

THE USE OF MODELS

We need to define models to help us understand what we are doing, provide a basis for defining goals, and provide a basis for measurement. In what follows we define **model** as an abstraction of a real world process. It attempts to explain what is going on by making assumptions and simplifying the environment. It gives us a viewpoint of the software development process, product or environment by classifying various phenomena, abstracting from reality, and isolating the aspects of interest. There are models of resource use, complexity, reliability, change, defects, etc. We define **metric** as a quantitative measure of extent or degree to which something possesses and exhibits a certain characteristic, quality, property, or attribute.

We need of people, (e.g., customer, manager, developer), processes, and products in order to understand what they are and how they work. Most importantly, we need to study the interactions of these models. These models do not have to be formal. We can have informal models that provide visibility into the items of interest. Let us consider a few such models.

Software Defect Prevention/Isolation/Detection Model

First consider a model of defects focused upon the idea of preventing, isolating and detecting defects. We begin with some definitions from of error, fault and failure [IEEE/Standard]. We will use the term defect to refer to an error, fault or failure.

Errors are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools.

Faults are concrete manifestations of errors within the software. One error may cause several faults and various errors may cause identical faults.

Failures are departures of the operational software system behavior from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure. This provides some insight into the difference between reliability and correctness. A system may be reliable but not correct, i.e. it may contain faults but if we never execute those faults it is reliable. On the other hand, if we define correctness as the conformance of the code to the specification, a system may be correct but not reliable because the user may try to use the system in ways not permitted in the specification and the system can crash.

Now consider the model in Figure 5. We are looking for methods, techniques and tools that will help us prevent errors and faults, isolate faults, and detect faults and failures in the software development process.

The top half of the figure is a model of defects. Errors cause faults which may cause failures. We can prevent errors and faults directly, but only prevent failures by preventing errors and faults or detecting

and isolating faults before they are executed. We can detect faults and failures after they have been put into various documents, and can isolate faults.

The bottom half of the figure is a model of the activities carried out during the software development process. Given a problem, an attempt is made to understand the problem in order to construct solutions. Those solutions are then documented and those documents may be analyzed. Based upon our understanding of the documents, we create lower level solutions, e.g. from specifications we create design, from design we create code, etc. One of those documents, the is executable and provides results from that execution. Those results can also be analyzed. During the entire process there are management activities that go on, dealing with the flow of information, for example.

We combine these two models in order to provide some insights for selecting methods, techniques and tools that will allow us to minimize the number of defects associated with software activities. Beginning on the left hand side of the diagram, we are interested in preventing errors. Where can we prevent errors or misunderstandings from ever entering the system in the first place. We can do that by preventing errors in our understanding of the problem, constructing of solutions, documenting of solutions, and understanding of various documents to create lower level solutions.

Methods, techniques and tools that prevent errors in problem understanding include education and training in the problem domain as well in the requirements of similar projects; simulations of the user's needs; clear, unambiguous, consistency and completeness checking notations for writing the requirements, etc. Approaches to preventing errors in the construction of solutions include education and training in the solution domain; prior experience with solving similar problems; etc. Approaches to preventing errors in documentation and management include the education, training and use of good methods that have been shown to be effective for this problem domain and environment.

Approaches to preventing faults in documenting or recording solutions, include syntax-directed compilers and or document preparation tools that forbid the recording of certain types of misunderstandings. These tools do not prevent all types of faults from entering the system but usually concentrate on a particular type of fault, e.g., a syntax-directed editor focuses on preventing syntactic faults and some semantic faults that can be detected by a compiler.

We can isolate and detect faults during the analysis of documents by the education, training, and use of good reading techniques and methods that have been shown to be effective for this problems domain and environment, as discussed in the last section. We can detect failures by good testing techniques and methods, tailored to the types of errors, faults and failures, expected in this environment.

Based upon the interaction of these two models, we can define various software defect classification schemes that allow us to better understand the kinds of defects we make so we can choose the appropriate techniques, methods and tools for our environment. For example, error classification schemes include:

Error origin - the phase or activity in which the misunderstanding took place. Example subclasses include: requirements, specification, design, code, unit test, system test, acceptance test, maintenance

Error domain - the object or domain of the misunderstanding. Example subclasses include: application, problem-solution, notation <semantics, syntax>, environment, information management, clerical where application is a misunderstanding of the problem or application domain, problem-solution is a misunderstanding of the solution space, semantics and syntax are misunderstandings of the semantics of syntax of the notation used to express the problem or solution, environment is a misunderstanding of the compiler, operating system or external interface, information management is a misunderstanding due to communication, and clerical is a mistyping or misspelling

Document error type - the form of misunderstanding contained in a document. Example subclasses include: ambiguity, omission, inconsistency, incorrect fact, wrong section

Change effect - the number of modules changed to fix an error

Fault classification schemes include:

Fault detection time - the phase or activity in which the fault was detected. Example subclasses include requirements, specification, design, code, unit test, system test, acceptance test, maintenance

Fault Density - number of faults per KLOC

Effort to Isolate/Fix - the time taken to isolate or fix a fault usually in intervals of time, e.g., 1 hour or less, 1 hour to 1 day, 1 day to 3 days, more than 3 days

Omission/commission - where omission is neglecting to include some entity and commission is the inclusion of some incorrect executable statement or fact

-
Algorithmic fault - the problem with the algorithm, e.g., control flow, interface, data <definition, initialization, use>, where control flow is an incorrect path taken, interface is a fault associated with structures outside the modules environment, definition is the incorrect definition of a variable or data type, initialization is a failure to initialize data on entry/exit, and use is the incorrect use of a data structure, or an erroneous evaluation of a variable's value

Failure classification schemes include:

Failure detection time - the phase or activity in which the failure was detected. Example, subclasses include: unit test, system test, acceptance test, operation

System Severity - the level of effect the failure has on the system. Example subclasses include: operation stops completely, operation is significantly impacted, prevents full use of features but can be compensated, minor or cosmetic

Customer Impact - the level of effect the failure has on the customer. Example subclasses re usually similar to the subclasses for system severity but filled out from the customer perspective so the same failures may be categorized differently because of subjective implications and customer satisfaction issues

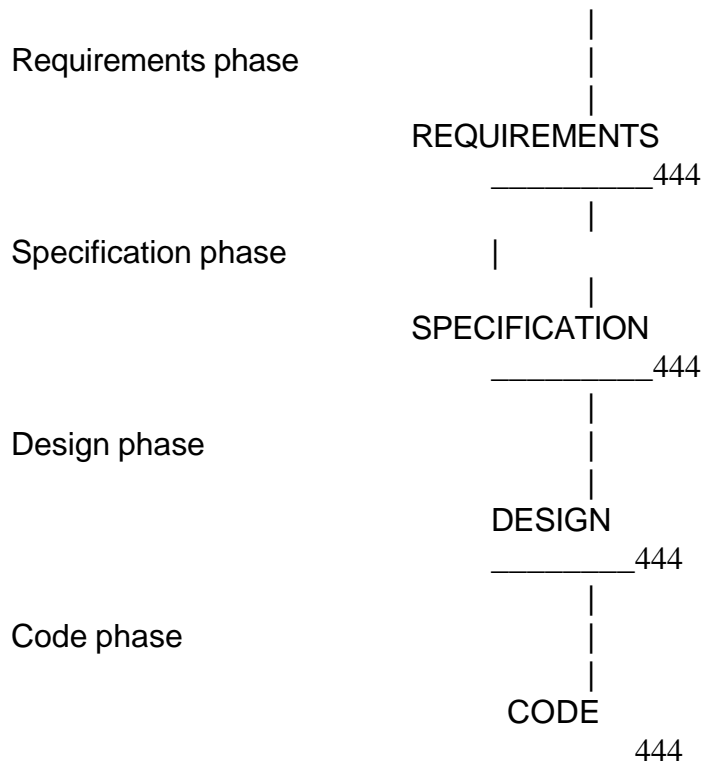
Although each of these classes may be interesting in their own right, we are often interested in the combination of classes of errors. For example, the relationship between system severity failures and faults of omission/commission. Are the highest severity faults caused by leaving something out or putting in the wrong thing? What is the average effort to isolate/fix a fault of omission or a control flow fault or a requirements error or an application error.

This model not only provides insights into how we can make and prevent defects in the software development process, it also provides a basis for defining measures and analyzing those measures.

Producer/Consumer Process Model

As another example, let us pick a model of the software process which we will call the producer/consumer model. For each document or intermediate product we must specify the consumers, their operational profiles, and their goals or needs for that product. Then based upon this model, each such document should be evaluated from the perspective of all it consumers or customers.

SOFTWARE LIFE CYCLE PROCESS



System analyst	4	4	4	
Designer		4	4	4
Coder			4	
System tester	4	4	4	
Maintainer		4	4	
Quality				4

Figure 7. Example Reviewers for Documents

Although this model does not directly generate metrics, it does define measurement points, measurement perspectives, and insights into the quality of the intermediate products. In the next section we will provide a basis for measuring these perspectives.

Productivity Factors Model

As one last model, consider a model of productivity. In most business environments, productivity is expressed as a ratio of values to costs, where value is either an increase in revenues attributable to the product or a cost reduction resulting from the use of the product. You will notice that this definition focuses attention on management decisions rather than programmer output.

Consider figure 8, in which the factors associated with productivity are related. Values is influenced by customer or corporate needs and the product being produced. These are hard values to quantify. We may chose to spend too much for a product because we wish to be first on the market with a particular product or please a particular customer in anticipation of further business. These decisions effect more than productivity, they may also effect quality, i.e. the company may be willing to release a less than high quality product to get into the marketplace first.

Cost on the other hand is influenced by capital investment and work productivity. Work productivity is the amount of function produced per unit of cost. It is influenced by the product, process and environment factors. For example, the amount of function we can produce depends upon such product factors as the newness to the state of the art of the application, the amount of reuse available, the complexity of the function,, etc.; such process factors as the quality of the development and maintenance processes and how well they are tailored to the application; environment factors include the type and availability of the development and target machines, the software development environment, the operating system support, etc. Capital investment in equipment, tools, and office environment also affect cost.

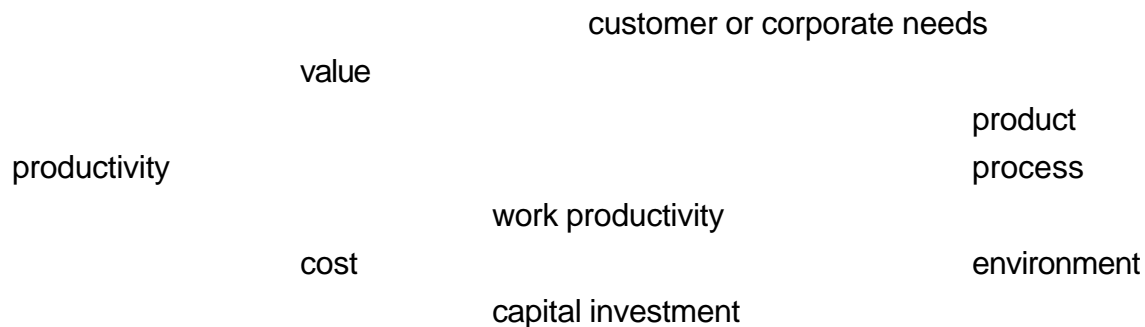


Figure 8. Factors in a Productivity Model

In many software environments, productivity is defined in terms of lines of code per unit of effort. In the context of this model, that definition is at best a definition of work productivity and at that is only a very weak approximation. For example how good a measure of function is lines of code? Is the product, process, and environment taken into account to differentiate the possible levels of productivity for different contexts?

Although this model is the least insightful with regard to metrics, it is very useful in terms of understanding whether we are measuring the right thing and does provide some insights into the fact that we cannot use lines of code per unit of effort as a measure across the industry for comparison because of the influence of the different product, process, and environment factors. In some sense it shows that lines of code per unit of effort is a complexity measure; it tells us how expensive it is to produce lines of code for a particular project given the context and individuals involved.

We have seen three different models, each offering some understanding of the software process and product and the nature of the business. Hopefully they have demonstrated the importance of modeling. Whenever we build a model we should ask certain specific questions: Is the model correct in principle? Does the model actually describe what we are doing? How can we improve the model based on theory, practice and analysis? How do we feed back what we have learned to improve the model or our adherence to it?

In order to improve the software development process and product, we need to better understand. This implies the building of models. We want to build descriptive models to explain what is happening. This allows us to better understand the nature of the business locally. Using these models we can analyze and experiment with improvements in technology. Then we want to define prescriptive models to motivate that improvement.

