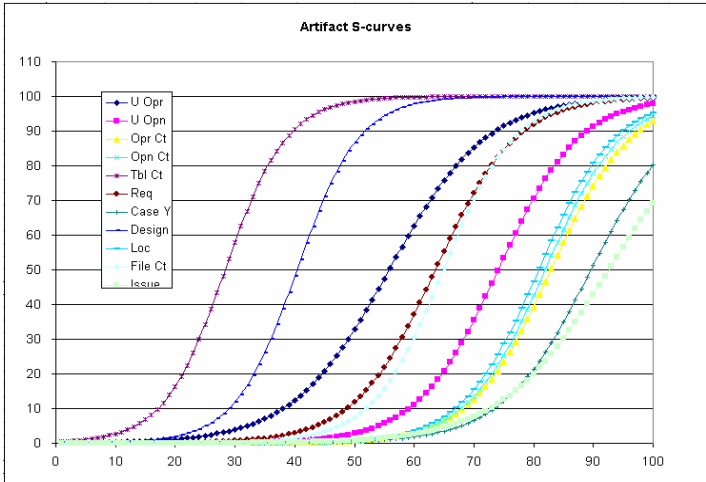
Fig. 35. S-curves of normalized experiment artifacts

The s-curves' corresponding *readiness* and *generation* values are listed in Table 7:

Table 7. S-curve Parameters of Experiment NPAG

	Requirements	Case	Design	LOC	Files	Issues
readiness	12160	239500	2443	321200	58280	27330
generation	0.1480	0.1350	0.1932	0.1568	0.1692	0.1103

	Unique operators	Unique operands	Operators	Operands	Tables
readiness	988.4	57280	239300	348600	211.9
generation	0.1235	0.1472	0.1494	0.1558	0.1928

We have fitted both s-curves, $c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$, and straight lines, $c = m \cdot t + b$, to

NPAG. The results are two graphs and four parameters. The two parameters that

describe the straight line fit are *readiness, r,* and *generation, g,* where smaller readiness

indicates earlier start of the artifact building activity and larger generation means faster

creation of artifacts. Similarly, the two parameters that describe the s-curves are also

readiness, $\dfrac{-b}{m}$, and *generation, m.* We present a sorted data table below and follow

with analysis of the sorted data.

Table 8. Artifacts Sorted According to Graph Parameters

descend		ascend		descend		ascend	
linear ready		**linear grow**		**S-curve ready**		**S-curve grow**	
Case Y	30.38	Oper	0.5932	Opern	348600	Issues	0.1103
Oper	28.47	LOC	0.6019	LOC	321200	U Oper	0.1235
Opern	28.2	Opern	0.6065	Oper	239300	Case Y	0.135
Issues	28.14	U Opern	0.8154	Case Y	179500	U Opern	0.1472
LOC	27.58	Case Y	0.8459	Soruce Files	58280	Requirement	0.148
U Opern	23.23	Issues	0.9424	U Opern	57280	Oper	0.1494
Requirement	9.121	Source Files	0.9823	Issues	27330	Opern	0.1558
Source Files	5.674	U Oper	1.045	Requirement	12160	LOC	0.1568
Design	2.833	Requirement	1.171	Design	2443	Source Files	0.1692
U Oper	-0.6571	Tables	1.234	U Oper	988.4	Tables	0.1928
Tables	-6.155	Design	1.363	Tables	211.9	Design	0.1932
better		better		better		better	

Table 8 lists the eleven sorted artifacts, according to the four graph parameters. The column labeled 'l ready' stands for liner fit readiness parameter. The column labeled 'l grow' stands for liner fit generation parameter. The labels 'S ready' and 'S grow' correspond to s-curve readiness parameter and s-curve generation parameter respectively. These parameters have been sorted so more desirable values are closer to the bottom of the table. The general idea of *better* is: 1) starting early in the project, and 2) creating artifacts quicker.

The first column lists artifacts sorted by *linear readiness* values. We note that the Table Count artifact has the best value (-6.155). The third column lists the sorted *linear (fit) generation* values with the Design Count as the best artifact with a parameter value of 1.363. The third fifth column lists sorted artifacts according to the *s-curve readiness* values, with Table Count as the best artifact with a value of 211.9. Lastly, in column seven are artifacts sorted according to the *s-curve generation* parameter; in this, the Design artifact count is at the top, with a value of 0.1932. The horizon line through the table is a 50 percent demarcation that separates the artifacts into a better performing group from the average performing group. For example, the better group artifacts would have started earlier in the project and grow at a faster pace.

We have shown in an earlier chapter that the s-curve visually fits better than linear regression, as in Figure 36 about the Lines of Code artifact.
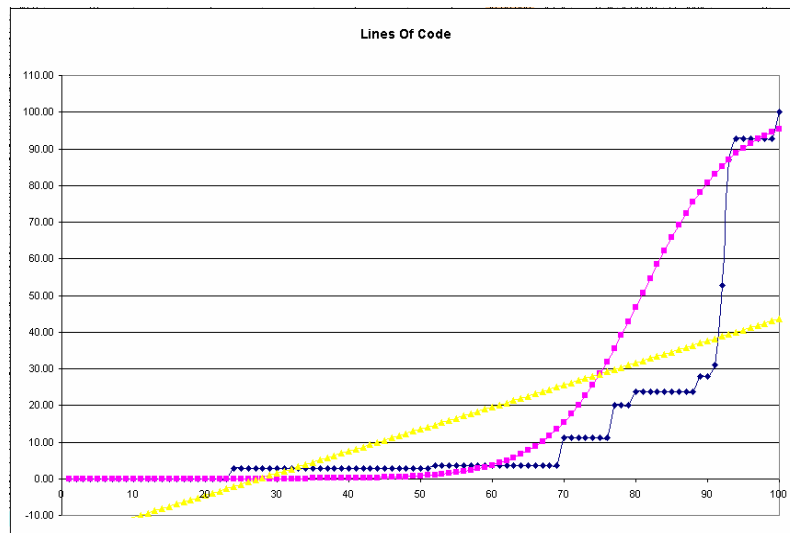
Fig. 36. Lines of code with linear and s-curve fit

The s-curve (pink-colored) shows the significant generation of the artifacts while the liner fitted straight line does not show that particular artifact characteristic. We now look at fitting curves to the artifact data from the parameter value point of view. Table 9 shows artifacts that have been selected using graph parameters as the selection criteria. We demarcate the field with a 50 percentile line for each of the four parameters in order to identify artifacts above the 50 percent ranking. We note that (in blue color) the readiness and the generation parameters of the liner regression are the same set of artifacts: Tables (Development), Unique Operator (Development), Design (Design), Source Files (Development), and Requirement (Requirement). The phases of the software engineering cycle are in parenthesis after the artifact. It is interesting that the same set is selected by the liner regression parameters. Since the project experiment follows the Extreme Programming practice, it is reasonable that many of these *readiness* artifacts are from the Development phase. The pink-colored artifacts are selected by the

s-curve parameters. We note that only two artifacts are in common, or in agreement, by the *readiness* and the *generation* parameters of the s-curves, Design, and Tables. This indicates that Design artifacts and Tables were ready at an earlier stage of the development cycle than other artifacts. These artifacts' selection by the s-curve generation parameter indicates that, once the artifacts started to be created by the team, their magnitude grew rather quickly.

Table 9. Favorable Artifacts Selected According to Common Graph Parameters

descend		ascend		descend		ascend	
linear ready		**linear grow**		**S-curve ready**		**S-curve grow**	
Case Y	30.38	Oper	0.5932	Opern	348600	Issues	0.1103
Oper	28.47	LOC	0.6019	LOC	321200	U Oper	0.1235
Opern	28.2	Opern	0.6065	Oper	239300	Case Y	0.135
Issues	28.14	U Opern	0.8154	Case Y	179500	U Opern	0.1472
LOC	27.58	Case Y	0.8459	Soruce Files	58280	Requirement	0.148
U Opern	23.23	Issues	0.9424	U Opern	57280	Oper	0.1494
Requirement	9.121	Source Files	0.9823	Issues	27330	Opern	0.1558
Source Files	5.674	U Oper	1.045	Requirement	12160	LOC	0.1568
Design	2.833	Requirement	1.171	Design	2443	Source Files	0.1692
U Oper	-0.6571	Tables	1.234	U Oper	988.4	Tables	0.1928
Tables	-6.155	Design	1.363	Tables	211.9	Design	0.1932
better		better		better		better	

Table 10 highlights artifacts selected by the same *type* of parameters from both the straight line and the s-curve graphs; that is, one set is selected by the *readiness* parameters of the straight line and the s-curve, and the other set has been selected by the *generation* parameters of both graphs. Artifacts selected in this table are strong candidates for that particular type of characteristic (in which artifacts are either created early, or they are generated quickly). For readiness, both liner and s-curve selection include Tables (Development), Design (Design), and Requirement (Requirement). This seems to be reasonable, since Extreme Programming were used during the experiment

and all the Requirement, Design, and Development phases are carried out in parallel. As for the *generation* characteristic -- Design (Design), Tables (Development), and Source Files (Development) -- artifacts were selected to indicate the quick magnitude increase. Specifically, the database table was developed quickly because electronic commerce databases have a standard pattern and, once one is familiar with it, they can be created quickly. The quick generation of the Source File (final count 3,541 files) could be due to the Component nature of the electronic application. The development team created many files (706 files, 20 percent of the total count) for the application from scratch; thus, the quick generation of the Source File count could be due to the downloading of the already created external component files.

Table 10. Favorable Artifacts Selected According to Common Type of Parameters

descend		ascend		descend		ascend	
linear ready		**linear grow**		**S-curve ready**		**S-curve grow**	
Case Y	30.38	Oper	0.5932	Opern	348600	Issues	0.1103
Oper	28.47	LOC	0.6019	LOC	321200	U Oper	0.1235
Opern	28.2	Opern	0.6065	Oper	239300	Case Y	0.135
Issues	28.14	U Opern	0.8154	Case Y	179500	U Opern	0.1472
LOC	27.58	Case Y	0.8459	Soruce Files	58280	Requirement	0.148
U Opern	23.23	Issues	0.9424	U Opern	57280	Oper	0.1494
Requirement	9.121	Source Files	0.9823	Issues	27330	Opern	0.1558
Source Files	5.674	U Oper	1.045	Requirement	12160	LOC	0.1568
Design	2.833	Requirement	1.171	Design	2443	Source Files	0.1692
U Oper	-0.6571	Tables	1.234	U Oper	988.4	Tables	0.1928
Tables	-6.155	Design	1.363	Tables	211.9	Design	0.1932
better		better		better		better	

Lastly, we investigate the artifacts that are selected based on all four parameters of the two graphs: Table (Development) and Design (Design). Their selection indicates that the two have been created at an early time of the project and grew quickly as shown in Table 11. The necessary number and types of database tables of the Table artifact is

fairly constant for electronic commerce based software product, thus the creation of database table artifacts is a pattern that can be repeated. As for Design, its selection can mean that design was done early in the software life cycle, created quickly, and without significant addition to the Design artifact. For a well known electronic commerce application using well know web service technology, this is to be expected.

Table 11. Favorable Artifacts Selected According to All Graph Parameters

descend		ascend		descend		ascend	
linear ready		**linear grow**		**S-curve ready**		**S-curve grow**	
Case Y	30.38	Oper	0.5932	Opern	348600	Issues	0.1103
Oper	28.47	LOC	0.6019	LOC	321200	U Oper	0.1235
Opern	28.2	Opern	0.6065	Oper	239300	Case Y	0.135
Issues	28.14	U Opern	0.8154	Case Y	179500	U Opern	0.1472
LOC	27.58	Case Y	0.8459	Soruce Files	58280	Requirement	0.148
U Opern	23.23	Issues	0.9424	U Opern	57280	Oper	0.1494
Requirement	9.121	Source Files	0.9823	Issues	27330	Opern	0.1558
Source Files	5.674	U Oper	1.045	Requirement	12160	LOC	0.1568
Design	2.833	Requirement	1.171	Design	2443	Source Files	0.1692
U Oper	-0.6571	Tables	1.234	U Oper	988.4	Tables	0.1928
Tables	-6.155	Design	1.363	Tables	211.9	Design	0.1932
better		better		better		better	

H. Foundation for Operational Software Process Measurement

This research provides a set of quantitative facts that are derived directly from artifact values; the latter can be used to construct metrics [16] for software project management using knowledge-based software tools [50]. These numbers can be predictably reproduced by fitting both a s-curve and a linear graphs to a standardized formatted artifact value. In addition, these numbers are without human-mediated adjustment or organizational-specific calibration, which is an important aspect of characterizing the

numbers as being factual. However, team-specific calibration can be obtained for in-process operation management. Below, we describe the quantitative values based on the artifact being collected $((t_0, a_0), (t_1, a_1), \cdots, (t_n, a_n))$. This sequence of values will be put into the NPAG format so both the range of the time variable and the magnitude variable are in proportion [0,100]. This format can serve as a standard for the analysis of software artifacts of similar software projects. The experimental project was 100 days, thus normalizing the time values would have little consequence to data. However, normalizing the time values would enable comparison of projects with different time duration.

For graphic representation, the normalized sequence is plotted on a 100 by 100 grid. This is a clean stage for the simultaneous presentation of all artifact values, as shown by the Lines of Code Figure 37 here:
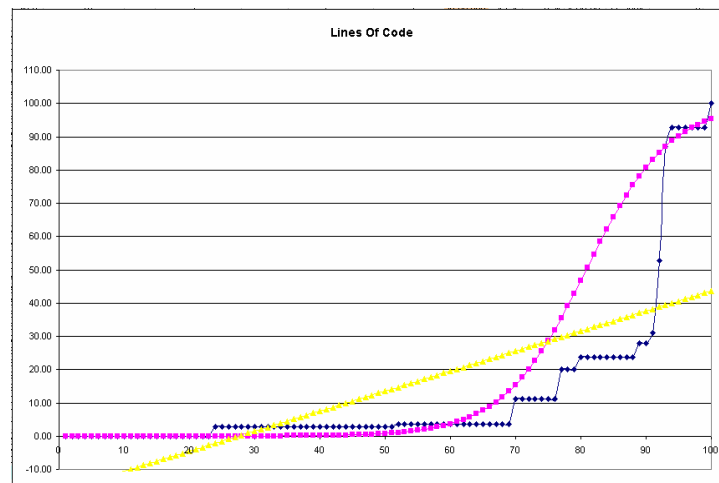

Fig. 37. Artifact data plotted in the NPAG format

The four parameters used to describe the s-curve are: $c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$, and the

liner line, $c = m \cdot t + b$, can be used as quantitative component for building artifact-

related metrics. For example, the *readiness* parameters of the two graphs can be

combined as $ready = w_1 \cdot r + w_2 \cdot \dfrac{-b}{m}$, where $1 = \sum w_i$ is the definition of readiness (as a

combination of both the liner and the s-curve readiness parameters). Similarly, the

generation parameters can also be combined as $growth = w_3 \cdot g + w_4 \cdot m$, when $1 = \sum w_i$.

These calculations are suggestions for future work and were not reviewed in this paper.

The researchers would like to emphasize that all numbers used are directly derived from

the factual recording from software tools; thus, they are grounded and can be used as

historic factual evidence of the software engineering process that have been carried out

during the project.

I. In Process Software Assessment

We give an operational procedure for the visualization and utilization of graph

parameters [44], which we define as the *readiness* and *generation* parameters of the s-

curve. We call these parameters the NPAG parameters, which include two parameters

from the s-curve and two parameters from liner regression. The in-process software

assessment [10] is composed of two steps: Bootstrapping and Assessment. Bootstrapping

gathers the team specific parameters for each artifact and Assessment gives an indication

of whether the team is progressing on target or behind target. First, we describe the

bootstrapping process. Due to the constraint of the s-curve equation,

$c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$. Initial artifact estimates are needed. This can be acquired through

estimation such as Line of Code artifact value estimation for object-oriented projects by

Ronchetti [39, 43]. However, due to the wide range of artifacts involved, the author

recommends running a Calibration Project to acquire the initial artifact values. These

artifact values are used as the Expected Maximum in the s-curve equation during the

second Calibration Project. Completion of the second project will generate the first set of

the s-curve parameters: r and g. This completes the bootstrapping step.

After the bootstrapping procedure, we have a set of s-curve parameter values for

each artifact. During a real project, we use the rearranged s-curve equation

$L = c \cdot (1 + r \cdot \exp(-g \cdot t))$ for in-process assessment of the generation behavior of the

particular artifact. Specifically, if $c \cdot (1 + r \cdot \exp(-g \cdot t)) \geq 100$, then the artifact is being

created as expected. On the other hand, if the value is less than 100, then the particular

artifact at that time is being created at a slower pace than expected. The managers can

use that data as a control value [13]. For example, perhaps use a 10 percent envelop

around the expected 100 value. Figure 38 is a more detailed operational procedure:

```
If the team's historic parameter record does not exist
        Carry out first Calibration Project to measure Maximum Artifact
        values.
        Carry out second Calibration Project to measure s-curve
        parameters
end
While not end of project period
        At each time period
                For each artifact value being collected
                        If c · (1 + r · exp(−g · t)) not within control range
                                Find out reason for the deviation
                Carry out process adjustment and correction
Store NPAG and NPAG parameter record
```
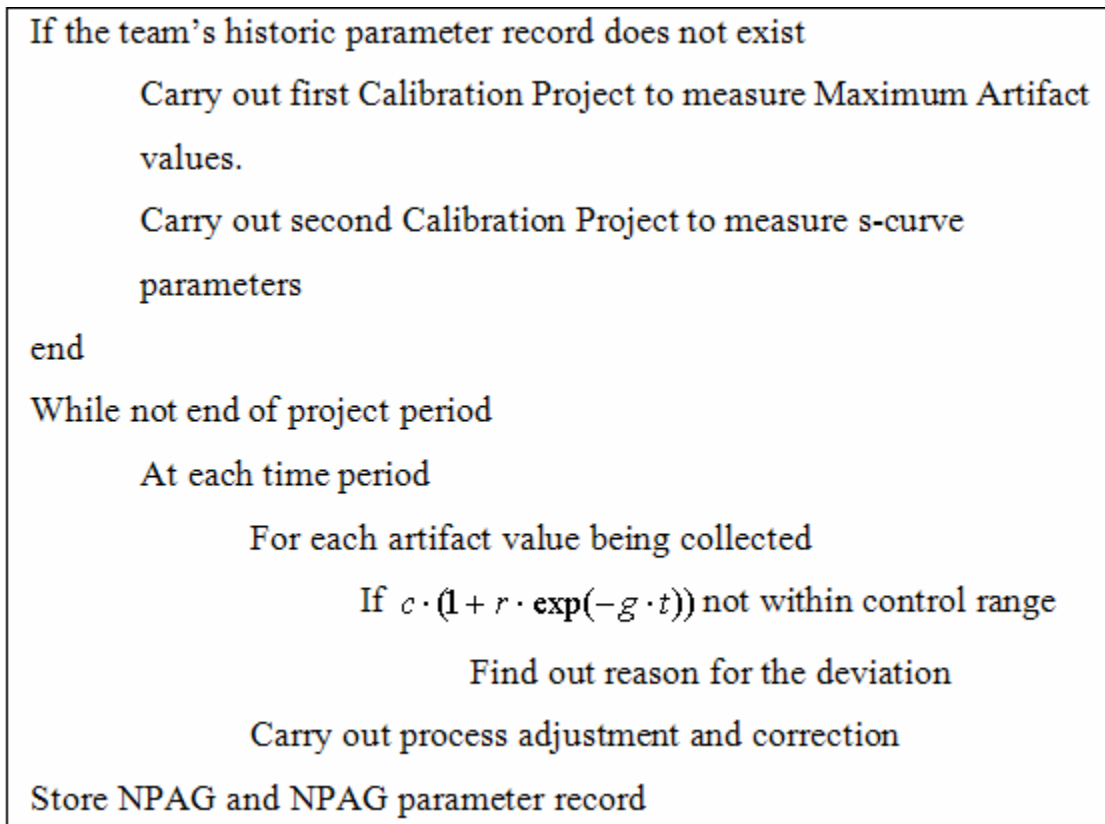
Fig. 38. Algorithm using NPAG measurement as process control variable

J. Conclusion

A display format called the Normal Proportion Artifact Graph (NPAG) was explained and used to display experimental software project artifact data. This eliminates the Unit Collision and Scaling Problem commonly encountered when displaying multi-unit artifacts in a common space. Two-graph methods were investigated to their display capability in representing the artifact data. The s-curve was found to be more fitting than liner fit. This can be due to the step-function nature of software project attributes. The

characteristic of the s-curve was described using the *readiness* and *generation*

parameters, and the effect of those parameters on the graph shape was described. The

readiness and the *generation* values were presented as possible quantitative numbers that

describe the generation of artifact magnitude, which can be used as grounded

quantitative values for the construction of project metrics. A novel process control

procedure was described based on calibrated graph parameters. This procedure gives a

easy-to-understand response for the in-process control of multiple artifact generation

processes.

CHAPTER V

CONCLUSION AND FUTURE WORK

Eleven artifacts that span the software engineering life cycle were collected from a software engineering project experiment and analyzed. The number of artifact instances created by software engineering activities were plotted in a normalized graph format and fitted using both s-curves and also straight-lines. The artifact values were formatted into the Normal Proportion Artifact Graph (NPAG) format, which the author believes should become a standard for displaying multiple type of artifact in a single display without either Unit Collision nor Scaling Problem. This recommendation is based on the observation of the step-wise generation pattern of artifact instances; that after identifying s-curves as a reasonably superior graphical abstractions of the step-wise artifact values.

This research defines and described *readiness* and *generation* parameters for the experimental data collected based on the s-curve model. The parameter values are grounded in quantitative summarization of software engineering activities, carried out during the experiment and the operational level of software engineering activity; that is, focusing on the software engineering processes that generates artifacts.

Based on the proposed NPAG format and the NPAG parameters that characterize software artifact generation, these ground parameter values can be valuable in modeling a team's artifact generation capability. For example, the readiness parameter may indicate the development team's process maturity level, as in the ability to generate artifact as planned. The growth parameter might indicate the experience or a team based

on the assumption that experienced team means quicker generation of artifacts. There are still interesting questions that should be investigated:

- What is the meaning when a fitted s-curve does not reach the final artifact value?

- What is the meaning when a fitted s-curve starts above the 0 percent mark at time 0?

- What is the effectiveness in using s-curve parameters as process control variable for operation-level software engineering processes?

These graph fitting techniques help to predict the timing and amount of resources used throughout an Extreme Programming project. Since significant software resources are devoted to the maintenance phase of the software life cycle [30, 31, 32], it is informative to investigate the effectiveness in the application of s-curve control variable to manage the maintenance phase of the software life cycle.

The experiment used to collect data was conducted over a 100-day period. Future research should address projects of different lengths. As Putnam has observed on software sizing, combination of sizing patterns are, in themselves, a larger version of the pattern [38] (in this case, Raleigh Curves). It would also be interesting to investigate the applicability of the s-curve fitting to a combination of software projects.

REFERENCES

[1]   V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. on Soft. Eng.,* vol. 12, pp. 733 – 743, 1986.

[2]   P. Béquin and Y. Clot, "Situated action in the development activity," *Activities,* vol. 1, pp. 50 - 63, 2004.

[3]   K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," http://agilemanifesto.org, 2001.

[4]   D. J. Berndt, J. L. Jones, and D. Finch, "Milestone markets: software cost estimation through market trading," *Proc. 39th Ann. Hawaii Int. Conf. Sys. Sci.,* vol. 9, pp. 230 - 237, 2006.

[5]   B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Ann. of Soft. Eng..* vol. 1, pp. 57 – 94, 2005.

[6]   B. Boehm and R. Turner, "Management challenges to implementing agile processes in traditional development organizations," *IEEE Soft..* vol. 22, p. 10, 2005.

[7]   B. W. Boehm, B. Steece, and R. Madachy, *Software Cost Estimation with Cocomo II*. Englewood Cliffs, New Jersey, Prentice Hall PTR, 2000.

[8]   B.W. Boehm and K. J. Sullivan, "Software economics: a roadmap," *Proc. Conf. Future of Soft. Eng.,* pp. 319 - 343, 2000.

[9]   E. Chang and T. S. Dillon, "A usability-evaluation metric based on a soft-computing approach," *IEEE Trans. on Sys., Man, and Cyber.,* vol. 36, pp. 356 - 372, 2006.

[10]   R. L. Chen, "Computerized life cycle advising, monitoring, and predicting (CLAMP),", Ph.D., Texas A&M University, College Station, 1985.

[11]   B. H. C. Cheng and J. M. Atlee, "Research directions in requirements engineering," *2007 Future of Soft. Eng. FOSE '07*, pp. 285 – 303, 2007.

[12]   Clayton M. Christensen, "Exploring the limits of the technology s-curve. Part1: Component technologies," *Prod. and Op. Manag..* vol. 1, pp. 334 – 357, 1992.

[13]   P. Donzelli, *"*Decision support system for software project management," vol. 23, pp. 67 – 75, July 2006.

[14]   M. R. Endsley and D. J. Garland, *Situation Awareness Analysis and Measurement*, Mahwah, New Jersey: Lawrence Erlbaum Associates, 2000.

[15]   R. Fantina, *Practical Software Process Improvement*. Boston: Artech House, 2005.

[16]   N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. on Soft. Eng.*, vol. 20, pp. 199 – 206, 1994.

[17]   R. Foster, *Innovation: The Attacker's Advantage*, New York: Summit Books, 1986.

[18]   Martin Fowler, *UML distilled, third edition: a brief guide to the standard object modeling language*, New York: Addison-Wesley, p. 68, 2004.

[19]   P. F. Gehring and U. W. Pooch, "Software development management", *Data Manag.*, vol. 1, pp. 14 – 18, 1977.

[20]   G. A. Hall and J. C. Munson, "Software evolution: Code delta and code churn", *Jour. of Sys. and Soft.*, vol. 54, no. 2, pp. 111 – 118, 2000.

[21]   IEEE, *IEEE software engineering collection on cd-rom, IEEE Software Engineering Standard*, IEEE, 2006.

[22]   IEEE, *IEEE standard glossary of software engineering Terminology, IEEE Standard*, IEEE, 1990.

[23]   ISO, "ISO/IEC 25000:2005 software product quality requirements." vol. 2006, ISO, Ed.: ISO, 2005.

[24]   A. Issa, M. Odeh, and D. Coward, "Software cost estimation using use-case models: A critical evaluation," *Info. and Comm. Techn., 2006. ICTTA '06 2nd*, pp. 2766 – 2771, 2006.

[25]   E. Johansson and M. Host, "Software architectures: Tracking degradation in software product lines through measurement of design rule violations," *Conf. Soft. Eng. Know. Eng. SEKE '02, 14th International*, Ischia, Italy, pp. 249 – 254, 2002.

[26]   M. Jorgensen and K. Molokken-Ostvold, "Reasons for software effort estimation error: Impact of respondent role, information collection approach, and data analysis method," *IEEE Trans. on Soft. Eng.,* vol. 30, pp. 993 - 1007, 2004.

[27]   M. Jorgensen and M. Sheppart, "A systematic review of software development cost estimation studies," *IEEE Trans. on Soft. Eng.,* vol. 33, pp. 33 - 53, 2007.

[28]   B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation*," IEEE Trans. on Soft. Eng.*, vol. 21, pp. 929 – 944, 1995.

[29]   C. P. Lecht, *The management of computer programming projects*, New York: American Management Association, Inc. 1967.

[30]   M. M. Lehman, "Approach to a theory of software evolution," *Eighth Intern. Workshop on Prin. of Soft. Evol.*, pp. 135, September 2005.

[31]   M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, pp. 1060 – 1076, 1980.

[32]   M. M. Lehman and J. F. Ramil, "Towards a theory of software evolution – and its practical impact," *Proc. of 2000 Inter. Symp. Prin. Soft. Evol.*, vol. 1, pp. 2 – 11, November 2000.

[33]   N. K. Ma and H. Fan, "Split team development project (A)," Case Study, Texas A&M University, Department of Computer Science, College Station, Texas, 2004.

[34]   Z. Mihajlovic and D. Velasevic, "Tracking software projects with the integrated version control in SMIT," *ACM SIGSOFT Soft. Eng. Notes*. vol. 26, pp. 38 – 43, 2001.

[35]   R. Miles and K. Hamilton, *Learning UML 2.0*, Sebastopol, California: O'Reilly, April 2006.

[36]   Object Management Group, *UML infrastructure specification, v2.1.1,*
         formal/2007-02-04, Needham, Massachusetts, Object Management Group,
         February 2007.

[37]   M. C. Paulk, "Extreme programming from a CMM perspective," *IEEE Soft.*, vol.
         18, pp. 19 – 26, 2001.

[38]   L. H. Putnam, "A general empirical solution to the macro software sizing and
         estimating problem," *IEEE Trans. on Soft. Eng.*, vol. SE-4, pp. 345 – 361, 1978.

[39]   M. Ronchetti, G. Succi, W. Pedrycz, and B. Russo, "Early estimation of software
         size in object-oriented environments a case study in a CMM level 3 software
         firm," *Info. Sci.*, vol. 236, pp. 475 – 489, 2006.

[40]   B. E. Scott, "Survey of computer program documentation practices at seven
         federal government agencies," Comp. Tech. Incorp., Arlington, Virginia, Report,
         March 1967.

[41]   R. L. Shaw, *Fighter Combat, Tactics and Maneuvering*. Annapolis, Maryland:
         Naval Institute Press, 1985.

[42]   D. B. Simmons, "Measuring and tracking distributed software development
         projects," *Proc. Ninth IEEE Works. Future Trends of Dist. Comp. Sys.*, pp. 63 -
         69, May 2003.

[43]   D. B. Simmons, "Art of writing large programs, The," *IEEE Comp.,* vol. 5, p. 7,
         1972.

[44] D. B. Simmons, N. C. Ellis, H. Fujihara, and W. Kuo, *Software Measurement, A Visualization Toolkit for Project Control and Process Improvement*. Englewood Cliffs, New Jersey: Prentice Hall PTR, 1998.

[45] D. B. Simmons and N. K. Ma, "Software engineering expert system for global development," *IEEE Inter. Conf. on Tools with Artif. Intel.* Washington, D. C.: IEEE, 2006.

[46] D. B. Simmons and C.-S. Wu, "Plan tracking knowledge base," *Twenty-Fourth Ann. Int. Comp. Soft. Appl. Conf.* Taipei, Taiwan, 2000.

[47] E. Stensrud and I. Myrtveit, "Identifying high performance ERP projects," *IEEE Trans. on Soft. Eng.,* vol. 29, pp. 398 - 416, 2003.

[48] K. Thackrey and J. Wright, "Experience tracking software development progress on a large Ada project (a window into the development process)," *Conf. TRI-Ada '91: Today's Accomplishments; Tomorrows Expectations*, San Jose, California, 1991, pp. 418 - 424.

[49] L. Voinea and A. Telea, "Multiscale and multivariate visualizations of software evolution," *Proceedings of the 2006 ACM symp. Soft. Visual. SoftVis '06,* pp. 115 - 124, 2006.

[50] N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson, "Preliminary results using scale-down to explore worm dynamice," *Proceedings of the 2004 ACM Works. WORM '04,* pp. 65 - 72, 2004.

[51]   L. Voinea and A. Telea, "Visualization: an open framework for CVS repository querying, analysis and visualization," *Proc. 2006 Int. Work. on Mining Soft. Repo. MSR '06,* pp. 33 - 39, 2006.

[52]   C.-S. Wu and D. B. Simmons, "Software project planning associate (SPPA): a knowledge-based approach for dynamic software project planning and tracking," *Comp. Soft. App. Conf., COMPSAC 2000*, The 24th Annual International, Taipei, Taiwan, p. 6, October 25 - 27, 2000.

[53]   S. Yun and D. B. Simmons, "Continuous productivity assessment and effort prediction based on Bayesian analysis," *Twenty-Eighth Ann. Int. Comp. Soft. App. Conf.* Hong Kong, 2004.

[54]   J. Zhang, D. Zage, and W. Zage, "Improving project planning/tracking for student software engineering projects through SOPPTS," *Proceedings 16th Conference on Soft. Eng. Edu. Train., 2003 (CSEE&T 2003)*, Madrid, Spain, 2003.

[55]   S. Zhang and Y. Wang, "An extension of SEMEST: The online software engineering measurement tool," *CCECE 2004 - CCGEI 2004* Niagara Falls, New York, 2004.

[56]   J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. on Soft. Eng.,* vol. 32, p. 14, 2006.

APPENDIX A

GLOSSARY OF TERMINOLOGY

A. Introduction

In this section, terms are defined. Each is listed entirely in capital letters, then followed by a corresponding definition. All terms are used in the context of software engineering and of software project; contextual support of many terms can be invoked by either precede or follow the term with the words 'software project'. The reading of the definition of a term proceed as "The definition of a <the specific term> is," followed by the definition of the term. Additional definitions can be found in:

1. IEEE (Institute of Electrical and Electronics Engineers) *Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990*.

2. Unified Modeling Language (UML) v2.1.1.

3. PAMPA Knowledge Base

All terms used in this dissertation intent to be consistent with these sources. However, the main purpose of this Glossary is to function as a supporting resource for this dissertation; thus, definitions herein can be different than a term's generally definition. Terms that are unchanged from the Standard Glossary are indicated with a star (*) symbol.

Due to the counting nature of this research, it is necessary to take additional time to distinguish between an abstraction and an instance. We define the description of items

as CLASS and the actual item as INSTANCE. For example, the class

SOFTWARE_ENGINEER describes software engineers in general, while an instance of

that class named 'Johnny Rocket' is a specific software engineer. Each class has an

attribute that indicates the number of class instances in existence. The manifestation of a

class needs not be physical. For example, one's thinking to use 'Agile software life

cycle' is an acceptable instance of the 'Software Project Process Ideas' class.

B. Glossary

ABSTRACTION LEVEL. An attribute of a software instance that contains a value

    indicating the closeness of a software instance to machine code. Machine code has

    the lowest abstraction value.

ABSTRACT CLASS. A class that cannot be instantiated.

ACTIVITY. A PAMPA class that is composed of an initial milestone and a final

    milestone.

AGGREGATION. see aggregation ordinary.

AGGREGATION COMPOSITE. of class. A whole/part relationship. "a strong form of

    aggregation that requires a part instance be included in at most one composite at a

    time. If a composite is deleted, all of its parts are normally deleted with it. Note that

    a part can (where allowed) be removed from a composite before the composite is

    deleted, and thus not be deleted as part of the composite." An example is shown

    below:
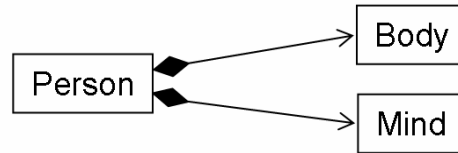
AGGREGATION ORDINARY. An example of an aggregation where the Testing Team class owns Software Engineers is diagrammatically shown below using the UML syntax
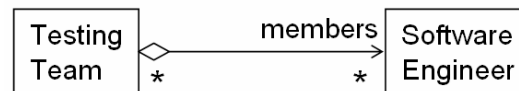
ARTIFACT. A PAMPA knowledge base class. An artifact is any object that is created and maintained by software tools. This research expands the definition of artifacts from objects maintained by direct software tools to include objects maintained by in-direct software tools. For example, a project plan is an artifact created by a project management software tool, issues, and problem reports are artifacts; design objects are also artifacts.

ATTRIBUTE. A characteristic of an object; for example, the object's color, size, or type. Some characteristics can be measurable, countable, or comparable.

CANONICAL ATTRIBUTE PROJECT SET (CAPS). CAPS is a set of software project attributes that can be used for retrospective project review to improve software processes or for team capability assessment.

CLASS. of unified modeling language infrastructure [36]. An element of the M2 layer of the Four-layer Metamodel Hierarchy where UML is defined. Instantiation of class from the M2 layer to the user model M1 layer results in a representation of a set of possible real-world elements.

CLASS ABSTRACT. See abstract class.

COMPLETION. A milestone.

COMPOSITION. class. A class that include other classes. Instance of the included class has only a single object as its owner. When the owner object is deleted, the instance is also deleted.

CRITERIA. A question that can result in a 'Yes' or a 'No' answer.

DEFECT. An set of incorrect instructions in the source code.

DEMONSTRATION. Running of executables of an in-progress software project for informative purpose.

EXECUTABLE. see object code.

FOUR-LAYER METAMODEL HIERARCHY. see UML. FOUR-LAYER METAMODEL HIERARCHY.

GENERATION. A graph parameter that indicates the growth rate of the dependent variable. For example, the graph below contains six plots with different generation parameter values. The different rate at which the vertical value increases is due to the different generation parameter values.
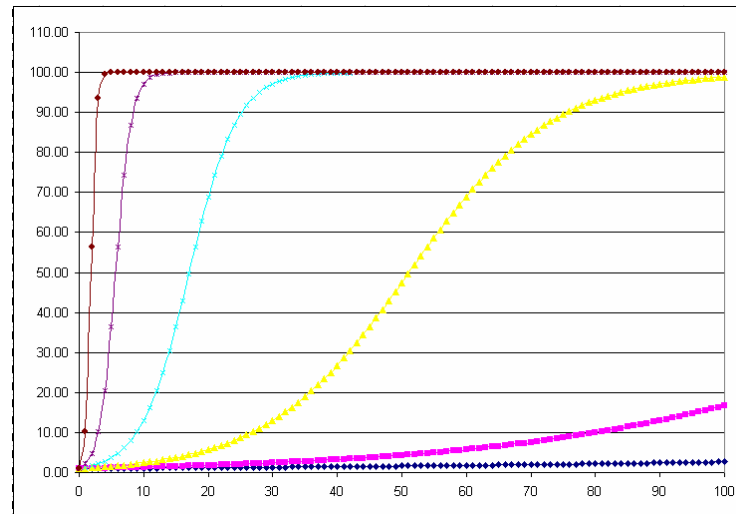
GROUNDED INSTANCE. An element of the M0 layer of Four-Layer Metamodel Hierarchy.

INSTANCE. of a class. An occurrence of a class. Each instance has the following attribute: instance name.

INSTANCE COUNT. CLASS. An attribute of a model that contains the number of instances in the M0 Four-Layer Metamodel Hierarchy.

ISSUE. Deviation from requirement or generally accepted behavior that causes material operation inefficiency. Issues usually occur during software operation or testing. In this particular experiment, software process-related issues are also recorded as issues.

METAMODEL (*) [36]. M2 layer of the Four-Layer Metamodel Hierarchy. For example, the Unified Modeling Language (UML) is a metamodel.

MODEL. equation. A mathematical equation that maps values from one set of values to another set of values.

MODEL [36]. graphic user. Define languages that describe semantic domains, i.e., to allow users to represent a wide variety of problem domains.

NPAG (Normalized Proportion Artifact Graph). A display of artifact records, wherein both the time axis and the vertical axis are measured from 0 through 100.

OBJECT CODE. A software instance in a format that can be recognized by a computing machine. A software instance of the lowest abstraction value.

PAMPA. An acronym for Project Attribute Monitoring and Prediction Associate. A computing system that monitors and predicts software project attributes. It is composed of a knowledge base and an expert system.

PAMPA KNOWLEDGE BASE. A UML-based user model of software project that is composed of 35 classes that define the Plan, Supplier, Organization, Software Product, and Customer areas of a software project and their relationships.

PARAMETERS. Fixed value in a mapping function between two set of numbers.

PROCESS. A PAMPA class that is composed of activities. (1) A sequence of steps performed for a given purpose; for example, the software development process. (2) An executable unit managed by an operating system scheduler. *See also:* task; job. (3) To perform operations on data.

READINESS. A graph parameter that indicates the starting point of significant dependent variable growth.

REQUIREMENT*. (1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other

formally imposed documents. (3) A documented representation of a condition or capability, as in (1) or (2). *See also:* design requirement; functional requirement; implementation requirement; interface requirement; performance requirement; physical requirement.

S-Curve. An equation that maps an independent variable that represents time to a dependent variable. The equation embodies the description of 3 stages: slow initial growth of dependent variable, followed by rapid growth, and finally by slow growth. An s-curve equation is defined as $c = \dfrac{L}{1 + r \cdot \exp(-g \cdot t)}$, where t is the independent variable and c is the dependent variable, g and r are parameters, and L is a constant. A possible usage of the equation is to fit a sequence of t and c pairs to derive the g and r parameters using linear regression fitting process. An S-curve can be used to describe adaptation of technology with the pass of time. In our context, we use an s-curve to describe the evolution of a software artifact.

SOFTWARE ENGINEERING TOOL. see Software Tool.

SOFTWARE LIFE CYCLE. The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. *Note:* These phases may overlap or be performed iteratively.

SOFTWARE PROJECT. A set of classes and relationships defined by PAMPA (Project Attribute Monitoring and Prediction Associate).

STORY. A text description that conveys the software requirement from the users' point of view. Each story is stated in such wording so it can be validated. Story is a requirement document that is used in the Extreme Programming software development process.

UML. Unified Modeling Language. a visual modeling language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET). http://www.omg.org/docs/formal/07-02-04.pdf

UML. FOUR-LAYER METAMODEL HIERARCHY. An Object Management Group (OMG) description of graphical modeling language. The detailed description is in reference [36]. The four layers are named M3/meta-metamodel, M2/metamodel, M1/model, M0/run-time instance. The Unified Modeling Language (UML) is an M2 layer element, software architect and software designers instantiates UML to create models in the M1 layer, and M0 layer contains run-time instances of M1 model. Below is an example of the Four-level Metamodel Hierarchy [36]:
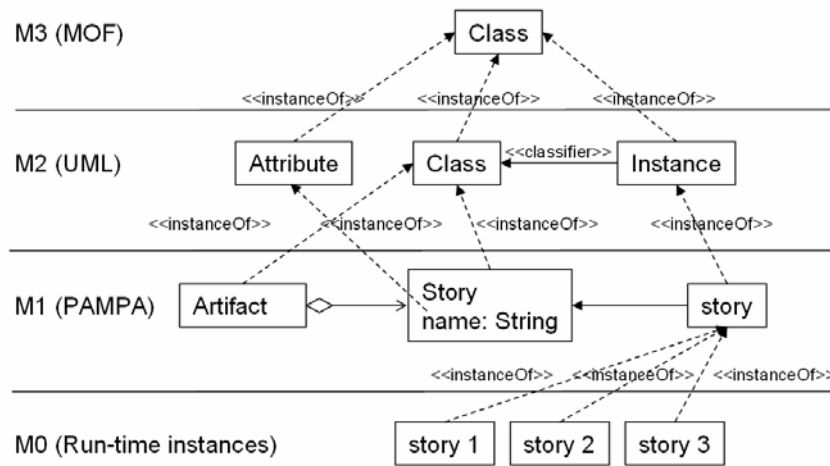
UNIT. MEASUREMENT. A set quantity of an instance attribute that has a name and is

generally known. The set quantity is used to describe the attribute. Defined in the

context of four items: 1) an instance, 2) an attribute, 3) a quantity with respect to the

attribute, and 4) a name. Example 1, after measuring the diagonal length of a laptop

screen, one writes down 23. To answer the question 'What is the unit of 23?', one

states "The instance being measured is a laptop screen, the attribute being measured

is length, the quantity is 23, and the name of the unit is 'inches'". Example 2, after

counting stars in the sky, one writes down 230. To answer the question 'What is the

unit of 230?', one states "No instance is being measured, instances are being

counted. No attribute is being measured. The quantity is 230, and the name of the

unit is 'count'". Example 3, after learning that the Darkness of a Night can be

defined by the number of viewable stars, one counted the number of stars in a sky

and wrote down 230. To answer the question 'What is the unit of 230?', one states

"The instance being measured is the sky, the attribute being measured is 'Darkness

of a Night', the quantity is 230, and the name of the unit is 'star'" We note that

while quantity and the physical act of counting for example 2 and example 3 are the

same. The question "What is the unit of 230?" warrants different answers because of

difference in context.

APPENDIX B

RECORDED ARTIFACTS

Below is the set of artifact data retrieved from a successful software project. The
project produced a successfully electronic commerce web site named Purchase
Tracker. This table only gives the final magnitude of each of the artifact. The
complete record includes the sequence of the artifact magnitude collect during each
day of the project.

Measurement	Lifecycle	Tool	Type of Tool	Final Size
Requirement Count	Requirement	Rational RequisitPro	Requirement tool	74 Tasks
Design objects	Design	Rational Software Architect	Design	7 Use cases, 6 Interaction Diagram, 1 Database Diagram.
Test Cases	Test	Excel	Testing	30 Test Cases, 29 Completed Test Cases.
Lines of Code	Development	Subversion	Configuration management tool	7758 Lines of Code
File Count	Development, Maintenance	Subversion	Configuration management tool	3541 Files
Issue Count	Development, Maintenance	ClearQuest	Issue Tracking Tool	27 Closed Issues, 38 Ending Issues.
Unique Operator Count	Development, Maintenance	Subversion	Configuration management tool	23 Unique Operators
Unique Operand Count	Development, Maintenance	subversion	Configuration management tool	1158 Unique Operands
Operator Count	Development, Maintenance	subversion	Configuration management tool	3940 Operators
Operand Count	Development, Maintenance	subversion	Configuration management tool	6622 Operands
Database Table Count	Development, Maintenance			6 Database Tables.

APPENDIX C

EXPERIMENT APPLICATION USER MANUAL

Below is part of the User Manual from the Application Project of the experiment. The application team carried through the extreme programming practice and developed a full function electronic commerce web application that manages a store's inventory. The system's name is Purchase Tracker.

Purchase Tracker User Manual
(Version 1.0)

Table of contents

Introduction to purchase tracker:

Purchase tracker is an application mainly designed to track the cash outflows with in the various accounts of a particular user. This application not only enabled the users to keep track of there daily purchase transactions, but also helps the vendors to maintain their inventory reorder levels. Key feature in this application is the graphical representation of the user inputs to understand the data more efficiently.

System Requirements:

Web Browser: Internet Explorer 5 or higher, Mozilla, or Opera 6 or higher

How to create a user account

Steps:
1. Click on the 'Add User' link in the opening webpage.
2. Add the user information into the new user request form and Click Next Button

3. Add the user account information and click 'Create Accounts'

3

4. A user account is created with the following the following information submitted by the user.

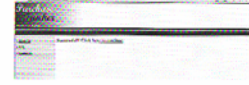**How to login into the purchase tracker application**

Steps:
1. Enter the username in the username / user id text field. Enter the corresponding password for the username in the password text field.

2. Click 'Login' button in the webpage.
3. If the login is successful, a success message is displayed. Else a error message is displayed.

4. A user account is created with the following the following information submitted by the user.

How to login into the purchase tracker application

Steps:
1. Enter the username in the username / user id text field. Enter the corresponding password for the username in the password text field.

2. Click 'Login' button in the webpage.
3. If the login is successful, a success message is displayed. Else a error message is displayed.

4. A user account is created with the following the following information submitted by the user.
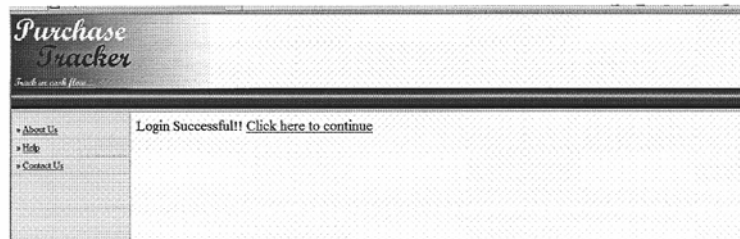

How to login into the purchase tracker application

Steps:
1. Enter the username in the username / user id text field. Enter the corresponding password for the username in the password text field.
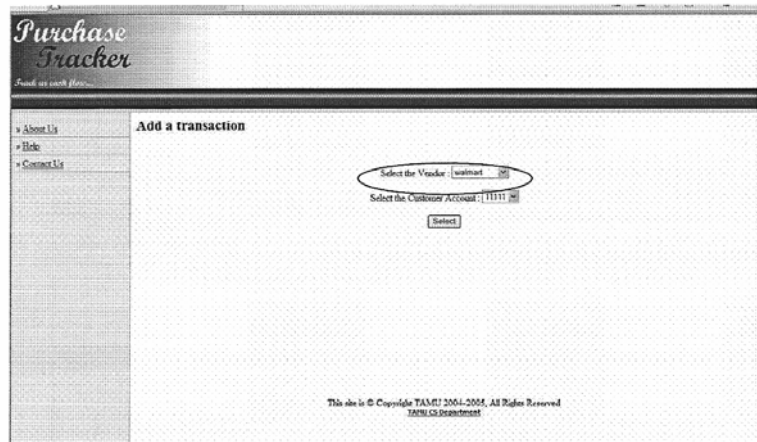

2. Click 'Login' button in the webpage.
3. If the login is successful, a success message is displayed. Else a error message is displayed.

Add a user's purchase transaction

Steps:
1. Click on the 'Add Transaction' link in the user homepage.
2. In order to add a transaction, user must select the store in which the items are bought.

3. Next the user must select the account information in which the transaction is made.

4. Based on this input, a dynamic screen is displayed showing the items in the store. The user must select the item that is purchased along with the quantity of purchase.

5. If more than one kind of item is purchased from the vendor, the user uses 'Add Item' button to one more entries into the screen.

6. If user wants to delete an item, user can click 'Remove Item' button.
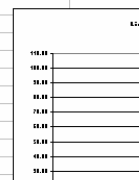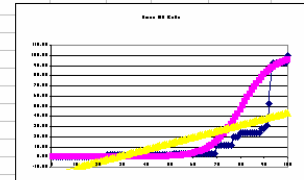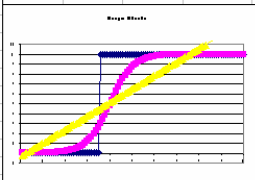

7. Once transaction information is added in the screen. Click 'Add Transaction' button to add the transaction to the database.

APPENDIX D

DATA FITTING SAMPLE

Below is a sample of the artifact values collected during the experiment.

	A	B	C	BQ	BR	BS	BT	BU	BV	BW	BX	BY	BZ	CA	CB	CC
1	Blue lines indicate Demonstration D				100								100			
2	slope1 , intercept 2			L	100							L	100			
3	blue are s-curve parameters			alpha	7.8009	a	2442.92					alpha	12.6799	a	321238.06	
4				beta	(0.1932)	b	0.1932					beta	(0.1568)	b	0.1568	
5	b (intercept)						(3.8612)	2.8333							(16.6009)	27.5830
6	m (slope)						1.3628								0.6019	
7							Design Object							LOC APP		
8						s-curve	linear							s-curve	linear	
9	Week No.	Project Day	Date	Design	trans	Design	Design	Churn of Design	Lines Deleted	Lines Added	LOC	Loc	trans	Loc	Loc	Churn of LOC
87	12	78	4/3/2007	99.90	-6.91	99.9302	102.4369	0	0	0	1560	20.11	1.38	39.0273	30.3436	0
88	12	79	4/4/2007	99.90	-6.91	99.9424	103.7997	0	0	0	1560	20.11	1.38	42.8170	30.9454	0
89	12	80	4/5/2007	99.90	-6.91	99.9525	105.1625	290	0	290	1850	23.85	1.16	46.6929	31.5473	290
90	12	81	4/6/2007	99.90	-6.91	99.9609	106.5253	0	0	0	1850	23.85	1.16	50.6091	32.1491	0
91	12	82	4/7/2007	99.90	-6.91	99.9677	107.8881	0	0	0	1850	23.85	1.16	54.5178	32.7510	0
92	13	83	4/8/2007	99.90	-6.91	99.9734	109.2509	0	0	0	1850	23.85	1.16	58.3717	33.3528	0
93	13	84	4/9/2007	99.90	-6.91	99.9781	110.6137	0	0	0	1850	23.85	1.16	62.1259	33.9547	0
94	13	85	4/10/2007	99.90	-6.91	99.9819	111.9765	0	0	0	1850	23.85	1.16	65.7402	34.5565	0
95	13	86	4/11/2007	99.90	-6.91	99.9851	113.3393	0	0	0	1850	23.85	1.16	69.1807	35.1584	0
96	13	87	4/12/2007	99.90	-6.91	99.9877	114.7021	0	0	0	1850	23.85	1.16	72.4207	35.7603	0
97	13	88	4/13/2007	99.90	-6.91	99.9899	116.0649	0	0	0	1850	23.85	1.16	75.4409	36.3621	0
98	13	89	4/14/2007	99.90	-6.91	99.9917	117.4277	311	0	311	2161	27.86	0.95	78.2298	36.9640	311
99	14	90	4/15/2007	99.90	-6.91	99.9931	118.7905	0	0	0	2161	27.86	0.95	80.7827	37.5658	0
100	14	91	4/16/2007	99.90	-6.91	99.9943	120.1532	239	0	239	2400	30.94	0.80	83.1009	38.1677	239
101	14	92	4/17/2007	99.90	-6.91	99.9953	121.5160	1693	0	1693	4093	52.76	-0.11	85.1907	38.7695	1693
102	14	93	4/18/2007	99.90	-6.91	99.9961	122.8788	2647	0	2647	6740	86.88	-1.89	87.0624	39.3714	2647
103	14	94	4/19/2007	99.90	-6.91	99.9968	124.2416	461	0	461	7201	92.82	-2.56	88.7287	39.9732	461
104	14	95	4/20/2007	99.90	-6.91	99.9974	125.6044	0	0	0	7201	92.82	-2.56	90.2046	40.5751	0
105	14	96	4/21/2007	99.90	-6.91	99.9978	126.9672	0	0	0	7201	92.82	-2.56	91.5058	41.1769	0
106	15	97	4/22/2007	99.90	-6.91	99.9982	128.3300	0	0	0	7201	92.82	-2.56	92.6481	41.7788	0
107	15	98	4/23/2007	99.90	-6.91	99.9985	129.6928	0	0	0	7201	92.82	-2.56	93.6476	42.3806	0
108	15	99	4/24/2007	99.90	-6.91	99.9988	131.0556	0	0	0	7201	92.82	-2.56	94.5192	42.9825	0
109	15	100	4/25/2007	99.90	-6.91	99.9990	132.4184	557	0	557	7758	99.99	-9.21	95.2772	43.5843	557
110	15	101	4/26/2007	99.99				0	0	0	7758	99.99				0
111																
112																
113																

VITA

Name

    Norman K. Ma

Education

    2007, Ph.D. in Computer Science, Texas A&M University at College Station,
        Texas
    2000, M.B.A., Southern Methodist University, Dallas, Texas
    1990, M.S. in Computer Science, University of Tennessee at Knoxville, Tennessee
    1986, B.S. in Computer Science, University of Illinois at Urbana-Champaign,
        Illinois

Professional Experience

    2007, System Engineer, The MITRE, Bedford, Massachusetts
    2003, Teaching Assistant, Texas A&M University at College Station, Texas
    2002, Software Engineer, IntelliSoft Corporation, Plano, Texas
    1995, Software Engineer, Raytheon Systems Company, McKinney, Texas
    1992, Software Engineer, Lockheed-Martin, Glendale, California
    1986, Software Engineer, Texas Instruments, Johnson City, Tennessee

Contact

    Norman K. Ma
    101 Great Rd #108
    Bedford, MA 01730

    Department of Computer Science
    Texas A&M University
    TAMU 3112
    College Station, TX 77843-3112

    www.web2076.net