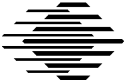
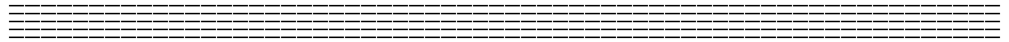
Defining Software Measures: Conference Handout (v1.1)

Software Process Program

Software Engineering Measurement and Analysis Project

© 1996 Carnegie Mellon University

To The User

Our aim with this conference handout is to get this work into the hands of software practitioners prior to final publication of any outputs. The goal is to, in return, receive some candid feedback on the models, methods, and the work in general. Our intent is to use that feedback to improve any outputs from this work so that those outputs are appropriate in an applied software development or maintenance environment.

Enclosed you will find

- a paper entitled, "A Data Definition Framework for Defining Software Measurements." This paper accompanied a presentation at the Software Engineering Process Group (SEPG) Conference, May 1996. The paper describes how to complete and apply a data definition form (DDF) for your measurement program. It has been modified slightly for synthesis into this handout, for example, the references were moved elsewhere to accommodate all references, not just those in the paper. The pages of this section start with "A."

- a DDF for software requirements including checklist, data dictionary, and EER diagram. The pages of this section start with "B."

- a DDF for software changes including checklist, data dictionary, and EER diagram. The pages of this section start with "C."

- references used throughout the conference handout. The references were moved from the SEPG paper to here to accommodate all references used in this handout. The pages of this section start with "D."

- a feedback form for getting your experiences back to us on using this handout. Your input is very important to us and is the primary reason we have assembled this handout prior to final publication of any outputs of this work. The pages of this section start with "E."

Thank you in advance for your input. I look forward to hearing from you. For updates on this work, surf the web to

http://www.sei.cmu.edu/technology/measurement

James A. Rozum

A Data Definition Framework for Defining Software Measurements

by James A. Rozum and Seshadri Iyer, Software Engineering Institute [1]

About this paper

The goals of this work are to develop a repeatable method for defining software measurements and to expand the number of measurements that are currently supported by a checklist-based framework approach to software measurement definition. The data definition framework (DDF) described herein is a prototype of an approach that is being explored. This paper briefly covers:

- our motivation in pursuing this work
- a description of the approach we are exploring

When this work comes to fruition, the SEI team exploring these methods expects to release reports on defining software measures using the DDF concepts. Planned SEI technical reports will describe:

- how to use the common data definition framework to define a set of measurements using a DDF. This report will introduce a common framework for defining software measurements and provide some implemented DDFs for software measures, including three of the four original SEI core measures [Carleton].

- a process to both create a new DDF for an item to be measured and to tailor an existing DDF.

Background on the measurement frameworks

The Software Engineering Institute (SEI) and others recognized the need for better definition of software measurement data. It was, and still is, common to define data that is collected in natural language. This often results in more than one interpretation of the item being measured. When the contents of a specific, seemingly same, data definition are analyzed, the underlying measurement process could often produce different measurement outputs. Thus, having better data definitions of software measurements facilitates repeatability of the measurement process.

To help with this problem, the SEI published a set of checklist-based frameworks for defining four key software measures. The work, published in September 1992, included materials and guidelines for a set of basic measures (size, effort, schedule, and quality) that were intended to help users plan, monitor, and manage their internal and contracted software projects [Carleton].

We have learned much since the release of the 1992 checklist-based frameworks through our field experiences [Rozum], teaching users to use and apply the frameworks, informal feedback, and, most recently, some independent analysis [Mattsson]. A key lesson learned from the collective experience was that the frameworks were very helpful, but there has been mention about improvements and revisions are needed.

[1] © 1996 Carnegie Mellon University

Also, since the release of the 1992 frameworks, the issue of verification and validation of software measurements has grown in importance for measurement practitioners [Fenton], [Kitchenham], [Schneidewind]. This increases the importance of clear measurement definitions.

Leveraging its field experience and lessons learned since the release of the 1992 frameworks, in 1995, the SEI initiated an effort to improve its current foundation by developing a common data definition framework for measurements and to expand the base of existing checklist-based frameworks.

What is a data definition framework?

Data definition frameworks (DDF) are primarily used to define measurements as well to communicate more effectively what a set of measurements represent. Secondary DDF uses include assistance for:

- identifying issues that can be used to focus data analysis
- designing databases for storing measurement data
- developing data collection forms

A DDF can be used to define a set of measurements. For example, a single DDF can be used to identify a line of code measurement, i.e., identify what is to be counted.

A DDF can also be used to help communicate what has been counted. A DDF does this by allowing a user to identify specifically what was included and excluded in a measurement. For example, if I have a count of lines of code, say 317,300 lines of code. The DDF helps me communicate what that number represents by identifying what types of code were counted and included in that number and what types of code were specifically not counted, i.e., excluded.

Our chosen approach?

Our fundamental desire when we began was to develop a framework that could be applied in a consistent and repeatable manner across many different software measures. Our research into how to accomplish this led us to explore methodologies used in the design of information systems.

Using this philosophy, we explored fundamental relational database methods as well as object oriented methods. We eventually chose relational database methods and, perhaps the most common technique for illustrating the design of relational databases, the entity relationship (ER) model [Elmasri], [Getta], [Coleman].

Our model uses an enhanced entity relationship (EER) like model. The ability of EER techniques to generalize and aggregate information is powerful in defining the characteristics of an item to be measured

---

Terminology

As mentioned above, the DDF approach described herein uses an EER-like modeling process. The list below describes the terminology used in the remainder of this paper [Elmasri] [Coleman].

- **Entity:** An entity is a "thing" in the real world with an independent existence. An entity type defines a set of entities that share a set of

common attributes. An entity may be an object with a physical existence: a car, an employee, or an executable line of code; or an entity may be an object of conceptual existence: a company, a job, or a software defect. A rectangle is used to identify an entity type in an EER model.

- **Attribute:** Attributes are the characteristics and properties that further describe the entity type. For example, the entity *employee* might have the attributes, *name*, *address*, *job classification*, etc. As another example, if we consider an entity called a *line of code*, an attribute of a line of code might be the *language* that it was written in. Each entity will have a value for each of its attributes. An attribute is illustrated in an EER model as an oval.

- **Relationship:** A relationship type among two or more entity types defines a set of associations among the entities from the entity types. In EER diagrams, a diamond is used to identify a relationship type and is connected by lines to the entity types that are said to participate in that relationship type. An arrow shows the direction of the relationship.

We often find that an entity type has subgroupings of its entities that are meaningful and need to be represented explicitly in the database. To model this situation, we use a process called generalization that allows an entity type called a **superclass** to be formed by factoring out the common properties of several entity types, called the **subclasses**. On the converse, we also use a process called specialization in which a new **subclass** is defined as a specialized version of the **superclass**.

For example, the members of the entity type called *employee* may be further grouped into subclasses called *management*, *document specialists*, *support staff*, *programmer*. An entity that belongs to one of these subgroupings also belongs to the entity type *employee*. Each of these subgroupings are called subclasses and the entity type *employee* is referred to as a superclass.

In a generalization, the subclasses "share" or "inherit" the attributes of the superclass as well as have attributes that are specific to that subclass. For example, an attribute, *social security number*, may be attached to the superclass *employment class.* Every person that is a programmer would also inherit the attribute. That is, every *programmer* would have an explicit value for the attribute *social security number* .

In our framework, we have created a superclass/subclass specialization from an entity type when we felt there might be a need to model more information.

How to Use the Data Definition Frameworks

DDF architecture

The figure below conceptually illustrates a DDF for defining software measurement data and how an EER model might map to a DDF. The columns of a DDF separate into one of two areas, data definition columns and data manipulation columns. Characteristics of the *measured area of interest* are separated into major sections that can be represented as either a superclass or as a measurement attribute.

Data Definition Columns		Data Manipulation Columns			
Measured Area Of Interest		**Total Count**		**Separate**	**Array**
		include	**exclude**	**Counts**	
Superclass #1					
subclass #1					
	attribute #1				
	attribute #n				
subclass #2					
	attribute #1				
	attribute #n				
Measurement Attribute #1					
	value #1				
	value #n				

Data definition columns

Data definition columns identify the characteristics that a measure of interest can be separated into. The characteristics become items (e.g., superclasses and subclasses) that are modeled in an EER model.

The figure below illustrates a portion of the line of code DDF. The figure has 1 superclass (*statement type*), 2 subclasses (*executable* and *non-executable*), and shows 3 of the possible attributes for the subclass *non-executable* (*declarations, compiler directives*, and *comments*)

Data Definition Columns

	Line of Code		Total Count		Separate	Array
			include	exclude	Counts	
superclass ⇒	**Statement Type**					
subclass ⇒	Executable					
	Non-Executable					
subclass attributes ⇒		Declarations				
		Compiler directives				
		Comments				
	↑ *first column*	↑ *second column*				

The intent of the data definition columns is to have a set of characteristics that, when assembled as a DDF, are collectively exhaustive in their representation of the measure of interest. The characteristics are mutually exclusive at each level in the data definition columns. That is, superclasses are orthogonal of each other; subclasses within a superclass are orthogonal. Orthogonal implies that every instance of the entity type being measured, e.g., a line of code, needs to be able to be represented in one and only one subclass attribute from the set of subclasses within a superclass. In EER modeling terminology, the concept of mutual exclusion is often referred to as disjointness.

Definition column descriptions

There are two columns used for data definition. The first column identifies the top level characteristics for a specific item to be measured. Each top level characteristic is either a superclass or a measurement attribute. These two types of characteristics each become a major section of the DDF. Superclasses are partitioned into subclasses in the first column.

Superclass and **Subclass:** When there is more than one subgrouping for the domain of values represented by the top level characteristic, we establish a superclass with subclasses. For each superclass, there will be two or more subclasses. Subclasses provide additional detail on the characteristic represented by the superclass. Each subclass has a set of attributes attached to it.

Measurement attribute: When a characteristic does not have more than one grouping, but rather a domain of values, we model that characteristic as an attribute of the item being measured. We refer to these types of attributes as measurement attributes. (From a modeling standpoint, if a superclass had only one subclass, the subclass would be redundant and we would model the characteristic as a measurement attribute rather than a subclass.)

The second column is used to identify either the subclass attributes or the values that a measurement attribute can reflect. In either case, the column is the lowest level of detail available for a characteristic of the measurement. Every instance of the measurement characteristic needs to have one and only one representation at this level for each major section. For example, every line of code needs to be able to reflect one and only one value for each measurement attribute; likewise, every line of code needs to also be able to reflect one and only one subclass attribute within every superclass.

Example of data definition columns

Figure 1 below illustrates an example using line of code as the item being measured. In the example the first major section is the superclass *statement type*. The *statement type* superclass is used for identifying the different types of statements that a line of code can reflect.

In the example, the subclass *non-executable* is elaborated with additional detail on the types of non-executable statements. The additional detail identified in the second column are attributes of the subclass *non-executable*. The subclass attributes shown are: *declarations, compiler directives, comments,* and *other* .

To check completeness of this superclass, every line of code needs to have one and only one instance in this section. That is, every line of code needs to be able to be identified as either *executable* or *non-executable*. If the line of code is identified as *non-executable*, then it must also be able to be identified as one and only one of the attributes shown for the subclass *non-executable*.

Line of Code Page 1 of		Total Count		Separate	Array
		include	exclude	Counts	
Statement Type					
Executable					
Non-Executable					
	Declarations				
	Compiler directives				
	Comments				
	Other				
How Produced					
	Programmed				
	Generated				
	Auto Translator				
	Copied (no change)				
	Modified				
	Removed				
	Not identified				

Figure 1: Example from a portion of the DDF on line of code

Figure 1 also illustrates an example using the first column for a measurement attribute. In the figure, *how produced* is a measurement attribute. From the figure we can see that *how produced* can take one of the following values: *programmed, generated, auto translator, copied (no change), modified, removed, not identified.* It is a measurement attribute because there are no subgroupings that can be formed from the different values of the characteristic (at least in this example). Furthermore, every line of code has to be able to reflect one and only of these values.

To reach this state of mutual exclusion among the values, certain counting and classification rules that embellish the definition may need to be established. As is often the case, simply identifying the superclass, subclasses, and subclass attributes is not enough to communicate clearly or to mitigate confusion of what the characteristics represent or how to count the characteristics. For example, more than one type of statement may exist on a physical line, as is the case when comments are embedded on the same line as an executable statement.

To mitigate any confusion or overlap that may exist, a textual definition of what the superclass and its subclasses represent is still required. The textual definition is given in the form of a data dictionary and includes all information that is necessary to apply the mutual exclusion rule. This information includes, at a minimum, a definition of the characteristic, any counting rules that may apply (as in case of how to count comments embedded in an executable line of code), and specific information relevant to the organization and its software process.

Data manipulation columns

In Figure 1, the columns labeled "Total Count," "Separate Counts," and "Array," are referred to as the data manipulation columns. The data manipulation columns are used to identify what went into obtaining the counts (what was included/excluded) for the measurements being made.

There are three basic types of counts that can be made of the item being measured:
- a single number that is a total count of the item being measured and represents a comprehensive view that includes many characteristics, e.g., the number of lines of code in a system
- a single number that is a separate count of a characteristic of the item being measured (i.e., a count of a measurement attribute value, subclass, or subclass attribute) and represents a one dimensional view of the item, e.g., the number of comment lines of code.
- a count that represents a multidimensional view, (or an array of characteristics of the item being measured), of two or more *major* sections of the item being measured, e.g., the number of executable and non executable lines of code that were programmed and the number of each that were generated by a code generator.

Total count"

The "Total count" column is used to identify what subclass attributes and measurement attribute values have been included and knowingly excluded from a count. For example, if a report stated that version 3.1 of the xyz software is 317,478 physical sources lines of code, a user could communicate or understand what that number represents by referring to the total count column and determine what was counted and included in that number as well as what was specifically not counted (i.e., the exclude column).

Whenever a single number that is an aggregate of subclasses and measurement attributes is requested or reported, the total count columns are used.

They are also used to indicate what characteristics are included and excluded, when separate and array counts are obtained.

It is the feature of trying to clearly communicate what is to be counted (or what was counted) that makes completeness of the checklist (identification of superclasses and measurement attributes) important. If the list is not collectively exhaustive, then confusion and miscommunication could occur. Because of the need to be collectively exhaustive, it is just as important to describe characteristics that will be specifically excluded as it is to describe what was included. It also brings to attention what specific actions that are required to be taken to exclude characteristics. This also increases the importance of the data dictionary and the definitions and counting rules included in it.

To use the total count columns, a user (requester of a measurement or reporter of a measurement) would simply check those characteristics that are to be counted in the include column and check the exclude column for those characteristics that are specifically not to be counted. The figure below uses a portion of the earlier line of code example to illustrate the use of the total count columns. Many characteristics are not shown because of space restrictions.

Line of Code		Total Count include	Total Count exclude
Statement Type			
Executable			
Non-Executable			
	Declarations		
	Compiler directives		
	Comments		
	Other		
How Produced			
	Programmed		
	Generated		
	Copied (no chg.)		

If all attributes under a subclass are to be as a complete set, either included or excluded, then a single check in the box for the subclass is allowed to designate that either all attributes are applicable or the level of detail needed/possible is acceptable at the subclass level. This is illustrated in the *how produced* measurement attribute in the example. If there are to be checks in boxes for measurement attribute values / subclass attributes in a section, there should not be a check in the measurement attribute / subclass row. This is illustrated in the *non executable* subclass in the example.

If the figure above represented a complete DDF, the number reported for the total lines of code would represent all statement types that are either *executable*, *declarations*, or *compiler directives*, regardless of how the statement was produced. Specifically excluded from the count would be *comments* and *other* (not *declarations*, *compiler directives* or *comments*); regardless of how produced.

"Separate counts"

Often we have a need for a count of certain characteristics of the item being measured. The framework uses the separate count column as the method to identify what characteristics to also count separately. This lower level, one dimensional view provides additional insight to the item being measured. The additional insight is usually needed to get at the root of an issue or problem of interest to the organization or project collecting the data. In this way, a DDF can be used to assist in the data analysis process.

Line of Code Page 1 of		Separate Counts
Statement Type		
Executable		
Non-Executable		
	Declarations	
	Compiler directives	
	Comments	
	Other	
How Produced		
	Programmed	
	Generated	
	Copied (no chg.)	

A check in the separate counts column gives the user a count for each characteristic checked in the Separate Counts column.

A count for a complete subclass (i.e., all attributes for a subclass) can be obtained by marking the separate count column in the row of a subclass. This is equivalent to saying, “I don't care about the specific attributes of the subclass, but am only interested in the subclass of the item being measured.” In the example above, there would be a separate count of all of the executable lines of code and another separate count of all the non-executable lines of code.

Each box checked in the Separate Counts column represents an independent count of the item being measured and therefore does not have to have a box checked in the same row in either the include or exclude columns of the total count columns. However other characteristics that affect this count need to be specifically included or excluded in the total count columns.

The figure above illustrates how the separate counts column can be used. Using that figure, four separate data items (counts) would be collected and reported. Separate counts would be made that represent:

- all executable lines of code
- all non-executable lines of code
- all lines of code that were programmed (i.e., written by individuals and not falling into one of the other measurement attribute values)
- all lines of code generated by code generators

“Arrays”

To analyze measurement data, users very often need to have multidimensional views of the item being measured. The multidimensional view is across more than one "major" section (i.e., an array involves a combination of more than one superclass or measurement attribute). For example, a user may have a need to know for both executable or non-executable lines of code, how many were copied without change, removed, and modified.

The Array column gives the user the ability to get that multidimensional view into an area of concern. We have defined a valid array as having a box checked in the array column for at least two major sections.

The figure below illustrates how the array columns could be used to define a 2x3 array that represents the data requirements for the described example.

As when making separate counts, rows checked in the array columns do not need to have a box checked in the include or exclude columns of the total count column. However other characteristics that affect this count need to be specifically included or excluded in the total count columns.

The number of actual measurements made is equal to the number of checks in one section multiplied by the number of checks in the other(s). In the below example, there would be 6 measurements made (3 x 2 = 6).

Line of Code		Array
Statement Type		
Executable		
Non Executable		
	Declarations	
	Compiler directives	
	Comments	
	Other	
How Produced		
	Programmed	
	Generated	
	Auto Translator	
	Copied (no change)	
	Modified	
	Removed	
	Not identified	

If more than one array is needed, the user can simply devise a coding scheme to distinguish the arrays. The most common scheme we use are alphanumeric letters.

If more than two dimensions are needed, the user simply uses a combination of characteristics for each dimension. A one dimensional array can be obtained using the separate count columns.

Putting it all together

The figure below illustrates how the DDF would look when the "parts" from above are put together. The DDF illustrated in the figure below would require that 12 measurements be taken. But will report 15 measurements: 1 totals count measurement, 4 separate counts measurements, and 2 arrays, a 3x2 (6 measurements) and a 2x2 (4 measurements).

Line of Code		Total Count		Separate Counts	Array
		include	exclude		
Statement Type					
Executable					a, b
Non Executable					a, b
	Declarations				
	Compiler directives				
	Comments				
	Other				
How Produced					
	Programmed				b
	Generated				b
	Auto Translator				
	Copied (no chg.)				a
	Modified				a
	Removed				a
	Not identified				

Figure 2: Filled out example of a portion of the line of code DDF

What if you are using the 1992 frameworks?

Remembering that this is work in progress, we do not recommend that those using the original frameworks published in 1992 discontinue using them or use them any differently than they are currently.

A self-imposed requirement that our team had was to add functionality, not reduce any functionality that may have existed in the 1992 versions. We do expect some changes; after all, the new work should be an improvement from the existing. We also do not expect that even this new work will be perfect; hence our need for input from you, the users.

Generally, we believe that any existing data definition using the 1992 frameworks can be mapped into the new frameworks when they are released. The benefits to those using the current frameworks will come from a common and repeatable approach to using a set of frameworks and, therefore, communicating with them—one of their original intents.

Summary

We have introduced a technique for defining software measures. This technique builds on earlier SEI work. The goal of this work is to provide a common framework for defining software measures. We believe that this work will lead us to accomplish a larger goal of developing a methodology to define (almost) any software measure.

In this paper we have introduced many rules for using the DDFs. In this summary, let us just say that rules are made to be broken. As others like [Luce], [Michell], and [Gaito] have pointed out, responsible data analysis must be open to anomaly if it is to support scientific advancement.

We believe that a few anomalies will always exist and rules will be broken. The rules on subclass attributes and measurement attribute values being mutually exclusive (i.e., disjoint) may be an example if we migrate closer to information systems design methodologies.

We hope to hear from you on your experiences of using this approach as well as your experiences with using the 1992 checklists.

If interested in providing comments or input on this paper or future drafts[2], you can do so through the SEI world wide web home page (http://www.sei.cmu.edu/technology/process.html). You can also send your requests for more information or comments on using DDFs, to jar@sei.cmu.edu or to:

James A. Rozum
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

[2] Refer to the feedback questionnaire at the end of this handout to submit comments.

DDF for Defining Measures of Software Requirements

As with most measurement definitions, one cannot expect to immediately recognize or distinguish the item being measured under operational conditions. For example, someone not familiar with source code cannot pick up a data definition or a completed framework for lines of code and be able to identify a line of code.

Measuring requirements also suffer this fate. Using the EER model, DDF, and data dictionary contained herein will not help you identify what a requirement is. Using the EER model, DDF, and the data dictionary will help you identify characteristics that software requirements must reflect (otherwise, the characteristic should not be on the DDF).

The data dictionary included herein is meant to illustrate what a data dictionary can contain. It is not definitive and as such, does not seek to prescribe how to identify a software requirement. There are too many ways of identifying requirements for us to be prescriptive. For example, some processes may use formal methods for documenting requirements. In this scenario, there may be several levels describing a requirement as it continually gets decomposed. Some software processes count “shall” statements; others may use a documentation format for their requirements specification that identifies requirements. Many software maintenance processes use artifacts from a configuration management process. Therefore, proposing a definitive or prescriptive approach on how or what a software requirement is would be futile and would serve only a small fraction of the needs of software practitioners.

Our recommendation is to apply the enclosed information at the level that you track and manage requirements. For most DDFs, the characteristics included on the DDF will be independent of the process. The DDF included herein is also independent of how a software requirement is identified.

When you use the information enclosed you will need to complete a data dictionary for the DDF. We recommend that in the foreword of your data dictionary you clearly define your methods, description, and counting rules for identifying a software requirement within your software process.

We are very interested in how you define requirements and your experiences in using this DDF; please use the enclosed questionnaire to tell us about your experiences.

A DDF for Software Requirements

System Software Requirements page 1 of 2		Total Count include	 exclude	Separate Counts	Array
Verifiable					
	yes				
	no				
Consistent					
	yes				
	no				
Stability					
	Never changed				
	1 change				
	2 changes				
	3 or more				
	Deleted				
Criticality					
	1st level				
	2nd level				
	3rd level				
	4th level				
	other				
Classification					
	Functional				
	Performance				
	Interface				
	Operational				
	Resource				
	Verification				
	Acceptance test				
	Documentation				
	Security				
	Portability				
	Quality				
	Reliability				
	Maintainability				
	Safety				
	Other				
Specification Meth					
	Formal				
	Informal				

System Software Requirements page 2 of 2		Total Count include	exclude	Separate Counts	Array
Introducing Activity					
	Reqt. Gathering				
	Reqt. Analysis				
	Design				
	Coding				
	Testing				
	Maintenance				
	Software Arch.				
	Release Prep.				
Status					
Open					
	Recognized				
Inactive					
	Suspended				
Active					
	Analyzed				
	Designed				
	Coded				
	Tested				
Closed					
	Approved				
	Released				
	other				
Elicitation Activity					
Interviews					
	Customers				
	Users				
	Other				
Existing Documents					
	Internal Sys.				
	External Sys.				
Prototypes					
	Internal to org.				
	External to org.				
Existing Systems					
	Internal to org.				
	External to org.				
Analysis of					
	System				
	Domain				
	Other systems				
	Other				
Change Requests					
	Internal				
	External				

Data Dictionary for Software Requirement Measurement

The information used in the DDF and the accompanying (portion of) its supporting data dictionary comes from the following primary sources [IEEE_830], [Mazza], and [Christel].

Verifiable

Verifiability is a measure of whether a requirement can be checked that it is implemented in the software system. Ideally, every requirement should be verifiable; however, we recognize that, often, this is not possible.

Our criteria for verifiable is testability, (i.e., there is a sequence of steps by which it can be demonstrated that the requirement has been implemented.) Projects will need to define criteria to determine whether a requirement is verifiable or not. For each requirement, the value that this attribute can take is binary; either the requirement is verifiable or not. An alternative that organizations may also pursue is to define the levels of testing that a requirement needs for verification in the implemented system and track each requirement to a level of verifiability.

Consistent

Consistency is a measure of the logical coherence and compatibility of a requirement with all the other requirements. A requirement is consistent if, and only if, no individual requirements conflict with each other. For example, the specified characteristics of real-world objects may conflict, or, there may be logical or temporal conflict between two specified actions. For each requirement, the value that this attribute can take is binary; either the requirement is consistent or not.

Stability

Stability is an indicator of change that a set of requirements undergoes. This attribute is used to track the number of changes made to a requirement. Some requirements are more stable over the expected life cycle of the software; others are more dependent on feedback from the phases of the software life cycle.

This is only a recommendation on how stability can be classified. Projects can use this as a starting point to determine their own classification. We use the following values associated with a requirement. It is expected that prior to implementation, each has criteria determined for it.

- ***never changed***
- ***changed once***
- ***changed twice***
- ***changed three or more times***
- ***deleted***

The *stability* attribute is unique for a requirement. The attribute is mandatory and single valued. The criteria and process used to define a change to a requirement would must be defined by an organization.

Criticality

Criticality is a measure of the importance of a requirement from the user's perspective. Criticality is measured in several levels, the most critical being absolutely essential, and the least critical being not immediate. Criticality becomes important in incremental software life cycles where only a subset of all the requirements are implemented and provided to the user.

This is only a recommendation on how criticality can be classified. Projects should define levels according to the systems mission and organization's software process. We have included four values associated with this attribute for measuring the criticality of a requirement:

- ***1st level:*** Most critical, the system needs this requirement to be correctly implemented because there are no alternative workarounds.
- ***2nd level:*** The requirement is needed, but there is a workaround for the short term.
- ***3rd level:*** The requirement will be needed in the near future.
- ***4th level:*** The requirement is not immediately required.

The value of the *criticality* attribute is unique for each requirement. The attribute is mandatory and single valued. As the system matures, it should be expected that for each release the criticality of all requirements should be reviewed to determine if changes are necessary.

Classification

Classification refers to the type of a software requirement. Values that we have identified that this software requirement attribute can take on include the following:

- **functional:** These requirements specify 'what' the software has to do. They define the purpose of the software. To satisfy criteria that requirements be quantitative, sometimes the functional requirements include performance attributes. When this occurs criteria need established for determining which value to use to track the requirement.

- **performance:** These requirements specify numerical values for measurable variables (e.g., rate, frequency, capacity, speed, etc.) of performance characteristics that the software must meet. Performance requirements may be incorporated in the quantitative specification of each function, or stated as separate requirements. Qualitative statements are typically unacceptable, e.g., "quick response" should be replaced with "response time must be less than x seconds for y% of the cases with an average response time of less than z seconds."

- **interface:** These requirements specify hardware, software, or user elements with which the system, or system component, must interact or communicate. Interface requirements can be classified into software, hardware, and communication interfaces. Software interfaces can include operating systems, software environments, file formats, database management systems and other software applications. Hardware interface requirements may specify the hardware configuration. Communications interface requirements constrain the nature of the interface to other hardware and software. For example, the requirements may demand the use of a particular network protocol.

- **operational:** These requirements specify how the system will run and how it will communicate with the human operators. Operational requirements include user-interface, usability and human-computer interaction requirements as well as logistical and organizational requirements. Examples include: the screen layout, the content of error messages, help systems, etc. It is often useful to define the semantics and syntax of commands.

- **resource:** These requirements specify upper and/or lower limits on the physical resources such as processing power, main memory, disk space, etc.

- **verification:** These requirements specify the constraints on how the software is to be verified. The verification requirements might include requirements for simulation, emulation, live tests with simulated inputs, live tests with real inputs, and interfacing with the testing environment.

- **acceptance test:** These requirements specify the constraints on how the software is to be validated prior to release.

- **documentation:** These requirements specify the project-specific requirements for documentation of the software being developed.

- **security:** These requirements specify the requirements for securing the system against threats to confidentiality, integrity, and availability. Examples of security requirements are the inhibiting of specific commands, read-only access, a password system, and computer virus protection.

- **portability:** These requirements specify the needs for modifying the software to execute on other computers and operating systems. Possible computers and operating systems, other than those of the target system, should be stated.

- **quality:** These requirements specify the quality standards of the product. Where appropriate, software quality attributes should be specified in measurable terms.

- **reliability:** These requirements specify the acceptable mean time interval between failures of the software. The reliability requirements may also specify the minimum time between failures that is acceptable.

- **maintainability:** These requirements specify the level of effort to repair faults and adapt the software to new requirements. The effort required for performing these tasks should be stated in quantitative terms, such as the mean time to repair a fault. The maintainability requirements may include constraints imposed by the potential maintenance organization.

- **safety:** These specify any requirements to reduce the possibility of damage that can follow from software failure. Safety requirements may identify critical functions whose failure may be hazardous to people or property.

The *classification* attribute may not be unique for a requirement. However, by defining *classification* as an attribute, we have prescribed it to be logically mandatory and single valued. We recognize that this may not be the case or may not meet the needs of organizations; there are several methods to design an information system to allow multiple values for this attribute conceptually. The criteria and process used to define these values must be defined by the organization.

Specification Methodology

The *specification methodology* attribute indicates how the requirement has been represented in documented specifications. Requirements elicitation is concerned with gathering information from various stakeholders to derive the requirements of the system. This collected information (requirements) needs to then be represented in some manner. Ideally, the gathered statements are expressed in a notation which elucidates their implications, prompts further questions, and facilitates detailed analysis and design of the system. Defined are two values that this attribute may reflect: *formal* and *informal*. Formal specification are methodologies that require the use of a predicate logic or other formal methodology. Informal methodologies for expressing requirements include textual documents (including standardized organization formats), picture, screens, menus, etc.

Introducing Activity

This measurement attribute is used to track the activity during which the requirement was introduced. All requirements are introduced during activities of the software life cycle.

This is only a recommendation for the introduction activities to use. Projects should determine their activities based on their software process. The values for this attribute that we have defined are the following:

- ***requirements gathering***
- ***requirements analysis***
- ***software architecture***
- ***design***
- ***coding***
- ***testing***
- ***maintenance***
- ***release preparation***

Status

The requirements *status* category identifies a stage in the requirements' implementation. It represents the state the requirements are in, and progress made towards their satisfaction.

We define four states with inherent substates: open, active, closed and inactive. A certain state or substate is achieved if certain criteria are met. These criteria are defined below. This is only a recommendation on how states can be classified. Projects may want to determine their own classification. The definitions of the requirement states and their substates are the following:

- **open:** This term means that the requirement is recognized. The *open* value has only one option.
  - ***recognized*:** To be in the recognized state, sufficient data has been collected to permit an evaluation of the requirement. Specific data items would need determined, that when available, the requirement would be considered in the recognized state.

- **active:** This term implies that there has been some level of investigation and action to implement the requirement. The active value is decomposed into substates to obtain detailed understanding of the work remaining in the implementation of the requirement. The *active* value has the following options:
  - ***analyzed:*** Sufficient data has been collected to determine details of its implementation. Depending on the organization, the amount of data required to satisfy the criteria for this state may vary significantly.
  - ***designed:*** This state indicates that the requirement has been implemented in the design of the software system satisfactorily. Depending on the organization, the criteria for this state may vary significantly. For example, the criteria may be satisfied only after an inspection/review of the design document is complete, the design baseline and a traceability exists between the design baseline and the requirement.
  - ***coded:*** This state indicates that the requirement has been implemented in the code of the software system satisfactorily. Depending on the organization, the criteria for this state may vary significantly. For example, the criteria may be satisfied only after an inspection/review of the code is complete, the code is baselined and a traceability exists between the code and the requirement.
  - ***tested:*** This state indicates that the requirement has been tested and verified. Depending on the organization, the criteria for this state may vary significantly. For example, the criteria may be satisfied only if all test cases have been executed and documented, and a traceability exists between the test cases and the requirement.

- **closed:** This term means that the requirements have been implemented and tested. The *closed* value is decomposed into the following substates:
  - ***approved:*** Sufficient data has been collected to ensure that the requirement is implemented satisfactorily in the software system and is approved for release to the customer.
  - ***released:*** These requirements have been released to the customer.
  - ***other:*** These requirements include requests by the customer that have been discarded by means of discussion between the customer and the developer.

- **inactive:** This term implies that a requirement has been suspended awaiting discussions between the customer and the developer. The *inactive* value has only one option.
  - ***suspended:*** These requirements include requests by the customer that have been temporarily suspended and require further discussion between the customer and the developer.

Elicitation Activity

The *elicitation activity* category refers to the activity, process, or operation taking place when the requirement is identified. Identified are six subcategories of activities, each unique in terms of the process used to elicit requirements. The following is only a recommendation on how elicitation activities can be classified. Projects should determine their own classification scheme. The definitions of the elicitation activities and their subcategories are as follows:

- **existing systems:** Studies and analyses of existing systems that are similar to the system for which requirements are being gathered result in requirements that may be missed otherwise. Examples might include user interface (screens, help text), performance issues etc. The existing systems are further refined into two subcategories: external and internal. The subcategories refer to whether the existing system reflect functionality of a system that is internal to the organization or functionality of a system that is external to the organization.

- **existing documents:** Studies of documents of existing systems that are similar to the system for which requirements are being gathered result in requirements that may be missed otherwise. The existing documents are decomposed into two subcategories: *external* and *internal*. The subcategories refer to whether the existing documents reflect functionality that exist in a document that is internal to the organization or external to the organization.

- **interviews:** Interviews are perhaps the most common technique used for gathering information during requirements elicitation. The information collected through interviews can address organizational and contextual factors provided that the right questions are asked. Likewise, if the right people are interviewed the information will represent multiple stakeholders' opinions across the different communities affected by the development of the proposed system. It is expected that each subcategory has further defined criteria to clearly parse individuals into one of the following:
  - ***users:*** This relates to all interviews conducted with the users of the software system being built.
  - ***customers:*** This relates to all interviews conducted with the customers of the software system being built.
  - ***others:*** This relates to all interviews of people other than users and customers.

- **prototypes:** The prototyping activity in which the objective is to define requirements is called exploratory prototyping. Prototypes are often used to test customer reaction and into design ideas. In this way, a prototype implements selected aspects of the proposed software system so that concept can be demonstrated and validated. Also, prototypes are often used to implement high risk functional, performance, or user interface requirements. Prototypes are decomposed into two subcategories: *external* and *internal*. The subcategories refer to whether the prototype was developed within the organization (internal) or externally, e.g., when an acquisition agency furnishes software or a system to a developer of an unprecedented system.

- **change requests:** Requests refers to the process of collecting information from parties at multiple levels within the development and

user communities for requirements for the software system being created. For example, information may be requested from a high-level commander for a strategic long term perspective or from an end user for the immediate perspective. Requests are decomposed into two subcategories: *external* and *internal*. The subcategories refer to whether the requests come from a person or group internal to the developing organization or from outside of it (external).

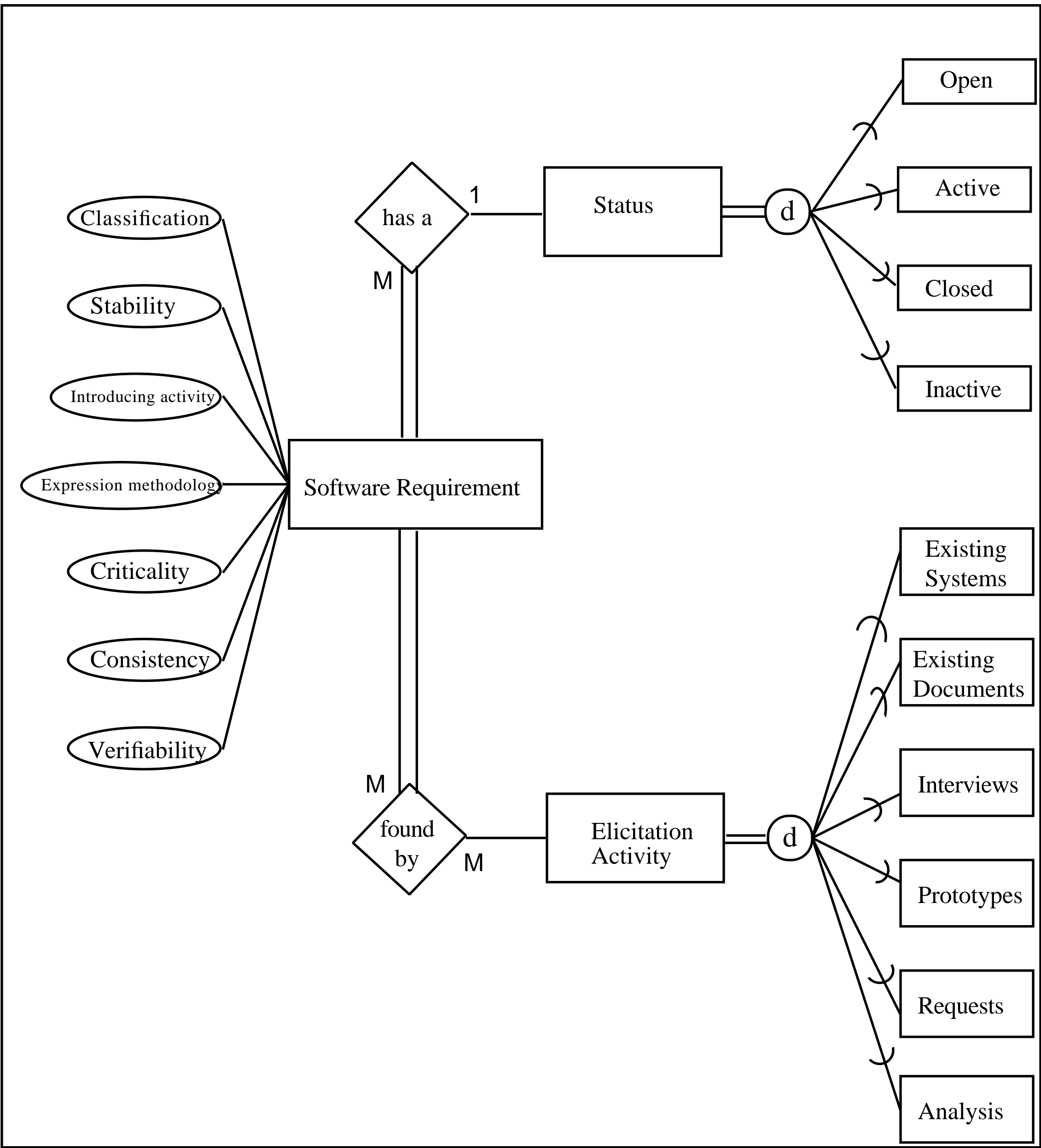
- **analysis:** Detailed analysis of gathered requirements results in reducing the problems of understanding and scope. This process may result in new and/or changed requirements for the system. As an elicitation activity that uncovers requirements, analysis can be further refined into: ***system, domain,*** and ***other systems***.

EER model for software requirements

The figure below illustrates the EER model for software requirements. For simplification of the figure, attributes are not shown for the subclass entities.

Regarding the superclass, *status*, a requirement cannot be in more than one state at any point of time. A certain requirement is always in at least, and at most one state; therefore, the cardinality of the relationship *has a* for the entity type, *software requirement* should be mandatory and single-valued, therefore, the cardinality for the relationship is one-to-one and participation is total. The *status* superclass options could have many requirements associated with them. Likewise, a *status* superclass option may not have any requirements associated with it, (e.g., there may be no requirements that are *inactive*.) Therefore, the cardinality for the relationship is one-to-many and participation is partial.

Regarding the superclass, elicitation activity, it should be recognized that requirements are identified during different elicitation activities. To model this, we use the relationship *found by* between the *software requirement* and *elicitation activity* entity types. A particular requirement may appear in one or more elicitation activities. This implies that the cardinality for the relationship *found by* are one-to-many. All requirements must have a means by which they have come into existence, therefore, the participation constraint is total. The cardinality for the inverse relationship could be zero-to-many. The minimum zero cardinality would mean that a certain elicitation activity might cause the identification of zero requirements. This implies that the cardinality is one-to-many and the participation constraint is partial.


Classification
Stability
Introducing activity
Expression methodolog
Criticality
Consistency
Verifiability
has a
1
M
Status
d
Open
Active
Closed
Inactive
Software Requirement
M
found by
M
Elicitation Activity
d
Existing Systems
Existing Documents
Interviews
Prototypes
Requests
Analysis

DDF for Defining Measures of Software Changes

This section uses the 1992 version of the framework for counting problems and defects [Florac] and reformats the information using the common DDF. Users of the enclosed DDF could refer to Chapter 3 of [Florac] for further description of the entities and attributes used in the model and the DDF.

This DDF is for defining measures of software changes to a system that have been requested. As in most development or maintenance efforts, it is not expected that all requested changes be handled immediately or in the next release (baseline), but that some number of changes will be backlogged. This is true for incremental development as well as maintenance of fully fielded systems.

Often, organizations to not separate the backlog of changes into enhancements and problems or defects types (erroneously from a process improvement perspective). This DDF puts an increased emphasis on enhancements (emphasized to a level equal to problem report and defects). Because of this equal emphasis of enhancements and problems this measure has been renamed from measures of software problem reports and defects as in [Florac] to software changes.

This DDF could be modified or another created with a similar structure for changes encountered during the software development or maintenance process. For example, the DDF could be modified or another similar DDF created where changes are action items from reviews, problems encountered during testing, etc.

Users of the 92 framework on counting problems and defects should be able to adopt the new DDF with minor or no changes. '92 users can continue to use the supplemental forms provided along with the '92 frameworks or migrate to a data dictionary methodology similar to that enclosed and used in information systems design. Also, '92 users will have the obvious changes associated with an expansion of the DDF to accommodate the expansion from only measuring problems and defects to also including equivalent coverage for enhancements.

A DDF for Software Changes

Software Changes page 1 of 2		Total Count include	Total Count exclude	Separate Counts	Array
Change Status					
Open					
	Recognized				
	Evaluated				
	Resolved				
	other				
Closed					
	Tested				
	Approved				
	Released				
	other				
Change Type					
S/W Problem/Defect					
	Requirements				
	Design				
	Code				
	Oper. Document				
	Test Case				
	other work product				
Other					
	Hardware				
	Operating Syst.				
	Human Mistake				
	other				
Enhancement					
	New Requirement				
	Modified Reqt.				
	other				
Unknown					
	Not Repeatable				
	Not Identified				
Uniqueness					
	Original				
	Duplicates				
	Not Identified				
Criticality					
	1st Level (most)				
	2nd Level				
	3rd Level				
	4th Level				
	5th Level				
	Not identified				

Software Changes page 2 of 2		Total Count include	Total Count exclude	Separate Counts	Array
Urgency					
	1st Level (most)				
	2nd Level				
	3rd Level				
	4th Level				
	Not Identified				
Finding Activity					
Synthesis of					
	Requirements				
	Design				
	Code				
	Test Document				
	User Document				
Inspection of					
	Requirements				
	Preliminary Design				
	Detailed Design				
	Code				
	Test Document				
	Oper. Document				
Formal Reviews of					
	Plans				
	Requirements				
	Preliminary Design				
	Detailed Design				
	Testing				
	Delivery / Acceptance				
Testing of					
	File / Unit				
	Module/Component				
	Configuration Item				
	Integration				
	System				
	Pilot/Beta				
	Acceptance				
	IV & V				
Post-Release					
	Production				
	Release / Deployment				
	Installation				
	Operation				
Finding Mode					
	Non-operational				
	Operational				
	Not identified				

Data Dictionary for Software Changes

The information for the data dictionary on problem reports comes directly from the original SEI technical report on measuring software quality: *Software Quality Measurement: A Framework for Counting Problems and Defects*. For more information or details on how to effectively measure staff hours refer to [Florac].

Change Status

Change status refers to a point in the problem corrective action process that has been defined to have met some criteria. The *change status* reveals information about the progress to resolve and dispose of the reported changes. A change cannot be in more than one state at any point in time. The process state of a requested change will shift as a change report moves through the corrective action process.

The recognition or opening of a change request is based on the existence of data describing the event. As the investigative work proceeds, more data is collected about the change, including information that is required to satisfy issues raised during the analysis. It is the existence of this data that is used as a set of criteria to satisfy moving from a status of open to a closed status. The two states open and closed are further decomposed as follows. Each state and its set of values would need to be thoroughly and specifically defined relative to an organization's software process.

- **open:** These changes refer to those that are recognized and some level of investigation and action will be undertaken to resolve it. The *open* value is often decomposed into sub-states to obtain a detailed understanding of the work remaining in the analysis and correction process to close the change. Values that we have identified for this subclass include:
  - ***recognized*:** Recognized refers to the state when sufficient data has been collected to permit an evaluation of the change to be made. This implies that the data has been verified as correct as well. This generally means the set of data that one can reasonably expect the change originator to provide.
  - ***evaluated*:** Evaluated refers to the state when sufficient data has been collected and an investigation completed of the reported change and the various software artifacts to at least determine the change type and scheduled for review of its disposition.
  - ***resolved*:** Resolved refers to the state when a change has been reported and evaluated, and sufficient information is available to satisfy the rules for resolution.
  - ***other***: Other refers to a state where a change may be open, but in a state other than the normal, change resolution process. For example, a change may be given a low priority and recognized but not evaluated.

- **closed**: This subclass refers to the state when the investigation of a change is complete and the action required to resolve the requested change has been proposed, accepted, and completed to the satisfaction of all concerned. In some cases, a change report will be recognized as invalid as part of the recognition process and be closed immediately. The values used in this document are described below. Note that even though these attributes are included here, the values could easily be

defined by an organization's process as *open*, rather than *closed.*
- ***tested***: Refers to the state when a change has been corrected and has completed a testing process that verifies that the implementation was successful.
- ***approved***: Refers to the state when a change has been corrected and the software has been approved, but not released to the users.
- ***released***: Refers to the state when the software that corrects a change has been released to the users.
- ***other***: Refers to the state when a change may be closed, but for some reason, does not fit into a state reflective of the normal, change resolution process.

Change Type

Change type is used to assign a value to the change that will facilitate the evaluation and resolution of the changes reported. The change type structure is arbitrarily divided the changes into four subclasses: software defects, other changes, enhancement, and unknown. In practice, each change data record should contain only one of the following change type subclasses.

- **software problem / defect:** This subclass includes all software defects that have been encountered or discovered by examination or operation of the software product. Possible values for this subclass are:
  - ***requirements defect*:** Requirements defect refers to when a mistake is made in the definition or specification of the customer needs for a software product. This includes defects found in functional specifications; interface, design, and test requirements; and specified standards.
  - ***design defect:*** Design defect refers to when a mistake is made in the design of a software product. This includes defects found in functional descriptions, interfaces, control logic, data structures, error checking, and standards.
  - ***code defect:*** Code defect refers to when a mistake is made in the implementation or code of a program. This includes defects found in program logic, interface handling, data definitions, computation, and standards.
  - ***operational document defect:*** Operational document defect refers to when a mistake is made in a software operational document. This does not include mistakes made to requirements, design, or coding documents.
  - ***test case defect:*** Test case defect refers to when a mistake in a test case causes the software product to give an unexpected result.
  - ***other work product defect:*** The value other work product defect issued when defects are found in software artifacts that are used to support the development or maintenance of a software product. This includes test tools, compilers, configuration libraries, and other computer-aided software engineering tools.

- **other:** This subclass includes those changes that contain either no evidence that a software defect exists or contain evidence that some other factor or reason is responsible for the change. Values that we have included are:
  - ***hardware problem:*** Hardware problem refers to when a problem occurs due to a hardware malfunction that the software does not, or cannot, provide fault tolerant support.

- ***operating system problem:*** This value refers to when a problem is such that the operating system in use is responsible.
- ***human mistake:*** Human mistake refers to when a change occurs due to a user misunderstanding, incorrect use of the software, or an error made by the computer system operational staff.

• **enhancement:** This subclass refers to a change that describes a new requirement or functional enhancement that is outside the scope of the software product baseline requirements. After the initial release of a system, users often enter new requirements or enhancements into the process through the change reporting process. By including enhancements as a separate subclass, the maintenance organization can use one system to track the functional changes to a system. Values that we have included are:

- ***new requirement:*** New requirement refers to the enhancement that is requested to implement a change that is a new function or requirement of the system.
- ***modified requirement:*** Modified requirement refers to the enhancement that is requested to implement a change that modifies existing functionality of the system.
- ***other:*** The value other refers to an enhancement that is requested to implement a change but cannot (at that point in time) be classified as a new or modified requirement.

• **unknown change:** This subclass is for when the type of change has not been determined. Values that we have included are:

- ***not repeatable (cause unknown)*****:** This option refers to when the information provided with the change description or available to the evaluator is not sufficient to assign a change type to the change.
- ***value not identified***: This option refers to when the change has not been evaluated.

Uniqueness

Uniqueness differentiates between a unique change and a duplicate. The possible values for this attribute are:

• **duplicate:** The change has been previously discovered.

• **original:** The change has not been previously reported or discovered.

• **not identified:** An evaluation has not been made.

Criticality

Criticality is for distinguishing levels of disruption a problem gives users when they encounter it. The value given to criticality is normally provided by the person or organization originating the change. Criticality is customarily measured with several levels, the most critical being a catastrophic disruption, and the least critical being at the annoyance level.

Urgency

Urgency is for communication to the developer the user's level of importance for resolution and closure of the change. Urgency is customarily measured with several levels, similar to criticality. The most critical level is that a user needs to have a correction or enhancement immediately and further work for that user is not possible. The least critical level is a level of annoyance that the user can work around.

Finding Activity

Finding activity refers to the activity, process, or operation taking place when the change was encountered and is used to distinguish between changes requested during various activities. Five categories of activities are identified, each unique in terms of the process used to detect requested changes. The categories included are:

- **synthesis**: Refers to the act of putting together the parts relative to a value below. Each of the values below are products. This subclass refers to the development of a product. The values included are:
  - ***requirements***: Refers to changes requested while developing the requirements.
  - ***design***: Refers to changes requested while developing the design.
  - ***code***: Refers to changes requested while developing the code.
  - ***test document***: Test document refers to changes requested while developing the test document (s).
  - ***user document***: User document refers to changes requested while developing the user document(s).

- **inspection**: This subclass refers to the act of inspecting or informally reviewing a product relative to a value below. Each of the values below are products. This subclass refers to the inspection of a product. The values included are:
  - ***requirements:*** Refers to changes requested while inspecting the requirements.
  - ***preliminary design***: Preliminary design refers to changes requested while inspecting the preliminary design.
  - ***detailed design***: Detailed design refers to changes requested while inspecting the detail design.
  - **code**: Refers to changes requested while inspecting the code.
  - ***test document:*** Test document refers to changes requested while inspecting the test document(s).
  - ***operational document***: Operational document refers to changes requested while inspecting the operational document(s).

- **formal reviews**: This subclass refers to the act of a formal review, typically with the customer, of a product relative to a value below. Each of the values below are products. This subclass refers to the formal review of a product. Sometimes, changes requested during formal reviews are tracked as action items from the review.
  - ***plans***: Plans refers to the changes requested during a formal review of the management plan(s) of the project undertaken.
  - ***requirements:*** Requirements refers to the changes requested during a formal review of the requirements.
  - ***preliminary design:*** Preliminary design refers to the changes requested during a formal review of the preliminary design.
  - ***detailed design:*** Detailed design refers to the changes requested during a formal review of the detailed design.

- ***testing:*** Testing refers to the changes requested during a formal review of the testing products prior to the start of testing.
- ***delivery / acceptance:*** Delivery/acceptance refers to the changes requested during a formal review of the products to be delivered after the completion of testing.

• **testing**: This subclass refers to the act of testing relative to a value below. Each of the values below are types of testing processes. This subclass refers to changes requested during one of these types of testing. The values included are:
- ***file / unit:*** File / unit refers to the changes requested during the functional testing of a file or unit.
- ***module / component:*** Module / component refers to the changes requested during the functional testing of a module or component
- ***configuration item:*** Configuration item refers to the changes requested during the functional testing of a configuration item.
- ***integration:*** Integration refers to the changes requested during the integration testing of a various components of the system.
- ***system:*** System refers to the changes requested during the functional testing of the complete system.
- ***pilot / beta:*** Pilot / beta refers to the changes requested during the testing of the system by users, but prior to the formal release of the software.
- ***acceptance:*** Acceptance refers to the changes requested during the acceptance testing of a system.
- ***IV&V:*** Refers to the changes requested during the independent verification and validation (IV&V) of the system.

• **post-release**: This subclass refers to the acts associated with operating the software after it has been released. This subclass refers to changes requested after release of the software. The values included are:
- ***production***: Production refers to the changes requested during production of the product.
- ***release / deployment:*** Release / deployment refers to the changes requested during release or deployment of the product.
- ***installation***: Installation refers to the changes requested during installation of the product.
- ***operation***: Operation refers to the changes requested during the operation of the product. This attribute would be used to describe the finding activity of changes requested by users or customers.

Finding Mode

Finding mode is used to identify whether the change was requested in an operational environment or in a non-operational environment. This superclass is essential to defining problem counts that are due to software failures. The choices for this attribute are:

• **dynamic:** Dynamic refers to a change that is requested during operation or execution of the computer program. If the change type is a software defect, the change is a fault by definition and the problem is due to a software failure.

• **static:** Static refers to a requested change that is found in a non-operational environment. Problems or defects found in this environment cannot be due to a failure or fault. Problems or defects found in this

mode would typically be found by formal reviews, software inspections, or other activities that do not involve executing the software.

- **not identified:** Not identified refers to changes where an evaluation has not been made.

Modeling Measures of Software Changes

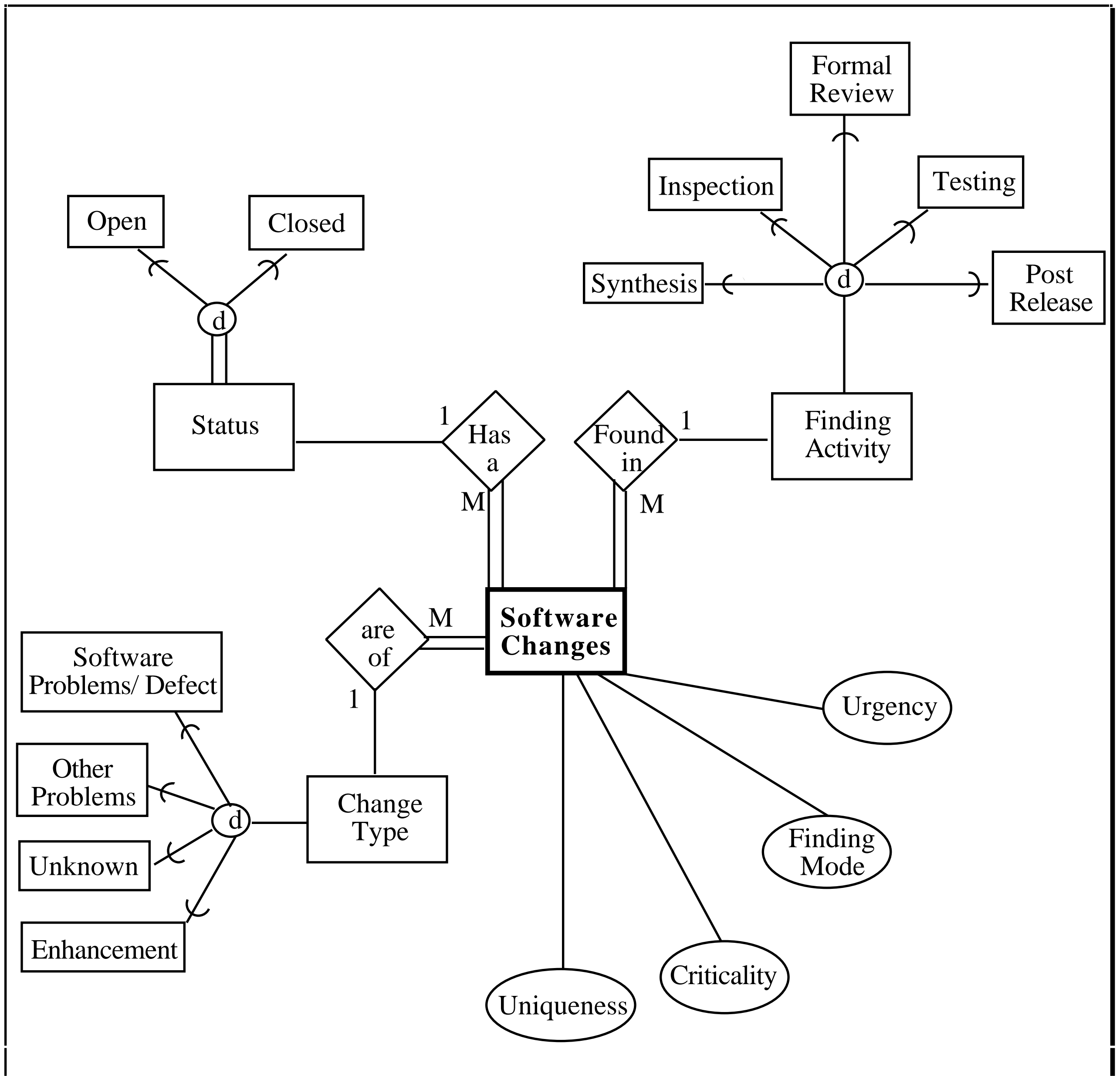
EER Model for Software Changes

The figure below illustrates an EER model for software changes. For simplification of the figure, attributes are not shown for the subclass entities.

Regarding the superclass, *status*, a software change that is requested can be in only one state at any point in time. A specific change is always in at least, and at most, one state; therefore, the cardinality of the relationship *has a* for the entity type, *software change,* should be mandatory and single valued. The cardinality for this portion of the relationship is one-to-one and participation is total. The *status* superclass options could have many changes associated with them. Likewise, a *status* superclass option may not have any changes associated with it, (e.g., there may be no *open* changes) Therefore, the cardinality for this portion of the relationship is one-to-many and participation is partial.

Regarding the superclass, *finding activity*, it should be recognized that changes can be originated as the result of many different activities. To model this, we use the relationship *found in* between the *software change* and *finding activity* entity types. A particular change is defined in this model as being originated by only one activity. This implies that the cardinality for the relationship *found in* is one-to-one. All changes must have a means by which they have come into existence, therefore, the participation constraint is total. The cardinality for the inverse relationship could be zero-to-many. The minimum zero cardinality would mean that a certain finding activity might be associated with the identification of zero changes. Therefore, this portion of the relationship has a cardinality of one-to-many and the participation constraint is partial.

Regarding the change type superclass, a single change should be associated with one and only one change type. In the cases where this does not hold for a requested change, it is very likely that more than one change is actually being requested. In these cases, it is highly recommended that the change request be separated into multiple single requests. Every change must have an associated change type; therefore, the participation constraint is total to go along with a cardinality constraint that is one-to-one. The other side of the relationship follows the same pattern as the above two superclasses and has a cardinality of one-to-many and the participation constraint is partial.


Formal Review
Inspection
Testing
Open
Closed
Synthesis
d
Post Release
d
Status
1
Has a
M
Found in
1
M
Finding Activity
Software Problems/ Defect
are of
M
1
Software Changes
Urgency
Other Problems
d
Change Type
Unknown
Finding Mode
Enhancement
Criticality
Uniqueness

References

[Carleton] Carleton, Anita D.; Park, Robert, E.; Goethert, Wolfhart B.; et. al., *Software Measurement for DoD Systems: Recommendations for Initial Core Measures*, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-92-TR-19, September 1992.

[Christel] Christel, Michael G. and Kyo C. Kang, *Issues in Requirements Elicitation*, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-92-TR-12, September 1992.

[Coleman] Coleman, Derek, et. al., *Object-Oriented Development, The Fusion Method*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[Elmasri] Elmasri, Ramez and Shamkart Navathe, *Fundamentals of Database Systems*, second edition, The Benjamin/Cummings Publishing Company, 1994, Redwood City, CA.

[Fenton] Fenton, Norman and Barbara Kitchenham, "Validating Software Measures," *Journal of Software Testing, Verification and Reliability,* Vol. 1, Issue No. 2.

[Florac] Florac, William, *Software Quality Measurement: A Framework for Counting Problems and Defects,* Software Engineering Institute, Pittsburgh, PA, CMU/SEI-92-TR-22, September 1992.

[Getta] Getta, Janusz R., "Translation of Extended Entity-Relationship Database Model into Object-Oriented Database Model," *Interoperable Database Systems*, Elsevier Science, New York, NY, 1993.

[IEEE_830] *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Standard 830-1993, December 1993.

[Kitchenham] Kitchenham, Barbara; Shari Lawrence Pfleeger; and Norman Fenton, "Towards A Framework for Software Measurement Validation, *IEEE Transactions on Software Engineering*, December 1995.

[Luce] Luce, R.D., et. al., *Foundations of Measurement (Vol. III)*, Academic Press, New York, NY, 1990.

[Mattsson] Kajko-Mattsson, Mira, *An Analysis of the SEI Software Quality Model,* The Software Metrics Laboratory, Royal Institute of Technology and Stockholm University, Kista Sweden, December 1995.

[Mazza] Mazza, C., et. al., *Software Engineering Standards*, Prentice Hall, UK, 1994.

[Michell] Michell, J., "Measurement Scales and Statistics: A Clash of Paradigms," *Psychological Bulletin*, 1986.

[Rozum] Rozum, James A. and William A. Florac, *A DoD Software Measurement Pilot: Applying the SEI Core Measures*, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-94-TR-16, May 1995.

[Schneidewind] Schneidewind, Norman F., "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, May 1992.

Defining Software Measurements: Your Thoughts and Experiences

Spring-Summer 1996

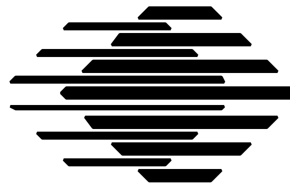
This document contains questions about the materials you just received in the packet entitled *Defining Software Measures: Conference Handout.* Your answers will be used to help assess the quality of the work described there, and identify areas where it may need improvement.

Please read and answer all of the questions. Feel free to write in the margins if you wish to comment on any questions or qualify your answers. Please also feel free to return your marked up copies of the entire conference handout. Your comments will be read and taken into account.

The questionnaire should take about 5 or 10 minutes to complete. Please complete and return it to the SEI as soon as you have had a chance to review the materials in the Conference Handout.

Your answers will be held in the strictest of confidence. Any information that might identify you or your organization will be kept separate from your answers, which will be used in summary form only. Specific answers will not be identified by organization, individual, or in any other manner.

Thank you for your cooperation.



Software Engineering Institute
Carnegie Mellon University

© Copyright 1996, Carnegie Mellon University

I. About the Data Definition Frameworks

- By "organization" we mean an entity within which (possibly many) projects or similar work efforts are organized under common management and policies.
- When thinking about your organization, please answer for the unit where you actually work - not for the larger entity of which it may be a part.

1 First of all, how would you describe the new data definition framework for defining software measurements? *(Please mark one box for each)*

		EXCELLENT	GOOD	FAIR	POOR
1.1	The overall approach...	❐	❐	❐	❐
1.2	The new data definition framework (DDF) checklists	❐	❐	❐	❐
1.3	The enhanced/extended entity relationship (EER) diagrams	❐	❐	❐	❐
1.4	The data dictionaries..	❐	❐	❐	❐
1.5	Its usability for defining and tailoring new software measures..	❐	❐	❐	❐
1.6	The likelihood that it will be used in your organization...	❐	❐	❐	❐

Further comments? *(Please describe)*

2 Is this new approach based on relational database methods... *(Please mark one box for each)*

		ALMOST ALWAYS	FREQUENTLY	OCCASIONALLY	RARELY IF EVER
2.1	Too complex / difficult to understand?	❐	❐	❐	❐
2.2	Appropriately complex -- given the complexity of defining / describing software measures?	❐	❐	❐	❐

Further comments? *(Please describe)*

3 Do you think that it would be a good idea to provide automated tool support for the new data definition framework? *(Please mark one box)*

- ❒ YES
- ❒ NO - AVAILABLE ELSEWHERE
- ❒ NO - NOT NECESSARY

Further comments? *(Please describe)*

4 How would you describe the SEI's previous work on checklist-based frameworks for defining software measures? *(Please mark one box for each)*

		EXCELLENT	GOOD	FAIR	POOR	DON'T KNOW
4.1	The overall approach................................	❒	❒	❒	❒	❒
4.2	The existing measurement check lists (for size, effort, schedule, and quality).........	❒	❒	❒	❒	❒
4.3	The supplemental forms	❒	❒	❒	❒	❒
4.4	Its usability for defining and tailoring new software measures..............................	❒	❒	❒	❒	❒
4.5	The likelihood that it will be (or was) used in your organization..................................	❒	❒	❒	❒	❒

Further comments? *(Please describe)*

5 Are the SEI's previous checklist-based frameworks... *(Please mark one box for each)*

		ALMOST ALWAYS	FREQUENTLY	OCCASIONALLY	RARELY IF EVER	DON'T KNOW
5.1	Too complex / difficult to understand?	❒	❒	❒	❒	❒
5.2	Appropriately complex -- given the complexity of defining / describing software measures?	❒	❒	❒	❒	❒

Further comments? *(Please describe)*

6 How would you describe the framework for defining measures of software requirements (Section C in the Conference Handout)? *(Please mark one box for each)*

		EXCELLENT	GOOD	FAIR	POOR
6.1	The data definition framework (DDF).....................	❒	❒	❒	❒
6.2	The enhanced/extended entity relationship (EER) diagram...	❒	❒	❒	❒
6.3	The data dictionary...	❒	❒	❒	❒
6.4	Its usability for defining and tailoring software measures	❒	❒	❒	❒
6.5	The likelihood that it will be used in your organization..	❒	❒	❒	❒

Further comments? *(Please describe)*

7 Is the requirements framework... *(Please mark one box for each)*

		ALMOST ALWAYS	FREQUENTLY	OCCASIONALLY	RARELY IF EVER
7.1	Too abstract to apply in your organization?	❒	❒	❒	❒
7.2	Too complex / difficult to understand?	❒	❒	❒	❒
7.3	Appropriately complex -- given the complexity of defining / describing software measures?	❒	❒	❒	❒

Further comments? *(Please describe)*

8 How would you describe the framework for defining measures of software changes (Section C in the Conference Handout)? *(Please mark one box for each)*

		EXCELLENT	GOOD	FAIR	POOR
8.1	The data definition framework (DDF)....................	❐	❐	❐	❐
8.2	The enhanced/extended entity relationship (EER) diagram...	❐	❐	❐	❐
8.3	The data dictionary...	❐	❐	❐	❐
8.4	Its usability for defining and tailoring software measures	❐	❐	❐	❐
8.5	The likelihood that it will be used in your organization..	❐	❐	❐	❐

Further comments? *(Please describe)*

9 Is the software changes framework... *(Please mark one box for each)*

		ALMOST ALWAYS	FREQUENTLY	OCCASIONALLY	RARELY IF EVER
9.1	Too abstract to apply in your organization?	❐	❐	❐	❐
9.2	Too complex / difficult to understand?	❐	❐	❐	❐
9.3	Appropriately complex -- given the complexity of defining / describing software measures?	❐	❐	❐	❐

Further comments? *(Please describe)*

10 In general, were the conference handout materials... *(Please mark one box for each)*

		ALMOST ALWAYS	FREQUENTLY	OCCASIONALLY	RARELY IF EVER
10.1	Understandable technically?	❒	❒	❒	❒
10.2	Clearly written?	❒	❒	❒	❒
10.3	Easy to skim and find the main points?	❒	❒	❒	❒
10.4	Sufficiently detailed?	❒	❒	❒	❒
10.5	Too detailed?	❒	❒	❒	❒)

Further comments? *(Please describe)*

11 Should the SEI continue its current work in defining software measures based on relational database methods? *(Please mark as many boxes as apply)*

❒ DEFINITELY

❒ PROBABLY

❒ PROBABLY NOT

❒ DEFINITELY NOT

Further comments? *(Please describe)*

II. About Yourself and Your Organization

12 Which of the following best describes your employment status in the software industry? *(Please mark one box)*

- ❒ EXECUTIVE
- ❒ MANAGEMENT
- ❒ SENIOR TECHNICAL
- ❒ TECHNICAL
- ❒ OTHER *(Please describe briefly)*

13 Approximately how many people are employed in your organization? *(Please mark one box for each)*

		TP TO 50	51 TO 100	101 TO 200	MORE THAN 200
13.1	Total number of employees:	❒	❒	❒	❒
13.2	Number primarily engaged in software development or maintenance:	❒	❒	❒	❒

14 In what major area / sector is your organization located? *(Please mark **one** box)*

- ❒ DEFENSE CONTRACTOR
- ❒ DoD / MiLITARY
- ❒ OTHER GOVERNMENT
- ❒ INDUSTRY / COMMERCIAL
- ❒ ACADEMIC
- ❒ OTHER *(Please describe briefly)*

15 How much expertise exists in your organization about software measurement?
*(Please mark **one** box for each)*

		EXTENSIVE	MODERATE	LIMITED	HARDLY ANY
15.1	In general..	❒	❒	❒	❒
15.2	About relational database methods.........................	❒	❒	❒	❒
15.3	About entity relationship modeling........................	❒	❒	❒	❒
15.4	About the SEI's 1992 checklist-based measurement frameworks......................................	❒	❒	❒	❒

16 To the best of your knowledge, what is the software process maturity level of your organization?
*(Please mark **one** box)*

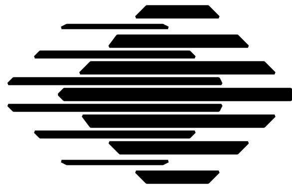
- ❒ CMM LEVEL 1 (initial)
- ❒ APPROACHING LEVEL 2
- ❒ CMM LEVEL 2 (repeatable)
- ❒ APPROACHING LEVEL 3
- ❒ CMM LEVEL 3 (defined)
- ❒ HIGHER
- ❒ DON'T KNOW

17 Is there anything else that you would like to tell us about your organization's needs and the SEI's work in software measurement? *(Please describe)*

Thank you very much for your time and effort!

Defining Software Measurements: Your Thoughts and Experiences

Spring-Summer 1996



Please return this form at your earliest convenience to:

James A. Rozum
Software Engineering Institute
4500 Fifth Avenue, 3rd Floor
Pittsburgh, PA 15213-2691
412/268-7770
jar@sei.cmu.edu
fax: 412/268-5758