CHAPTER I

Introduction

In interacting with the environment, with others, and with the artifacts of technology, people form internal, mental models of themselves and of the things with which they are interacting. These models provide predictive and explanatory power for understanding the interaction. These statements hardly need be said, for they are consistent with all that we have learned about cognitive processes and, within [the book *Readings in Human-Computer Interaction*], represent the major underlying conceptual theme. Nonetheless, it does not hurt to repeat them and amplify them, for the scope of the implications of this view is larger than one might think.

— Donald Norman [49, p. 241]

As Donald Norman indicates, the human-computer interaction (HCI) community has benefited from exploring the implications of mental models. This perspective has aided their ability to create more usable, understandable user interfaces for computer programs, as exemplified by the popular “desk top” metaphor.

Researchers also have used the concept of mental models to examine how programmers design new software [23][64][14] and understand existing software [32][41]. The next logical step, as it was for the HCI community, is to ask how focusing on mental models can help us create software that is more usable and understandable to programmers. This is particularly important in the coming age of “software reuse,” where software parts will be written with the goal of being utilized by different programmers in quite disparate contexts. In this dissertation, we turn our attention to the question of how to help create software that is more comprehensible, with interesting results.

In particular, asking questions about how programming languages support the formation of effective mental models leads one to conclude that conventional programming languages are *inadequate* for constructing large, sophisticated software
systems that are “understandable.” Furthermore, there is not a generally-accepted
theory of what the building-blocks of software systems are, or how they are envisioned
as contributing to the understandability of the software comprising them.

This dissertation explains the inadequacy of conventional programming languages
in supporting the formation of effective mental models. It then presents a new model
of component-based software that provides concrete support for recording critical in-
formation about each software structure, information that can form the basis for a
programmer’s own mental model of that structure. This new model, termed the ACTI
model, for “Abstract and Concrete Templates and Instances,” is both mathematically
formal and language-independent. It captures and formalizes the underlying concep-
tual view of software architecture embedded in modern module-structured languages
while simultaneously providing support for forming mental models. As a result, it
can serve as a general-purpose theory of the nature of software building-blocks and
their compositions.

1.1 Why Conventional Languages Fail

In short, modern programming languages have evolved from their predecessors with
the primary purpose of describing instructions to computers. They were never de-
digned to help explain to people the meaning of the software that they can describe.
This has led to two critical problems with programming languages today: modules are
considered to be purely syntactic constructs with no independent meaning, and those
parts of programs that are deemed meaningful (usually procedures, in imperative
languages) have a “hierarchically constructed” meaning.

First, most modern programming languages have some construct that is intended
to be the primary “building-block” of complex programs. This building-block may
be called a “module,” a “package,” a “structure,” or a “class.” Unfortunately, these
constructs are rarely given meaningful semantic denotations. Conventional wisdom in
the computer science field indicates that these constructs are primarily for grouping
related definitions, controlling visibility, and enforcing information hiding. For exa-
ample, when speaking of module-structured languages like Ada or Modula-2, Bertrand
Meyer states:

In such languages, the module is purely a syntactic construct, used to
group logically related program elements; but it is not itself a meaningful
program element, such as a type, a variable or a procedure, with its own
semantic denotation. [46, p. 61]

In this view, there is no way for one to make such building-blocks contribute directly
to the understandability of the software comprising them. While object-oriented
languages usually give a stronger meaning to the notion of a “class,” they also fail
to provide any vision of how the meaning of individual classes can contribute to a
broader understanding of the software systems in which they are embedded.

Second, those program elements that are given a real semantic denotation are often
given a meaning that is “hierarchically constructed,” or synthesized. In other words,
the meaning of a particular program construct, say a procedure, is defined directly in
terms of its implementation—a procedure “means” what the sequence of statements
implementing it “means.” The meaning of its implementation is defined in terms of
the meanings of the lower-level procedures that it calls. Thus, a procedure’s meaning
is “constructed” from the meanings of the lower-level program units it depends on, and
the meanings of those lower-level units in turn depend on how they are implemented,
and so on.

This simple synthesis notion of how meaning is defined bottom-up is adequate from
a purely technical perspective. It is also very effective when it comes to describing
the semantics of layered programming constructs. Unfortunately, it is at odds with
the way human beings form their mental representations of the meaning of software
parts, as described in Chapter II.

The result of these two features of programming languages is that they are in-
adequate for effectively communicating the meaning of a software building-block to
people (programmers, in particular). The semantic denotations of programming con-
structs in current languages only relate to how a program operates. They fail to
capture what a program is intended to do at an abstract level, or why the given
implementation exhibits that particular abstract behavior. In order to address these
concerns, it is necessary to assign meaning to software building-blocks, to separate
the abstract description of a software part’s intended behavior from its implementa-
tion, and to provide a mechanism for explaining why the implementation of the part
achieves behavior consistent with that abstract description.

1.2 Correcting the Deficiency

The ACTI model is centered around the notion of a “software subsystem,” a general-
ization of the idea of a module or a class that serves as the building-block from which
software is constructed. A subsystem can vary in grain size from a single module up to
a large scale generic architecture. ACTI is designed specifically to capture the larger
meaning of a software subsystem in a way that contributes to human understanding,
not just the information necessary to create a computer-based implementation of its
behavior.
ACTI is not a programming language, however. It is a formal, theoretical model of the structure and meaning of software subsystems. It has two features that specifically address the inadequacies described in Section 1.1:

1. In ACTI, software subsystems (building-blocks) have an intrinsic meaning; they are not just syntactic constructs used for grouping declarations and controlling visibility. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.

2. The meaning of a software subsystem is not, in general, hierarchically constructed. In fact, it is completely independent of all the alternative implementations of the subsystem.

Thus, ACTI provides a mechanism for describing what a subsystem does, not just how it is implemented. The meaning provided for a subsystem is a true abstraction—a “cover story” that describes behavior at a level appropriate for human understanding without explaining how the subsystem is implemented. Further, ACTI provides a formally defined mechanism, called an interpretation mapping, that captures the explanation of why an implementation of a subsystem will give rise to the more abstractly described behavior that comprises the meaning attributed to the subsystem—in short, an explanation for why the cover story works.

1.3 Contributions

The primary contribution of this research to the field of computer science is the development of the ACTI model. As a model of software structure, ACTI is unique because:

1. It promotes the creation of large, sophisticated software systems that are understandable. This is achieved by both supporting the formation of mental models of software parts, and addressing the inherent human (cognitive) limitations in dealing with those mental models.

2. It gives a real semantic denotation to “modules,” which are the building-blocks from which software is composed. This denotation identifies just what “modules” should be so that they are meaningful, allow sophisticated composition, and present simple (not synthesized) conceptual models.

3. It identifies “interpretation mappings” as the mechanism that allows one to clearly represent why an abstraction (a specification with a simple model) correctly describes the composite behavior of a complex combination of lower-level software parts (an implementation).
4. It is language-independent, and unifies the concepts behind object-oriented pro-
gramming and more traditional module-based programming. Thus, it solves the
problem of understandable software composition across module- and class-based
languages.

5. It solves the problems normally associated with “inheritance” [8] by separating
“programming by difference” from type-to-type and module-to-module subtyp-
ing relationships.

Because of these achievements, ACTI can serve as a general-purpose theory of the
nature of software building-blocks and their compositions.

1.4 Organization

Chapter II begins with a discussion of mental models and how programmers under-
stand software. The topic of mental models leads to an explanation of the inadequacy
of conventional programming languages, which provide no real aid to forming effec-
tive mental models of software parts. Chapter II concludes by explaining how ACTI
addresses this inadequacy. The discussion in Chapter II can be appreciated by anyone
with a general background in computer science.

Next, Chapter III provides an explanatory introduction to the ACTI model of
software subsystems. This introduction is presented at an informal, intuitive level,
using an extended code example written in RESOLVE (Section A.3). Chapter III
concentrates on what is contained in an ACTI subsystem, and examines a typical
software component specification. While Chapter III is aimed at a general computer
science audience, a basic familiarity with the RESOLVE language [66] will aid in
getting the most out of the example.

Chapter IV then provides a complete formal definition of the ACTI model of
software subsystems. The definition is centered around the four types of ACTI
subsystems—abstract instances, concrete instances, abstract templates, and concrete
templates—which can be used to provide semantics for module or class specifications,
implementations, generics, and so on. ACTI is defined in formal mathematical terms,
so a certain degree of mathematical maturity and familiarity with the concepts of
denotational semantics is very helpful in obtaining a full appreciation of the model.

Given the formal definition of ACTI, Chapter V continues the example introduced
in Chapter III to illustrate some of the more advanced aspects of ACTI. While Chap-
ter III focuses on one subsystem, Chapter V looks more at the relationships between
subsystems. It demonstrates the use of interpretation mappings between abstract and
concrete instances, and explains the semantics of module-level program statements.
CHAPTER I. INTRODUCTION

(such as instantiation of a parameterized component) in terms of the ACTI model. In addition to familiarity with the formal definition presented in Chapter IV, the reader will need a good understanding of the RESOLVE language to fully appreciate Chapter V.

Chapter VI brings the main body of the dissertation to a close by summarizing the work, discussing the contributions, and pointing toward future research.

For the reader interested in the details of the development of the ACTI model, Appendix A discusses the research approach. Included is an examination of four modern programming languages chosen for the broad spectrum of software construction philosophies they embody: OBJ, RESOLVE, Eiffel, and Standard ML. Appendix A gives summaries of these languages and discusses the issues they raise for the software structure modeling problem.

Given this collection of languages, the next step is to compare and contrast the extent to which each addresses the intuitive notions about software architecture and composition that practitioners have observed. For example, the separation of specifications from implementations, the possibility of multiple implementations for one specification, and the use of generics for parameterized programming are all structuring techniques that practitioners have used in software systems. In order to perform a comparison based on such criteria, however, it is necessary to capture these “intuitive notions” in a more rigorous form. Appendix B addresses this problem.

Finally, Appendix C analyzes ACTI using the notions formalized in Appendix B to show how it measures up against the requirements derived from previous work. This appendix also shows how each of the example languages in Appendix A is “subsumed” by the new language-independent ACTI model, or alternatively, how the structures in each of the languages can be interpreted within the ACTI mathematical spaces.

Taken as a whole, this dissertation follows the path that Norman advocates software professionals should follow, this time in the realm of software engineering rather than HCI:

People’s mental models are apt to be deficient in a number of ways, perhaps including contradictory, erroneous, and unnecessary concepts. As designers, it is our duty to develop systems and instructional materials that aid users to develop more coherent, usable mental models. [49, p. 244]