UC Irvine
ICS Technical Reports

Title
Towards a theory of software engineering

Permalink
https://escholarship.org/uc/item/8cs973zv

Authors
Freeman, Peter
Staa, Arndt von

Publication Date
1984

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
NO. 242

*NOTE: This paper has been submitted for journal publication; any reference to it should so indicate.*

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

TR#242

# TOWARDS A THEORY OF SOFTWARE ENGINEERING*

Peter Freeman
University of California
Irvine, CA, USA

Arndt von Staa
Pontificia Universidade Catolica
Rio de Janeiro, Brasil

November, 1984

* Support for this research has been provided by IBM Brasil, National Science Foundation (US) grant MCS-83-04439, CNPq (Brasil) grant 40.16336/84, and FINEP (Brasil) grant 62.84.0416.00.
©Copyright 1984 by P. Freeman and A. von Staa

## ABSTRACT

A theory of software engineering (SE) is presented and its application to explaining and analysing SE situations is illustrated. The theory is based on a characterization of SE representations and the fundamental activities that are applied to them. Motivations for developing a theory and means of establishing its validity are also discussed.

## MOTIVATION

We are researchers interested in understanding phenomena in the area of software engineering — that human activity involved with the development of software in a systematic, engineering-oriented way. We are also people interested in the pragmatic aspects of software development — that is, in seeing that software engineering is effective in reaching the goals set for it. The ideas and analysis presented in this paper are intended to contribute to scientific understanding and engineering practice by providing a theoretical basis for both.

A precise definition of software engineering (SE) is not necessary (in general), but it may help if the reader understands the context in which we are working. To us, software engineering is:

*The systematic application of methods, tools, and knowledge to achieve stated technical, economic, and human objectives for a software-intensive system.*

We would emphasize that, in our view, SE applies (in some form) to *any* software development, not just large, complex, technically sophisticated systems.

Theories in science, and in human affairs more generally, serve a valuable role of encapsulating knowledge about some topic in compact form. Further, they often permit an economy of action and thought, and provide an ability to predict (in limited ways) the future of events in the realm dealt with by the theory.

We see several specific reasons to support the need for a theory of software engineering, including:

Methods and tools for software engineering abound and one of the pressing questions of the day is "How do they compare to each other?". A theory about the basic processes of software engineering will provide us a basis for evaluating some of the artifacts of the software engineering world with which we come into contact. If there is no underlying

theory or conceptualization of software engineering, then any such comparisons are limited to shallow, feature comparisons. The existence of a theory, however, will permit a deeper analysis of methods and tools that can begin to evaluate these artifacts not in terms of comparison to their neighbors, but rather, in comparison to what the theory tells us is important in SE.

A second, very important reason for having a theory is to provide rational guidance for research efforts. A good theory will indicate areas where our knowledge is insufficient, thus leading us in useful research directions. It will also indicate areas where pragmatic advantage can be gained based on the theory, thus contributing to the practical aspects of the field. We feel that all too often current R&D work is oriented just toward examples, rather than towards basic issues.

A third consideration is the importance of theories in teaching and, more generally, explanation. Computer science, for the most part, progressed beyond being simply a descriptive science some years ago. We no longer just teach specific algorithms, languages, or machine architectures and believe that we are teaching computer science. Instead, we try to present our students with characterizations of classes of algorithms, general principles of languages, and fundamentals of machine organization.

We would submit that in SE we do not have many theories that permit us to teach the subject in a compact and general manner. Likewise, we are often at a loss to explain SE conceptually to those not directly familiar with it (*e.g.* high-level managers). Thus, one of our prime motivations is to provide this kind of basis for the transmittal of software engineering knowledge.

These are the main motivators (for us) for developing a more theoretical understanding of SE. Before presenting the core of a theory, however, we will quite briefly outline it and discuss the all-important topic of theory validation.

## AN OVERVIEW OF THE THEORY

Figure 1, taken from Kerola and Freeman [1981], partitions the elements of the development world into three classes. Our theory concerns portions of Class II — the people who do development — and Class III of that figure — the tools of development (used in a broad sense that includes concepts as well as physical artifacts). This subset of Classes II & III is shown in Figure 2. It contains two items of interest to us here: representations and activities.

Figure 1: Elements of Concern in Development

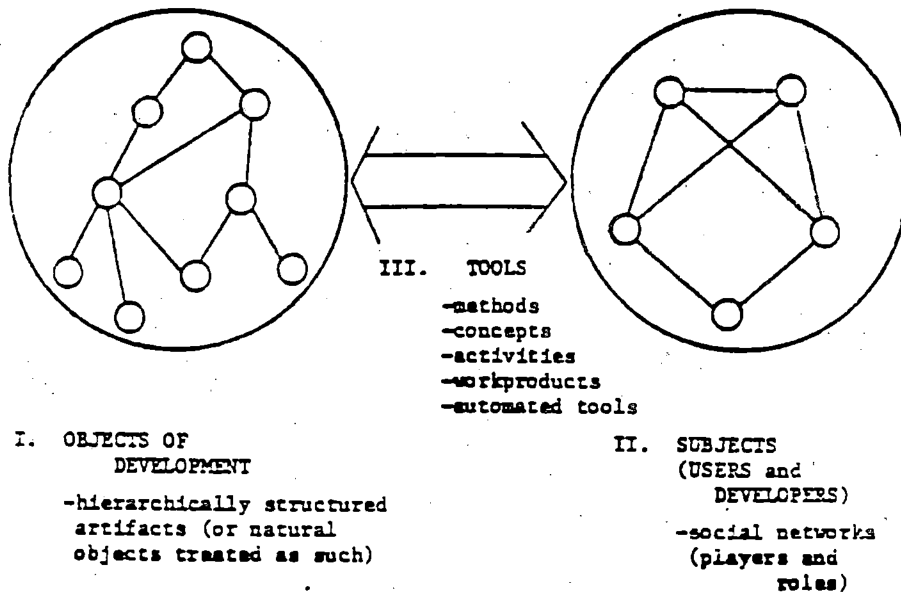
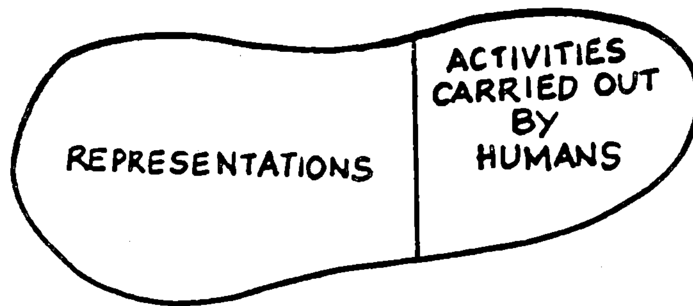


Figure 2: Elements of the Theory



Our underlying purpose here is to make the software development process sensible through the use of a small number of underlying concepts. In that context, we believe that careful explanation of these two elements will provide the basis for understanding most of what goes on in software engineering.

Activities and representations that are largerly managerial in nature (size prediction, project planning, budgets, manpower allocations, negotiation of acceptance standards, and so on) are not covered in our theory. They are essential and their integration with the technical activities is key.* However they are outside the scope of our concern here.

The first element of the theory concerns the representations used in SE. A cornerstone of the theory is that software engineering can be viewed entirely as a process of creating and manipulating representations. This view is central to understanding the theory.

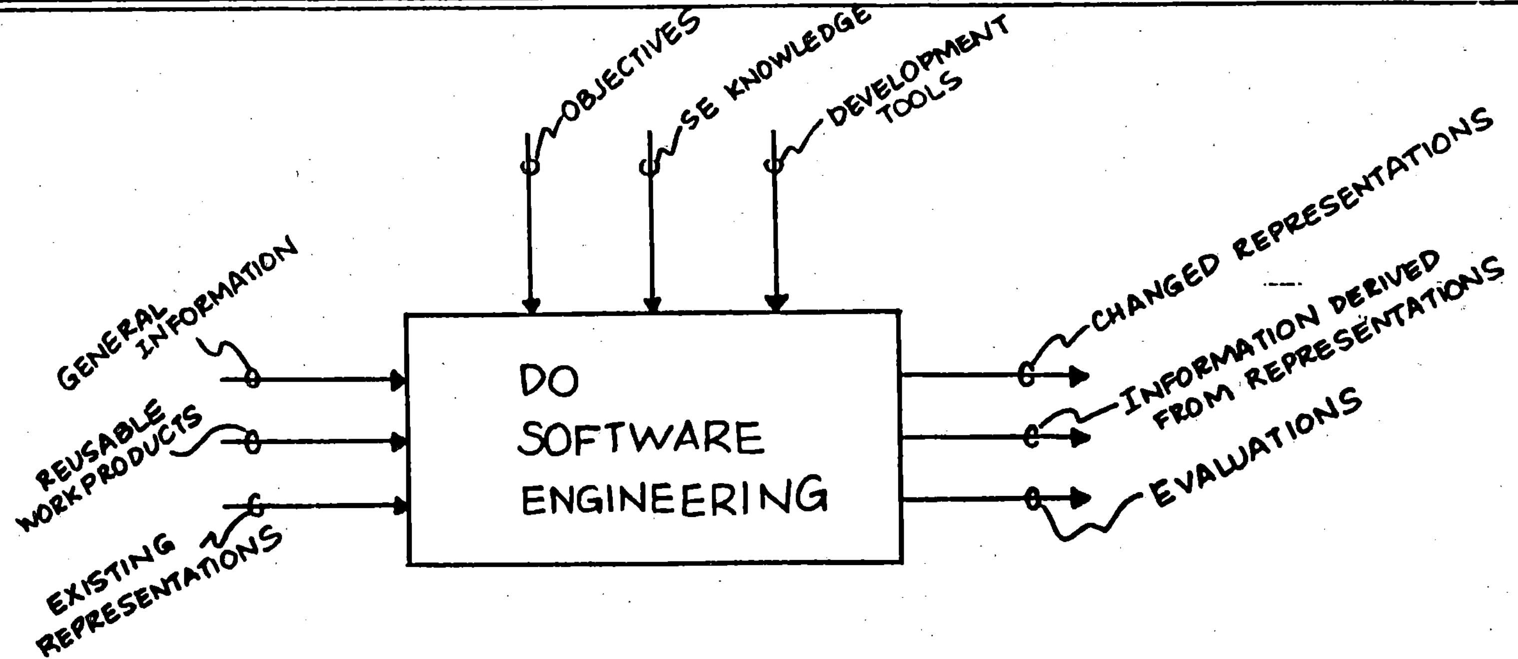
The second element of the theory, then, is composed of the fundamental intellectual activities that people carry out in dealing with software engineering representations. (We do not exclude automated versions of these activities.) The analysis criterion used to identify the activities is that they all deal with representations or the process of creating and manipulating them in some way.

* One of us (PF) is currently exploring the detailed relationship between the technical and managerial aspects of software engineering.

SADT™ DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

| USED AT: | AUTHOR: P. FREEMAN<br>PROJECT: TIJUCA<br>NOTES: 1 2 3 4 5 6 7 8 9 10 | DATE: 3 OCT 84<br>REV: | | WORKING | READER | DATE | CONTEXT: |
|---|---|---|---|---|---|---|---|
| | | | | DRAFT | | | — |
| | | | | RECOMMENDED | | | |
| | | | ■ | PUBLICATION | | | |




PURPOSE: Model the intellectual activities utilized by a person doing software engineering.

VIEWPOINT: Researcher viewing a broad range of SE phenomena.

| NODE: A-Ø | TITLE: DO SOFTWARE ENGINEERING (CONTEXT) | NUMBER: PF11 |
|---|---|---|

Figure 3: Do SE (Context)

Using SADT notation, Figure 3 ties together these elements and relates them in the following way. We view SE as a process of transforming objectives into systems under the constraint of development tools (*e.g.* representations) and SE knowledge (information about development objects - class I in Figure 1). It is important to remember that Figure 3 *could* represent *any* software engineering activity from a single design change to an entire development lifecycle.

We intend that this theory be fundamental - that is, implied by the nature of the task. It is not dependent on any particular methodology (*e.g.* structured development [Yourdon, 1979] or Jackson design [Jackson, 1983]); if it bears similarity to particular approaches, then we take that as evidence of its power to explain different techniques.

We are well aware that this is not the first time someone has attempted to theorize about systems development. Boehm [1976] and Mills [1980] both present some general principles of SE; Parnas [1972,1976] has long dealt with principles of design and specification; text book writers such as Jensen and Tonies [1978], Pressman [1982], and Fairley [1985] try to present general explanations; every methodology developer is inherently dealing with a "theory" in a specific area (see [Freeman and Wasserman, 1983] for a selection).

Without giving a precise review of other attempts or comparison to the ideas in this paper (that is better left to others), we would observe that most of the above-cited instances are attempts to abstract or generalize. The theory presented here, on the other hand, cuts below the surface and addresses issues of underlying processes.

In that sense, there are a smaller number of similar works. One of us has previously theorized about design process and representation [Freeman and Newell, 1971, Freeman, 1978, 1979, 1983], forming a clear base for some of the theory. While not explicitly utilized here, the paper by Ross, Goodenough, and Irvine [1975] has goals close to our own and must be counted as an intellectual ancestor. Similarly, while rather different in realization, the work of Kerola and his collegues [1979] is quite similar in motivation to our own. Their

dedication to providing an intellectual foundation for systems development has certainly increased ours.

Many others have contributed to our intellectual ancestry and a few others have attempted some rigorous theorizing. A thorough review should properly come later, however, since our theory is not directly based on the work of others.

Before presenting it, we must address the question of how one validates such a theory. That is the subject of the next section.

## THEORY VALIDATION

It is important to keep in view the distinction between theory and reality. A theory is at best a "...coherent set of general propositions used as principles of explanation for a class of phenomena"* and at worst may be a conjecture that is totally false.

The purpose of validation is to provide objective information concerning the degree of correspondence between a theory and the reality it proposes to explain. Validation of a theory is always important if the theory is to be used in any meangingful way. We believe that in the case of SE theory, validation is especially important because of the newness of the field. Theories that are largely wrong, not just ineffectual, could do serious damage to our ability to develop software if they are followed.

An indirect way of validating the correctness of a theory is to determine whether or not it can answer questions about the pheonomena of interest. A theory of SE should permit us to answer a question such as "Does improvement in the ability of developers to perform certain activities greatly improve the overall quality of software?" If we can determine the accuracy of answers to such questions, then we can have some indirect information

---

* *The Random House Dictionary of the English Language*, Random House, New York, 1968.

concerning the validity of the theory. Later in this paper we present an initial validation of this sort.

The classical way of validating a theory is to take observations or perform experiments and test whether the data taken agrees with what the theory predicts. The difficulty of running experiments in SE is well-known [Basili, 1980] and probably is not an appropriate way to test the theories proposed here. Observations in which we collect data about some naturally occuring event rather than setting up a controlled situation offer more hope.

In general, theories of this type must be validated largely through comparison to observations of actual software development. Can the activities listed here be identified in the behavior observed? Are there other activities? Do software representations have characteristics besides the fundamental ones we propose? Are there overall processes being used in practice that are not captured by the theory?

Having dealt with the preliminaries, we are now ready to present the theory.

## REPRESENTATION

In software engineering, representations are everything. The artifacts that we build — systems — exist in physical form only as states of various electronic or magnetic devices. Thus, the representations we build are all we really have of our systems.

In this section we present the first element of our theory, a set of nine statements that characterize SE representations:

- the difference between information, representational forms, and representations;
- insufficiency of a single representation;
- quality measures of representational forms (capacity, communicability, and verifiability);

- levels of detail;
- scope;
- hierarchy;
- viewpoint;
- quality measures of representations (syntactic correctness, completeness, consistency, and validity);
- transformability of representations.

In the following, we will expand on these characterizations of SE representations.

1. *There are three aspects: information, representations, and representational forms.*

*Information* is what goes into a representation and is the knowledge (facts, etc.) about a system that we wish to communicate to others (including ourselves at a later time). *Representations* contain information about a particular system and can be considered models of that system. These are instances of *representational forms* (or, rep forms for short) which are models of representations; that is, they specify the format of information for a particular type of representation. We usually think of representational forms as being simple (*e.g.* structure charts) or compound (*e.g.* a document outline that specifies the use of specific forms in each section). In the following we sometimes do not differentiate between representations (instances) and representational forms (types) when the distinction is unimportant; if we make statements about the instances, they are also assumed true of the types.

As an example, when designing an operating system, we may have a list of commands the system must support, interface constraints to other systems, and a description to the interrupt structure of the underlying machine; this is *information* relevant to this particular situation. As a *representational form* for the design specification we may choose Petri nets

and finite-state machines with a simple pseudo-code for actions, organized in a particular document format. A *representation* results when we place this particular set of information into this representational form.

2. *No single representation or form (except a simple concatenation) will suffice to hold effectively all the information relevant to a particular system.*

There are many aspects of a system to understand (performance, static structure, data representations, and so on) and this leads to many different representational needs. A corollary to this is that a structured collection of representations is normally needed to describe a particular system in a coherent fashion.

The word "effectively" in statement 2 leads to:

3. *There are three inherent measures of quality of SE representational forms: capacity communicability, and verifiability.*

*Capacity* addresses the issue of whether a particular form can hold specified information. For example, if we must represent the interconnection of modules and the conditions under which they are called, then traditional structure charts will have insufficient capacity (since they do not show calling conditions).

*Communicability,* on the other hand, refers to how effective a given form is for communicating information to the intended reader. For example, the communicability of structure charts, for showing interconnections, is generally considered to be greater than a linear language (such as a pseudo-code).

*Verifiability* is a property of a form that measures how easily any property of a representation constructed from the form may be checked for some property. For example, natural language is much harder to check for consistency than is a formal language.

We are painfully aware of the absence of any real measures of representational quality (in any dimension). That does not detract, however, from this being a fundamental property of SE representational forms.

4. *Representations exist at different levels of abstraction.*

This should be obvious, given that we use representations to model something (the system) that exists at various levels. Normally we have, co-existing multiple representations of a system that are at different levels of abstraction. Some of the variation is for effectiveness of communication (a functional specification being presented to the customer for approval should not have complete design detail) and some is due to the chronological order of creation of the representation. Although there is no general way at this point to measure level, it is clear that there are some intuitive ways in which good developers choose a level of representation to best suit a particular task.

5. *A representation has scope.*

Often there will be information about a system, at the level of abstraction in a given representation, which is not included in it. For example, the description of the operating instructions for a system might be at the same level as the description of the screen formats the system produces; but, this information might not be included in a particular representation because it is outside the scope.

Scope is not the same as completeness. Scope is a property that establishes a boundary — like the border of a road map. Completeness refers to whether all appropriate information that is within the defined scope of a representation is, in fact, included — corresponding on a map to whether all roads at the defined level of detail and within the border are actually shown.

In practice, however, incorrectly defined scope and incompleteness often lead to the same result: needed information missing from a representation.

6. *Representations form a natural hierarchy.*

A piece of information in one representation may be related to another representation in a parent-child relation, in which the information in the parent is expanded in the child. For example, the information at one step of a step-wise refinement is expanded in later representations. This might be termed a ***natural or semantic hierarchy.***

There may also be a "defined" hierarchy in which a representational form is compound, being made up of several other forms. For example, a requirements definition document (a representational form) might include a section utilizing data-flow diagrams and another utilizing decision tables. There would be a hierarchy of forms in this case, defined for convenience rather than resulting from any inherent property of the representation.

A more extreme example of defined hierarchy involves PDL and structure charts. In some cases, an overall representational form utilizes PDL at the highest level (in order to show control) and structure charts to show the structure of subsystems; other forms reverse this. utilizing PDL within structure charts to show control within individual modules.

7. *A representation has a viewpoint.*

The information in a representation can be related, scoped, and detailed in a way that reflects a particular point of view. For example, a design may leave out information about the operational aspects of the system because it is not presented from the viewpoint of the system operator, whereas a maintenance manual will include it.

Vantage point is not often taken into account in preparing representations (see Ross and Schoman [1977] for an exception), but can be extremely important. It provides a way of improving the consistency of a representation, thus improving its communicability.

8. ***Representations have four basic quality properties in addition to the quality properties of forms: syntactic correctness, completeness, consistency, and validity.***

*Syntactic correctness* means that a representation matches its associated rep form. A representation is a model of something and is considered ***complete*** if it contains all the information necessary to model the external reality at the desired scope and level of abstraction. We say the information is ***consistent*** if all models of any given external item have the same interpretation. A representation is ***valid*** if the model it forms does not permit incorrect conclusions to be drawn about the external reality (within the limits of scope and level of abstraction). Validity corresponds most closely to semantic correctness although completeness and consistency are involved also.

9. *Representations may be changed into other representations.*

The degree of syntatic correspondence between the two may vary from very high (in the case when we simply correct a syntax error) to very low (in the case where we produce a system structure chart from a natural language description of the requirements for the system). The semantic correspondence may also vary in a similar fashion.

A deeper and more extensive* theory of SE representation could be developed. The characterizations presented here, however, capture the essence of the situation at a level consistent with the other parts of the theory.

## ACTIVITIES OF SOFTWARE DEVELOPMENT†

The last point in the previous section is the key to understanding this theory. In its simplest, most basic form, the theory comes down to:

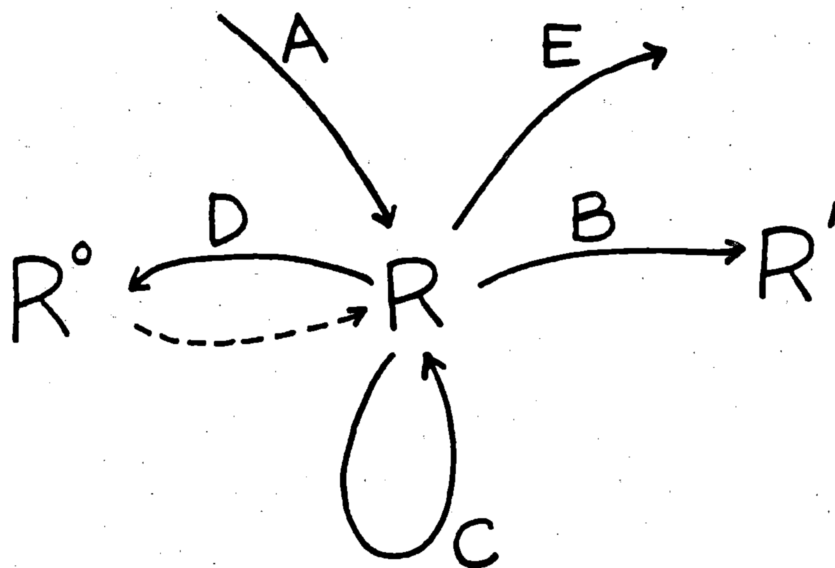
> ***Software engineering consists of the creation and manipulation of (system) representations.***

* A pragmatic aspect is the set of requirements for SE representation forms (languages) that is implied by our theory. We are actively exploring this in another paper [Von Staa and Freeman, 1984].

† A rudimentary version of some of the material in this section can be found in [Freeman, 1983] and its earlier editions.

Given this underlying philosophy, we derive a set of basic SE activities as follows.

Figure 4: Operations on a Representation



Consider a SE representation, R, as shown in Figure 4. There are several things that can be done to it or that involve it:

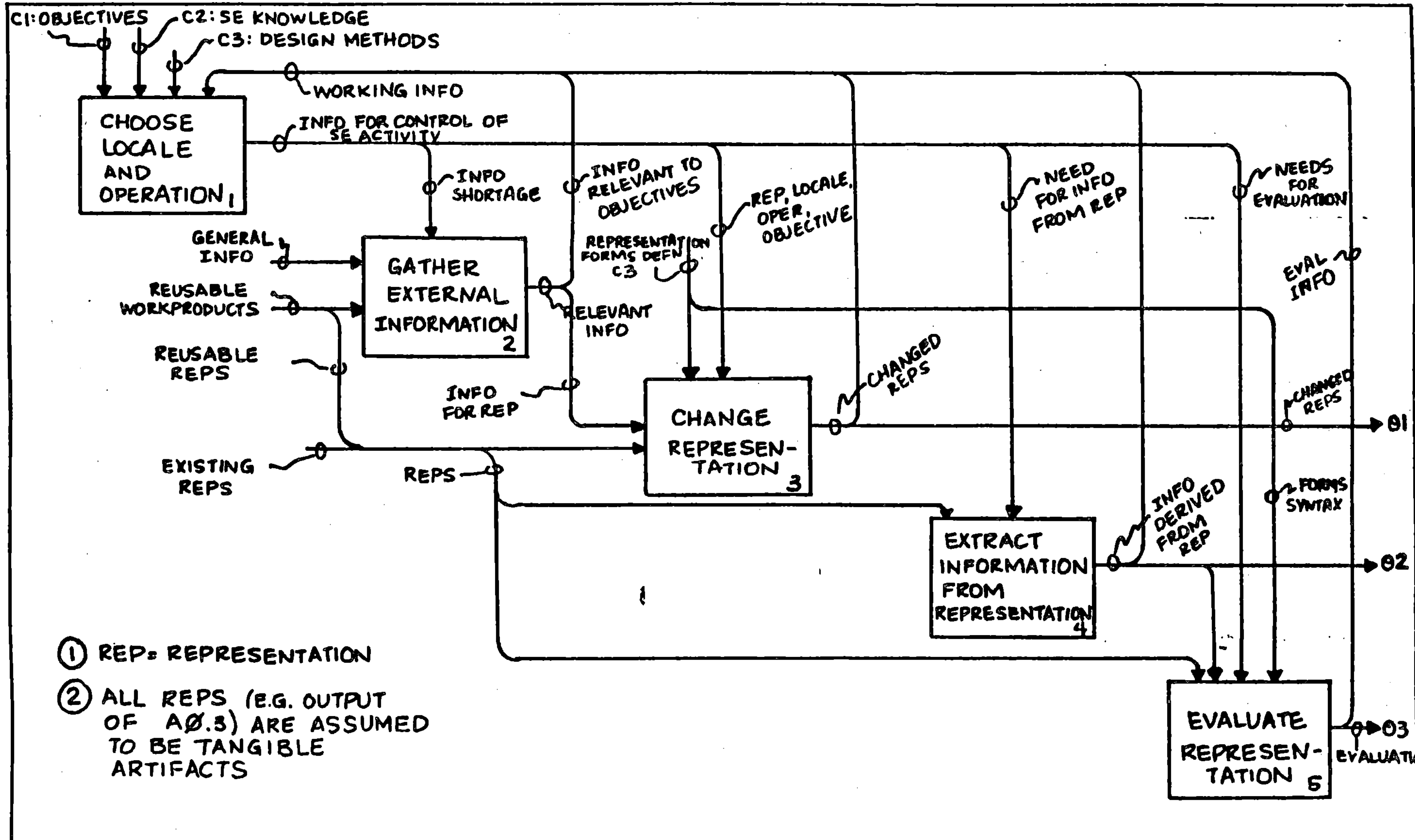
1. Create it *ab initio* (A);
2. Change it into another SE representation R′ (B);
3. Modify it (C);
4. Derive a representation $R^o$ which could have been used in obtaining it initially (D);
5. Derive some information from it (and, perhaps, other representations) (E);
6. Gather information from sources outside the representation to be put into it;
7. Decide what to do to the representation next.

The first four of these operations can be abstracted to the single operation of "Change Representation." Doing so leads to Figure 5 that succinctly models the activities of SE as described in the theory.

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

| USED AT: | AUTHOR: P. FREEMAN<br>PROJECT: TIJUCA<br>NOTES: 1 2 3 4 5 6 7 8 9 10 | DATE: 3 OCT 84<br>REV: | WORKING<br>DRAFT<br>RECOMMENDED<br>PUBLICATION | READER DATE | CONTEXT: |
|---|---|---|---|---|---|




| NODE: AØ | TITLE: DO SOFTWARE ENGINEERING | NUMBER: PF10 |
|---|---|---|

Figure 5: Software Engineering Activities (Classes)

These five basic activities can be broken down as follows, yielding a set of activities that are sufficient to decribe any (technical) SE activity:

- Choose Locale and Operation
  - search
  - scout
  - check
  - decide
- Gather External Information
  - read
  - observe
  - listen
  - experiment
  - obtain feedback
  - filter
- Change Representation
  - partition
  - elaborate
  - abstract
  - translate
  - delete
  - modify
  - compose
- Extract Information from Representation
  - detect changes
  - derive
  - identify corruption
  - measure
  - predict
- Evaluate Representation
  - verify
  - validate
  - apply quality control
  - approve

A careful study of Figure 5 will reveal that our basic model adopts the information processing view of human problem-solving [Newell and Simon, 1972]. Figure 5 is essentially a model of the (relevant portions of the) mind of a software engineer; bear in mind that we are trying to explain the *intellectual* processes of SE.

In this context, then, our model is organized in an overall (SE) control part (box A0.1) and four functional elements (boxes A0.2–A0.5) that act on or help us act on representations. (We do not claim, necessarily, that this is precisely how the brain works; we are only interested here in the observable behavior of a software engineer).

The class of control activities, "Choose Locale and Operation," are concerned with what to do next. They are defined in terms of finding a (part of a) representation to work on, the change to be effected in it, and the operation to carry out (including getting more information).

"Gather External Information" is a straight-forward class of activities. They are involved in obtaining information from outside the immediate focus of attention that is needed to change or evaluate representations.

"Change Representation" activities are the core of the SE process. They are primarily concerned with effecting specific changes in a representation (including creating one from scratch). Although the activities are mental, the changes must always be made in an explicit, tangible representation. Thus the activities of "writing" or otherwise recording changes to a representation are a very important subpart of "Change Representation."

The "Extract Information" class is composed of those operations that derive some information from an SE representation, but do not change it.

"Evaluate Representation" activities also extract information from a representation, but in a specialized way. They are concerned with evaluating the quality of a representation against various expectations.

The remainder of this section will briefly discuss each of the elemental activities. One is tempted to define strict levels of cognitive functions in modeling human activity in order to make the theory more orderly and regular. The fact that our theory includes rather low-level functions such as "decide"* and high-level functions such as "experiment" gives the impression of a shot-gun approach. However, considering our objectives, that is not the case. Our focus here is on explaining observable SE activity, not on building a complete cognitive theory.

* Notationally, we will try to always quote the actual operator or its derivative.

### Choose Locale and Operation

*Search.*

*Look for a representation having a given set of characteristics.*

The action of searching for something is, of course, a fundamental mental activity that is used constantly. In the context of SE, searching means to look in one's mind for a representation that could meet our current goals.† A common example is when we are producing a structure chart; we imagine several alternative structures (in general) before writing one down.

*Scout.*

*Explore possible alternatives without fully completing the corresponding alternatives.*

In development, as in other endeavors, we often employ a form of look-ahead. Scouting is an activity in which we explore the ramifications of possible decisions before actually making them.

Typically, when engaged in a scouting activity, the developer does not produce the complete representation implied by the decision being explored. Rather, only enough of it is produced to collect the required information. (For example, in scouting the addition of a new module for a function versus simply modifying an existing one, we probably would not produce a completely changed structure chart but only sketch the proposed changes.)

There is a close relationship between searching and scouting. The essential differences are in the size of the operation and result. Searching is wholly a mental activity (no visible

† "Goals" and "objectives" are often used synonymously; we use "goals" to denote internal mental states describing a new state of the external world to be achieved and "objectives" to denote externally stated targets for an SE task.

output) and typically small in scope (*e.g.* a part of a structure chart) while scouting connotes producing tangible and larger results.

*Check.*

*Determine if a given representation and set of goals (for the representation) match.*

This is a low-level "comparison" operator that permits us to determine if we have achieved the representation we are seeking. There are much higher-level comparisons that are very important in SE (*e.g.* acceptance testing) which are based on this elemental operation.

An example of checking can be seen when we have in mind a certain sequence of actions, produce a flow chart, and then "check" whether it properly represents the actions. There are, of course, much lower-level comparison operators (at the physiological level, for example) on which checking is based.

*Decide.*

*Choose among alternatives.*

Decision-making is an activity found throughout SE at all levels. We decide what requirements are important and should be put into a requirements definition and which should be left out; we decide which form of an algorithm to use in a program; we decide if a set of tests is adequate to provide a desired level of confidence. These observable forms of decision-making are based on "decide" but also include many other operations.

The "decide" operator at a fundamental level is concerned with choosing alternatives relating to other basic operators. Goals for representations must be chosen; information to include in a representation must be chosen from what has been obtained; sequences of operations necessary to achieve a development objective must be formulated; and so on.

Clearly, there is much fine structure of "decide." Careful study of how we make development decisions at the level of changes to a representation will provide us with a scientific basis for understanding more complex decision-making in this arena. (See [Levin, 1976] for an early start in this direction.)

**Gather External Information**

*Read, observe, listen.*

These actions are the common ones of human perception.

*Experiment.*

> *Produce a representation or situation from which needed information can be obtained.*

Prototyping is the archetypical experimental operation in SE. We build prototypes in order to obtain information useful to later stages of development (*e.g.* to learn if proposed screen formats are easy for users to deal with). Building a simulation model of a proposed design to determine critical operational characteristics is another form of experimenting. If we create a scenario depicting typical use of a system and present it to the customer, we have created a situation from which we can obtain needed information (*e.g.* on possible omissions from the proposed system).

*Obtain feedback.*

> *Communicate an understanding of some set of information to others for the purpose of checking the accuracy of the understanding.*

The last example of "experiment" could also be an example of "obtain feedback." Based on our understanding of the requirements for a system, we might produce a scenario for the customer in order to check the accuracy of our understanding of them. During

information gathering, there is usually frequent "obtain feedback" actions to check accuracy; for example, a primary function of the SADT review cycle is to check the accuracy of the information in the model.

Review of SE workproducts is often done to obtain feedback. That activity is covered below in a more detailed manner. The "obtain feedback" operation has the connotation of creating a new representation of information (*e.g.* spoken words) in one's head for the purpose of communicating to someone else.

*Filter.*

> *Remove from a set of information any that is not needed for a given set of goals.*

Information is gathered in the context of some set of goals for building system reps (*e.g.* to produce a functional specification or finding an appropriate algorithm). As one gathers information (*e.g.* by talking with potential users or reading a description of various algorithms), the information not relevant to those goals is removed (or, at least, ignored). We would note that the ability of one to filter is a key determiner of success in many development situations.

## Change Representation

*Partition.*

> ***Segment a representation into pieces, each containing a part of the information in the original.***

The modules of a system (representation) form a partitioning of the system (although the dependencies between modules have an aspect of hierarchy that suggest more than a

simple partition). Segmentation of a large system into overlays that can fit into a given memory is, perhaps, a clearer example of partitioning.

We also partition on a smaller scale when we divide the information in a functional specification into subsections, group the functions in a design into facilities or cluster a set of format statements into a single subroutine. The key idea is that of establishing boundaries or interfaces between subsets of information in a representation.

This means that the concepts of coupling and cohesion are closely tied to the operation of partitioning. They often serve as guidelines for partitioning (telling us where to place the cut-lines in the representation) and are used to characterize the result.

Partitioning is perhaps the most critical operation on a representation because of the great impact different partitionings may have on our understanding of the information in it. This, in turn, leads to the fact that many design methods are primarily ways of achieving partitionings with specific properties (*e.g.* information hiding).

*Elaborate*

*Add information to a representation.*

To "elaborate" something means to make it more detailed, to add features, to work it out to perfection. Top-down design procedures make heavy use of elaboration, but other approaches and aspects of design involve elaboration as well. For example, if we are following a most-critical-component-first design approach, then, after devising the critical components, we elaborate our design by adding additional less-critical parts. When we take a gross representation of control flow and add the housekeeping details we are elaborating.

Decomposition, an activity frequently discussed in SE, is (in the terms of our theory) a partitioning followed by an elaboration. For example, the information contained in an SADT diagram is partitioned into a set of boxes representing functions (or data) and then

each box is elaborated (inside its boundary) with details on another diagram; the result is called a decomposition.

There are other cases called decomposition which do not fit this definition, leading us to not include decomposition as a basic activity. For example, decomposition of a set of specs into individual work assignments is simply a partitioning while top-down decomposition in programming is almost entirely elaboration with little or no partitioning.

*Abstract.*

*Take information out of a representation by grouping (parts of it), naming the groups, and removing the detail.*

"Abstraction" is the opposite of "elaboration." "Abstraction" is the operation of generalizing, of throwing away irrelevant details, of separating the essentials of a situation from the inessential, of considering something as a general quality unrelated to any particular concrete object. Abstraction is always done with respect to some reference point, in SE this is typically the underlying machine.

We abstract in many situations in order to see the overall structure of something. Theorems in mathematics, physical laws, and generalizations of all kinds are abstractions that permit us to better understand individual situations by relating them to other situations having similar characteristics. Abstraction has long been used in programming through the use of subroutines, in which we abstract a set of operations into a single name. The power of "abstraction" in SE is widely, and properly, appreciated.

*Translate.*

*Take the information in a given rep form and put it into a second rep form different from the first.*

During development we often must convert a representation written in some language into another representation written in a different language; maintaining, however, the same meaning in the output representation as is in the input representation. We commonly call this process "translation." The purpose of performing a translation is usually to obtain a new representation of the system that has some desired properties. For example, we may have represented the modular structure of a system using structure charts in order to gain the advantage of a graphical representation. At a later time, we may then need to translate the connections shown on the chart and the accompanying interface definitions into statements in a module interconnection language in order to facilitate constructing the physical system.

Translations are supposd to be perfect (with respect to information preservation) but often are not — leading to corruptions in the representation. Measuring the accuracy of the translation is sometimes important.

When we convert a problem model expressed as a data-flow diagram into a structure chart, we are translating and elaborating (by adding control information). When we decompile a piece of machine code into equivalent higher-level language, we are translating and deleting (details at the machine level).

*Delete*

*Remove information from a representation.*

Deletion takes place at many levels in SE. We may delete a function from a specification, an entire sub-branch of a structure chart, or a comma from a program. All have the effect of removing some information from the representation.

Deletion gives rise to a number of other operations involved with checking that the deletion maintains correct syntax, fixing it if not, and so on. This fine structure offers

an interesting and well-bounded situation from which some deeper cognitive studies could profit.

*Modify*

*Change information in a representation without translating, abstracting, or elaborating.*

The most common form of modification is the correction of syntax. We correct a misspelled identifier, add a missing operator, or move a line on a DFD. Semantic changes may also be made when adding another function to a specification, adding parameters to a module interface, or changing the computation of a quantity in a program. The essential aspect of modify is that it does not change the rep form being used nor add or remove information through abstraction and deletion.

*Compose.*

*Create a representation out of pieces.*

As noted in the discussion of rep forms, we often use compound forms such as for a requirements definition. The action of taking several completed representations (*e.g.* a DFD, a data dictionary, and a set of process descriptions) and joining them to form a composite whole is composition.

**Extract Information from Representation**

*Detect changes.*

*Determine changes needed to make a given representation fulfill its goals.*

This operation takes place in the context of two representations, one of which forms a set of goals for the other (*e.g.* the external specification of a module and the corresponding

detailed design). Typically, a change has been made at the lower level of abstraction (*e.g.* the detailed design) and this must be *reflected* into appropriate changes at the higher level (*e.g.* the external spec) so the two representations match.

*Derive.*

*Determine the specifications from which a given representation was produced.*

It sometimes occurs that a representation, as built, is inconsistent with the stated goals (the representation from which it was derived) but is nonetheless acceptable. The action of obtaining the revised goals from the representation, we call derivation. A common example is the situation in which functions not in a design are added by the implementors. The design corresponding to the system as built must then be derived.

*Identify Corruption.*

*Find in what way, if any, two representations differ in meaning.*

We use the term *corruption* to denote a difference in meaning between two representations that are supposed to have the same meaning. For example, a program specification may indicate parameters A, B, and C in that order; the actual implementation of the program may erroneously list the parameters in order A, C, B. This would be a corruption. "Identify corruption", then, is the activity of examining two representations (or sets of representations) to find corruptions in the output representation as compared to the input representation. A typical example is the comparison of the external and internal specifications of a program to ascertain if the interfaces are consistant.

*Measure.*

*Determine the value of some quantity or property in a given representation.*

There are many properties of a representation which can be measured, ranging from a simple count of its symbols to complex notions such as the cohesion between its elements.

We call any activity that ascertains values of some externally defined quantity, a "measuring activity." An example is the determination of the level of coupling present in a structural representation of a system.

*Predict.*

> *Make projections on the basis of a representation that indicate the characteristics of the final representation of the development.*

Prediction is another fundamental mental activity that is used in many ways. Here we use it to indicate an action that is central to the validation process — the determination of whether a specification, if eventually implemented without error, describes a system which will meet the current goals for the system. Perhaps the most common example of this activity is the customer sign-off on a set of system specifications.

### Evaluate Representations

*Verify.*

> *Test if a given representation fulfills its stated specification.*

Generally, verification means the process of determining that a specific workproduct meets the specifications from which it was derived. For example, we may verify that an architectural design fulfills the functional specifications or that a set of detailed module designs conforms to the architectural design. This operation is a direct result of the underlying view that SE is a process of mapping one representation into another.

*Validate.*

> *Test if a given representation fulfills a given set of development objectives.*

We differentiate between "goals" which apply to a representational mapping and "objectives" which relate to the external targets for a given SE development. A common activity (sometimes formalized) is the comparison of the current system rep (*e.g.* a functional spec) to the development objectives. Typically, this happens at the start of a project (once the functional specs are written) and at the end (acceptance testing). More careful development is sometimes needed (*e.g.* in large and complex projects) resulting in validation at several intermediate stages.

*Apply Quality Control.*

*Compare a set of measurements of a representation to a set of standards.*

A representation, just as any other object, can be compared to a set of external quality criteria. This activity which may employ measuring to produce characterizations to be compared, we call representation quality control. This is what we are doing when we check a specification for consistency of data definitions, for example.

*Approve.*

*Certify that a representation meets a set of quality standards.*

A neglected activity of SE is that of issuing a piece of information that a given representation has been inspected and meets (or fails to meet) a set of quality criteria. An example is the certification that a functional spec is complete and consistent.

## APPLICATION OF THE THEORY TO DEVELOPMENT PROCESSES

There is presently considerable interest in the subject of the software process [Lehman, 1984], to the point that the 8th. International Conference on Software Engineering is dedicated to the subject. This interest is largely driven by pragmatic concerns, but forms an ideal point of connection with the theoretical notions presented here.

Before applying the theory to several specific situations, we will connect it to the development processes that tie together the individual activities and representations of development. Development lifecycles are the most common and obvious manifestations of development processes, but any sequence of activities, aimed at producing software, qualifies. At this stage, the connection is informal and incomplete, but suffices to begin the process of utilizing the theory to explain pragmatic software engineering.

In this context then, the following statements characterize development processes in terms of our theory:

- *A SE development process is fundamentally one of gathering information and putting it into a representation.*

Our model is that there is a lot of information "out there," some of which is relevant to the system model we are currently developing. The activities of development, when formed into a coherent process, can be properly viewed as a process that filters out the relevant information and puts it into the developing representation of the system.

- *There is a representation of the system at all times.*

At first, it may be a single sentence describing the concept or need of the commissioner of the system; at the end, it is the code and all the accompanying documentation. As noted above, we assume the representation is always a tangible one.

- *There is a partial ordering of the representations with respect to their degree of completeness and a complete ordering with respect to time at which they were produced.*

Given these orderings, we can think of a "direction" of development which goes from the earlier (less complete) representations to the later (more complete) ones. This corresponds to the utilization of the elaboration operator to carry us from more abstract (with respect to the machine) to less abstract representations of the system.

- *A common variant of the "normal" process is that of reflection in which we go back to a previous representation.*

Sometimes this iteration of steps in development consists of an explicit "derive" operation to obtain the corresponding, more abstract specification that led to the representation. Other times, we simply shift attention to the previous representation and make changes to it based on information that has become available only after creating the "later" representation.

- *Development is a process of transforming one representation into another.*

In this context, all of the activities of development can be viewed as either carrying out such transformations or making them possible.

- *As the representation of the system progresses (i.e. gets closer to the final form) it "soaks up" information.*

Thus, we think of there being "more" information in later versions. As with abstraction, in which it is important to note the point of reference, the "more" here refers to machine-relevant information. Indeed, a common problem of development is that as the system is built, we *lose* information relevant to the application domain.

- *Progression is not always foward with respect to the partial degree of completeness.*

We often must iterate and return to an earlier (in time) representation. However, this iteration may be progress since the newer representation will be produced (by some activity) from the later representation. It may have more information in it, or the information may be more accurate. In either event, the iteration may produce a better version. This is not always the case, however, since it is possible that the iteration will introduce errors or in some other way produce a worse representation.

- *Development processes are often non-predictive.*

That is, we cannot always tell what will result from applying a particular activity. Further, we often cannot say in advance which activity should even be applied.

- *Development processes are complex combinations of fundamental development activities.*

The complexity of a development process in terms of the amount of information, number of pieces of representation, and number of possible activities that can be applied at any step lead to the need for having systematic technical and managerial control of the development activity. This complexity is also what leads to the need for mathematical rigor wherever possible.

- *Development processes are only activities in even larger processes.*

For example, the process of automating a factory has as one activity, the development of software. Business planning and management of systems development surround a development process, which in turn is composed of fundamental activities.

There are many additional things that one can observe about pragmatic development processes. This set however, provides a tie between the abstractions of the theory and the pragmatic world of software engineering.

## APPLICATION OF THE THEORY TO SPECIFIC SITUATIONS

In this section we present some examples illustrating usage of the theory. We have picked simple examples so as to not get bogged down in a mass of details (and to keep their length manageable). We have also used situations described in the literature in order to emphasize our usage of the theory as an analysis and understanding tool.

Our presentation will consist of a statement of the SE situation, interleaved with a translation into the terminology of the theory. This will then be followed by an analysis.

## Step-Wise Refinement

Wirth's original formulation of the step-wise refinement method [Wirth, 1971, p.226] was succinctly summarized at the end of his paper and can be paraphrased as follows (leaving out only original commentary as indicated by (...)):

1. Program construction consists of a sequence of refinement steps. In each step, a given task is broken up into a number of subtasks. Each refinement description of a task may be accompanied by a refinement of the description of the data, which constitute the means of communication between the subtasks. Refinement of the description of program and data structures should proceed in parallel.

1′. Program construction consists of a sequence of refinement (*partitioning* followed by *elaboration*) steps. At each step a given task is *partitioned* into a number of subtasks. Each *elaboration* in the *representation* of a task may be accompanied by the *elaboration* of the *representation* that permits communication among the subtasks. *Elaboration* of both *representations* should proceed so that their *level of detail* are made more concrete by a like amount in a given step.

2. The degree of modularity obtained in this way will determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is executed.(...)

2′. The degree of modularity of the program will determine the ease of applying *change activities* to the finished program *representation* in the event of changes or extensions to the purpose or changes in the environment (language, computer) in which it is executed.

3. During the process of stepwise refinement, a notation which is natural to the problem in hand should be used as long as possible. The direction in which the notation develops during the process of refinement is determined by the language in which the

program must ultimately be specified, *i.e.* with which the notation ultimately becomes identical. This language should therefore allow us to express as naturally and clearly as possible the structures of program and data which emerge during the design process. At the same time, it must give guidance in the refinement process by exhibiting those basic features and structuring principles which are natural to the machine by which programs are supposed to be executed. (...)

3′. The *representation* should maintain a *viewpoint* oriented toward the problem as long as possible. The *representational forms* used during refinement will be influenced by the *form* of the target language. They must be highly *communicative* of the structures of program and data which are *elaborated*. At the same time, these *representational forms* must *communicate* features of the target machine.

4. Each refinement implies a number of design decisions based upon a set of design criteria. Among these criteria are efficiency, storage, economy, clarity, and regularity of structure. (...)

4′. (no change)

5. (...)

The following analysis is based on the theory:

- It is left implicit in the original statement that the *initial representation* should have a *scope* that encompasses the entire problem to be solved and that each step should preserve this scope. It is our understanding, however, that this is an essential aspect of the method and hence should not be left implicit.

- Guidance is needed on how to change the *representational forms* (language) as the process proceeds in order to achieve the desired viewpoint and communication characteristics.

- No reflection activities are indicated, although, again, we believe they are intended.
- No evaluation activities are indicated (such as "verify" or "validate," although implicitly they are essential.
- It is not indicated whether future changes to the problem should operate on the final representation or on an intermediate one in the hierarchy.

Although additional comments can be made, our purpose here is to illustrate use of the theory, not to tear apart a particular presentation. In fairness, we should note that some of the items not mentioned in the quoted material above are covered implicitly in the presentation in Wirth's paper.

However, one of the motivations for our theory is to make it easier to give such presentations. The theory can provide a checklist to remind us to deal with issues (such as reflection and evaluation in this case) that are indicated as important in the situation.

### Top-Down Programming

Ledgard [1974, p.64] provides a concise definition of a top-down programming method that builds on step-wise refinement:

1. *Exact Problem Definition:* The programmer starts with an *exact* statement of the problem. (...)

1′. The *scope* of the initial problem statement should include all elements of the problem to be solved, and its *validity* should be as high as possible consistent with the level of detail chosen. No "derivation" should be permitted to act on or produce a new initial representation.

2. *Initial Language Independence:* The programmer initially uses expressions (often in English) that are relevant to the problem solution, even though the expressions cannot be derectly translated into the target language. From statements that are *machine and*

*language independent*, the programmer moves toward a final machine implementation in the target language.

2′. Starting with a representation that has a ***viewpoint*** independent of machine and later ***representation forms***, the programmer applies a series of ***operators*** to arrive at a final representation in the target language.

3. *Design in Levels.* The programmer designs the program in *levels*. At each level, the programmer considers alternative ways to refine some parts of the previous level. The programmer may look a level or two ahead to determine the best way to design the level at hand.

3′. The programmer produces an explicit design in ***levels of detail.*** The programmer considers alternative "elaborations" at each level. The programmer may "scout" ahead two or three levels to gain information on how to best make "decisions" at the current level.

4. *Postponement of Details to Lower Levels.* The programmer concentrates on critical broad issues at the initial steps, and postpones details (...) until lower levels.

4′. The programmer ***makes decisions*** ("decides") at ***early steps*** of development (low level of detail) that are critical and affect the entire ***scope*** of the representation, the programmer postpones decisions that affect only part of the entire scope or narrow parts of the representation until ***later steps*** (high level of detail).

5. *Insuring Correctness At Each Level.* After each level, the programmer rewrites the "program" as a correct formal statement. This step is critical. He must ***debug*** his program and insure that all arguments to unwritten procedures or sections of code are explicit and correct so that further sections of the program can be written ***independently*** without later changing the specifications or the interfaces between modules.

5′. After each step, the programmer *translates* the representation into a correct, formal *representational form*. He must debug and "verify" his program by "identifying corruptions" involving references to unwritten sections of code or procedures.

6. *Successive Refinements.* Each level of the program is successively refined and debugged until the programmer obtains the completed program in the target language.

6′. Each representation is further "elaborated" and "transformed" into new representations closer to the target *representation form*. It is also "verified" until a *representation* in the target *representation form* is obtained.

Casting this method into the terms of our theory leads to the following observations:

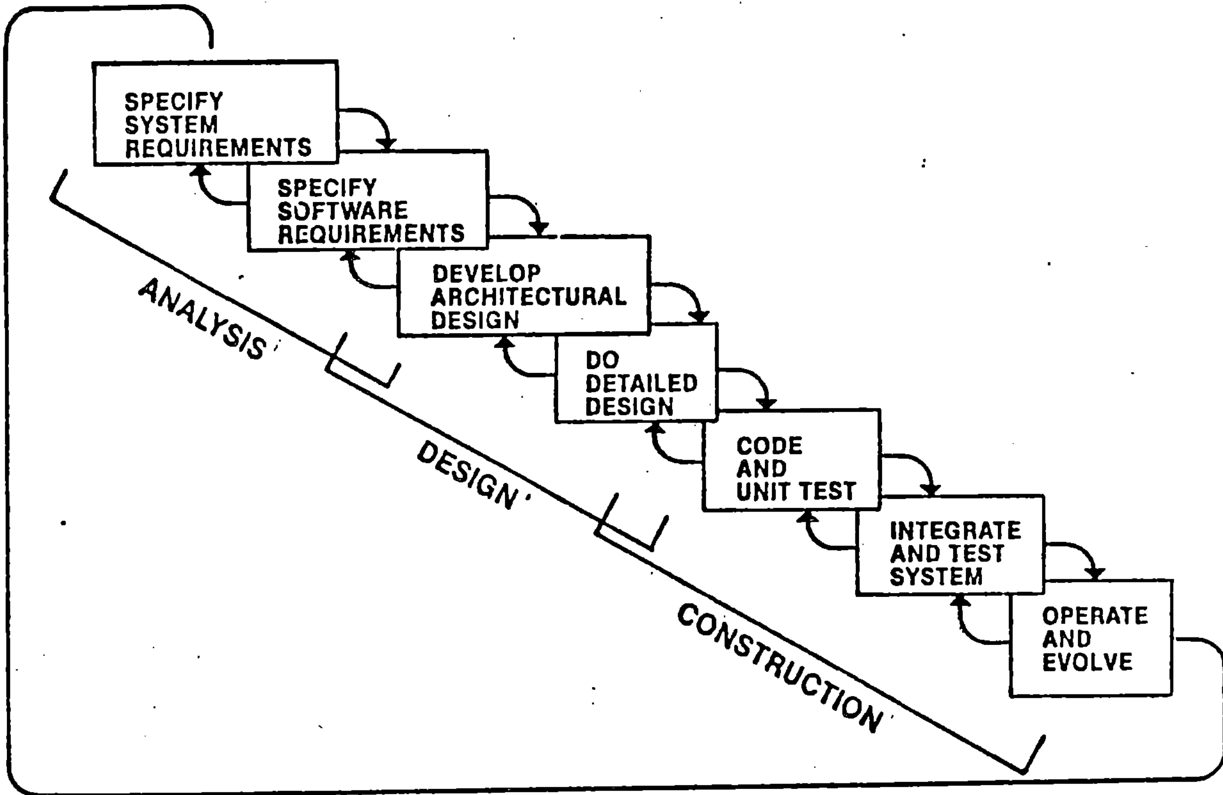
- The original formulation just restates in places (*e.g.* step 6) the basic ideas of the theory. An individual method should be able to build on the underlying concepts rather than having to restate them each time.
- Though relatively well stated, Ledgard required two additional pages of prose to explicate the approach. For example by itself, step 5 says nothing about what to do with the formal statements. Using the concepts to the theory, the entire method could be stated more concisely.
- The prohibition against changing the initial problem representation is unreasonable when the method is viewed as part of a larger system development method in which validation activities may show inconsistencies between the developing program and development goals.
- Little guidance is given on choice of representation elements to "partition" and "elaborate".

In general the top-down method, as stated here, captures in a pragmatic form much of the underlying theory.

## Waterfall Model

Figure 6 shows the well known "waterfall" lifecycle model. A fuller description of it can be found in [Freeman, 1983] and numerous other places. It is used so broadly, that it is impossible to find a concise statement of it.

**Figure 6: Software Lifecycle**



The following, then, is a synthesized description, expressed in the terms of our theory:

> The lifecycle of a software system can be divided into a number of stages characterized by the *activities* applied and the *representation forms* produced. The activities are actually *developement processes* focused on the production of a particular *representation form* (*e.g.* detailed design).
>
> A flow of *representation forms* between stages is expected in in a "forward" direction (*i.e.* in growing level of detail). It is further expected that a return to a previous stage may be necessitated. *Representation forms* for each stage maybe specified.

Some of the complaints commonly lodged against the waterfall model can then be explained in terms of the theory:

- Development does not proceed in such a strict, top-down manner. (The model implies a time-line in the forward direction. Reflection — which *is* shown — destroys this.)
- The workproducts passed from stage to stage are not well-defined. (Representation forms are not defined and, in general, are not related to the activities in a stage.)
- The model does not explain how to develop the specified workproducts. ("Representation change" activities are not composed into development sequences in the model.)
- Quality assurance is not addressed. ("Evaluation" activities are not directly linked to the model.)
- Prototyping is not included in the model. (Only a single development sequence is shown even though many alternatives may be utilized in practice.)

Additional analysis of this traditional model can be developed on the basis of the theory, but we will not do so here.

## Jackson System Design

Jackson [1983, p.38] succinctly presents a new approach to software development:

1. Entity action step: In this step the developer defines the real world areas of interest by listing the entities and actions with which the system will be concerned.

1′. The *scope* of a model of the areas of interest is defined by "deciding" with which entities and actions the system will be concerned.

2. Entity structure step: In this step the actions performed or suffered by each entity are arranged in their orderings by time. The orderings are representated by Jackson Structure Diagrams.

2′. The *representation* of step 1 is "partitioned" and "elaborated" into one in which (a) all actions associated with each entity are explicitly given and (b) these activities are ordered by time.

3. Initial model step: In this step, the description of reality, in terms of entities and actions, is realized in a process model and connections between the model and the real world.

3′. The *representation* of step 2 is utilized as the basis for a new *representation* produced by "elaborating" the descriptions of entities and actions. The *representation form* is one of parallel, cooperating processes.

4. Function step: In this step, functions are specified to produce the outputs of the system, additional processes being added to the specification as necessary.

4′. The *representation* of step 3 is "elaborated" with additional function specifications and processes.

5. System timing step: Here, the developer considers some aspects of process scheduling which might affect the correctness or timeliness of the system's functional output.

5′. The *representation* is "validated" against customer goals and "predictions" of performance are made. "Decisions" are made, but informally documented (this last taken from more detailed explanation in Jackson's book).

6. Implementation step: In this step, the developer considers what hardware and software is, or should be, provided for running the system and applies the techniques of transformation and scheduling, along with techniques of database definition, to allow the system to be efficiently and conveniently run.

6′. "Decisions" are made regarding needed software and hardware for running the system. The representation is "translated" into one that can be efficiently and conveniently run.

The steps given here are taken from Jackson's outline of his entire book, in which each step is thoroughly treated. Some of the points below are treated in the fuller presentation:

- Nothing is said of "verification" activities to be applied between steps.
- The transformational view of development of our theory is very similar to Jackson's basic model of development.
- Reflection is not discussed.
- The viewpoint taken in the first four steps is that of the customer or user.
- We believe presentation of the method could be improved by emphasizing the information that must be added at each step (following that aspect of our theory).

Again, the analysis here is intended to be illustrative of use of the theory, not a definitive critique of a particular method.

### Miscellaneous Characterizations

As a final illustration of use of the theory, we list without comment several characterizations of software engineering situations and issues:

1. The emphasis in almost all research concerned with the constructive activities of SE is on the representation change activities. Yet, the theory implies and practice confirms that extraction and evaluation activities are just as important, if not more so. Further, the research tends to focus on only a few of the activities such as "abstraction" and "elaboration," ignoring others such as "partitioning."

2. There has been much discussion of the value of prototyping. That is not surprising when one looks at the traditional lifecycle as only one, highly restrictive development process out of the multitude that can be formed. Further, that lifecycle can be seen as not incorporating some of the basic activities relevant to SE.

3. Languages (rep forms) are central to SE. Yet, few are designed to meet requirements based on any underlying notion of the nature of SE (*e.g.* that the SE process is one of continually transforming a representation of the system). That is not to say that the developers of modern languages such as Ada [DoD, 1980], Modula-2 [Wirth, 1983], and Plain [Wasserman, 1981] do not understand SE. We are saying, however, that languages could be improved if they are based on a fundamental view. Further, current languages do not support many of the characteristics of representations or basic activities of SE.

4. Education in SE is all too often focused on *particular* development processes and representations rather than on their general properties. When they are no longer valid, new ones must be learned, sometimes with great difficulty. The situation is akin to teaching how each piece of electronic equipment works, rather than teaching the fundamentals of electronics.

5. Management of SE is often considered difficult since there may be no way to tell when the design is finished or how much it should be iterated. This is a result of having no measures of the properties of a representation (scope, correctness, validity, and so on) nor clear understanding of the results expected from applying certain activities or development processes.

We offer these characterizations as further examples of the explanatory power of our theory.

## CONCLUSION

We can make several conclusions from the development and presentation of this theory:

1. It has a certain internal consistency and completeness;
2. It is capable of explaining common situations in a clear manner and of providing some insight to more complex situations;
3. It needs to be validated;
4. It needs to be further developed and refined;
5. It needs to be used.

Our own research (individually and cooperatively) will focus on application of the theory to a number of situations, since (for us, at least) it is a working tool that greatly aids our understanding of SE. We also will be expanding it, especially in the area of development processes.

The ultimate conclusion will be drawn by others. If it aids in the further study and understanding of the SE process, then it will have met its goal.

## ACKNOWLEDGEMENTS

The financial support of IBM Brasil that made possible an extended period of collaboration between the authors was especially instrumental in bringing this work to fruition. The word processing support of Miss Sheila Zokner of IBM-Rio, Miss Lynn Caverly and Ms. Pat Harris of UC Irvine is most appreciated. Finally, the probing questions of students, colleagues, and practitioners over the years have served to keep alive and sharpen our interest in what makes SE work.

## REFERENCES


Basili, Victor R. "Product Metrics," *Tutotial on Models and Metrics for Software Management and Engineering*, 1980. p. 214–217.

Boehm, Barry. W. "Software Engineering," *IEEE Transactions on Computers*, December 1976. p. 1226–1241.

DoD. *Reference Manual for the Ada Programming Language*. July 1980.

Fairley, Richard. *Software Engineering*. McGraw Hill, 1985.

Freeman, Peter. "Software Design Representation: Analysis and Improvements," *Software Practice and Experience*, Vol. 8, 1978. p. 513–528.

Freeman, Peter. "Towards a Theory of Software Engineering," unpublished lecture notes, January 1979.

Freeman, Peter. "Fundamental of Design," *Software Design Techniques 4th Edition*, P. Freeman and A.I. Wasserman (eds.). IEEE Computer Society, October 1983. p. 2–22.

Freeman, Peter and Allan Newell. "A Model for Functional Reasoning in Design," *Proceedings of the Second International Joind Conference on AI.* London, September 1971. p. 621–633.

Freeman, Peter and Anthony Wasserman. *Tutorial on Software Design Techniques — 3rd Edition*. IEEE Computer Society, 1980.

Freeman, Peter and Anthony Wasserman. *Tutotial on Software Design Techniques — 4th Edition*. IEEE Computer Society, 1983.

Jackson, Michael. *System Development*. Prentice-Hall International, 1983.

Jensen, Randall and Charles Tonies. *Software Engineering*. Englewood Cliffs, New Jersey: Prentice Hall, 1978.

Kerola, P. *Summary Report of the Systemeering Research Seminar of Tampere*. Helsinki: Finish Data Processing Association, 1979.

Kerola, P. "On Infological Research into Systemeering Process," *Proceedings of IFIP TC. 8 Working Conference in Bonn*. Amsterdam: North Holland, 1980.

Kerola, P. and Peter Freeman. "A Comparison of Lifecycle Models," *Proceedings of the 5th International Conference on Software Engineering*. IEEE Computer Society, 1981.

Lehman, M. *Proceedings of the Software Engineering Process Workshop*. IEEE Computer Society, 1984.

Ledgard, Henry. *Programming Proverbs*. Rochelle Park, N.J.: Hayden Book Co., 1974.

Levin, Steven L. *Model of the Problem Selection Process in the Design of Computer Programs*," Ph.D. thesis UC Irvine, 1976.

Mills, Harlan D. "The Management of Software Engineering. Part I: Principles of Software Engineering," *IBM Systems Journal* Vol. 19, No. 14, 1980.

Newell, A. and H. Simon. *Human Problem Solving.* New York: Prentice-Hall, 1972.

Parnas, David L. "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, December, 1972. Reprinted in Freeman and Wasserman (1983).

Parnas, David L. "Some Hypotheses About the 'Uses' Hierarchy for Operating Systems," *Technische Hochschule Darmstadt*, Darmstadt, West Germany. Tech. Report. March 1976.

Pressman, Roger. *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1982.

Ross, Douglas T. and Kenneth E. Schoman, Jr. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, January, 1977. Reprinted in Freeman and Wasserman (1983).

Ross, Douglas, John Goodenough, and Al Irvine. "Software Engineering: Process, Principles, and Goals," *Computer*, May 1975. Reprinted in Freeman and Wasserman (1980).

von Staa, Arndt and Peter Freeman. "Requirements for Software Engineering Languages," in preparation. October, 1984.

Wasserman, Anthony I. et al. "Revised Report on the Programming Language PLAIN," *ACM Sigplan Notices*, Vol. 16, No. 5 (May 1981). p. 59–80.

Wirth, Niklaus. *Programming in Modula-2.* New York: Springer-Verlag, 1983.

Wirth, Niklaus. "Program Development by Step-wise Refinement," *Communications of the ACM*, April 1971. p. 221–227. Reprinted in Freeman and Wasserman (1983).

Yourdon, E. *Managing the Structured Techniques.* New York: Yourdon Press, 1979.