5-2010

The Pursuit of an Unequivocal Primary Representation

Delroy A. Brinkerhoff
Utah State University

Follow this and additional works at: http://digitalcommons.usu.edu/etd
Part of the Computer Engineering Commons

Recommended Citation
THE PURSUIT OF AN UNEQUIVOCAL PRIMARY REPRESENTATION

by

Delroy A. Brinkerhoff

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Instructional Technology and Learning Sciences

Approved:

Dr. Mimi M. Recker  Dr. Andrew S. Gibbons
Major Professor  Committee Member

Dr. M. David Merrill  Dr. David A. Wiley
Committee Member  Committee Member

Dr. Scott R. Cannon  Dr. Donald H. Cooley
Committee Member  Committee Member

Dr. Byron Burnham
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010
ABSTRACT

The Pursuit of an Unequivocal Primary Representation

by

Delroy A. Brinkerhoff, Doctor of Philosophy

Utah State University, 2010

Major Professor: Dr. Mimi M. Recker
Department: Instructional Technology and Learning Sciences

A chief human characteristic is the desire and ability to change the world. Prior planning is crucial when those changes are complex and extensive, and require the cooperation of many people. To satisfy this need, many disciplines have developed specialized notations for representing the plans. Developers in one discipline, computer-based instruction, are burdened by the current need to use two separate notations. Instructional experts design the instruction and represent the design with a primary representation. The instruction described in a primary representation is easy to see, which makes the representation suitable for evaluation, communication, and enhancement. Programmers translate the primary representation into a computer program, which is able to run on a computer but is a secondary representation.

The problem with this process is that the primary representation is equivocal or ambiguous. Equivocal representations are subject to multiple interpretations; it is also possible for programmers to introduce errors during translation. Alternatively, the computer program is unequivocal, but the instruction that is evident in the primary
representation diffuses into the program, becoming obscure and difficult to use for further evaluation, communication, or enhancement. A representation that is both unequivocal and primary benefits computer-based instructional development by eliminating ambiguity and translation errors while preserving the instructional details for later use.

A representation is unequivocal if it is computable, and it is primary if it is able to represent the dynamic behaviors of complex instruction and its use as a design language can be demonstrated in published literature. My research evaluated and compared two design languages, PEAnets (networks of processes, entities, and actions) and the Unified Modeling Language, as potential unequivocal primary representations. Two translators, one for each language, were developed as a part of this research, and four complex computer-based instructional examples were created and translated into operational computer-based instruction. The translators demonstrated that both representations are computable, and the examples demonstrated that both languages are sufficiently robust to represent complex computer-based instructional systems. Both languages have been used successfully for designing instruction or general computer systems. I concluded, based on these observations, that both languages qualify as unequivocal primary representations.

(241 pages)
To Kathryn, with love and appreciation
ACKNOWLEDGMENTS

It is difficult for me to imagine reaching the end of a doctoral program without a great deal of help and support from many people. Throughout my life I have received constant support and encouragement from my family, my friends, my colleagues, and my teachers. My success at this time is also your success.

I owe more to each member of my committee, Mimi Recker, Andrew Gibbons, Dave Wiley, David Merrill, Scott Cannon, and Donald Cooley, than words can express. Each member has had a profound impact on my life. I owe special thanks Dr. Recker, who graciously offered to fill the position as the chair of my committee when personnel changes in the Department of Instructional Technology and Learning Sciences left that position vacant. More importantly, she also changed the way that I think about knowledge and learning, and therefore the way that I teach. I also offer special thanks to Dr. Cannon, my major professor for my master of science degree. That experience taught me what I was capable of doing.

I am grateful to Warren Hill, the dean of the College of Applied Science and Technology at Weber State University, for his substantial support. I am also grateful to the faculty and staff of the Department of Computer Science at Weber State University, who provided substantial support and who were a never-ending source of encouragement.

My parents, Ann and Cameron Brinkerhoff, were my earliest supporters. They started me down a path of life-long learning and had faith that the path would take me where I needed to go. For that start in life, and for so many more things, I will always love you.
My most profound and heartfelt gratitude goes to my family: my children Diane, Mark, Michelle, and Daniel, and my wife, Kathryn. Yours has been the greatest sacrifice and leaves me with the largest debt. I love you all.

Kathy, after thirty years, our life together becomes richer and more beautiful each day. I promise to share more of that life with you now, and to love you forever.

Delroy A. Brinkerhoff
## CONTENTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT ....................................................................................................................... iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS ................................................................................................. vi</td>
</tr>
<tr>
<td>LIST OF TABLES ............................................................................................................ xii</td>
</tr>
<tr>
<td>LIST OF FIGURES ......................................................................................................... xiii</td>
</tr>
</tbody>
</table>

### CHAPTER

1  INTRODUCTION .......................................................................................................... 1
   Research Questions and Purpose ................................................................. 2
   Definitions ........................................................................................................ 5
   Computability ............................................................................................... 6
   Primary versus Secondary ..................................................................... 7
   Definition Summary ................................................................................ 10

   Procedure .................................................................................................... 10
   Instructional Demonstrations ................................................................ 14
   Dynamic Versus Visual CBI Components ............................................. 19
   Shells, Frames, and Instructional Delivery ............................................. 21
   Research Objectives .............................................................................. 27
   Conclusion ................................................................................................. 31
   References ................................................................................................. 32

2  AN EXPLORATION OF PEANETS: THE PURSUIT OF AN UNEQUIVOCAL PRIMARY REPRESENTATION: ........................................... 35
   Primary and Unequivocal ...................................................................... 37
   PEAnet History and Overview .............................................................. 40

   The Entity Table ..................................................................................... 41
   The Network View ............................................................................... 43

   Implementing a PEAnet System ............................................................... 47

   Entity Extensions .................................................................................... 48
   Activity Extensions ................................................................................. 49
LIST OF TABLES

Table ........................................................................................................................................ Page

1     A Comparison of PEAnet and UML Diagram Elements .............................................. 105
LIST OF FIGURES

Figure .............................................................................................................................. Page
1   A TM with four states and three program rules. ................................................................. 6
2   The canal lock demonstration ..................................................................................... 15
3   The MP3 Player demonstration .................................................................................. 16
4   UML class diagram showing the shell delivery system architecture
    and its main classes. .................................................................................................. 23
5   UML class diagram showing the packaging of and articulation
    between the shell presentation system, the PEAnet system, and the
    MP3 player example. ............................................................................................ 24
6   UML class diagram showing the packaging of and articulation
    between the shell presentation system, UML state machine library,
    and the MP3 player example. ................................................................................ 25
7   The PEAnet property table illustrating the entities, properties, and
    property-values that appear in the boat and canal lock example. ....................... 43
8   Canal lock activities, processes, consequences, and conditions ......................... 44
9   An empty activity that creates consequences that may be used as
    conditions. ............................................................................................................ 49
10  An internal activity triggered by a timer within the CBI. ........................................ 50
11  The internal “_START” activity is triggered once at CBI startup. ....................... 50
12  Valued consequence ...................................................................................................... 53
13  Valueless consequence .................................................................................................. 53
14  Tool tip showing the property values necessary for a process to run .................... 54
15  Condition editor pick list: available conditions are on the left and
    currently set conditions are on the right. ............................................................... 54
16  The PEAnet sequencing problem illustrated by the difficulty of
    implementing a toggle button (a) and the limited solution (b). ......................... 56
17  How to return to a previous state when multiple paths enter the current state? ..... 60
18 The UML class diagram describing the canal lock CBI example. .........................70
19 The state machine describing a digital watch represented by two tables. ..........73
20 A graphical representation of the state machine describing a digital watch. ....74
21 A Moore machine state. ....................................................................................76
22 A compound state with three substates and a history state. .........................78
23 UML state diagram for the Boat object in the canal lock example. ..............80
24 UML state diagram for the Lock object in the canal lock example. .............81
25 UML state diagram for the two Gate objects in the canal lock example. .........81
26 UML state diagram for the Valve object in the canal lock example. .............82
27 The components of the UML CBI development system. ...............................89
28 Partial CFG for evaluating arithmetic expressions. ........................................98
29 Expressions: (a) an arithmetic expressions, (b) a pointer-valued expression. .....................................................................................99
30 Parse tree for the expression: (x + 3) * 5.......................................................100
31 Properties in the PEAnet version of the canal lock example..........................107
32 Classes, and their associated attributes and behaviors; the UML version of the canal lock simulation defines one Gate class but has two Gate objects named lowerGate and upperGate. ........................................107
33 PEAnet describing the dynamic behavior of a CBI system triggered by the user pressing the Close Valve Button.........................................................109
34 UML state diagram describing the dynamic behavior of the canal lock valve. ..............................................................................................................110
35 Unintended consequences: the Play process changes mode to play; this allows the Pause process to run, which changes the mode back to pause. .....................................................................................117
36 UML state diagram: pressing the play button alternately triggers the play and pause actions. .................................................................................117
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>The UML state diagram solution to the memory problem: add more states.</td>
</tr>
<tr>
<td>38</td>
<td>Modified PEAnet entity table grouped by entity, and containing both properties and processes.</td>
</tr>
<tr>
<td>39</td>
<td>Modified PEAnet diagram for the Valve entity.</td>
</tr>
</tbody>
</table>
Present-day practice of computer-based instruction is characterized by working with twofold representation: script and program. The instruction expert designs and produces the script or storyboard which forms the base for the programmer to develop the program that is intelligible for the computer. The storyboard is the primary representation. Unfortunately it is not unequivocal either for the programmer or for the expert critic, and therefore not suitable for intersubjective professional communication. It is only the programmer who develops an unequivocal (secondary) representation, i.e. the program. However, as a basis for communication it is absolutely useless: what is important in terms of subject-matter or instructional method disappears in the jungle of codification. (Eckel, 1993, p. 7)

Within the context of computer-based instruction (CBI), Eckel’s (1993) observations implied that equivocation and rank (primary or secondary) form two independent dimensions of instructional representation. The instruction contained in an unequivocal representation is unambiguous and interpretable in only one way; conversely, an equivocal representation is ambiguous, and therefore, subject to multiple interpretations. Similarly, instruction represented in a primary form is readily apparent and, therefore, suitable for use by instructional experts, while an arcane notation, suitable for computers, shrouds secondary forms. I propose that the best and most obvious solution to this problem is to identify a CBI design representation that unambiguously and clearly presents instructional intent – a representation that is both unequivocal and primary.

Understanding the development and use of storyboards highlights the essential characteristics of a primary representation. Prior to storyboards, stories were developed in a comic-strip format with three or six drawings or scenes per page. Disney animator Webb Smith created the first storyboards in the early 1930s.
Smith . . . hit upon the idea of making each of the drawings on a separate sheet of paper and pinning them all, in sequence, to a bulletin board. The story for an entire short could be accommodated on a single board and thus the director, or anyone else concerned with the production, could see the plot of an entire movie spread out in front of him. If changes had to be made, drawings could be moved or taken down and replaced by others. . . . Disney himself must have been delighted by this innovation. The storyboard enabled him to participate even more closely in the development of his cartoons, allowing him to walk into a music room and see at a glance exactly what needed to be done. (Finch, 1973, pp. 82-84)

By offering storyboards and scripts as examples of primary representations, Eckel (1993) suggested the role that primary representations play in CBI development. Storyboards, scripts, and other primary representations allow developers to define, communicate, and experiment with the sequence of and the interaction between individual CBI components. However, storyboards and scripts are equivocal. I maintain that it is possible to describe the dynamic or behavioral components of CBI with primary representations other than storyboards and scripts. Furthermore, if the representations are made to satisfy one well-defined requirement, I maintain that they are also unequivocal.

**Research Questions and Purpose**

“A problem occurs when a problem solver wants to transform a problem situation from the given state into the goal state but lacks an obvious method for accomplishing the transformation” (Mayer & Wittrock, 1996, p. 47). This definition evokes the common view of problem solving as discovering a path, from a starting point to an ending point, through a maze (Simon, 1998); once discovered, the path represents the problem solution.

A single, fundamental question defines my starting point: Is a representation that is both unequivocal and primary possible? Although a negative answer would quickly
end this line of research, a positive answer by itself would do little to advance the science of CBI development. Two auxiliary questions clarify and focus my research. First, is it possible to create or to identify an unequivocal primary representation that is suitable for specifying the dynamic or behavioral components of CBI? Second, if such a representation is created or identified, is it practical to develop CBI with it?

The overall purpose of the research I present here is to answer these questions. Specifically, my ending point or goal state is the verification that an unequivocal primary representation is both theoretically and practically achievable, and to demonstrate that it is both possible and practical to implement the dynamic components of CBI in such a representation. I follow the often-used strategy of decomposing the overall problem into subproblems, with the solution of each subproblem strengthening the overall verification.

The first level of decomposition naturally produces two subproblems.

1. The first subproblem addresses the potential for creating an unequivocal primary representation. This problem is further decomposed into two additional problems.

1.1. The first problem is to define and characterize unequivocal and primary representations. The definitions are needed to establish that a proposed representation is either unequivocal or primary. My definitions are sufficient but not necessary, which means that it is possible that my definitions might exclude a representation that is unequivocal or primary, but a representation that satisfies my definitions is guaranteed to be unequivocal and primary.

1.2. The second problem is to either create or to identify a candidate representation. I make no attempt to define a new representation, but I do evaluate two existing
representations, one created specifically for instructional development and the other for general software development, in the context of the established definitions. These representations are offered as answers to the first auxiliary question.

2. The second subproblem explores the feasibility of using the candidate representations to implement dynamic CBI components. Three specific kinds of dynamic components are considered: First are instructional controls that implement the method upon which the instruction is based. Second are the user controls that implement the user interface. Third are the models and simulations at the center of experiential learning. I demonstrate the feasibility of implementing dynamic CBI components with the candidate representations by creating a set of CBI examples, which entails solving three subproblems.

2.1. The first problem is determining how a description of a dynamic CBI component is physically expressed in a candidate representation. It is possible but not practical to design these instructional components manually with “paper and pencil.” A manual solution makes editing the design tedious and precludes an automated translation into executable CBI components. It is more practical to use a software solution to create and maintain CBI designs. I use existing software when possible and create new software when needed.

2.2. Once the designs are created, the next problem is converting them into a machine-executable format that can be incorporated into CBI. Although it is possible to do this manually, doing so risks introducing additional, human-caused
errors. I solve this problem by creating software that automatically and consistently translates the designs into a suitable executable form.

2.3. Finally, given the means for designing and translating dynamic CBI components, a set of instructional examples complete the practical demonstration. The examples illustrate how instructional controls and instructional models and simulations can be implemented in the candidate representations. The examples answer, at least in part, the second auxiliary question.

**Definitions**

Instructional development consists of at least five major activities: (1) analysis of the setting and learner needs, (2) design of a set of specifications for an effective, efficient, and relevant learner environment, (3) development of all learner and management materials, (4) implementation of the resulting instruction, and (5) both formative and summative evaluations of the results of the development. (Gustafson & Branch, 2002, p. xiv)

Depending on the size and the scope of the instruction, many people may be involved with it during each activity. The instruction’s form or representation is a function of the current activity and, possibly, of individual preference. Equivocal representations are ambiguous and, therefore, subject to multiple interpretations. The ambiguity implies that anyone involved in the instructional development process could choose an interpretation that deviates from the meaning intended by the original developer. In the case of CBI, the risk of misinterpreting equivocally represented instruction is perhaps greatest when a programmer, with limited training in instructional theory and technique, must translate the representation into a computer program (the fourth activity in the Gustafson and Branch, 2002, list). An unequivocal representation,
with a single interpretation, is a more suitable starting point for program development, and is a more desirable vehicle for carrying instructional information through all development activities. Eckel (1993) equates an unequivocal representation with a computer program, and thereby operationally defines an unequivocal representation as one that is computable.

Computability

Following common practice, I define computability in terms of a Turing Machine (TM), which is a theoretical model of a computer. A TM consists of six components: (a) an input alphabet, (b) a tape divided into cells, (c) a tape read/write head, (d) an output alphabet, (e) a finite set of states, and (f) a program consisting of a set of rules (Cohen, 1997, pp. 435-436). Although TMs consist of simple components, they are complex and a familiar metaphor is useful. A TM is similar to a roadmap: the cities correspond to states (component e) and the (one-way) roads between them to the program rules (component f). Figure 1 illustrates a simple TM with four states (the circles) and a program consisting of three rules (the labeled arrows).

Figure 1. A TM with four states and three program rules.
Each program rule consists of three elements: (a) the input character read from the tape, (b) the output character written to the tape, and (c) the tape movement (either left or right). Program rules define a crucial TM characteristic: TMs are deterministic, meaning that rules leaving the same state may not begin with the same input character. That is, a TM’s current state and input uniquely specify the next program rule that executes.

Informally, an operation is computable if it is possible to construct a TM that performs the operation (Cohen, 1997, p. 601). TM programs are precise: there is only one way to interpret a program rule – there is no ambiguity about what actions to perform, nor is there any choice about the next step to take. This precision implies that a computable representation consisting of computable operations must have exactly one interpretation and, therefore, must be unequivocal. To be useful, it is not enough that a representation be unequivocal, it must also be easy to understand the behaviors that it represents. Although TMs are constructed from few and simple components, they are challenging to understand, which makes them unsuitable as a primary representation.

**Primary versus Secondary**

Distinguishing between a primary and a secondary representation is more challenging than establishing that a representation is unequivocal. The difference between a primary and a secondary representation is most evident in the different ways that they are used. In the same way that a storyboard allows “the director, or anyone else concerned with the production . . . [to] see the plot of an entire movie spread out in front of him. . . . [and to] see at a glance” (Finch, 1973, pp. 82-84) the structure and events of the movie, subject matter and the instructional method are apparent in a well-formed
primary representation. This makes a primary representation an appropriate medium for designing, evaluating, refining, and communicating instruction. A primary representation may consist of multiple subrepresentations, including an abstraction of the subject matter (e.g., a storyboard or a script) and a specification of the dynamic controls (e.g., a flowchart or a list of rules). The controls govern the instructional sequence, and regulate feedback, scaffolding, and diagnostics as directed by the underlying instructional methods.

A secondary representation incorporates all of the information available in the primary representation but may do so in a translated form. Within the CBI, the subject matter may remain distinct (e.g., as text or an image) or it may be translated into a dynamic form (e.g., a simulation or other program). However, it is always necessary to translate the controls into computer-executable actions. In this way, the subject matter and the instructional method that are easily discernable in the primary representation, shape the secondary representation while dissolving into it. A full translation from the secondary representation back to the primary is not always possible, and may be difficult or costly when it is possible.

Similar to the Gustafson and Branch (2002) activities noted previously, Eckel (1993) specified an instructional development process that consisted of four phases: (a) design and production, (b) preimprovement, (c) delivering of instruction, and (d) revision. Instructional experts design and refine instruction during the initial steps of both design models and frequently record and communicate their results using a primary notation. In both models, the activities may begin linear but then form an endless cycle
that consists of delivery, evaluation, and revision activities (see Eckel, 1993, Figure 1, p. 2; Gustafson and Branch, Figure 1, p. 3).

Eckel (1993) further asserted that evaluation and revision rely on two key CBI requirements. The first requirement is that a “teaching function is laid down to the last detail and can therefore be criticized” (p. 1). Expressing the instructional method in an unequivocal or computable form satisfies this requirement. The second requirement is for “the recording of instructional results” (p. 3) during instruction. Computers can easily and accurately capture every action that a learner makes and then correlate the actions with the current instructional content and method. A cycle of continuous instructional improvement is only possible when both of these requirements are satisfied.

However, both the teaching function and the instructional results are products of the computer program, that is, of the secondary representation. If the primary representation was faithfully translated into the computer program (i.e., into the secondary representation) and if there were no subsequent changes to the program, then the primary representation may again be used during many of the development activities. Conversely, if the two representations have drifted apart, then modifying and retranslating the primary representation risks losing changes made directly to the secondary representation. Furthermore, it may prove too difficult and too costly to translate the secondary representation back to the primary. A single representation, one that clearly presents instructional intent and that directly implements the instruction, solves the problems caused by multiple representations.
Definition Summary

Computability is sufficient, though not necessary, to establish that a representation is *unequivocal*. Furthermore, an automatic and consistent mechanism that translates the representation into a computer-executable form is sufficient to establish computability. A representation is *primary* if (a) it embodies a notation sufficiently rich to describe the dynamic behaviors of CBI and (b) its use as a CBI design language or as a design language in a similar domain, is demonstrable in the literature. A practical demonstration of designing and implementing instruction with a candidate representation will operationally establish the notational requirement.

Procedure

Eckel’s (1993) observations underscore a contemporary problem with CBI development: The representations currently in use are either equivocal and primary, or are unequivocal and secondary. It is difficult to assess the impact of this situation or the benefits of its solution on instructional development generally and on CBI development specifically. These and other questions arise naturally from a consideration of Eckel’s (1993) observations, but the one upon which all others rest is clear: Is it possible to define and to implement an unequivocal primary instructional representation?

The solution to a similar problem in software engineering suggests a solution to the problem of identifying an unequivocal primary instructional representation. Fowler (2004) suggested that the driving force behind the use of graphical modeling languages in the software industry “is that programming languages are not at a high enough level of abstraction to facilitate discussions about design” (p. 1). The meaning of a computer
program is difficult to see once it is expressed as programming statements. To solve this problem, software engineers use graphical design and modeling languages. Similarly, graphical design and modeling languages can serve as unequivocal primary representations for the specification of CBI behaviors.

Software engineers and instructional developers create one or more diagrams during analysis and extend them during the other development activities. However, when a programmer manually translates a diagram into a computer program, the diagram remains the primary representation and the program is secondary. Furthermore, any manual translation process is subject to two kinds of errors: Equivocation errors occur when the programmer misinterprets the meaning of a diagram, and implementation errors occur when the programmer incorrectly translates a diagram. Manual translation is a major obstacle to using diagrams as a primary unequivocal representation.

Automatically translating the diagrams overcomes this obstacle. Automatic translation allows the diagrams to retain their role as a primary representation while also serving as an unequivocal, executable representation. Like any program, the diagrams may be flawed and require correction, but once corrected, the diagrams are no longer subject to equivocation or implementation errors. Like any compiler, the translators may also be flawed. When a conventional compiler is flawed and cannot be repaired or replaced, programmers modify the program to “work around” the flaw; in essence, the compiler effectively defines the unequivocal meaning of a program. Similarly, the automatic diagram translator is the final, but consistent, judge of the diagram’s meaning.
Aside from defining an unequivocal primary instructional representation, the diagrams may play two additional roles. First, they define a notation or set of symbols that an instructional developer uses to construct an external representation of an instructional design problem. Second, the diagrams constitute a “problem space” in which instructional design problems may be solved. Jonassen (2000) maintained that “problem solving requires some activity-based manipulation of the problem space” (p. 65). The diagrams and the symbols they contain form the basis of a manipulatable problem space. I explore two graphical design language candidates: PEAnets and the Unified Modeling Language (UML).

Merrill and the ID² Research Team (1993) introduced PEAnets\(^1\) as “a knowledge structure consisting of processes, entities and activities (or actions in some references) related in such a way as to provide an integrated whole” (p. 5). Merrill (1999) later specifically proposed using PEAnets as a language for specifying instructional strategies and simulation engines. They also acquired their current, semigraphical appearance at that time. Alternatively, the UML is a contemporary, general-purpose software modeling language that “was born out of the unification of the many object-oriented graphical modeling languages that thrived in the 1980s and early 1990s” (Fowler, 2004, p. 1). Although created for and capable of representing the structure and the behavior of general software systems, serving as an explicit instructional strategy and simulation design language is a less-studied role for the UML.

\(^1\) Variations in the capitalization and the hyphenation of the acronym PEAnet occur in the cited literature, but all variations refer to the same concept.
PEAnets are currently a research language with little automated support. Drake (1998) developed IDVisualizer™, a simulation tool whose inputs are derived from a PEAnet, as a part of his Ph.D. research. However, to use this tool, an instructional designer must externally organize the processes, entities, and activities of a PEAnet and then manually enter the results into IDVisualizer. Drake (1998) implemented the tool in what is now an outdated version of Asymetrix™ Multimedia ToolBook, which is necessary to run the final simulation. Unfortunately, IDVisualizer does not operate in current versions of ToolBook. To fill this void, I created an editor/translator based on published examples of the language (Merrill, 1999). My editor/translator closely follows the 1999 notation, but it was necessary both to extend the published notation and for the tool to provide additional views of the PEAnet.

In contrast, the UML is widely used throughout the software industry and many tools, both commercial and open source, are available for creating and editing UML diagrams. Beginning with the UML 2.0 standard, compliant tools must support the importing and exporting of the diagrams as text files in the extended markup language (XML) metadata interchange (XMI) format (Lear, 2000, p. 26). I use a commercial UML editor to draw the diagrams and then export them as XMI files. My UML compiler reads the exported files, reconstructs the UML information as an in-memory tree, extracts the dynamic components, and translates them into an executable engine. The resulting engine operates independently of the compiler and of the UML editor. I created four instructional systems to demonstrate these tools and to explore the feasibility of using PEAnets and UML diagrams as an unequivocal primary representation.
Instructional Demonstrations

Drake (1998) utilized a canal lock example to demonstrate the feasibility of using PEAnets to design and implement instructional simulations. Merrill (2001) later referred to this example while describing the “Components of Instructional Design.” I based the first two demonstrations on a subset of the Drake (1998) and Merrill (2001) canal lock example, creating versions of the canal lock instruction both with PEAnets (Appendix B) and with UML state machine diagrams (Appendix C). Both versions model the behavior of a boat navigating through a canal lock but lack the instructional features that Merrill (2001) described. This simplification results in a “path simulation” (Gibbons & Fairweather, 1998), which has a “single correct path that students. . . [must] follow to accomplish a solution” (p. 100), and that displays “a corrective feedback message” (p. 300) whenever students attempt an action that is off the correct path.

The full canal lock example (Figure 2) contains five active components: the boat, the value, the water level in the lock, and the two gates. The simulation begins with a boat below the lock in the canal, with both gates and the valve closed, and with the lock full (i.e., the water level is high). The user navigates the boat through the lock into the lake by pressing the buttons that control the position of the boat, the valve, and the gates; the position of the valve controls the water level in the lock.

The user opens the valve (bottom left oval) to drain water from the lock, lowering the water level. Once the water levels in the lock and in the canal are the same, the user can open the lower (left) gate and move the boat into the lock. The user must close the lower gate before closing the value, which fills the lock. The user may open the upper
(right) gate when the water levels in the lock and in the lake are the same, and then move the boat into the lake. Although the canal lock example is a path simulation, it consists of multiple, intersecting paths, and the simulation can switch paths at any intersection. For example, once the boat is in the lock, the user may reverse its direction and move it back into the canal. Nevertheless, the natural constraints of the system (e.g., not being able to move the boat through a closed gate) are always enforced. Simulations, even one as simple as the canal lock, allow learners to explore and experience these constraints in a safe environment.

The second set of demonstrations, implemented with PEAnets (Appendix D) and UML state machine diagrams (Appendix E) respectively, rely on a high resolution, high fidelity (Gibbons, 1998/2001) model of a complex physical device, a personal MP3 player (Figure 3). Resolution and fidelity are measures of the detail and the faithfulness

Figure 2. The canal lock demonstration.
of a model when compared with the modeled entity or environment. An instructional approach suitable for training someone to use an MP3 player quickly and accurately is also suitable for training users of complex medical devices, navigation instrumentation, automated process control, and so forth. The MP3 player demonstrations are based on two instructional models: Four-Component Instructional design (van Merriënboer, 1997) and Model-Centered Instruction (Gibbons, 1998/2001).

The Four-Component Instructional Design model (4C/ID) proposes that complex cognitive skills are composed of many constituent skills. The 4C/ID model further distinguishes between recurrent and nonrecurrent constituent skills. Recurrent skills must be performed quickly with few or no errors, may be performed simultaneously with other constituent skills, and are generally not easily adapted to new situations (van

![Figure 3. The MP3 player demonstration.](image-url)
Merriënboer, 1997, p. 91). Conversely, nonrecurrent skills are more broadly applicable but are performed more deliberately and with more focus. The MP3 player instruction focuses on training the recurrent skills necessary to run the device quickly and accurately.

The 4C/ID model also distinguishes between part-task and whole-task training. During part-task training “the learner is taught only one or a very limited number of constituent skills at the same time” (van Merriënboer, 1997, p. 95). Alternatively, whole-task training values authenticity and presents complete tasks as seen by an expert practitioner. However, “it may be impossible to find an ‘authentic’ case that is simple enough to start instruction with. . . . [therefore] a combination of part-task and whole-task approaches is typically required” (van Merriënboer, 1997, p. 96). The MP3 player demonstration utilizes part-task instruction to introduce the player’s controls and their basic operation, and then switches to whole-task instruction to train the user to operate each feature of the player.

Following the design approach prescribed by model-centered instruction (MCI), a model of the MP3 player is the central component of the whole-task instruction. Describing MCI, Gibbons (1998/2001) asserted that “the most effective and efficient instruction takes place through experiencing realia or models in the presence of a variety of instructional augmentations designed to facilitate learning from the experience” (p. 512). A key instructional augmentation is the posing of “one or more carefully selected and sequenced problems that are within the learner’s ability to solve, defined with respect to the model” (Gibbons, 1998/2001, p. 526). The MP3 player demonstration presents a sequence of problems to the learner. Each problem requires the learner to explore a
different aspect or feature of the MP3 player. By taking this approach, MCI recognizes that “the information that even a simple model contains is extensive and cannot be discovered by a learner all at once” (Gibbons, 1998/2001, p. 526). Using problems to focus the learner’s attention on specific features of the MP3 player, MCI supports the learner’s construction and organization of knowledge about the player in a whole-task-oriented context.

Although MCI provides the underpinnings for the MP3 player instruction, especially the whole task-instruction, the demonstration only utilizes a subset of the MCI model. MCI also specifies a learning companion, another augmentation that provides “assistance to the learner in the formation, conduct, and interpretation of otherwise self-directed learning actions” (Gibbons, 1998/2001, p. 517). Some instructional components of the MP3 player demonstration provide a limited amount of learning companion functionality. For example, a narrative introduces each problem and guides the learner’s exploration of the MP3 player, and the instruction provides a small amount of learner feedback. These features implement some basic learning companion functionality but lack the sophistication and the full range of operations of a true companion.

Finally, the two instructional demonstrations (Figure 2 and Figure 3) make clear the need for CBI to include both user interface and visual representation components. However, neither PEAnets nor UML diagrams include features for describing the appearance of either of these components, nor does either of my systems directly address these issues. Nevertheless, a practical demonstration of instruction described with either language is not possible without a user interface and a visual representation.
Dynamic Versus Visual CBI Components

Numerous practitioners divide the organization of instructional models and simulations into two distinct components. Winsberg (2003), for example, described computer-based models by first observing that they “depict the time-evolution of the system being studied” (p. 107), and later noting that they often use “graphical techniques . . . to transform the output into graphics and videos that sometimes resemble images such as might be produced by laboratory instruments trained upon the system in question” (p. 108).

Similarly, Gibbons and Fairweather (1998) observed that instructional simulations, which they maintain are just another form of CBI, mimic or “create three things: a model of a system, a model of an environment, or a model of expert behavior” (p. 24). They later asserted that simulations communicate through a “representation” that may include “all of the sensory channels: graphic and motion graphic, tabular, verbal and non-verbal auditory, tactile, kinesthetic, and olfactory” (p. 318).

At the same time, Gibbons (1998/2001) suggested that instructional “models are assumed to consist of two parts: (a) a set of abstractions of cause-effect or time-space sequence, and (b) a media representation of the abstractions” (pp. 514-515). Finally, Merrill (2001) asserted, “Almost all subject matter content can be represented by entities, . . . actions, . . . processes, . . . and properties” (p. 295). He then added that, “A portrayal is how the student senses the component. A given portrayal may be symbolic, verbal, graphic, video, animation, audio, olfactory, or kinetic” (p. 296; see also Merrill, 1998, 1999, 2000).
The forgoing observations make it clear that instructional models and simulations consist of at least two distinct components: the logic that calculates the trajectory of the system based on established laws and initial conditions, and an interface that presents that trajectory to the learner. Although the portrayal or representation is a vital part of any instructional model or simulation, PEAnets and UML diagrams only describe the dynamic components of instruction. However, those dynamic components are not limited to the modeling or trajectory-calculating logic.

Gibbons and Fairweather (1998) noted, “The concept of an instructional simulation sharing control of and working in coordination with an instructional strategy is becoming increasingly important” (p. 24). The diagrams also provide a representation for specifying and communicating instructional strategies and controls. Following this approach, developers may implement the two CBI components as two independent steps: (a) They specify and implement the dynamic components with diagrams, and (b) they implement the appearance and interface with traditional programming languages.

Contemporary authoring tools typically allow developers to control both the instructional display and sequence (Gibbons & Fairweather, 1998). Many of these same authoring tools now provide a programming interface sufficiently robust to support the execution engines of both languages. A plausible process for creating CBI is: (a) Create the visible components of instruction with an authoring tool, (b) create a set of PEAnets or UML diagrams with an appropriate diagramming tool, (c) translate the diagrams into an executable engine, and (d) implement the control engine in the tool’s native programming language. As an additional advantage, PEAnets and UML diagrams can
also support the necessary dialog between the CBI programmer and the instructional
designer: The programmer describes the available portrayal methods that the designer
may utilize in the diagrams, and the designer requests additional methods of the
programmer as needed. I chose a similar development process but implemented the
demonstration examples in the Java™ programming language rather than using an
instructional authoring tool, and created my own display and sequencing system.

**Shells, Frames, and Instructional Delivery**

The need to establish both computability and notational sufficiency through a
practical demonstration implies the further need for a delivery mechanism. The delivery
mechanism must support a portrayal or representation, the PEAnet- or UML-controlled
instructional sequence, and a user interface. To maintain compatibility with the other
components – the models (the canal lock and the MP3 player), the user interfaces, the
translators, and the engines – I implemented the delivery system in Java. This approach
offers several advantages.

The Java programming language is portable and architecturally neutral (Gosling
& McGilton, 1996). Portability means that Java programs will run on a wide variety of
computer systems – any system for which the Java Runtime Environment™ (JRE) has
been written, and neutrality insures that all program data is platform independent. Java
provides platform independence by defining data types at the language level rather than
at the hardware level. The language (a) specifies the size, byte order, and format of
numerical data and (b) defines a common structure for strings and other textual data.
CBI, when developed with Java, is platform independent and can be deployed on a wide
variety of computers. Finally, Java is a stable, open-source language: In 2007 Sun Microsystems, which holds the Java trademark, “announced that future versions of Java will be available under the General Public License. . . . a very courageous move that will extend the life of Java by many years” (Horstmann & Cornell, 2008, p. 12).

I based my delivery system on a construct variously called a shell or a frame. Merrill, Li, and Jones (1991) defined a system, which they call a transaction shell, that consists of two subsystems: an authoring environment and a delivery mechanism. They describe the system simply as “a piece of computer code that, when delivered to a student via an appropriate delivery system, causes [an instructional] transaction to occur” (p. 7). Alternatively, Gibbons and Fairweather (1998) used the term frame to denote a similar entity that they describe as “an authoring object which allows the author to define either or both: (1) an expressive display element (something which can be seen, heard, or felt), and (2) a pattern of unseen computer branching logic” (p. 59).

I developed a shell delivery system similar to frames and transaction shells, but my shells only implement logic for loading and displaying content. Figure 4 illustrates the major shell system classes as a UML class diagram. The InstructionManager class and its subclasses assume all responsibility for implementing the instructional controls and, therefore, the instructional sequence. Several kinds of presentation shells appear on the right-hand side of the diagram. The most general is a media shell that can contain one or more media-related panels that display images and text (plain, HTML, and RTF).

Developers can create specialized panels, (e.g., the AudioPlayer) by realizing the ShellPanel interface. Similarly, they can create specialized shells by extending either the
Shell or the *MediaShell* classes. The *ShellPlayer* class defines a general interface between the operating system and the shell system. The *InstructionManager* loads and sequences the instructional presentation and specialized managers implement specific instructional controls. The *GeneralManager* class defines a set of generic functions, which support PEA.net and UML instruction managers. (The MP3Engine class uses the open source MP3 player, JLayer 1.0, downloaded from www.SourceForge.net).

Figure 5 is a UML diagram illustrating how the shell system articulates with the generated PEA.net engine through CBI components (i.e., the MP3 player example). Each tabbed rectangle represents a UML package, which corresponds to a Java package. The
shell and PEAnet systems are independent of one another but can function cooperatively. The MP3 player example includes three separate PEAnets. The first implements the instructional controls, the second drives a review of the MP3 player’s controls, and the last models the MP3 player itself. The PEAnet translator creates three classes, AbstractMP3Control, AbstractButtonTest, and AbstractSimulation, which correspond to the PEAnets and which interface the visible CBI components with the PEAnet system.

Figure 6 is a UML diagram illustrating how the shell system articulates with the translated UML state machine diagrams. The full UML system is composed of two subsystems: the translator and the UML package. The translator, which is not illustrated,
translates the exported UML diagrams into an executable engine composed of objects instantiated from classes in the UML package. The View class plays the role of the portrayal or representation. A class named MP3Manager appears in both mp3player packages; these classes share a great deal of code but are not the same class.

The shell player package, used in both examples, is not formally a part of my research; however, it is a necessary part of the demonstrations. The package defines a host environment that contains and displays individual shells. Many of the shells present instruction that is traditional in the sense that it is not based on either PEAnets or UML.

Figure 6. UML class diagram showing the packaging of and articulation between the shell presentation system, UML state machine library, and the MP3 player example.
diagrams. But the two MP3Manager classes (Figure 5 and Figure 6) are driven directly by PEAnets and UML diagrams respectively.

Using the modular, object-oriented design allows easy modification and extension. For example, the shell system does not currently provide for Eckel’s (1993) “recording of instructional results” (p. 3), but this feature can be easily added to the instructional manager. However, I deliberately chose to make the demonstrations “safe” and “simple.” The demonstrations are safe in the sense that they do not access the host computer’s file system, and they are simple in the sense that they do not maintain any learner information on the server. This safe and simple design eases the task of delivering the demonstrations via the World Wide Web by simplifying the Java Network Launch Protocol™ (JNLP) files used to download the demonstrations. The alternative is to create JNLP files that request additional permissions on the host computer. Such JNLP files must be digitally signed and the signing key must be registered with a third party, which is costly.

The PEAnet editor/translator, the UML diagram translator, and the uml package, together constitute the main body of my research. I couple these two components with the shell player system and the MP3 player examples to complete the practical demonstrations. The practical demonstrations show that it is possible to specify CBI behaviors with diagrams and automatically translate the diagrams into computer-executable engines. The two main components also satisfy several objectives established at the beginning of the research.
Research Objectives

“The engineer, and more generally the designer, is concerned with how things ought to be—how they ought to be in order to attain goals, and to function” (Simon, 1998, pp. 4-5).

“Engineers, to carry out their task . . . must work to very concrete objectives” (Vincenti, 1990, p. 213).

The success of an engineering or design project is the degree to which it solves the stated problem and how well it attains the concrete goals and objectives established at the onset. I set for myself the following goals and objectives at the beginning of the research reported here.

1. Establish that PEAnets and UML diagrams are unequivocal by demonstrating that they are computable. In harmony with Eckel’s (1993) observation that it is “the programmer who develops an unequivocal . . . representation” (p. 7), establishing that both diagrams form, in the limited context of instructional simulations and controls, a computable programming language, is an appropriate method of demonstrating that they are unequivocal. Computability is sufficient to demonstrate that both diagrams are unequivocal but it may not be necessary. Nevertheless, the tools needed for a practical demonstration of computability are also necessary for realistically creating instruction from the diagrams. This approach and practical necessity entail several subobjectives:

1.1. Develop an automatic translator for each diagrammatic language. The translator automatically and consistently converts an instructional design,
specified as a PEAnet or UML diagram, into an instructional simulation engine or into a control system that reflects the underlying instructional method. The translator operationally establishes computability and reduces implementation errors.

1.2. *Create a PEAnet editor*. Objective 1.1. implies the need for a PEAnet editor: Translating a PEAnet into an executable form implies that it must first exist in a machine-readable format. Drake’s (1998) approach of manually entering and updating PEAnet information reinforced the current distinction between a primary and secondary representation. A general-purpose drawing tool, which represents drawing elements, as lines, boxes, and text, does not satisfy the need established by 1.1.: A general drawing tool does not assign sufficient meaning to the primitive drawing elements to support the translation process. Furthermore, an editor will support Jonassen’s (2000) process of problem solving, applied to instructional design, by allowing a designer to easily manipulate a representation of the design problem expressed as a PEAnet.

1.3. *Identify an appropriate UML diagramming tool*. Objective 1.1. similarly implies the need for a UML diagramming tool. Unlike PEAnets, UML diagrams and diagramming tools are currently in wide use. UML editors are available that represent the UML notation at a sufficiently high semantic level to allow automatic translation. An appropriate UML diagramming tool will also support Janassen’s problem solving by manipulation.
1.4. *Create a UML translator.* Many modern UML diagramming tools automatically generate code in several computer-programming languages. The most commonly generated code is a class framework created from class diagrams (such as Figure 4, p. 23). At least one tool does generate Java code from a UML state machine diagram; however, at the beginning of my research, the tool did not support many of the complex state machine features needed for the demonstration programs. Aside from the need to export the diagrams from the diagramming tool, the translator must be completely automatic. The final executable must operate independently of all UML diagramming tools.

2. *Develop a complex instructional example to demonstrate the notational strength of PEAnets and UML Diagrams.* This example will fulfill the “national richness” requirement set forth for establishing that the two languages are primary. I address the second requirement, the historical use as a design language, later in this paper.

3. *Develop diagnostic and debugging support.* Diagnostic and debugging systems are not necessary to demonstrate that a representation is unequivocal or primary. However, people, especially when solving complex problems, make mistakes, and a practical system, which is suitable for day-to-day instructional development, anticipates this by providing error detection and reporting. Furthermore, I believe that these systems are essential for creating large, complex programs like the MP3 player demonstrations.

4. *Support the most common instructional platforms.* Although many operating systems are currently in use, four dominate the classroom, office, and home: Windows™,
Unix™, Linux, and the Apple™ Macintosh™. For example, Nicholas Negroponte launched The One Laptop Per Child (OLPC) project in 2005 with the goal of providing inexpensive computers to students; the OLPC computer, the XO laptop, runs on Linux (Barack, 2008; BBC, 2007a). Furthermore, Intel’s Classmate (Fildes, 2007), a low-cost computer similar to the XO laptop, and Beijing-based Lenovo’s “low-cost computer aimed at the country’s vast rural population” (BBC, 2007b) run on Windows. Likewise, a small set of processors are in wide use. This set of processors is dominated by the x86 architecture, but other, incompatible, processors are also common, including Motorola™ and SPARC™. A flexible, cost-effective CBI solution should easily support multiple operating systems and processors.

5. **Simplify concurrency.** Instructional simulations and controls, like any computer program, consist of a set of tasks. These tasks may be completed either sequentially or concurrently. Simple CBI may not require concurrency but examples that are more extensive surely will, and contemporary graphical user interfaces (GUIs) already rely extensively on concurrent operations. When discussing some of the challenges of implementing MCI, Gibbons (1998/2001) noted that solving concurrency or multitasking problems often “requires computer and programming skills not possessed by the average [instructional] designer” (p. 536). The PEAnet editor and both translators must simplify how task completion order is specified and implemented.


**Conclusion**

During my initial pursuit of an unequivocal primary representation in chapter 1, I presented definitions of an *unequivocal representation* and of a *primary representation*. The definitions will serve, in later chapters, as a standard for evaluating PEAnets and UML diagrams. My pursuit implies the need for a system of automated tools that either directly or indirectly demonstrates the satisfaction of some definitional requirements or that advances the feasibility of developing CBI with PEAnets or UML diagram. Finally, my pursuit has also produced a list of objectives that I will use later to evaluate the success of the research.

In chapter 2, intended for journal publication, I present a detailed examination of PEAnets and the development of a PEAnet software system. I evaluate PEAnets based on the previously stated definitions of unequivocal and primary representations. Based on this evaluation, I conclude that PEAnets are unequivocal, but, following an extensive PEAnet-based CBI example, that they have several limitations that hinder their use as a primary representation and demonstrate that they do not scale well to large instructional solutions.

In chapter 3, also intended for journal publication, I present a detailed examination of UML state machine diagrams, including a software system that translates the diagrams into executable CBI components. From this research, I conclude that UML state machine diagrams are unequivocal. Moreover, I conclude that with appropriate training, UML state machine diagrams are suitable as a primary representation.
In chapter 4, which I will submit for possible publication, I compare the two representations, highlighting their similarities and their essential differences, and analyzing both representations in terms of formal languages. I offer extensions to ameliorate some PEAnet limitations, and describe an alternate approach to the translation process used by both software systems. I conclude the chapter by evaluating both representations for the purpose of designing and implementing instructional behaviors.

Finally, in chapter 5, I summarize the research presented in the central three chapters.

References


CHAPTER 2
AN EXPLORATION OF PEANETS: THE PURSUIT OF AN UNEQUIVOCAL PRIMARY REPRESENTATION:

If you want to build a dog house, you can pretty much start with a pile of lumber, some nails, and a few basic tools such as a hammer, saw, and tape measure. In a few hours, with little prior planning, you’ll likely end up with a dog house that’s reasonably functional, and you can probably do it with no one else’s help. (Booch, Rumbaugh, & Jacobson, 2005, p. 4)

At a time when many design disciplines are benefiting from formalism and automation, instructional technology remains unsure of their value. Four decades ago, Suppes (1969) noted a concern then “beginning to be widely discussed . . . [was] the claim that the deep use of technology, especially computer technology, will impose a rigid regime of impersonalized teaching” (p. 44). More recently, Rieber (1998) admitted during his 1998 Peter Dean Lecture at the annual AECT convention in St. Louis, MO., that he had “long been sensitive to our field disavowing the artistic side of IT and instead overemphasizing . . . its scientific aspects” (para. 3). Ely (1999) summarized these concerns by observing, “Professionals in the field . . . cringe at the more common interpretation of instructional technologists as ‘engineers’ or ‘technicians’. They would prefer to be ‘designers or ‘architects’” (p. 308).

Ironically, architecture exemplifies the harmony of art and engineering. I take the position that a formal approach to instructional design is not necessarily antithetical to an artistic approach and does not necessarily lead to rigidity. More specifically, that in the context of designing and producing Computer-Based Instruction (CBI), such an approach is necessary to solve an outstanding instructional design problem.
In the early days of Computer-Assisted Instruction (CAI), Stolurow (1969) recognized that “a description of teaching in a computer language is less equivocal than one in natural language” (p. 66). Eckel (1993) later reiterated and enlarged the problem when he described the contemporary technique of creating CBI as involving two representations: one primary and one secondary. Instructional experts, he maintained, produce the primary representation that contains the subject matter and the instructional method. Programmers translate the primary representation into a program or secondary representation. In his view, the problem with this process is that the primary representation is equivocal, that is, it is ambiguous and subject to multiple interpretations. In Eckel’s opinion, “It is only the programmer who develops an unequivocal (secondary) representation, i.e. the program” (p. 7). Unfortunately, when the instruction is translated from a primary to a secondary representation, instructional details diffuse into the program. The details become obscure in the program, which, in his assessment, is “absolutely useless” (p. 7) for communication, evaluation, or further improvement.

An instructional design expressed in a representation that is both unequivocal and primary benefits CBI development in three ways. First, it establishes the unambiguous meaning contained in the representation. Second, it eliminates errors introduced into the CBI at the time of translation. Finally, it preserves the instructional details embodied in the design for assessment and future instructional development. To better understand how instructional designers may achieve these benefits, it is necessary to put both dimensions of instructional representation in context and then define litmus tests for determining when a representation is primary and when it is unequivocal.
Primary and Unequivocal

The set of blueprints for a house is a useful metaphor for instructional development. One kind of blueprint specifies the structural framework, another the floor plan, the elevation shows how the finished house looks, and the service plan specifies the wiring and plumbing. Similarly, CBI can be roughly divided into three components: the model data, the external view, and the control logic. This well-known design pattern, the Model/View/Controller (MVC), was used to implement user interfaces in Smalltalk-80 (Gamma, Helm, Johnson, & Vlissides, 1995) and is currently used for the same task in Java™ (Horstmann & Cornell, 2008). Software that is implemented following the MVC design pattern is more flexible and dynamic than when the functions are tightly coupled. CBI development can also follow the MVC design pattern. Specifically, I propose that an unequivocal primary representation is useful for describing and implementing CBI controllers. CBI controllers may define the behaviors of instructional simulations or models, user interface controls, and instructional methods (i.e., the CBI controls based on instructional theory and practice). Following this strategy, the model (i.e., the data) and the view are still implemented in a traditional programming language.

Demonstrating that a notation is primary is the most challenging objective to measure because it is somewhat subjective. A primary representation is an abstraction that allows the viewer to see the design as a whole, to see the relations between the individual components, and to quickly identify the events that account for the instruction’s behavior. Specifically, the subject matter and the instructional method are apparent in well-formed primary representations, which make them appropriate media for
designing, developing, maintaining, evaluating, improving, and communicating instruction. How apparent this information is to viewers is a function of their understanding of the abstraction technique – how familiar they are with the elements used to encode the information. Unfortunately, the most common and familiar abstractions, prose for example, are typically the most equivocal.

The number and type of abstractions used in a primary representation is a function of the type of CBI, the tools available, and the designer’s preference. Typically included are abstractions of the subject matter (text, diagrams, images, simulations, etc.) and of the instructional method (storyboard, script, flowchart, rules, etc.). The instructional method implements the principles used to design the instruction, governs the presentation sequence, and regulates feedback, scaffolding, and diagnostics. Abstractions in the primary representation may also describe the user controls that, in conjunction with the method, determines and implements the actions the learner can perform.

A programmer or a computer program translates the primary representation into the secondary representation, which includes as possible subrepresentations program source code, machine code, or other executable forms. The translation process retains the information present in the primary representation, but converts it into an obscure, arcane form (e.g., source or machine code) suitable for computer execution. Following the translation, the subject matter may remain distinct (e.g., as text or an image) or it may be translated into an executable form (e.g., a simulation or other program). However, the translation process always converts the set of actions specified by the instructional method and the dynamic controls into an executable form.
Their intended audience and use distinguish primary and secondary presentations. Instructional experts work with primary representations and use them to view and communicate instructional intent, while computers process and ultimately execute secondary representations. Unfortunately, these differences are still insufficient to distinguish clearly between the two representations. Scripts and storyboards are Eckel’s (1993) standard for a primary representation, but users must be trained and gain experience before using them effectively. This observation suggests that it is unlikely that any representation, particularly one dealing with complex, abstract, and intangible concepts, can be primary in the sense that all of the information that it contains is completely discernable without some training or experience. Therefore, a practical primary representation must be as intuitive as possible, be easy to learn, and help the viewer to quickly locate the contained information and to understand the overall design.

In general, it is also challenging to establish that a given representation is unequivocal. However, one approach can demonstrate that some representations are unequivocal and at the same time make that representation practically useful for specifying instructional behaviors (i.e., the controller of the MVC design pattern). Stolow (1969) and Eckel (1993) both equated an unequivocal representation with a computer program. Therefore, if a representation is computable in the same way that a computer program is, then the representation is unequivocal. Furthermore, if the representation is computable, it directly and automatically implements the instructional controller.
A representation is computable if it is possible to translate it automatically, consistently, and unambiguously into the actions that implement the instructional intent on a computer. That is, for any element in an unequivocal representation, there is exactly one way to translate that element into a computer-executable operation. From a practical perspective, computability also implies two additional requirements. The first requirement is for a notation that can be stored, displayed, and manipulated by a computer. The second requirement is for a program or a system of programs that can edit and store the representation and that can automatically translate the represented instruction into CBI. Although not required, syntax checking and runtime diagnostics are desirable features.

In summary: A representation is primary if (a) it is intuitive, (b) it is easy to learn, and (c) it helps the viewer to quickly understand the design. A representation is unequivocal if (a) it is possible to automatically and consistently translate it into a directly executable form, and (b) it is possible to edit and store it on a computer. Together, these requirements form the standard against which I judge PEAnets, a candidate unequivocal primary representation.

PEAnet History and Overview

The components named by a PEAnet, processes, entities, and activities, began as different kinds of knowledge frames used to represent knowledge during instructional analysis (Jones, Li, & Merrill, 1990). Merrill and the ID2 Research Team (1993) established the purpose of frames saying, “We propose to represent knowledge with objects that we call frames; each frame has an internal structure (slots which contain
values for the structure) and links to other frames” (p. 6; see also Minsky, 1975, who used the term *frame* to name a semantic network with a similar description and purpose). Merrill and the ID₂ Research Team (1993) also introduced the term “PEA-Net” (later PEAnet) at that time to describe a structure of process, entity, and activity frames used “to automate a variety of instructional interactions” (p. 5). ID Expert, an expert system for helping subject matter experts identify and capture instructional information (Merrill, 1998a), was a successful example of one such automated instructional interaction.

In 1993, PEAnets were presented as a set of tables or frames. Process frames consisted of seven slots or attributes that are represented by seven table columns: (a) name, (b) location, (c) condition, (d) transformation, (e) property, (f) owner, and (g) to value (Merrill & ID₂ Research Team, 1993, Tables 1 & 2). Activity frames were similar except column (d) was named act (Table 3). Entity frames contained five slots: (a) name, (b) properties, (c) symbol, (d) dynamic value, and (e) default value (Table 4).

By 1998, after several years of refinement and simplification, PEAnets began to exhibit much of their present-day semantics (Merrill, 1998b). The following year, PEAnets assumed the semi-graphical format so long suggested by their name (Merrill, 1999). A later PEAnet example (Merrill, 2000, Table 6) presents a slightly altered appearance but is equivalent to the 1999 version. My PEAnet system and examples are patterned after the 1999 version, which divides a PEAnet into two views.

**The Entity Table**

PEAnets present two views of the represented instructional elements. The first view, the entity table, associates properties with each entity and defines the allowed
values for each property. *Entities* are the things (i.e., the nouns) that participate in the CBI and that exhibit time-varying behavior. Each entity has one or more *properties* that are the features or attributes that characterize the entity. Each property has a set of allowed *values* that capture the instructionally relevant and (typically) dynamic characteristics of the entity. The dynamic, time-varying behavior of the system is represented by the property-value changes.

As a part of his doctoral research, Drake (1998) developed IDVisualizer™, a simulation authoring and presentation tool whose inputs are the data contained in a PEAnet. Designers manually enter a PEAnet into IDVisualizer, which does not support editing or automatic translation. He demonstrated the tool by developing a simulation of a boat navigating through a canal lock. That example is reconstructed here from his work and from Merrill’s (2001) example of SHOW, ASK, TELL, and DO.

The example contains five entities: the boat, a valve, the lock, and two gates (Figure 7). The boat entity has three properties: *location*, *direction*, and *level*; location has three allowed values: *in the canal*, *in the lock*, and *in the lake*; direction and level each have two properties: *left* and *right*, and *low* and *high* respectively. The properties and property values associate with the valve and with the gates illustrate that each entity defines a unique scope or namespace. The same property names may appear in the descriptions of multiple entities, but they nevertheless represent distinct properties. Similarly, the same property value names may be used with different properties, while denoting different CBI characteristics. At any given time, the current set of property values completely describes the state of the CBI and its current appearance.
The Network View

The second view is a graphical network of activities, processes, consequences, and conditions. Figure 8 illustrates the network for the canal lock example. Changes are initiated by an activity and propagate through the network from left to right and from top to bottom. Activities are typically learner-initiated and represent the ways that learners can operate the CBI. Each activity triggers or activates a process. Processes are the dynamic agents that change the CBI state by changing the value of a property; the change is shown in the network view as a consequence.

Consequences play a dual role in PEAnets. Their first role is as the externally visible (i.e., instructionally significant) result of a process. Each consequence has a label, a property, and a single property value. For example in

\[ B \text{ location} = \text{in lock} \]

“B” is the label, “location” is a boat property, and “in lock” is one of the allowed values for that property. In this context “=” is interpreted as the assignment operator. A process may implement any appropriate CBI operation (e.g., animating boat movement) as a side effect, but from the standpoint of a PEAnet, its consequence is a property value change.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boat</td>
<td>location</td>
<td>in canal/in lock/in lake</td>
</tr>
<tr>
<td></td>
<td>direction</td>
<td>right/left</td>
</tr>
<tr>
<td></td>
<td>level</td>
<td>low/high</td>
</tr>
<tr>
<td>Valve</td>
<td>position</td>
<td>closed/opened</td>
</tr>
<tr>
<td>Lower Gate</td>
<td>position</td>
<td>closed/opened</td>
</tr>
<tr>
<td>Upper Gate</td>
<td>position</td>
<td>closed/opened</td>
</tr>
<tr>
<td>Lock</td>
<td>level</td>
<td>full/empty</td>
</tr>
</tbody>
</table>

*Figure 7. The PEAnet property table illustrating the entities, properties, and property-values that appear in the boat and canal lock example.*
Figure 8. Canal lock activities, processes, consequences, and conditions.
A second kind of consequence also appears in Merrill’s (1999) PEAnet example:

*show accident message*

Merrill (1999) neither distinguished nor named the two versions, but I refer to the former version as a *valued consequence* and the latter as a *valueless consequence*. That is, a valued consequence, which may have a side effect, always sets a property to the specified value; a valueless consequence only has a side effect and, therefore, does not change the state of the simulation. A valueless consequence does not have a label or a property/value pair.

Valued consequences play the second role by forming conditions. A *condition* is a boolean-valued expression (i.e., it must evaluate to either true or false) that is denoted by a consequence label. Each label is a shorthand notation used in place of the entire consequence, which, in this context, is treated as a test of equality (i.e., “=” is now interpreted as the equality operator). For example, if the label “B” appears as a condition, then the corresponding consequence

*location = in lock*

is evaluated: If “location” is equal to “in lock,” then the condition is true, otherwise it is false. Using the consequence label as a condition results in a compact notation but represents a reduction in expressive power from earlier PEAnets versions.

The original PEAnet notation (Merrill & ID2 Research Team, 1993, Table 1, p. 9) has evolved extensively. The appearance and operation of more recent PEAnets (Merrill, 1999, Figures 17.4 & 17.5, pp. 413-414) is quite different from earlier versions. The original condition notation was more flexible and more expressive as property values
could be examined with many relational operators: \(<, \leq, >, \geq, =, \) and \(\neq\). However, as PEA nets evolved, \(=\) became the only supported operation, which allows \(=\) to be interpreted both as assignment operation and as a test for equality, which further allows using consequence labels as conditions. Another simplification also significantly reduced the expressive power of PEA nets.

Traditional programming languages support two related branching statements:

1. \(\text{IF } \text{expression} \text{ THEN } \text{action-1}\)

2. \(\text{IF } \text{expression} \text{ THEN } \text{action-1} \text{ ELSE } \text{action-2}\)

In both statements, \(\text{action-1}\) takes place if the boolean-valued \(\text{expression}\) evaluates to true. If the \(\text{expression}\) is false, no action takes place in statement 1, but \(\text{action-2}\) takes place in statement 2. The original PEA net definition supported constructs functionally equivalent to both if-statements (Merrill & ID2 Research Team, 1993, Table 2, p. 9), whereas the newer definition only supports the first version (Merrill, 1999, Figure 17.5, p. 414). In the current PEA net network view, the expressions are represented by condition lists, and processes correspond to or carry out the actions.

Each process is “gated” by a condition list: a coma-separated list of conditions where each condition is represented by a consequence label. An empty list is true by default; a list with one condition assumes the logical value of that condition; when a list contains multiple conditions, they are joined by implicit conjunctions (i.e., logical AND operations), and all of the conditions must evaluate to true for the condition list to be true. If the condition list is true, then the process is allowed to run and the property value
change takes place. An additional notational element makes it is possible for a single action to trigger multiple processes.

One process may trigger another (denoted by the downward pointing arrows in Figure 8, p. 44), causing a “chain reaction” of property value changes that propagate through the network. Downward triggering takes place independently of the condition list, that is, if a list is false, the process does not run but it still triggers the next process in the sequence. For example, in Figure 8 the “close valve” process triggers the “fill lock” process, which then triggers the “raise boat” process.

PEAnets have a well-established notation with an associated, well-defined semantics. Are these qualities sufficient to claim that PEAnets form an unequivocal primary representation? I implemented a PEAnet system based on the two PEAnet views and on the semantics described above in an attempt to answer this question.

**Implementing a PEAnet System**

The most visible component of the PEAnet system is the editor, which allows an instructional designer to enter and edit a PEAnet, and to store it in a machine-readable file. The editor also includes the translator that automatically and consistently translates PEAnets into executable controllers. As an examination of a potentially unequivocal primary representation, the extensions made to the PEAnet notation, the features added to solve or ameliorate specific problems, and the instructional demonstrations are more significant.

I implemented two examples using the PEAnet system. The first example is the canal lock described previously. The complete PEAnet for this example consists of a
short entity table and three pages of activity/process networks. The second example is a complete CBI centered on a complex electronic device: a portable MP3 player. This example consists of three separate PEAnets: the first modeling the MP3 player, the second controlling a review of the player’s controls, and the last sequencing the instruction. Together, the three PEAnets are thirty-six pages long. The need for many of the extensions added to the PEAnet system only become apparent when developing an extensive CBI system such as the MP3 player.

**Entity Extensions**

Initially, entity frames (Merrill & ID2 Research Team, 1993, Table 4) specified, or could specify, the default or initial value of a property. This feature was removed from later entity tables (Merrill, 1999, Figure 17.4). Correct computer programs, however, require that all data must be appropriately initialized, either explicitly or implicitly. I took a simple approach of using the first value appearing in the values column (see Figure 7, p. 43) of the entity table as the initial value. This simple approach is sufficient in most cases but not all.

Assigning the first allowed property value as the initial value does not create a corresponding consequence. That is often not a problem because processes in the PEAnet typically cycle through all possible property values. However, there may be situations in which a system does not return a property to its initial value but where the corresponding condition (i.e., the label of a consequence assigning that value to a property) is nevertheless needed. This problem is solved simply by allowing an “empty” activity that defines the needed consequences (Figure 9) but that is never triggered. This
feature is useful during early CBI development, before complex consequences that return the system to its initial state are added, and for representing irreversible consequences where a modeled entity never returns to its initial state.

**Activity Extensions**

Activity names are derived from the actions that learners take to operate the CBI. This derivation causes the names to occupy a single, unstructured scope or name space. Each complete activity name must, therefore, be unique as is illustrated in (Merrill, 1999, Figure 17.5, p. 414). I take advantage of the uniqueness of each activity name when converting from Java™ events to PEAnet activities. Whenever a learner operates a Java control, an event is generated; the event handler looks up the activity name in a simple database and sends the corresponding activity to the PEAnet execution engine. Two related extensions allow activities to also originate from within the CBI.

The need in complex CBI for an *internal activity* can arise in two situations. The first situation is an asynchronous change in the CBI system. For example, in the MP3 player, one track completes playback and the next track, if there is one, should begin

![Figure 9](image-url). An empty activity that creates consequences that may be used as conditions.
playing. Similarly, some CBI may require internal timers to present an accurate simulation or to control certain instructional behaviors. Figure 10 illustrates an internal activity triggered by a CBI timer. The use of a leading and a trailing underscore in the activity name is a suggested style to distinguish between internal and external activities, but a designer may select any appropriate name.

The second situation is a special case of the first. It is common for a program to run a set of initialization methods following startup and before turning over control to the user. Complex CBI may have similar requirements. I introduced a special internal activity named \_START\ (Figure 11) that provides this functionality. This activity is automatically triggered during the initialization of the PEAnet engine. Each activity, internal or external, triggers a process that potentially changes the CBI’s state.

**Process Extensions**

When one process triggers another, the most commonly needed behavior is for the triggering process to complete before the triggered process begins. However, another

---

**Figure 10.** An internal activity triggered by a timer within the CBI.

**Figure 11.** The internal “\_START” activity is triggered once at CBI startup.
meaning is possible. For example, the process “fill lock” raises the lock water level, and the process “raise boat” elevates the boat in the lock. If both processes update a portion of the display, which is the case in this example, then executing them at the same time will provide a realistic simulation of the boat floating on the water and rising with the water level as the lock fills.

Although Merrill (1999) did not explicitly address process execution order, his intent seems clear: “When a process is triggered, it evaluates its conditions; if they are true, it executes; . . . it then triggers the next process in the chain” (Merrill, 1999, pp. 411-412). I modified the PEAnet notation to distinguish between, and added buttons to allow a designer to select, the two execution orders. The more common sequential process execution is denoted by a single-lined downward pointing arrow (Figure 8, p. 44) and concurrent execution is denoted by a double-lined downward pointing arrow (Figure 8, p. 44). (Concurrent execution is the illusion that a computer’s central processing unit, or CPU, is simultaneously executing multiple tasks; rapidly switching the CPU between tasks maintains the illusion.) For example, the “close valve” process triggers the “fill lock” process, which then triggers the “raise boat” process. Once started, all three processes run concurrently, rapidly interleaving their instructions to produce a smooth animation.

Both sequential and concurrent processes typically affect a property change. Properties and their allowed values are bound to entities and are defined within their scope. To affect a property change, a process must also be bound to an entity. However, PEAnets do not include specific notation to denote the binding but instead bind activities
and processes to entities by name. Each activity name must include the associated entity name (Merrill, 1999, Figure 17.5, p. 414); the entity name also often appears, perhaps in an abbreviated form, in the process name. I used a similar approach when implementing PEAnets, often naming activities after the controls that learners use to operate the CBI, and binding processes to entities by name. In my implementation, the entity name must be included as a part of the process name.

My final process extension is the addition of a built-in process named \_Break that conditionally interrupts or breaks a process sequence. Inserted into a sequence, a \_Break process triggers the next process only if its condition list is false – interrupting the sequence if the list is true. In some cases, interrupting the sequence simplifies subsequent condition lists: If condition $A$ appears in numerous condition lists, then breaking on not-$A$ allows the developer to remove condition $A$ from subsequent lists, simplifying them.

**Consequence Extensions**

I did not change the basic consequence notation or semantics, but I did generalize consequences without values to allow them to call any named method, which can accept any number of parameters of any available data type. I added a dialog box to the editor that allows designers to choose between valued and valueless consequences, and to configure each version. Designers deploy the editor by double-clicking the consequence they wish to edit in the network view, and then selecting the appropriate consequence type with the two radio buttons. Figure 12 illustrates the valued consequence editor. The designer selects the appropriate property and value from the two pull-down lists. Figure
13 illustrates the valueless consequence editor. The designer must enter a method name (defined in the CBI) and the actual parameters.

**Condition Extensions**

I did not make any changes to the condition notation or semantics. However, I did attempt to ameliorate a weakness with the notation. It is relatively easy to understand a condition list when the PEAnet fits on a single page, as it does in Figure 8 (p. 44). However, when the network view spans multiple pages, as it does in the case of the comparatively small canal lock example, it becomes difficult to find the consequences corresponding to the labels in a given condition list. The result is a considerable increase in the amount effort needed to understand the conditions under which a process will run. Viewing the PEAnet with an automated tool, rather than on paper, reduces but does not eliminate the effort.

Wood (1998) maintained that “to be usable, a user interface must provide access to the functions and features of an application in a way that reflects users’ ways of thinking about the tasks that a potential application will support” (p. 2). Intuitively, an
instructional developer attempting to understand the logic governing whether a process will execute or not, anticipates finding the necessary information in one location near the process. A PEAnet condition list, located adjacent to its associated process, contains the consequence labels that represent this logic, but the information is contained in the individual consequences, which are scattered throughout the network view.

A partial solution to this problem is suggested by usability engineering. Norman (2002) offered this as a fundamental principle of designing for people, to “make things visible” (p. 13). Similarly, Nielsen (1993) maintained that efficiency is one measure of usability, “The system should be efficient to use, so that once the user has learned the system, a high level of productivity is possible” (p. 26). To help clarify condition lists, the editor collects and displays the information encoded in the condition list: the entities that own each property, the property names, and the values necessary for the process to run. The information is displayed briefly when the mouse pointer hovers over a condition list (Figure 14), or is displayed indefinitely in the condition editor (Figure 15).

Developers launch the condition editor (Figure 15) by double-clicking on a
condition list in the network view. The right-hand panel displays the conditions currently in the condition list, while the left-hand panel lists the remaining conditions defined in the PEAnet. Selected conditions are moved from one panel to the other by pressing the appropriate button between the panels. The challenge of quickly and easily understanding the conditions that must exist before a process is allowed to execute is inherent in the 1999 PEAnet notation. The PEAnet editor, providing additional ways of viewing the condition list, can help clarify the meaning embodied in a set of conditions, but it does not completely solve the problem and several fundamental limitations remain.

**PEAnet Limitations**

Three related, fundamental characteristics detract from PEAnet’s ability to convey instructional meaning clearly: unintended consequences, inadequate conditions, and indistinct states. Individually, each characteristic obscures the contained information to some degree. Acting together, which happens when the PEAnet has relatively few actions but has many processes, their impact is compounded beyond their individual effects. This situation is exemplified by the MP3 player demonstration.

The player has eight controls through which the user can initiate eleven distinct activities. For each activity, the player’s behavior is a function of its current condition or state, that is, what the player is currently doing. The correct behavior is deduced from and implemented by a long sequence of processes and their associated conditions. In each sequence, some processes relate to the present behavior while others are currently irrelevant. Processes, and their consequential property changes at or near the beginning of the sequence impact the conditions and processes occurring later in the sequence.
Unintended Consequences

When an activity triggers a sequence of processes, conditions are evaluated and processes are executed from the top of the sequence to the bottom. A property value changed by one process affects the evaluation of subsequent conditions and, therefore, the execution of later processes. This effect is seen while trying to describe the toggle behavior of the MP3 player `play` button (Figure 16 a). If the player is playing, `mode = play` and condition `B` is false, so `MP3Player Play` does not run, but `MP3Player Pause` does run, which sets `mode = pause`. But the reverse operation does not work: now `mode = pause` and condition `B` is true, so `MP3Player Play` runs, which sets `mode = play`, but then `MP3Player Pause` runs and sets `mode = pause` and pauses playback. The effect of the `Play` process unintentionally interferes with the subsequent `Pause` process. Reversing the order of the two processes does not solve the problem but creates a situation where the player cannot be paused.

![Diagram](a)

![Diagram](b)

*Figure 16. The PEAnet sequencing problem illustrated by the difficulty of implementing a toggle button (a) and the limited problem solution (b).*
The problem can be solved (Figure 16 b) but at the cost of lost specificity. A single process toggles between playback and pause mode. The process *MP3Player Toggle Play* must perform the tasks accomplished previously by two processes. An instructional developer looking at the *Toggle Play* process must consider two possible outcomes when the process executes, making it is less specific than two processes. Merrill (1999) presented the same solution, which is adequate for small PEAnets and for short process sequences. However, as the number of entities grow and the sequences lengthen, it becomes increasingly difficult to ensure that unintended consequences do not introduce errors into the PEAnet and, therefore, into the CBI.

**Inadequate Conditions**

Four traits of the 1999 version of PEAnet conditions effectively reduce their expressive power. The first two are, in part, simplifications of earlier, more robust conceptions. The first trait is the absence of an ELSE clause. It takes at least one additional condition, and perhaps more, to implement the same logic as is possible with the IF-THEN-ELSE statement of a typical programming language. Consider a property that has many allowed values. The statement

$$IF \text{ property} = \text{value-1} \ THEN \ process-1 \ ELSE \ process-2$$

executes *process-2* for all property values except *value-1*. The PEAnet equivalent requires a condition list for each possible property value. The PEAnet can implement the same logic as the IF-THEN-ELSE statement, but the implementation is longer, more complex, more difficult to understand, and more prone to error.
The second trait is the limited number of logical operations allowed when forming conditions and condition lists. Within individual conditions, the current value of a property may only be examined with the equality, or $=$, operator. Many other relational operators are often used in other languages: $<$, $\leq$, $>$, $\geq$, and $\neq$. By excluding these operators, PEAnet properties are constrained to discrete values, and conditions are constrained to strict tests of equality (i.e., range tests are not supported). Furthermore, when combining individual conditions, PEAnet condition lists only support one operation: logical AND. Other languages also support the logical OR and NOT operations, define a precedence or order of evaluation for the three operations, and provide grouping symbols to alter the evaluation order. The additional operators permit clear and succinct specifications that simplify and control system complexity.

Traditional programming languages allow a block or compound statement in the place of process-1 and process-2. Block statements may contain multiple statements, controlling them with a single test. The third trait is that PEAnets only allow a single process to be executed when a condition list is true. Executing many processes requires a similar number of conditions, with a concomitant increase in complexity and providing more opportunities for unintended consequences.

Finally, programming languages permit IF-ELSE statement to be nested:

```plaintext
IF condition-list-1 THEN
  IF condition-list-2 THEN
    process-1
  ELSE
    process-2
```
That is, a process may itself be an IF-THEN-ELSE statement, and this nesting can take place to any desired depth. The condition list of an inner IF-THEN-ELSE statement is never considered if the condition list of the outer statement is false: if \textit{condition-list-1} is false, \textit{condition-list-2} is not evaluated. This structure serves to simplify the overall logic and reduce the complexity of a system.

**Indistinct States**

Each of the previous limitations result, at least in part, from the large number of possible states that a PEAnet may be in at any time. For example, a simple light switch has two states, on and off, and a PEAnet description of the switch requires one property with two allowed values. The PEAnet description of a device with two such switches has four possible states: off/off, off/on, on/off, and on/on. The number of states increases rapidly with each switch added to the device: \(2^{\text{switches}}\). The problem is compounded further by properties with many allowed values.

Within a PEAnet, it is easy to see the state of an individual property, but the overall state of the device, formed from the combined state of all properties, is less clear. There is no single, summary view that allows the reader to assess the state of the device with a glance at the PEAnet description – the states are indistinct. Furthermore, many of the property value combinations possible with the PEAnet description may not correspond to valid device states. Although there are well-known algorithms (Aho, Lam, Sethi, & Ullman, 2007) for removing these unused states, they do nothing to clarify the original PEAnet description. This problem manifests itself in several ways.
The “memory” problem, illustrated in Figure 17, is one example of a problem arising from indistinct states. An MP3 player displays a menu list that allows the user to select tracks for playback based on categories: artist, album, genre, year, or from all songs. The user presses the *select* button to advance from the menu to a filtered list of tracks. However, the user may return to the previous menu or list (i.e., to the previous device state) by pressing the *previous* button. In two cases, the album list and the song list states, the system does not “remember” the previous state or menu item, which makes it impossible to complete the backup operation.

In other languages, the problem is often solved by adding intermediate states that represent the path traversed to reach the current state. However, in a PEAnet, such states must be *synthesized* from properties and from property values, and the synthesis often requires additional, otherwise extraneous, properties. Although each additional property and each additional value causes the number of states to grow exponentially, it is the unintended impact of the properties on the PEAnet conditions that cause the greatest increase in system complexity.

![Figure 17. How to return to a previous state when multiple paths enter the current state?]
PEAnet states are not clearly delineated by sharp, well-defined boundaries, but blur together and overlap. This overlap results in an inter-state coupling similar to the inter-function coupling of functions that share global data. In a process familiar to programmers, one function can alter the shared data in way that induces an error in another function that uses the same data. The complexity resulting from inter-function coupling increases with, but at a faster rate than, the number of coupled functions.

Similarly, in PEAnets the inter-state coupling or overlap is caused by states sharing properties, with the resulting complexity increasing more quickly than the number of shared properties. The complexity becomes acute when, as in the case of the MP3 player demonstration, there are a relatively few activities that trigger a large number of behaviors. When an activity occurs, the correct behavior of the player is a function of the player’s current state and the activity. The appropriate behavior is deduced from and implemented by a sequence of condition lists and processes. Unfortunately, because the states are not distinct and share properties, processes early in the sequence may cause property changes that inadvertently trigger or suppress subsequent processes.

Paradoxically, solving an inter-state coupling problem often requires adding more properties. For example, to change many of the MP3 player’s settings, the user must press the select button twice to commit the change. The second press of the select button triggers the same sequence of condition lists and processes as does the first press. Furthermore, any other button press cancels the change. To implement this behavior, it is necessary to add a property that tracks the selection process: started, in progress, or
finished. Managing this one property adds a large number of conditions to the select button activity sequence.

Coupling-caused complexity is a function of the problem size but increases at a faster rate than does the size. Each activity sequence becomes increasingly complex and fragile as its length increases. At some size, dictated by the length of each activity sequence, the number of properties, and the number of property values, it becomes impractical to further modify the PEAnet. Each modification requires careful and extensive validation, not just of the activity sequence in which it is made, but of all sequences in the PEAnet. The structure of the PEAnet effectively limits its own size.

The self-limiting structure of a PEAnet is analogous to the square-cube law first described by Galileo Galilei (1638/1914). He reasoned that the strength of a structure increases as the square of its size, but that its weight increases as the cube. He concluded that, “you can plainly see the impossibility of increasing the size of structures to vast dimensions” (p. 130). At some point the structure collapses under its own weight. Inter-state coupling is the fundamental PEAnet limitation. It results in excessive complexity, limits the size of PEAnets, and underpins all of its other limitations.

Conclusion

Booch et al. (2005) suggested that the task of building a world-class high rise office building is a larger, more complex, and more demanding task than building a dog house. The requirements and specifications of those paying for the project must be satisfied and followed, while simultaneously maintaining the structural integrity of the building and adhering to local codes. Many people, with a wide variety of skills, must
work together: architects, designers, engineers, and construction workers representing countless skills. Along the way, many problems will be discovered and solved, and some improvements will be made. Although the following assessment is of a specific design discipline, it is generally true of many: “Curiously, a lot of software development organizations start out wanting to build high rises but approach the problem as if they were knocking out a dog house” (Booch et al., 2005, p. 5).

The success of any large, complex design project, including instructional design, demands continuity throughout. Continuity is possible only when the design is specified, communicated, and interpreted clearly and unambiguously. These necessary design tasks require a design representation shared by all participants. In the construction industry, blueprints are the primary representation that is universally recognized. Unfortunately, no completely automatic process exists to consistently convert a set of blueprints into a finished house, and, as anyone who has had a laundry chute in their house plans turn into a service conduit in the finished house knows, blueprints are equivocal.

Fortunately, the prospects for CBI are better. I have demonstrated a tool that automatically and consistently converts PEAnets into instructional controls, which further demonstrates that PEAnets are computable. By demonstrating that PEAnets are computable, I have shown that they are also unequivocal. However, their status as a primary representation is in doubt. The literature demonstrates that they were conceived as an instructional development language and that they have been used for that purpose in the past. Furthermore, I have successfully used them as the basis for implementing complex CBI. Although PEAnets are generally intuitive and easily learned, I have
argued that some PEAnet notation is inherently obscure: It is difficult to quickly and
easily understand some of the represented meaning, and the difficulty increases rapidly as
the size of the PEAnet grows.

Perhaps the greatest lesson that PEAnets teach is that instructional design
languages are possible and beneficial. I still believe that an unequivocal primary
representation can benefit CBI development. And I believe that such a representation is
within the reach of instructional developers.

References


CHAPTER 3

AN EXPLORATION OF THE UNIFIED MODELING LANGUAGE:

THE CONTINUED PURSUIT OF AN

UNEQUIVOCAL PRIMARY

REPRESENTATION:

Objectivist conceptions of learning assume that knowledge can be transferred from teachers or transmitted by technologies and acquired by learners. . . . Constructivist conceptions of learning . . . assume that knowledge is individually constructed and socially coconstructed by learners based on their interpretations of experiences in the world. Since knowledge cannot be transmitted, instruction should consist of experiences that facilitate knowledge construction. (Jonassen, 1999, p. 217)

The “real world” is an appropriate environment for many learning experiences (e.g., learning to drive by driving a real car). However, many environments in which instructional experiences are worthwhile may not be feasibly entered due to danger or cost. Commercial airline pilots, for example, must periodically practice the maneuvers required to recover from rare in-flight emergencies. Other learning environments are inaccessible. The Apollo astronauts, for example, were unable to practice landing on the moon by flying a real spacecraft to the lunar surface or on the earth. Finally, many potential learning environments are abstract. Learners can, for example, experience the operation of a computer’s central processing unit (CPU) by observing the signals on its external connections, but this experience provides little understanding of the program that is currently running on the computer. Alternatively, an abstract environment can represent data and algorithms at a higher semantic level and is more suitable for learning programming than is the real environment of electrical signals.
Whenever a learning environment is dangerous, expensive, inaccessible, or abstract, computer-based instruction (CBI) can provide a more appropriate experiential environment. Even when the real learning environment is an appropriate host for learning experiences, CBI is often an effective adjunct, particularly for special cases such as introductory, preparatory, assessment, or hard-to-create experiences. CBI supports experiential learning in at least two important ways. First, Mayer (1999) suggested that “learning as knowledge construction is based on the idea that learning occurs when a learner actively constructs a knowledge representation in working memory” (p. 144). CBI encourages learners to be cognitively active by mimicking the dynamic aspects of a natural environment, allowing the CBI system to behave naturally during observation and to react naturally to experimentation.

Second, CBI supports experiential learning by providing a problem-solving environment. Problem solving is often described as finding a path or a transformation from a current state to a goal state (Jonassen, 2000; Mayer & Wittrock, 1996; Russell & Norvig, 2003; Simon, 1998). Some researchers also suggest that problem solving involves manipulating an external representation of the problem (Jonassen, 2000; Mayer & Wittrock, 1996). By mimicking the dynamic features of a problem, CBI defines a problem space where the learner can explore and examine potential solution paths. CBI can also provide an external representation of the problem that the learner can manipulate as a part of the problem-solving process. The manipulations can take many forms such as reading and interpreting displays, operating controls in complex sequences, or rearranging symbols according to strict rules.
CBI’s support of experiential learning is based on its ability to mimic the natural dynamic behaviors of the learning environment. The challenge facing CBI developers is how to represent those behaviors while analyzing a problem, and how to ensure that the results of the analysis are precisely communicated and accurately translated into CBI operations. Douglas (2006) suggested that a language that software engineers use to analyze and design complex software systems may also serve instructional developers. “The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system” (Booch, Rumbaugh, & Jacobson, 2005, p. xiii). I propose that it is possible to use UML diagrams to analyze, design, and produce the dynamic components of a CBI system.

The Unified Modeling Language

“A modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system” (Booch et al., 2005, p. 14). The UML’s vocabulary is a set of thirteen semantically rich, articulated diagrams. Five diagrams describe the static structure of the system, six diagrams describe its dynamic behavior, and two diagrams describe the grouping and deployment organization of the system components. Additionally, the diagrams include annotational elements that document diagram features and clarify the relations between them. The diagrams are articulated in the sense that they provide different perspectives or views of the same system, with some features of one diagram mapping to the features of another diagram. Many of the diagrams are also semantically rich in the sense that they encode a large amount of information using a compact notation.
Although the UML was originally conceived as a software development language, its origin and history suggest that it is also appropriate for CBI development. The UML project began officially in 1994 with the unification of two graphical object-oriented modeling languages: Booch’s Booch Technique and Rumbaugh’s Object Modeling Technique (OMT); Jacobson’s Object-Oriented Software Engineering (OOSE) technique was officially unified with the others in 1995 (Booch, et al., 2005; Fowler, 2004; Schmuller, 2002). In 1997, these three prime developers solicited input from the software engineering community and formed a consortium to guide, direct, and support further development. The Object Management Group (OMG) issued a request for proposals for a standard modeling language in January 1997 and adopted UML 1.1 as that standard in November 1997. The OMG continues to direct the development of the UML today and adopted the most recent version, 2.2, in 2009.

The way that the UML was created impacts its proposed use as a CBI development language in three ways. First, the unification of three successful techniques enhances the UML’s clarity and intuitiveness by utilizing the best notation from each technique. Second, the soliciting of diverse software engineering concerns amplifies the UML’s expressiveness by providing a broad range of features, which makes it suitable for all software engineering design tasks, including CBI development. Finally, including the features needed for a broad spectrum of software development suggests that it is unlikely that any software engineering project, including CBI, will use all of the UML diagrams or features. Based on these three observations, I maintain that UML class and state diagrams are sufficient to describe the dynamic behaviors of a CBI system.
Class diagrams (Figure 18) form the static structure of the modeled system and are therefore the framework upon which the system is developed. The basic elements of a class diagram are the individual classes (the rectangles), and the relations between them (the decorated lines connecting the class symbols). Each class is typically divided into three sections: the class name is placed in the top section, the attributes or CBI variables are listed in the middle section (underlined names denote symbolic constants), and the operations, behaviors, or CBI functions are listed in the bottom section. It is the CBI functions that implement or carry out a CBI system’s dynamic behaviors and update its appearance. In an object-oriented program, each object is an instance of a class.

Figure 18. The UML class diagram describing the canal lock CBI example.
In previous chapters, I presented an example CBI system centered on the simple simulation of a boat navigating through a canal lock. The class diagram for that example includes six classes (Figure 18). Four of the classes, the Boat, the Valve, the Lock, and the Gate represent the dynamic CBI components. The CanalLock class provides the visible simulation elements, and the Simable class joins the simulation to an object-oriented UML CBI development system that will be described later in this paper.

Object-oriented programs are collections of objects bound together by five relations, which are each denoted by decorated lines joining the class symbols in class diagrams. Inheritance (the triangle-decorated lines in Figure 18) models the “kind of” or “type of” relation. Aggregation (the diamond-decorated lines in Figure 18), composition, and association model subtly different variations of the “whole-part” relation. Finally, dependency models the relation that arises when one object relies on the services of another to fulfill its own obligations. The objects forming a program communicate, and therefore cooperate, by sending messages to each other along these binding relations.

Individual classes in a UML class diagram may be articulated with other UML diagrams. Whenever a class exhibits complex, time-varying behavior, that behavior is described by either a state or an activity diagram. Activity diagrams describe the system flow of control – the sequence of function or method calls – as it passes from one operation to another and from one object to another. State diagrams describe the behavior of individual objects using an abstract model called a state machine. These state diagrams are the main UML diagram that I am proposing to describe the dynamic components of CBI systems.
State Machines

A state is the condition, attitude, configuration, or mode of an object at any point in time. The behavior of dynamic objects is described by how they change states over time. A state change or a transition is caused by a stimulus called an event. A transition may include an action, which is the object’s response to the event. A state machine is a collection of states and transitions. More formally, a state machine is a model “that specifies the sequence of states an object . . . goes through during its lifetime in response to events, together with its responses to those events” (Booch et al., 2005, p. 22).

A subset of the states, transitions, events, and actions describing a digital watch illustrate the operation of a state machine. Each state corresponds to one of the watch’s operational modes and each event to the pressing of one of the watch’s buttons. As the watch changes mode, it performs a task, such as starting a timer running. Various techniques for representing state machines are currently in use.

A textual or script representation of a state machine is compact but difficult to understand: The watch starts in the Display Time state. Pressing the Mode button causes the watch to change mode or to transition to the Count Down timer state. While in the Count Down timer state, the watch can respond to two events, Mode and Adjust. The Mode event sends the watch to the Alarm state; the Adjust event sends it to the Timing state and then Starts the timer running. From the Alarm state, a Mode event sends the watch back to the Display Time state. Finally, from the Timing state, an Adjust event returns the watch to the Count Down timer state and then Stops the timer. It is very tedious to try to understand the complete behavior of the watch described in this way.
Another common technique for representing state machines is as a pair of tables (Figure 19). Each state corresponds to a row in the Transition Table and each event corresponds to a column. The intersection of the current state and event denotes the next state; the state machine ignores events when the intersection is empty. The Action Table describes the action that takes place during a transition. Rows in the Action table correspond to the current state (where the transition begins) and columns correspond to the next state. The intersection of the current row and the next column denotes the action associated with that transition; empty intersections denote transitions without actions.

Representing a state machine as tables has three benefits. First, tables are more precise than scripts and are completely unambiguous. Second, given a simple, general algorithm, tables are executable or computable:

\[
\begin{align*}
next\_state &= transition\_table[current\_state][current\_event] \\
action &= action\_table[current\_state][next\_state] \\
do\_action(action) \\
current\_state &= next\_state
\end{align*}
\]

<table>
<thead>
<tr>
<th>Event State</th>
<th>Mode</th>
<th>Adjust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Time</td>
<td>Count Down</td>
<td></td>
</tr>
<tr>
<td>Count Down</td>
<td>Alarm</td>
<td>Timing</td>
</tr>
<tr>
<td>Alarm</td>
<td>Display Time</td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td>Count Down</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next State</th>
<th>Time</th>
<th>Count Down</th>
<th>Alarm</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count Down</td>
<td></td>
<td></td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td>Alarm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td></td>
<td>Stop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19. The state machine describing a digital watch represented by two tables.
Finally, tables are easy to understand, which implies that CBI developers can utilize them with little prior training. Although it is easy to determine from the tables how a state machine will behave in a given situation, it is difficult to grasp the state machine as a whole. Specifically, it is difficult to determine if the state machine is complete, to see if there are errors, and to correct errors when they are identified. The UML uses state diagrams to solve these problems.

**UML State Diagrams**

UML state diagrams are based on a graphical state machine notation called a Mealy machine (Figure 20). Mealy machines consists of a state machine where states are denoted by rectangles with rounded corners and transitions are denoted by arrows from the current or source state to the next or target state (Cohen, 1997, p. 152). A start state, a solid disk, indicates the initial state in which the machine begins. Mealy machines encode the event, the action (inside the square brackets), and the guard condition (following forward slash) on the transition arrow based on the notation:

\[ \text{event} \ [ \text{guard condition} ] / \text{action} \]

*Figure 20. A graphical representation of the state machine describing a digital watch.*
Transition Notation

The UML defines “four kinds of events: signals, calls, the passing of time, and a change in state” (Booch et al., 2005, p. 298). Signal events (e.g., Mode and Adjust in Figure 20) only convey a small amount of information – something happened – and are denoted only by name. Call events also have a name but may convey more information through a parameter-list following the name. There are two kinds of time events that are denoted by the *at* and *after* keywords respectively: absolute and relative. Both of these events require a single time parameter. Change events are denoted by the keyword *when* and require a single boolean-valued parameter. (A boolean-valued parameter is true in some cases and false in others; for example, *level == EMPTY*.) An event is ignored or lost when the event is received while the object is in a state that does not define a transition for that event. Events are optional, and transitions that do not have a triggering event are called null or lambda transitions; null transitions permit a state change as soon as the state is entered, if the change is permitted by the guard condition.

A *guard condition*, denoted by the square brackets, is a boolean-valued expressions based on an object’s attributes and operations, or on the attributes and operations of other accessible objects. If a transition has a guard condition, that condition must be true before the transition is allowed. If two transitions are triggered by the same event, including null events, then the guard conditions must be exclusive (i.e., no two conditions may be true at the same time) – at most, one transition leaving a state may be valid at any time. The valid transition determines the action that the state machine performs in response to the current event.
In the UML, an action is an operation (i.e., something that an object can do) and is denoted by a forward slash character preceding the action name. Conceptually, actions are atomic and instantaneous – they run uninterruptedly from the time they start until they are complete and their execution time is short compared to the overall program execution. The keyword *send* represents a special action that permits one object to send a signal event to another object.

**Specialized UML States**

Mealy machines form the foundation of UML class diagrams, but class diagrams also utilize a second state notation based on Moore machines (Booch et al., 2005, p. 344). Formally, the definition of a Moore machine is based on a transition table and an output or action table (Cohen, 1997, p. 150), but Moore machines are often represented graphically with the actions written inside of the state rather than on the transition arrow (p. 151). Figure 21 illustrates the features that may appear within a Moore-based UML state (Booch et al., 2005, pp. 301-302). The state name (in the top section) is the only required feature; the other features adhere to the notation: *event / action*

![Figure 21. A Moore machine state.](image)
The keywords *entry* and *exit* are special events that trigger actions when a state is entered or exited respectively. Entry and exit events, and their associated actions, are a convenience notation: The information that they encode could also be specified on the transition arrows. Putting the entry or exit events, or both, inside the state is more parsimonious in those cases when all transitions entering or exiting a state trigger the same action.

The *do* keyword denotes an activity that starts when the state is entered and stops when the state is exited. An activity is an asynchronous operation that begins executing when an object enters a state and ceases when the object leaves that state. An activity is similar to an action – it corresponds to a class operation – but unlike actions, which are atomic and instantaneous, activities are nonatomic and ongoing.

Deferred events are indicated with the special action keyword *defer*. Following *defer* is a list of events whose handling is deferred or postponed until the machine enters another state. Deferred events are examined before entry actions and do activities, and if a transition matches a deferred event, it triggers an immediate state change.

Internal transitions are the only state feature that does not utilize a keyword. For example, in Figure 21, *error* and *showHint* name an event and an action, respectively. An internal transition is similar to a *self transition*, which begins and ends with the same state (i.e., both ends of the transition arrow are anchored to the same state). However, self transition trigger entry and exit actions and internal transitions do not. Together, Mealy and Moore machines are sufficient to model simple objects. However, more complex objects may require states that exhibit more complex structure.
A compound state (Figure 22) has one or more substates embedded or nested within it. Transitions within a compound state (i.e., between substates) look and behave like transitions between noncompound states. Compound states simplify situations that require identical transitions, either to or from, all of the substates. For example, in Figure 22, the event 1 signal triggers a transition from any of the substates to State 6. The event 2 signal triggers a transition from State 5 to one of the substates, but it is not clear to which substate. A compound state may optionally contain a history state (denoted by a circle with an “H” inside). An explicit transition arrow connects the history state to the default state – the target for the first transition into the compound state. A history state “remembers” the last occupied substate and an in-bound transition that does not explicitly target a substate, implicitly targets the last occupied state. Finally, a transition may explicitly target any substate within a compound state: The event 3 signal triggers a transition from State 4 to State 2. This background is sufficient to understand an example CBI system developed with UML state diagrams.

Figure 22. A compound state with three substates and a history state.
UML CBI Example

The canal lock example presents the problem of navigating a boat between a canal and a lake. The UML description requires five diagrams: a class diagram (Figure 18, p. 70) and four state diagrams (Figure 23 through Figure 26). The Boat (Figure 23) is the most complex object in the problem. Three states, In Canal, In Lock, and In Lake, represent the boat’s stationary locations. Four states, Canal to Lock, Lock to Lake, Lake to Lock, and Lock to Lake, describe the boat while it is in motion. Self-transitions that begin and end in the motion states animate the boat by updating its position every 50 ms. The self-transitions are triggered by a time event (i.e., after(50u)) and continue while the boat is not in the target location (e.g., xPos != IN_LAKE). Each self-transition animates the boat by executing either a moveRight or a moveLeft action that makes a small, incremental change to the boat’s location on the screen and then repaints the screen.

The canal lock is an example of a path simulation (Gibbons & Fairweather, 1998, p. 299). Path simulations constrain learners to a well-defined path of correct responses and display correcting messages whenever the learners’ response leads them off the path. The canal lock consists of four paths that intersect in the In Lock state, where the learner may change paths. Each path consists of three transitions that begin in a stationary state, loop in an animation state, and terminate in a second stationary state. The first transition is initiated when the user presses a button that sends a left or right event. The second transition continues until the boat is in the correct location, which triggers the last transition. An advantage of the UML’s graphical notation is that the paths and the correcting messages are clearly evident in the Boat state diagram (Figure 23).
Figure 23. UML state diagram for the Boat object in the canal lock example.
after(50u) [level != EMPTY] / drainLock(); send boat.lower

after(50u) [level != EMPTY] / drainLock(); send boat.lower

Figure 24. UML state diagram for the Lock object in the canal lock example.

after(50u) [level != FULL] / fillLock(); send boat.raise

after(50u) [level != FULL] / fillLock(); send boat.raise

Figure 25. UML state diagram for the two Gate objects in the canal lock example.
Manually creating and editing these intricate diagrams is a prohibitively tedious task that is made unnecessary by the UML’s success. The UML is arguably the most successful and the most widely used object-oriented modeling language that is currently in use. As a consequence, many UML modeling tools are available. These tools span a wide spectrum of cost and capability. Unlike generic diagramming tools that manipulate diagram elements – lines, rectangles, text, and so forth – as primitive graphical shapes, UML modeling tools “understand” the diagram elements at a higher semantic level and manipulate the elements as the visible representation of high-level concepts – classes, relations, states, transition, and so forth.

**Figure 26.** UML state diagram for the Valve object in the canal lock example.
Like many UML modeling tools, the tool that I used associates each UML state diagram with a class. This association benefits CBI developers in two ways. First, the tool provides a set of navigation controls that allow developers to easily manage a complex set of diagrams and to quickly locate specific subdiagrams. The second benefit is a function both of the current UML standard and of the modeling tool’s ability to manipulate the notation at a high semantic level. UML modeling tools that conform to the OMG’s standards support exporting the diagrams as Extensible Markup Language (XML) files in the XML Model Interchange (XMI) format (Lear, 2000). The exported XMI files can then be processed to create executable CBI components. I propose that these benefits enable the UML to serve as a CBI development language that resolves an instructional design problem identified by Eckel (1993) nearly two decades ago.

The UML is an Unequivocal Primary Representation

Eckel (1993) asserted that the contemporary process of developing CBI produces and utilizes two representations of the CBI. The first or primary representation is instruction-oriented. The notation and the construction – the very purpose – of the primary representation is intended to make the represented instruction easily accessible to everyone involved with its development. Eckel (1993) offered storyboards and scripts as examples of primary representations but maintained that they are ambiguous or equivocal. Alternatively, computer programs, his example of an unequivocal representation, are couched in an arcane or secondary representation that is “absolutely useless” (p. 7) for further development. I maintain that the UML can resolve this problem by serving as a representation that is both primary and unequivocal.
The UML is an Unequivocal Representation

The UML lies at the end of a spectrum of software engineering languages. At the opposite end of the spectrum are low-level programming languages; the features of low-level languages are closely aligned to the hardware on which the programs run. Between these two extremes are high-level programming languages; the features of high-level languages are based on abstractions that divorce the programs from the underlying hardware, and “are characterized by resembling problem-solving notation rather than machine languages” (Pittman & Peters, 1992, p. 3). Problem-solutions are more easily seen and therefore better understood when expressed in a high-level language than when expressed in a low-level language. Nevertheless, high-level “programming languages are not at a high enough level of abstraction to facilitate discussions about design” (Fowler, 2004, p. 1). The UML satisfies the need for a language that is capable of specifying the details of a software system at a level of abstraction which is suitable for design.

The UML currently specifies thirteen distinct but articulated diagrams. Each diagram describes a different aspect of the system: static structure, dynamic behavior, deployment organization, or annotational elements. Many of the diagrams are “semantically rich” in the sense that they encode a large amount of information using a compact notation. Significantly, each of the symbols has a precise, established meaning:

The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously. (Booch et al., 2005, p. 15)
The well-defined semantics makes the UML a candidate for an unequivocal representation. Booch et al. (2005) maintained that the UML is not a graphical programming language but claims that, “It is possible to map from a model in the UML to a programming language” (p. 16). That is, the symbols appearing in a correct, well-formed model expressed in the UML have exactly one interpretation, and the meaning expressed by those symbols can be expressed as a computer program. Conversely, Fowler (2004) suggested that it is possible to use the UML as a programming language.\(^2\)

When used as a programming language, “developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code. Obviously, this usage of UML demands particularly sophisticated tooling” (p. 3).

The UML does lack some features, input-output statements for example, that are often included in high-level programming languages. Nevertheless, the UML is capable of representing many crucial high-level constructs, like branches and loops, and it shows those constructs in a visual notation that is understandable to nonprogrammers. The solution, I believe, is to use the UML as a partial programming language: CBI developers analyze, design, and implement the dynamic components of a CBI system with UML diagrams, and implement the user interface with traditional computer programming languages.

CBI obviously requires computer programming, but this approach recognizes a unique role that programming can play in CBI development. Four decades ago, Stolurow (1969) noted that “a description of teaching in a computer language is less equivocal than

\(^2\) He also stated, “I see the UML as programming language as a nice idea but doubt that it will ever see significant usage. I’m not convinced that graphical forms are more productive than textual forms for most programming tasks” (p. 6).
one in natural language” (p. 66). Significantly, computer programs remain Eckel’s (1993) “gold standard” for an unequivocal representation. The ability of the UML to serve as a programming language implies that it is an unequivocal representation.

**The UML is a Primary Representation**

Establishing that a representation qualifies as primary is more difficult than establishing that it is unequivocal. An analogy illustrates this problem: A house is accurately described by a set of blueprints. Elevations describe the appearance of the house using graphical illustrations. Elevations are widely understood because they present familiar concepts using intuitive symbols. Electrical plans describe the location and operation of switches, lights, outlets, and circuit breakers. Electrical plans are more difficult to understand than are elevations because they represent less familiar concepts and often use abstract symbols. Finally, floor plans describe the layout, size, and arrangement of rooms; they show the location of doors, windows, fixtures, and appliances. Floor plans are generally more concrete and, therefore, easier to understand, than are electrical or plumbing plans, but they may use some abstract symbols that untrained readers may not understand. The information that participants (architects, contractors, financiers, buyers, etc.) are able to extract from the various blueprints is a function of their individual training and experience.

Similarly, a set of UML diagrams describe or model a complex software system, and the diagrams are frequently analogized as software blueprints (Booch et al., 2005; Fowler, 2004; Schmuller, 2002). Unlike blueprints, UML diagrams typically represent concepts that are abstract and intangible. As a consequence, the UML notation is both
abstract and arbitrary, which makes UML diagrams less intuitive than many blueprints. However, the concepts represented by an electrical plan are familiar to experienced electricians and they are able to learn the notation quickly. Similarly, the concepts represented by a UML diagram modeling a CBI system are familiar to experienced CBI designers and they should be able to learn to use UML diagrams quickly by mapping familiar concepts to a new notation.

In his introduction of the UML, Schmuller (2002) noted that in the past, few programmers performed detailed analyses of a problem before writing code and contrasts that practice with contemporary industry needs.

Today, a well-thought-out plan is crucial. A client has to understand what a development team is going to do, and must be able to indicate changes if the team hasn’t fully grasped the client’s needs (or if the client changes his or her mind along the way). Also, development is typically a team-oriented effort, so each member of the team has to know where his or her work fits into the big picture (and what that big picture is). (p. 6)

Schmuller’s observations pressing the need for a language like the UML apply equally well to CBI design, and in that context describe many of the requirements of a primary representation.

UML diagrams are based on symbols and semantics that are designed to clearly represent general software systems. Fowler (2004) maintained that UML diagrams “help communicate ideas and alternatives about what . . . [a developer is] about to do” (p. 2). This is the essence of a primary representation and parallels Finch’s (1973) description of the role storyboards play in the production of animated films: “the director, or anyone else concerned with the production, could see the plot of an entire movie spread out in front of him” (p. 82). To better understand state diagrams, think of them as geographical
maps of cities and roadways, with states replacing the cities and transitions replacing the roads. Just as routs are “spread out in front” of a traveler on a map, the time varying behavior of a complex object is “spread out in front” of a CBI developer in a state diagram. Roadmaps are a familiar representation that are used and understood by many people. That familiarity suggests that CBI developers are capable of learning and using state diagrams. Without discounting the subjective nature of what constitutes a primary representation, I maintain that by intent and design, the UML qualifies as a primary representation.

The efficacy of the UML as an unequivocal primary representation rests on two claims: The first claim is that it is possible to unambiguously translate the diagrams into an executable form. Implicit in this claim is the assumption that the executable form can be combined with additional computer instructions and with instructional content to form a finished CBI product. The second claim is that it is possible to use UML diagrams to design and communicate select CBI components. This second claim assumes that the UML is sufficiently expressive so that it can describe the selected components. To substantiate these claims, I developed a two-part CBI development system that is capable of transforming UML class and state diagrams into executable CBI components and then used the system to create an extensive CBI example.

**Implementing CBI with the UML**

The first part of the UML CBI development system consists of a library of Java™ classes. The notational elements of UML state diagrams, as described previously, are implemented by classes in the library. The library, named *jsm* (for Java State Machine),
is organized as a Java package. The second part of the system is a compiler or translator that converts the UML diagrams into executable components. The translator builds a symbol table of the attributes (CBI variables) and the operations (CBI functions) from the class diagram. The translator instantiates objects, based on information in the symbol table, that realize each notational element in the UML state diagrams and then connects those objects together to implement the behavior of a specific state machine. Figure 27 illustrates the interaction of and the flow of data between the components of the UML CBI system and highlights the major steps in creating CBI with the UML system.

Creating the final CBI involves two major tasks: The CBI developer’s first task is to create the UML diagrams (item 1 below). The developer then writes the user interface code (item 6 below); currently, this must be done in the Java programming language to be compatible with jsm and the UML compiler. These two tasks can be done concurrently.

*Figure 27.* The components of the UML CBI development system.
or in either order. In practice, the usual starting point is the creation of a subset of the diagrams, and then the process proceeds iteratively as both components are extended and refined. The interactions between the system components are as follows:

1. **UML Diagrams.** These are created with a commercially available UML modeling tool.

2. **XML File.** The diagrams are exported as XML files in the XMI format. About 20% of the information contained in the state diagram XML file pertains to the function of the described state machine; the balance of the information relates to detail that is irrelevant in this context (e.g., the physical location and size of diagram elements).

3. **State Machine Compiler.** The compiler or translator constructs an abstract syntax tree from the state machine XML file. From the tree, the compiler extracts the information about the elements and structure of the specific state machine. The compiler instantiates and organizes the objects to realize the specified state machine, which is then saved in a binary file.

4. **Binary State Machine File.** The executable state machine consists of objects and object references expressed as Java bytecode, which is the name given to the virtual machine code produced by a Java compiler (Horstmann & Cornell, 2008). The binary file is created with the standard Java serialization mechanism that preserves the object references and permits the CBI to read and reconstruct the state machine very quickly.
5. *jsm*. The Java state machine package defines a set of classes, also expressed as bytecode, corresponding to each element of a state machine. The state machine compiler draws classes from this package as needed to construct a specific executable state machine implementing a dynamic CBI component.

6. *User Interface Code*. Although the UML can describe the behavior of a user interface, it does not include a diagram or other mechanism to describe the appearance of the interface. The CBI developer must provide, as one or more Java classes, the visual and aural CBI components. The classes must also provide the external user controls but the UML defines the behavior of the controls. Each of the user interface classes is compiled with the standard Java compiler.

7. *CBI Components*. Each of the preceding six steps is necessary to create each dynamic CBI component. At program startup, class loaders within the Java Virtual Machine™ (JVM) load the compiled interface code. References in the interface code to the CBI components causes the class loaders to load the CBI components, forming a complete, executable CBI system.

The complete CBI system is a stand-alone executable that requires neither awareness of the UML nor of any UML modeling tool. The CBI system may be stored on a computer, delivered on a CD-ROM, or delivered over the Internet. All that is required to run the CBI is a compatible version of the Java Runtime Environment™ (JRE). The JRE is available from Sun Microsystems at no cost and is legally distributable with Java applications.
To demonstrate the feasibility of using the UML CBI development system and to demonstrate that the UML is sufficiently robust to describe dynamic CBI components, I created a complete CBI system that presents training in the use of a portable MP3 player. The player’s small size limits the number of physical controls and the size of the display screen. Together, these two limitations greatly increase the complexity of the device by making it necessary to access the player’s functionality through a series of deeply-nested menus. The CBI make use of three UML state diagrams: one diagram to model the behavior of the player, one diagram to implement the instructional method, and one diagram to define the behavior of a complex learner assessment device.

The MP3 player state machine is the largest and the most complex in the MP3 player CBI. The organization of this state machine parallels the player’s menu system. When printed on standard-sized paper, the state machine spans fifteen pages. An MP3 player class defines the visual appearance of the player and the user controls. The MP3 player class also aggregates ten additional classes with state machines. These state machines are simple, some consisting of only one or two states, and easily fit on a single page. The instructional method and assessment state machines are much smaller and less complex than the MP3 player state machine. The instructional method state machine is two pages long and the assessment state machine is one, although very full, page in size.

Conclusion

Training astronauts and airline pilots through authentic experiences is difficult. Fortunately, the need for less exotic experiential learning environments is more common. Spacecraft and aircraft simulators require specialized user interface devices that create a
simulated environment that is nearly indistinguishable from reality. The UML CBI system presented here focuses on CBI that can run on a typical personal computer – one with a keyboard, mouse or other pointer, optical drive, sound card, and graphic display. CBI running on computers without specialized interface components is necessarily limited, but with a modest amount of abstraction can still mimic many learning environments, both real and abstract, faithfully.

Generally, the UML CBI system is well-suited to training conceptual cause-effect and time-space relations. By abstracting appearance, size, manipulation, and tactile sensations, it is possible to apply these relations to physical systems. This application extends the domain of the UML CBI system to training for operating and maintaining physical devices. The ability of the UML CBI system to implement cause-effect and time-space relations is a consequence of the language itself.

The many similarities between software engineering and instruction development – the abstract and intangible nature of their products, the need to analyze a problem domain, the importance of deliberate design, continuously evolving requirements, and the need to communicate and collaborate – suggest that tools developed in one discipline may benefit the other. The UML provides to software engineers a bridge between the initial analysis and design of a problem and the final programming of the resulting system. The UML can provide the same service to CBI developers by bridging the same gap. With the addition of Fowler’s “sophisticated tooling,” the UML can also serve as an unequivocal primary representation, further narrowing the gap.
The *jsm* library and the state machine compiler implement the needed tooling, and by so doing, demonstrate that the UML is unequivocal in the same way that a computer program is unequivocal. The UML was created to serve as a common language for describing and communicating the details of complex software systems; its wide adoption suggests that it satisfies this purpose for many software developers. Similarly, a primary representation is a common language for describing and communicating the details of complex CBI systems. The UML’s success in the software engineering domain builds confidence that it can also be successful in the CBI domain. Finally, the MP3 player CBI example demonstrates that it is both possible and feasible to use the UML as a CBI development language. Together, I believe that the tools and the completed CBI example demonstrate that the UML qualifies, both theoretically and practically, as an unequivocal primary representation.

**References**


CHAPTER 4
UNEQUIVOCAL PRIMARY REPRESENTATIONS:
PEANETS, UML DIAGRAMS, AND
FORMAL LANGUAGES

Suppose I want to understand the “structure” of something. Just what exactly does that mean?
It means, of course, that I want to make a simple picture of it, which lets me grasp it as a whole.
And it means, too, that as far as possible, I want to paint this simple picture out of as few elements as possible. The fewer the elements there are, the richer the relationships between them, and the more of the picture lies in the “structure” of these relations.
And finally, of course, I want to paint a picture which allows me to understand the patterns of events which keep on happening in the thing whose structure I seek. In other words, I hope to find a picture, or a structure, which will, in some rather obvious and simple sense, account for the outward properties, or the pattern of events of the thing which I am studying. (Alexander, 1979, pp. 81-82)

Eckel (1993) observed that contemporary computer-based instruction (CBI) design techniques create and utilize two representations. The first representation is primary but equivocal while the second representation is unequivocal but secondary. Alexander’s (1979) quest for “a simple picture” that allows the viewer to grasp the structure of a complex system “as a whole” parallels and succinctly characterizes both the purpose and the goal of a primary representation. Furthermore, if the picture elements and the relationships between those elements are sufficiently precise, then the picture may also be unequivocal.

Alexander (1979) encompassed both the unequivocal and the primary aspects of his simple picture by formally defining a language as “a system which contains two sets: 1. A set of elements, or symbols. [and] 2. A set of rules for combining these symbols”
(1979, pp. 183-184). The elements or symbols form the visible part of the representation and are largely responsible for making it primary. The combining rules determine the overall structure of a picture; that is, the rules determine what symbols will appear in a specific picture and the relations that join the symbols. The rules are not directly visible but are largely responsible for making the representation unequivocal.

Both the unequivocal and primary aspects of a representation are functions of the language forming the representation. Nevertheless, it is more difficult to establish that a representation is unequivocal. For example, two people can see the same picture, and one will “grasp it as a whole” while the other does not. This situation suggests factors such as training and experience also play a role in determining when a representation is primary.

**Languages, Representations, and Experience**

The primary and unequivocal dimensions of a representation are often in conflict. That is, simple and easy-to-understand representations may lack the precision needed to form an unequivocal representation, while computably precise representations may be too complex for anyone but a specialist to understand. The complexity of a representation is a product of the symbols, the joining relationships, and of the combining rules. The rules themselves may be either simple or complex but whether a representation is primary or not, is more of a function of the visible symbols and of the user’s familiarity with the represented concept than it is of the rules. This assertion can be demonstrated by examining how computer programs perform arithmetic.
Formal Languages

Contemporary computer programming languages are based on very precise definitions. One class of simple languages is defined with context free grammars (CFG). CFGs are often denoted by a quadruple: \((V, \Sigma, R, S)\), where \(V\) is a set of symbols (both terminal and nonterminal), \(\Sigma\) is a set of terminal symbols, \(R\) is a set of replacement rules, and \(S\) is the start symbol (Lewis & Papadimitriou, 1998, p. 115). The set of nonterminal symbols is the set difference \(V - \Sigma\). Replacement rules follow the pattern: \(N \rightarrow r\), where \(N\) is a single nonterminal symbol and \(r\) is a string of terminal and nonterminal symbols. \(V\) and \(R\) correspond to Alexander’s (1979) symbols and combining rules respectively.

Figure 28 is an example of a partial CFG that defines how computers evaluate arithmetic expressions. The CFG is expressed using Backus-Naur form (BNF), which is a meta-language used to define specific languages, especially programming languages (“Backus–Naur form,” 2003): that is, it is a language for defining languages. In this example, \(\Sigma = \{., *, /, +, -, 0, 1, \ldots, 9\}\); \(V - \Sigma = \{\text{digit}, \text{fact}, \text{term}, \text{expr}, \text{variable}, \text{start}\}\); \(S = \{\text{start}\}\); and \(R = \{\text{digit} \rightarrow \ldots, \text{fact} \rightarrow \ldots, \text{term} \rightarrow \ldots, \text{expr} \rightarrow \ldots, \text{start} \rightarrow \ldots\}\) (where the ellipses represent the right hand side of each rule for brevity). This CFG is partial because it does not provide a production for the nonterminal “variable.”

Although the CFG that describes how computers evaluate arithmetic expressions

digit → "0" | "1" | "2" | . . . | "8" | "9"
fact → digit | variable | "(" expr ")"
term → term "*" fact | term "/" fact | fact
expr → expr "+" term | expr "-" term | term
start → expr

*Figure 28. Partial CFG for evaluating arithmetic expressions.*
may be unfamiliar to nonprogrammers, the resulting expressions are more widely understood. For example, Figure 29 (a) illustrates an expression based on addition and multiplication. The asterisk denotes multiplication and the value of variable \( x \) is not specified in the example, but the meaning of the expression is otherwise clear. Figure 29 (b) uses nearly the same symbols as the first example but is more difficult to understand.

The grammar of the second example is larger and more complex than the first example, but the greatest hurdle to understanding Figure 29 (b) is that the underlying concept is unfamiliar to nonprogrammers: typecasting a pointer to a pointer and dereferencing the result. Explaining the operation in greater detail does little to help nonprogrammers understand the expression: Variable \( s \) in an untyped double pointer—the address in memory of another address in memory of some data. The double pointer value is cast or converted into the address of a character-string and the resulting string is dereferenced or extracted. Based entirely on familiarity with the represented concepts, the first expression is primary to a wider audience than is the second expression.

\[
(x + 3) \times 5
\]

(a)

\[
*(\text{char **}) s
\]

(b)

*Figure 29. Expressions: (a) an arithmetic expression, (b) a pointer-valued expression.*
It is possible to automatically translate representations based on formal languages into a format that can be executed on a computer by building a structure called a parse tree (Figure 30; Aho, Lam, Sethi, & Ullman, 2007, pp. 45-47). A parse tree explicitly presents the relationships that exist between the language terminals and nonterminals, which, as Alexander noted, creates “more of the picture” from “the ‘structure’ of these relations” (Alexander, 1979, p. 81). The computer carries out actions, which implement the meaning inherent in the representation, at the root of each subtree. The ability to translate a representation into a parse tree and to carry out the actions at the subtree roots is responsible for producing a computable, and therefore an unequivocal, representation. The disadvantage of this approach is that this structure is only implicit in the final representation, which is antithetical to the fundamental goal of a primary representation.

![Figure 30. Parse tree for the expression (x + 3) * 5.](image)
Graphical Languages

Formal language definitions are the foundation of computer programming and related languages but are not the only options available. For example, Alexander et al. (1977) defined a pattern language for designing “towns and neighborhoods, houses, gardens, and rooms” (p. ix). Each instance or pattern in the language has the same format, which in practice, is described as a combination of text and graphics (pp. x-xi) and consists of seven distinct sections.

Sections 1 and 2 form a high-level overview of a specific pattern; Sections 3-6 define the main content of the pattern; and section 7 contains a set of links to patterns that may potentially complete this pattern. The full pattern language consists of 253 separate patterns. When instances of the patterns are joined together they form a network or web-like structure. Individual threads in the web form sequences of patterns “always from the larger patterns to the smaller, always from the ones which create structures, to the ones which then embellish those structures, and then to those which embellish the embellishments. . . .” (Alexander et al., 1977, p. xviii).

The pattern language has many features in common with a CFG. Pattern independent regions (1) corresponds to the start symbol. Some patterns – for example, things from your life (253) – do not have links to other patterns and so must be terminal patterns. Most patterns, however, may link to many others and so behave like productions. For example, the pattern city country fingers (3) may link to one or more of agricultural valleys (4), mosaic of subcultures (8), web of public transportation (16), or
ring roads (17). Finally, some patterns support recursion; for example: accessible green (60) ➔ grave sites (70) ➔ quiet backs (59) ➔ accessible green (60).

The underlying concepts embodied in the pattern language are naturally familiar to architects and urban planners who easily grasp as a whole the designs expressed in the language. Novices can also understand the general concepts and can therefore understand designs expressed in the language with little coaching. This suggests that Alexander’s (1979) pattern language is a primary representation.

Complex and intricate designs are often too difficult to put into words alone and so architects, engineers, and other designers may resort to graphical notations. Charts and graphs, for example, are well known in business, mathematics, and the sciences. Graphical notations have also been developed to improve project management (Gantt, 1961; Martino, 1968; Riggs & Heath, 1963; Wiest & Levy, 1997). These network techniques, including PERT (program evaluation and review technique), CPM (critical path method), CPS (critical path scheduling), Gantt, and others, represent the temporal relations between “activities, events, and predecessors” (Wiest & Levy, 1997, p. 1).

Specialized notations, including graphical notations, often arise from specific needs. For example, Gantt charts evolved from logistical supply problems arising during WWI (Gantt, 1961), and more recently PERT charts were created to expedite the development and delivery of the Polaris ballistic missile (Wiest & Levy, 1997). Similarly, the need to train large numbers of designers and shop workers in the aircraft industry during 1941-1943 drove aircraft manufactures to compile “process specification” manuals detailing proprietary riveting technology:
Since much of the knowledge in any shop process is difficult or impossible to put into words, these books included many pictures and drawings. An especially impressive effort . . . contained 226 pages of detailed photographs and drawings. . . . Only a small fraction of the material was written text. (Vincenti, 1990, p. 188)

Other domains also provide examples that further demonstrate the diverse use of graphical notations. Waters and Gibbons (2004) contrasted two choreographic notations, one geometric and the other iconic. They noted that while the two notations provide a similar amount of information, people generally prefer working with the iconic notation. Waters and Gibbons (2004) suggested that the iconic notation is more intuitive and the symbols more “clearly call up the mental image of a dance” (p. 63). They further suggested that an increased “degree of match with the designer’s mental image of the product can facilitate comprehension and eliminate needless mental translation” (p. 63).

The exact notation used to express a design is a function of many variables: the problem domain, the traditions within that domain, the complexity of the designs, the tools available, and the need to communicate design details between stakeholders with different interests in a project. Although both textual and graphical notations are often used in practice, either notation by itself is generally suitable only for small, simple designs. In complex designs, “Some things are best modeled textually; others are best modeled graphically” (Booch, Rumbaugh, & Jacobson, 2005, p. 15). Complex designs are generally easier to understand and are more complete when they are expressed using a combination of textual and graphical notations. Both PEAnets and UML diagrams use a combination of graphical and textual notations; from this standpoint, both PEAnets and the UML can potentially represent complex designs.
Although dancing and riveting aircraft parts together seem completely unrelated, they share a significant common denominator. Dance movements consist of the precise time-varying position of the dancer’s body, frequently synchronized with music and the bodies of other performers. Riveting parts together requires that the parts are correctly positioned, that the correct rivet and tool are used, and that appropriate tolerances are maintained. Both dancing and riveting involve concrete, physical entities and processes. PEAnets and UML diagrams can describe the behavior of concrete entities and processes, but they are also frequently called upon to describe more abstract concepts.

People prefer to work with intuitive notations that correspond to the features of the represented entities and that “eliminate needless mental translation” (Waters & Gibbons, 2004, p. 63). However, intangible entities and abstract concepts may lack clear correspondences. Is it possible to reconcile human preference with need to represent ideas that do not present obvious images, and if reconciliation is possible, how does the respective levels of intuition and mental translation compare between PEAnets and the UML? Answers to these questions are based on the notation of each representation. PEAnets and the UML provide notational elements that correspond to and denote the entities, both concrete and abstract, found in a wide variety of problem domains. Both languages also include notational elements that describe the relations and interactions between those entities. How the entities, relations, and interactions are partitioned and named is largely arbitrary. As a result, there are many similarities between PEAnets and some of the thirteen UML diagrams but not always a direct one-to-one correspondence.
Chapters 2 and 3 detail the PEAnet and the UML notations respectively, and Table 1 summarizes the articulation between the two notations. Entities, properties, and property values describe problem domain components in PEAnet designs; classes, objects, attributes, and behaviors or operations do the same in UML diagrams. The remaining notational elements (including the five additional elements that are available in UML class diagrams) describe the relations and interactions, exiting in the problem domain, between the components. In both representations, each notational element has an impact on the intuitiveness of the representation and on the amount of mental translation that is necessary to understand the representation, which further impacts the ability of each language to serve as a primary representation.

**Representing Static Structure**

Entities and classes describe the static structure of a system by defining the system’s components. System descriptions include the instructionally significant details

Table 1

*A Comparison of PEAnet and UML Diagram Elements*

<table>
<thead>
<tr>
<th>PEAnet Element</th>
<th>UML Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Class and object (i.e., instance of a class)</td>
</tr>
<tr>
<td>Property</td>
<td>Attribute</td>
</tr>
<tr>
<td>Property value</td>
<td>Static attribute (implemented with a symbolic constant)</td>
</tr>
<tr>
<td>Activity</td>
<td>Event</td>
</tr>
<tr>
<td>Process</td>
<td>Behavior or operation (in class diagrams)</td>
</tr>
<tr>
<td></td>
<td>Action (in state diagrams)</td>
</tr>
<tr>
<td>Concurrent processes</td>
<td>Activities</td>
</tr>
<tr>
<td>Consequence</td>
<td>Attribute value change</td>
</tr>
<tr>
<td>Condition</td>
<td>Guard condition</td>
</tr>
<tr>
<td>Sum of all property values</td>
<td>State</td>
</tr>
<tr>
<td>Property value change</td>
<td>Transition</td>
</tr>
</tbody>
</table>
or characteristics of each component and do not change during system execution. In this sense, entities and classes are “containers” that contain and organize the information that characterizes the entity. For example, entities contain properties and property values. Similarly, classes contain attributes, and may optionally contain symbolic constants. These constants denote the legal values that may be stored by the attributes. The properties and attributes must be distinct with their respective containers; however, two entities may contain properties with the same name or two classes may contain attributes with the same name.

Although they are similar and serve the same purpose, UML class diagrams do have a significant advantage over PEAnets. It is common for designs to contain multiple instances of a given entity or class. For example, the canal lock simulation introduced in chapter 1 contains two gates, one at each end of the lock. The PEAnet description of the simulation includes two distinct entities named Lower Gate and Upper Gate (Figure 31). In contrast, the corresponding UML class diagram (Figure 32) contains a single class named Gate. The complete canal lock simulation, represented by a class named CanalLock, contains two instances of the Gate class; the instances or objects are named lowerGate and upperGate. The value of this difference becomes more evident as the size and the complexity of the classes increase and as the number of instances of each class increases.
Figure 31. Properties in the PEAnet version of the canal lock example.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boat</td>
<td>location</td>
<td>in canal / in lock / in lake / canal_to_lock / lock_to_lake / lake_to_lock / lock_to_canal</td>
</tr>
<tr>
<td></td>
<td>direction</td>
<td>right / left</td>
</tr>
<tr>
<td></td>
<td>level</td>
<td>low / high</td>
</tr>
<tr>
<td>Valve</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Lower Gate</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Upper Gate</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Lock</td>
<td>level</td>
<td>full / empty / draining / filling</td>
</tr>
</tbody>
</table>

Figure 32. Classes, and their associated attributes and behaviors; the UML version of the canal lock simulation defines one Gate class but has two Gate objects named lowerGate and upperGate.
PEAnets organize entities and their associated properties and property values in a table. Tables are familiar constructs and experienced CBI developers should quickly grasp the contained information. This observation suggests that PEAnet entity tables qualify as primary representations. UML classes organize their contained attributes in a list, but they also encode additional information with a compact, unfamiliar notation. Grasping all of the information contained within a UML class symbol requires an understanding of the object-oriented paradigm. However, object-oriented concepts are similar to concepts that are already familiar to experienced CBI developers. This observation suggests that developers can gain a sufficient understanding of the UML with a minimum amount of tutoring. Given that CBI developers are familiar with the underlying object-oriented concepts and can learn the UML class notation quickly, I assert that UML class diagrams also qualify as primary representations.

**Representing Dynamic Structure**

Entities and classes describe the system’s nouns or actors. Additionally, PEAnets and the UML include notational elements that describe system verbs – the reactions to various stimuli. Each of these reactions is described and controlled by a set of subcomponents in both languages. Chapters 2 and 3 describe in detail the operation of PEAnets and UML state diagrams, respectively. PEAnets represent dynamic system behavior with a cause and effect chain of operations (Figure 33). An activity represents a stimulus, often from the user, that triggers a process. When a process executes, it carries out the entity’s reaction to the stimulus, implements the intended instruction, updates the CBI’s appearance or portrayal, changes a property value as a consequence of its
execution, and optionally triggers another process. A process runs only if its associated conditions are true.

Each entity forms a unique scope – a region where named subcomponents (i.e., properties and attributes) are visible or accessible. It is for this reason that two entities may have properties, or property values, with the same name but not be in conflict. However, all activities and processes are defined in a single scope or namespace, which requires that each activity name and each process name be unique. Furthermore, activities and processes are bound to entities by name, which implies the additional requirement that activity and process names include the name of the associated entity. From a software engineering perspective, entities follow the object-oriented paradigm while activities and processes follow the procedural paradigm.

![PEAnet diagram](image)

*Figure 33. PEAnet describing the dynamic behavior of a CBI system triggered by the user pressing the Close Valve Button.*
Experienced CBI developers are familiar with the concepts (perhaps with different names) underlying activities, processes, consequences, and conditions. From this perspective, PEAnet diagrams are generally intuitive. The meaning of an activity (e.g., Press Close Valve Button) is obvious and the “trigger” labelled arrow clearly suggests the activation of a dynamic procedure. The notation that indicates one process triggering another, especially the distinction between a sequential and a concurrent triggering, is less obvious, as is the notation for consequences and conditions. Process chains, consequence, and conditions are not as intuitive as are activities and the initial process triggers. Nevertheless, PEAnet diagrams are easily understood with only a brief tutorial and I conclude that they qualify as a primary representation.

The UML describes the dynamic behavior of an object with a state diagram (Figure 34). As the object responds to stimuli, or events, it performs a state change or

![UML State Diagram](image)

*Figure 34. UML state diagram describing the dynamic behavior of the canal lock.*
transition. Transitions are allowed if the associated guard condition is true. If the transition is allowed, it can trigger an optional action. Like a PEAnet process, a UML action implements the intended instruction and updates the CBI’s appearance.

As with PEAnets, experienced CBI developers are familiar with the concepts underlying most of the UML state diagram notation. Nevertheless, the degree to which UML state diagrams may be considered intuitive is a function of the viewer’s previous experience with state machines. While state machines are a common device used in many domains, they are not universal. The basic features and operations of a state machine can be learned and understood with little effort but the detailed notation appearing in UML state diagrams may require more extensive instruction. UML state diagrams are clearly more complex and extensive than are PEAnet diagrams, which reflect the UML’s ability to represent a wider range of situations than can be represented with PEAnets. This increased power of expression is notable in four ways: more flexible guard conditions, different kinds of triggering actions, the ability to handle multiple events with a single transition, and a notation that clearly defines distinct states.

Comparing Expressive Power

The first way that UML diagrams provide an increase in expressive power is with guard conditions that, unlike PEAnet conditions, support a large set of relational and logical operations, and can access variables or attributes located in any object. UML guard conditions can include multiple operations, variables, and constant values. For example, the guard condition for the Closed to Opening transition in Figure 34 is:

\[
[ \text{upperGate.position} == \text{upperGate.CLOSED} \&\& \text{lock.level} == \text{lock.FULL} ]
\]
This guard condition consists of two subexpressions based on the equality operator (==): The first expression is true if the upper gate closed and the second expression is true if the lock is full. Guard conditions can be made arbitrarily complex by combining subexpressions with the logical operators AND (&&) and OR (||). In the guard condition example above, both subexpressions must true before the transition is allowed.

UML events are similar to PEAnet activities, but the UML defines many kinds of events where PEAnets define only a single kind of activity. The increased number of events is the second way that the UML demonstrates an increased expressive power over PEAnets. Signal events, which are equivalent to PEAnet activities, convey no additional information beyond the initial stimulus. Call events are similar to signal events but also carry additional information in the form of event or function parameters. Change events initiate a transition whenever a predefined condition becomes true, that is, whenever an object’s state changes. The UML also supports two kinds of time events. The first time event triggers a transition at an absolute, or wall-clock, time. The second time event triggers a transition after an elapsed time; for example, after(50u) – after 50-milliseconds. PEAnet-based CBI systems are capable of performing similar behaviors but these behaviors must be subsumed into the implementation of the PEAnet processes (i.e., into the code of the CBI program) where they are not visible. In this case, UML diagrams are “more primary” than are PEAnets.

The third way that UML diagrams provide an increased expressive power is that a single transition may be initiated by more than one event. This situation is denoted by a comma-separated list of events; the occurrence of any event in the list initiates the
transition (if the guard condition is true). PEAnets can represent the same design but only by including a separate entry for each activity and by duplicating all of the dynamic structure that is associated with the entries. This duplication can greatly increase the size and, therefore, the complexity of the design if the duplicated structure includes many processes, consequences, and conditions.

Finally, the UML is an object-oriented language that clearly identifies each object in the system. When an event occurs, it is sent to a specific and easily identified object. Whenever one object refers to another, the reference is unambiguously specified with the dot operator (i.e., the period). The dot operator separates the name of the referenced object (the left operand) from the name of the signal, variable, or symbolic constant (the right operand). For example, as illustrated in Figure 34, the valve object can send a signal to the lock object with the action lock.drain or it can examine the current value of an attribute stored in the lock object with the expression lock.level == lock.FULL.

Conversely, PEAnet diagrams are procedurally-oriented and the process names are defined in a single, global scope or namespace (unlike entities, which are object-oriented). When an activity occurs, it triggers a specific process but that process is bound to an entity only by name, that is, the entity name must be a part of the process name (e.g., Close Valve). This requirement places an additional burden on the diagram reader who must decipher the process name to determine which word or words denote the associated entity.

Furthermore, consequences are not explicitly bound to an entity. Instead, each consequence inherits its entity binding from the process that changes it. For example,
three of the entities, *Valve*, *Upper Gate*, and *Lower Gate* (Figure 31, p.107), have a property named *position* with a legal value of *closed*. Therefore, the meaning of the consequence \( K: \text{position} = \text{closed} \) (Figure 33, p. 109) cannot be fully appreciated until the reader follows the “change” arrow back to the *Close Value* process. This ambiguity is exacerbated when interpreting a condition. For example, condition \( K \) controls the *Fill Lock* process (Figure 31, p. 107). To fully understand the meaning of this condition, the reader must (a) locate the corresponding consequence, (b) identify the associated process (*Close Valve*), and then (c) mentally construct the represented meaning: *the lock can be filled with water only if the valve is closed*.

The differences in their respective powers of expression notwithstanding, a casual comparison of PEAnet (Figure 33, p. 109) and UML state diagrams (Figure 34, p. 110), suggests that PEAnet diagrams are more intuitive than are UML state diagrams. If PEAnet diagrams are more intuitive than are UML state diagrams, then they are better able to serve as a primary representation. However, the two representations do not scale at the same rate. The complexity of a PEAnet representation of a system increases at a higher rate as the size and complexity of a system increases than does the complexity of a corresponding UML representation. The disparity in scaling rates between PEAnet and UML diagrams is an artefact of their respective organizational themes.

**Organization, Clarity, and Scaling**

UML diagrams are organized around system nouns, while PEAnet diagrams are organized around system verbs. PEAnet processing begins with an activity (a verb) that stimulates a system change, and the complete description of that change follows the
activity in the PEAnet diagram (Figure 33, p. 109). Activities serve as the “entry points” of a PEAnet diagram; that is, someone searching for and trying to understand a specific system behavior represented by a PEAnet diagram first searches for the activity that initiates or triggers that behavior.

UML processing begins with an event (also a verb) that stimulates a system change. However, events do not form the entry points of a UML state diagram. Someone searching for and trying to understand a specific system behavior represented by a UML state diagram first locates the object or system subcomponent that provides that behavior, and then locates the state diagram that is associated with that object (Figure 34, p. 110). A state diagram succinctly summarizes the behavior of an object: the overall behavior can be understood at a glance and the detailed behavior comprehended with a more deliberate study of the diagram. For example, it is quickly evident from Figure 34 that the Valve in the canal lock CBI can be open, closed, or somewhere in between. This behavior is trivial, but the diagram also quickly clarifies arbitrary or more complex behaviors: In this CBI the valve cannot reverse direction while it is opening or closing.

UML state diagrams control complexity by separating the behaviors of individual objects or system subcomponents into separate diagrams. Therefore, as the size and complexity of a system increases, most of the added complexity is represented by new state diagrams and the complexity of individual diagrams increases little. In contrast, a single PEAnet diagram must describe all of the complexity that is inherent in a system. As the size and complexity of a system increases, the complexity of the PEAnet increases at a moderate rate if the new behavior is initiated with a new activity – this amounts to
adding a new section to the PEAnet table, which is similar to creating a new diagram. However, if the new behavior must be subsumed into an existing activity, then the complexity of the PEAnet diagram increases at a high rate.

I presented an example CBI system in chapter 1 that exhibits this problem, and presented a detailed analysis of the problem in chapter 2. The example is centered on a portable MP3 player, which has many functions but few controls. The imbalance between the number of functions and the number of controls implies that a PEAnet description of the player consists of few activities, one for each control, but that each activity initiates a long sequence of processes. The correct player behavior (i.e., the desired player function) must be deduced from the sequence of processes while rejecting incorrect behavior. The inability of PEAnets to scale or manage complexity in this special case derives from the three PEAnet limitations described in chapter 2: unintended consequences, inadequate conditions, and indistinct states. I consider inadequate conditions above as a limitation of expressive power. The problems of unintended consequences and indistinct states are a direct result of organizing PEAnet diagrams around activities.

In chapter 2, I illustrated the problem of unintended consequences with the task of pausing and restarting audio playback on the MP3 player. The player only has one button, play, which initiates both behaviors. The behaviors are implemented either with PEAnet processes or with UML actions – or generally with functions. It is desirable to make functions simple, with a single purpose, which is specified by the name of the function.
In a PEAnet diagram, the two functions correspond to a two-process sequence, where the first process triggers the second (Figure 35). Each process is controlled by a condition that corresponds to the consequence of the other process: The Play process is allowed to run if the player is in the pause mode and Pause is allowed to run if the player is in the play mode. However, when Play changes the mode to play, it allows Pause to run and it becomes impossible to restore playback. (The PEAnet solution is to define only one process that toggles between the two modes.) The corresponding UML state diagram solution is uncomplicated (Figure 36), and defines two separate functions.

**Figure 35.** Unintended consequences: the Play process changes mode to play; this allows the Pause process to run, which changes the mode back to pause.

**Figure 36.** UML state diagram: pressing the play button alternately triggers the play and pause actions.
The toggling problem is a minor, yet easy to understand, example of unintended consequences. Unintended consequences are more difficult to see when the conflicting processes are part of process sequences that are triggered by different activities. It is difficult to anticipate unintended consequences and it is difficult to correct them when they are located. Both difficulties increase steeply as the size of the PEAnet increases.

The moment-to-moment behavior of a system can be described by a finite set of states. Each state defines a specific mode or condition in which the system can exist. Transitions between the states define how the system can change or evolve over time. PEAnet diagrams do not explicitly define system states or transitions. Nevertheless, PEAnets implicitly represent the current state of a system but do so with the current values of all the properties defined in the property table. This representation does not support the viewer’s task of understanding system behavior in terms of states and it makes it difficult to add new states when they are needed.

I illustrated this problem in chapter 2 with the task of navigating through a set of menus. The represented system allows the user to return to the previous menu at any time, which requires the system to “remember” how it got into a particular state. A typical solution is to add new states that remember the previous location, but synthesizing those states from property values in a PEAnet diagram is difficult due to the likelihood of creating unintended consequences. The UML state diagram solution (Figure 37) adds enough states so that the path from the menu to the final state is unique, which makes the return path unambiguous.
It is understandable that a state-oriented representation better supports editing and otherwise maintaining these states than does a nonstate-oriented representation. That difference is result of the two organizational themes and underscores another advantage of the UML, which becomes more important as system complexity increases. Complex CBI systems are composed of many subsystems, and the subsystems may be composed of smaller subsystems. The subsystems often interact with each other but may carry out many, and perhaps most, of their operations independently.

The UML represents each subsystem as a distinct object with its own state diagram. Although the state diagrams can be articulated and can interact with each other, they remain distinct and able function independently. Furthermore, complex state diagrams can be decomposed into subdiagrams that separate complex behaviors. By separating complex behavior, that behavior is easier to understand and easier to manage. Conversely, PEA.net diagrams do not separate processes, consequences, and conditions into subcomponents, but maintain them in one monolithic namespace where their

![Figure 37](image)

*Figure 37. The UML state diagram solution to the memory problem: add more states.*
interactions are uncontrolled. The lack of separation also implies that anyone reading the
diagram must attempt to understand it as a monolithic system without the benefit of
subsystem boundaries and independence.

Although UML state diagrams and PEAnet diagrams serve nearly identical
purposes, their differences make a direct comparison regarding which is the better
primary representation difficult. When applied to small, simple CBI systems, PEAnets
are more intuitive and involve less mental translation than do UML state diagrams. In
these cases, PEAnet diagrams are a better primary representation for a larger group of
CBI developers than are UML state diagrams. However, when applied to larger, more
complex systems, the lesser expressive power of PEAnet diagrams, the likelihood of
unintended consequences, the problem of indistinct states, and the monolithic structure of
PEAnet diagrams, detracts from their ability to serve as a primary representation. In
these cases, UML state diagrams are the better primary representation.

Tables, lists, and roadmaps are widely familiar devices. PEAnets are composed
of entity tables and PEAnet diagrams, both of which are specialized tables; because tables
are well understood, I assert that PEAnets qualify as primary representations. UML class
diagrams are similar to lists and, therefore, qualify as a primary representation. UML
state diagrams are similar to roadmaps but also include a large amount of more abstract
information. While some of the notation used on a UML state diagram is not familiar,
the overall roadmap-like structure is familiar and I assert that state diagrams qualify as a
primary representation. UML state diagrams represent large and complex CBI better
than do PEA diagrams, but the latter can be improved with few modifications.
Proposed PEAnet Modifications

The aim of the following five proposed modifications is to make PEAnets more object-oriented. My underlying assumption is that object-orientation can mitigate many of the limitations described above and thereby enable PEAnets to better serve as a primary representation.

1. List process names in the entity table. PEAnet entity tables are currently partially object-oriented in that they bind properties to entities. This modification (Figure 38)

<table>
<thead>
<tr>
<th>Boat</th>
<th>In canal / in lock / in lake / canal_to_lock / Lock_to_lake / lake_to_lock / lock_to_canal</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td>right / left</td>
</tr>
<tr>
<td>level</td>
<td>low / high</td>
</tr>
<tr>
<td>Face Left( )</td>
<td></td>
</tr>
<tr>
<td>Face Right( )</td>
<td></td>
</tr>
<tr>
<td>Move Left( )</td>
<td></td>
</tr>
<tr>
<td>Move Right( )</td>
<td></td>
</tr>
<tr>
<td>Raise( )</td>
<td></td>
</tr>
<tr>
<td>Lower( )</td>
<td></td>
</tr>
<tr>
<td>Valve</td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Close( )</td>
<td></td>
</tr>
<tr>
<td>Open( )</td>
<td></td>
</tr>
<tr>
<td>Lower Gate</td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Close( )</td>
<td></td>
</tr>
<tr>
<td>Open( )</td>
<td></td>
</tr>
<tr>
<td>Upper Gate</td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Close( )</td>
<td></td>
</tr>
<tr>
<td>Open( )</td>
<td></td>
</tr>
<tr>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td>level</td>
<td>full / empty / draining / filling</td>
</tr>
<tr>
<td>Drain( )</td>
<td></td>
</tr>
<tr>
<td>Fill( )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 38. Modified PEAnet entity table grouped by entity, and containing both properties and processes.
completes the object-orientation by also binding processes (names with parentheses) to entities, accruing three direct benefits. First, it simplifies the task of understanding the system’s behavior by collecting the properties that describe a system and the processes that affect those properties in one location. Second, unambiguously binding a process to an entity makes that entity’s properties the default targets of the process, that is, those properties can be accessed without qualification. When a process accesses properties that are bound to other entities, those properties are qualified by the other entity’s name. Third, binding the processes and properties to an entity simplifies and supports the task of translating PEAnets into executable engines.

2. *Partition the PEAnet diagram into multiple diagrams.* Each diagram is associated with one entity, which is the default target for the entity’s processes. That is, any unqualified feature (process name, property name, or property value) must belong to the bound entity. The features of other entities are accessed with the dot operator (i.e., the period). The dot operator joins an entity (the left operand) with one of its features (the right operand). For example, in the modified *Valve* PEAnet diagram (Figure 39), the process *Close* refers unambiguously to the *Valve* entity and *Boat.Raise* denotes the *Raise* action defined in the *Boat* PEAnet diagram. Similarly, the property and the value in the condition *position == opened* refers to the *Valve*, but the property and the value in the condition *Lower Gate.position == Lower Gate.closed* clearly refers to the *Lower Gate* entity.

3. *Send activities to all responding entities.* An activity appears exactly once in current PEAnet diagrams but under this proposed modification, an activity would appear in
any PEAnet diagram where the associated entity exhibits instructionally significant behavior in response to the activity. The order that an activity is delivered to multiple entities is unspecified and concurrent. Whenever the activity must be delivered to entities in a specific order or whenever the triggered processes must execute sequentially, then one PEAnet diagram will trigger a process in a second diagram. To accomplish this interaction, the PEAnet diagrams must be articulated with the dot operator, the name of the receiving entity, and the name of the target process in the

<table>
<thead>
<tr>
<th>Activity / Process</th>
<th>change</th>
<th>Consequence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press Close Valve Button</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Begin</td>
<td></td>
<td>display(Wait for lock to drain)</td>
<td>Lock.level == Lock.draining</td>
</tr>
<tr>
<td>Break</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td></td>
<td>position == closed</td>
<td>position == opened &amp;&amp; Lower Gate.position == Lower Gate.closed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>display(close lower gate)</td>
<td>Lower Gate.position == Lower Gate.closed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>display(Wait for lower gate to open)</td>
<td>Lower Gate.position == Lower Gate.closing</td>
</tr>
<tr>
<td>Lockfill</td>
<td></td>
<td>Lock.level = Lock.filling</td>
<td>position == closed</td>
</tr>
<tr>
<td>Boat.Raise</td>
<td></td>
<td>Boat.level = Boat.high</td>
<td>Boat.location == Boat.in lock &amp;&amp; Lower Gate.position == Lower Gate.closed</td>
</tr>
</tbody>
</table>

**Figure 39.** Modified PEAnet diagram for the Valve entity.
receiving PEAnet diagram. This notation is described in number 2 above.

4. *Replace consequence labels in conditions with full boolean-valued expressions.*

Conditions are currently represented by consequence labels, and the designated consequences are each treated as a test for equality. The comma in a comma-separated list of conditions is implicitly treated as the logical AND operation. This notation is very compact but it is difficult to read, especially in long PEAnet diagram where the labels and the associated consequences are separated by many pages. This modification (Figure 39) will sacrifice compactness but will improve clarity and eliminate the need to search for the consequences matching a given condition. This modification solves one of the most pressing problems with PEAnets described in chapter 2. In conjunction with this replacement, add support for all relational operators EQUAL, NOT EQUAL, LESS THAN, LESS THAN OR EQUAL, GREATER THAN, and GREATER THAN OR EQUAL (==, !=, <, <=, >, and >=) and the common logical operators AND and OR (&& and ||).

5. *Add standard programming structures.* Imperative programming languages consist of statements that instruct the computer to perform a task. PEAnet diagrams form a simple imperative language. That language can be made more complete by adding three kinds of notational elements. (a) Add two special grouping processes: _Begin, and _End. The standard processes that are grouped together between these special processes are triggered only if the condition associated with the _Begin process is true (Figure 39). (b) Permit the process-to-process triggering arrow to return to a previous process in the sequences, which will form a loop at the bottom of the process
sequence. Once entered, the loop will continue indefinitely until it is terminated by
the following special process. (c) Add a special interrupting process named _Break.
I implemented the _Break process in the current version of the PEAnet system
described in chapter 2, and its basic functionality remains unchanged. The _Break
process may be placed at any position within the loop. Whenever a _Break process is
allowed to run (i.e., whenever its corresponding conditions are true), it terminates the
process sequence, including a process loop.

The significance of these modifications is underscored by an early description of
the object-oriented paradigm. “Object-oriented modeling and design is a new way of
thinking about problems using models organized around real-world concepts. The
fundamental construct is the object, which combines both data structure and behavior in a
Modification 1 results in PEAnet entity tables that “combine both data structure and
behavior,” and that parallel the content and organization of UML class diagrams.
Modifications 2 through 5 result in PEAnet entity diagrams that are similar to UML state
diagrams.

If the modified version of PEAnets and UML diagrams serve essentially the same
purpose, is there a compelling reason to choose one or the other? A modified PEAnet
entity table (Figure 38) and a UML class diagram (Figure 18, p. 70) both describe data
structure and behavior, but the class diagram includes far more detail. Similarly, a
modified PEAnet diagram (Figure 39) and a UML state diagram (Figure 23, p. 80)
describe the dynamic behavior of a system but the state diagrams contain more detail.
The increased detail of both UML diagrams and the free-form organization of state diagrams suggest that the PEAnet versions are easier for nonsoftware engineers to read and are, therefore, a more suitable primary representation. Nevertheless, there are two additional compelling arguments in favor of adopting the UML.

Both of these arguments in favor of the UML are a product of the language’s success and popularity. First, availability, there are many UML modeling tools currently available that span broad ranges of cost, features, and support. Most of these tools conform to the current UML standard. The tools are, therefore, consistent and, at least to a degree, compatible. The availability of full-featured and well-supported UML modeling tools obviates the need to develop and maintain a specialized modeling tool as is the case for PEAnets.

The second and the most fundamental argument in favor of the UML is that its wide use implies that it is a nearly universal modeling language. That is, the UML is understood and utilized by software engineers working a variety of domains. Within a specific domain, architects, analysts, designers, programmers, managers, and others each use some, but typically not all, of the UML features. By adopting the UML as a primary representation, CBI developers gain a design language or a representation that is already in use and understood by the software engineering community that they frequently partner with in the CBI development process.

**Conclusion**

Architects, choreographers, and engineers have developed design notations that utilize symbols that mimic, to some extent, the physical elements of the “real world” to
which they correspond. Two characteristics of these and similar disciplines make this mimicry possible. First, many of the elements that these disciplines deal with are concrete and the symbols mimic the appearance or some other physical property of the elements. Second, the scopes of these disciplines are restricted in the sense that each deals with a limited set of elements that are known in advance. Conversely, PEAnets and UML diagrams deal with abstract concepts for which there are no tangible, physical elements and both representations are frequently called upon to represent concepts conceived long after their symbols were finalized. To satisfy these constraints, PEAnets and UML diagrams define a limited set of abstract symbols based on few pervasive and highly generalized concepts.

PEAnets and UML diagrams use a combination of graphical and textual notations to convey the information embodied in a design. The notations are abstract but follow familiar organizations such as tables and lists. I maintain that the familiar organizations underlying the notations, coupled with a modest amount of training, are sufficient to make PEAnets and UML diagrams primary to experienced CBI developers. The ways in which the notational elements of both representations are combined are governed by strict, well-defined rules. The example CBI systems described in the previous chapters demonstrate that the combining rules are sufficiently robust to make both representations unequivocal.

Although I favor UML diagrams, that preference should not be interpreted as denigrating PEAnets. PEAnets were created by instructional developers, specifically by CBI developers, and not by software engineers. From that perspective, PEAnets have
endured well the two decades of their existence. PEAnets were created by CBI developers who saw a need for a CBI development representation. And PEAnets demonstrate that the needed representation is possible. This is not a bad role for PEAnets to play in the history of CBI.

The UML was born from the union of three successful object-oriented modeling languages. During its development and since its introduction, the UML has enjoyed considerable support from the software engineering community. That support has taken many forms, notably in receiving input that has shaped the notation and the semantics of the language so that it satisfies the needs of most software engineers. It is reasonable to presume that if the UML satisfies the needs of general software engineering, that it should also satisfy the needs of CBI development. Although neither discipline is a proper subset of the other, there is considerable overlap between the two, particularly in the area of actually implementing CBI. I firmly believe that an unequivocal primary CBI representation is possible and I believe that the UML is a strong candidate for that representation.

References


“Design is a creative process of engineering synthetic solutions to satisfy human needs” (Suh, 2000, para. 1). This broad definition encompasses a multitude of disciplines from general engineering (Vincenti, 1990, pp. 6-7) to instructional development (Gustafson & Branch, 2002, p. xii; Seels & Richey, 1994). Suh (2000) maintained that tools are one of the primary elements of design and suggested that “designers use computers as a tool to augment human capability” (para. 3). However, two prerequisites must be satisfied before computers are able to serve as a design tool. The first prerequisite is a symbolic representation of the evolving design that designers can understand and control, and that a computer can store, display, and manipulate. The second prerequisite is a software system that interfaces the human designer with the computer hardware.

The first prerequisite is problematic in the CBI design subdomain. Eckel (1993) observed that CBI designers are faced with not one, but two representations: “Present-day practice of computer-based instruction is characterized by working with twofold representation: script and program” (p. 7). He characterized scripts as primary and implied that they are a useful “basis for communication” (p. 7), but he also characterized scripts as equivocal. It was his view that programs are unequivocal but that programs are also secondary, meaning that “what is important in terms of subject-matter or instructional method disappears in the jungle of codification” (p. 7).
Contemporary CBI developers must choose between a representation that is primary but equivocal or a representation that is unequivocal but secondary. This dichotomy is the root of my principal research question: Is a representation that is both unequivocal and primary possible? In conjunction with this research question, I also established a series of project objectives to direct my research and to serve as the final evaluation criteria. Based on these objectives, I decomposed my principal research question into four more simple and more easily answered auxiliary questions.

Is it Possible to Rigorously Define *Unequivocal* and *Primary*?

The concepts represented by the terms *unequivocal* and *primary* are independent, which makes it possible and appropriate to examine their definitions individually. First, I argue that it is sufficient to demonstrate that a representation is computable to assert that it is unequivocal. This argument is based on observations that instruction expressed as a computer program is unequivocal, or at least less equivocal than instruction expressed in scripts or natural language (Eckel, 1993, p. 7; Stolurow, 1969, p. 66). However, I do not claim that computability is a necessary condition for an unequivocal representation. Arguing for sufficiency but not for necessity may set the standard needlessly high for an unequivocal representation; that is, the argument may exclude representations that are in fact unequivocal but that are not computable. Unequivocal but noncomputable representations are less useful and are more difficult to establish as unequivocal than are computable representations.

My first research objective, which served as the guide for answering the first two auxiliary questions, was to establish that PEAnets and UML diagrams are unequivocal.
This objective consisted of four subobjectives: (a) develop an automatic PEAnet translator, (b) create a PEAnet editor, (c) identify an appropriate UML diagramming tool, and (d) develop an automatic UML translator. All of these software systems were successfully completed. The systems satisfying objectives (a) and (d) demonstrate that PEAnets and UML diagrams are computable; the systems also demonstrate that it is possible to define an unequivocal representation and to establish that a candidate representation is unequivocal. Together, the four software systems also satisfy the second prerequisite for computers to able to serve as a design tool by providing the tools and interfaces that allow a human designer to manipulate a symbolic representation of a CBI system design.

Rigorously defining a primary representation proved to be a more difficult task than defining an unequivocal representation. My definition of a primary representation begins with Eckel’s (1993, p. 7) example of storyboards as a primary representation. The definition continues by noting that the value of a storyboard is that it allows “anyone. . . concerned with the production. . . [to] see the plot of an entire movie spread out in front of him” (Finch, 1973, p. 82). A primary representation is well-characterized by Alexander’s (1979) attempt “to understand the ‘structure’ of something” through “a simple picture of it, which lets me grasp it as a whole” (p. 81). In the context of CBI, a primary representation should make the dynamic components of instruction easy to see and easy to understand.

A primary representation that allows the viewer to grasp the behavior of system as a whole is intuitively valuable but elusive. Primary representations are elusive because
they are also subjective – what one viewer grasps as a whole, another viewer will not. I maintain that a representation is primary if (a) it embodies a notation that is sufficiently rich to describe the dynamic behaviors of CBI, and (b) its use as a CBI design language or as a design language in a similar domain is demonstrated in the published literature.

Part (a) of the definition requires that that the representation is sufficiently robust to describe a CBI system. A representation, no matter how simple and easily understood, is of no value if it cannot practically and usefully describe a realistic CBI system. The intent of part (b) of the definition is to establish that at least one independent observer considers the representation to be understandable and a suitable design language.

My second research objective was to develop a set of complex, realistic CBI systems. I developed four example CBI systems, two designed with PEAnets and two designed with UML diagrams, to satisfy this objective. The CBI examples also demonstrate that PEAnets and UML diagrams are sufficiently robust to describe CBI systems, which satisfies part (a) of the definition of a primary representation. Chapters 2 and 3 outlined the development of PEAnets and UML diagrams respectively. Each chapter included citations detailing the development and use of both representations as design languages. The cited research implies that practitioners within their respective design communities consider the representations understandable and therefore primary. The citations satisfy part (b) of the definition of a primary representation.

The PEAnet and UML translators demonstrate that it is possible to specify a computable and, therefore, an unequivocal representation. The CBI examples demonstrate that the PEAnet and UML notations and semantics are sufficiently powerful
to represent CBI systems. Finally, there are sufficient references in the literature to PEAnets and to UML diagrams to establish that at they are viewed as appropriate design languages by some design practitioners. I conclude from these observations that it is possible to define the terms _unequivocal_ and _primary_ with sufficient precision to evaluate a candidate representation and determine if it meets the criteria to be classified as an unequivocal primary representation. Answering the first auxiliary question sets the stage for answering the remaining questions.

**Is it Possible Either to Identify or to Create an Unequivocal Primary Representation?**

In the course of answering the first auxiliary research question, I established that both PEAnets and UML diagrams are unequivocal primary representations. These two languages are surely not the only candidates available, but they are sufficient to answer the second question affirmatively. I have also argued strongly in favor of using the UML to represent the dynamic components of CBI systems. A cautionary parallel can be drawn from the early history of high-level computer programming languages.

Prior to the development of high-level computer programming languages, “programmers wrote symbolic machine instructions exclusively (some even used absolute octal or decimal machine instructions)” (Knuth & Pardo, 1976, p. 61). Even experienced programmers were forced to laboriously reconstruct the meaning represented by these programs from the arcane symbols. Numerous “automatic programming” languages were proposed or developed to make programming faster and the final programs easier to read. As early high-level programming languages proliferated,
working groups from Europe and America met in 1958 and began developing Algol, “the language: the universal, international, machine-independent language for expressing scientific algorithms” (Webber, 2003, p. 538).

Algol was, in almost every way, superior to the 1957 FORTRAN programming language, but Algol was unable to unseat the older language. “Algol never succeeded in becoming the language for scientific computation. . . . The time for a universal language, even within a single application area such as scientific computation, has not yet come. It may never come” (Webber, 2003, p. 541). Similarly, I have argued for the UML as a universal unequivocal primary representation. To its advantage, the UML is more abstract and, therefore, more broadly applicable than is any programming language. Nevertheless, PEAnets or some other representation may be more suitable in some situations. It is possible that a universal unequivocal primary representation will never be developed; it is also possible that such a representation has already been developed and is in wide use. That conundrum can only be solved retrospectively.

**Is it Feasible to Represent CBI Systems with an Unequivocal Primary Representation?**

The CBI examples presented previously clearly demonstrate that it is possible to implement CBI systems with either representation. The examples also demonstrate that PEAnets and UML diagrams satisfy the first prerequisite for using computers as a design tool by providing a symbolic notation for representing, manipulating, and storing a design. However, the shortcoming of a practical demonstration is that it fails to distinguish between what can be done and what should be done. It is difficult to fairly
compare the time and effort needed to complete the PEAnet and UML versions of the example CBI systems. Broadly, however, it did take longer and was more difficult to implement the examples with PEAnets than it was to implement the same examples with UML diagrams. Again, an illuminating parallel with computer programming languages clarifies the relation between PEAnets and UML diagrams.

It took an experienced programming team three years to write the first FORTRAN compiler, which was incomplete and flawed when it was first delivered (Knuth & Pardo, 1976, pp. 60, 87). Today, in contrast, the task of writing compilers is understood well enough and compiler development tools are sufficiently available, so that undergraduate students in computer science typically create a simple compiler by themselves in a single semester. No one interprets the difference as suggesting that FORTRAN should not have been developed. Rather, much of what is known today about how to write a compiler, especially the meta-language used to specify a language’s grammar, has its roots in FORTRAN. PEAnets have similarly made their contributions to the search for an unequivocal primary representation.

Is it Possible to Automatically Translate an Unequivocal Primary Representation into Executable CBI Components?

The PEAnet and UML development systems, in conjunction with the example CBI systems, demonstrate that is possible to automatically and unambiguously translate these representations into executable components. In both cases, classes defined in the Java™ programming language represent the various notational elements: Activity, Entity, Consequence, State, Transition, and so forth. The translators for both PEAnets and UML
diagrams consist of two parts. The first part examines the representation and extracts the
details about the represented system. The second part dynamically creates or instantiates
objects from the classes that define the notational elements and configures the objects to
implement the system. In the PEAnet system, some of the objects are chained together
and the head of the chain is stored in a fast-lookup data structure (i.e., a hash table). In
the UML systems, the objects are chained together to form state machines and one
variable per machine maintains a reference to each machine’s the current state. Many
techniques are possible, but one is sufficient to answer the question in the affirmative.

The source code for the translators and the demonstration CBI systems is
available under the GNU General Public License (GPL). Appendix A describes how to
retrieve and build all of the system and example source code described here dissertation.

Secondary Research Objectives

Three of my research objectives are secondary in the sense that they did not
directly support my research question. The features that satisfy these objectives are,
evertheless, important from a practical perspective.

1. *Develop diagnostic and debugging support.* The development systems satisfy this
requirement in two ways. First, each system provides error detection and reporting
during the translation process. Although neither system detects and reports all errors,
the errors that are detected and reported are the ones that proved to be most useful
while developing the example CBI systems and are, therefore, likely to represent the
most common errors. Second, each system provides devices for symbolically
following system execution and for examining the values of system variables on demand. These devices are activated by enabling Java™ assertions.

2. **Support the most common instructional platforms.** Today, four systems dominate industry and education: Unix®, Linux®, Windows®, and Macintosh®. I chose the Java programming language as the development language because it runs on all four systems, and the development and runtime systems are available at no cost. The executable CBI components created with either system are integrated with a complete CBI system implemented in Java, which satisfies this research objective.

3. **Simplify concurrency.** Concurrency is the illusion that two or more activities are taking place at the same time, and, possibly cooperating. This is a programming technique that is familiar to software engineers, but it may be less familiar to CBI designers (Gibbons, 1998/2001, p. 536). Although CBI developers may lack experience implementing concurrency, it is an essential feature in large, complex software systems. I simplified concurrency in PEAnets by extending the process-to-process triggering notation. This extension, described in chapter 2, allows a CBI developer to select either sequential or concurrent process triggering. Sequential or synchronous processes execute one at a time. For example, if process P₀ triggers process P₁ (P₀ → P₁), P₀ completes before P₁ begins. Alternatively, concurrent or asynchronous processes appear to execute at the same time (an illusion created by quickly switching the central processing unit between the processes). Concurrency is inherent in UML state diagrams. Actions, appearing on the transition arrows are sequential; activities, appearing inside of states, are on-going, concurrent operations.
Additionally, change and time events are implemented with concurrent operations and execute asynchronously. (UML state diagrams include explicit notation for the concurrency operations of fork and join, but this notation is not supported by the present UML translator.) In summary, all of the research objectives are satisfied, and all of the auxiliary questions have been answered. From this position, it is possible to make a final, fundamental conclusion.

**Conclusion**

So, the real work of any process of design lies in this task of making up the language, from which you can later generate the one particular design. . . . But of course, once you have it, this language is general. If it has the power to make a single building live, it can be used a thousand times, to make a thousand buildings live. (Alexander, 1979, p. 324)

In the first chapter, I proposed that the task of identifying an unequivocal primary representation is better understood and is more easily completed by resolving a candidate representation into two independent dimensions: unequivocal-equivocal and primary-secondary. I also argued that demonstrating that a representation is computable is sufficient, although not necessary, to assert that it is unequivocal.

In chapters 2 and 3, I described PEAnets and UML state diagrams in detail and established that both are unequivocal by demonstrating that they are computable. I established the computability of both representations with practical demonstrations of systems that automatically, unambiguously, and consistently translate the diagrams into executable components that can be incorporated into a CBI system. At the same time, I argued that by intention and use, PEAnets and UML diagrams also qualify as primary representations.
In chapter 4 I compared and analyzed PEAnets and UML diagrams in the context of their visible symbols and their underlying combining rules. I argued here that the symbols of both representations are organized in ways familiar to most CBI designers and that the combining rules of both representations are sufficiently well defined to support computation. Based on these arguments, I concluded that both PEAnets and UML diagrams are primary and unequivocal representations.

The answer to my fundamental research question is that an unequivocal primary representation is possible; furthermore, it is also desirable. A practical representation allows CBI developers to design, communicate, and evaluate CBI systems. It also allows them to automatically convert the representation into executable CBI components in such a way that human-introduced translation errors are avoided. Two languages, PEAnets and UML diagrams, are fully established as unequivocal primary representations.

References


APPENDICES
Appendix A. System Source Code and Construction
The software that I developed for and describe in this dissertation consists of six separate Java packages and two example CBI systems. Each CBI example consists of a UML and a PEAnet version. The eight software components are:

1. **common** A collection of classes in common with many system components
2. **shellplayer** A general CBI delivery program
3. **jsm** Java State Machine Library
4. **smc** State Machine Compiler
5. **pea** PEAnet editor and translator
6. **expr** An expression evaluator used by UML guard conditions
7. **Canal Lock** The complete canal lock example
8. **MP3 Player** The complete MP3 player example

Additionally, the software uses the JLayer 1.0 MP3 decoder library. The source code for both the 1.0 version and the current version of the decoder is available from SourceForge: http://sourceforge.net/projects/javalayer/

**Source Code Availability**

The source code for the eight components listed above may be downloaded from http://icarus.cs.weber.edu/~dab/dissertation

All project source code is released under the current version of the GNU General Public License (GPL) detailed at http://www.gnu.org/licenses/gpl.html

**Project Build Tools**

I used several different tools to compile the project code. Most of these tools are available at no cost from the Internet. A Java compiler is required to build the project.
An automated construction utility, either make or ant, is strongly recommended. If you modify the XML parser embedded in the state machine compiler (smc), you will also need to install JFlex.

- **Java.** The Java Development Kit (JDK) contains the Java compiler and runtime system. Alternatively, the Java Runtime Environment (JRE) is sufficient to just run the CBI examples. (The JDK includes the JRE.) Current versions of the JDK and the JRE may be downloaded from Oracle’s Sun Developer Network:
  
  http://java.sun.com/javase/downloads/index.jsp

- **Automated build utility.** I developed the translators and CBI examples with a proprietary version of the make utility. I have included all of the original makefiles with the source code. I have also recently written an ant script to manage the build process. As make and ant perform essentially the same task, you only need one of them.

  - Public domain versions of make are often available on the Internet or the Microsoft version, named “nmake”, may be used. However, it may be necessary to modify the make-files when using nmake as it may not be fully compatible with the provided make-files. A binary version of make may be downloaded from ftp://ftp.delorie.com/pub/djgpp/current/v2gnu/00_index.txt
    
    The make source code is available from http://ftp.gnu.org/pub/gnu/

  - The ant program may be downloaded at no cost from The Apache Ant Project:
    
    http://ant.apache.org/
• *JFlex*. JFlex is a parser generator that generates Java code. The most recent version may be downloaded from http://jflex.de/

**Project Construction**

I developed the research project system on a Windows XP® system using the Java™ programming language (originally version 5 and later version 6). A set of programs, implemented as batch files, makefiles, and ant scripts, automate the project construction process. The construction programs are included with the source code. The construction programs target the Windows operating system but can also serve as a guide for project construction on other platforms.

1. Download the software, packaged in a jar file named representation.jar, from http://icarus.cs.weber.edu/~dab/dissertation
2. Place the jar file in a dedicated directory and extract the contents with the command
   ```
   jar xvf representation.jar
   ```
3. Specify the MP3 files for the simulated MP3 player.
   3.1. Edit the files `representation\mp3player\pea\tracks.txt` and `representation\mp3player\uml\tracks.txt` (or edit one file and copy it) so that it describes your MP3 files. (128kbps is the only bit rate supported at this time). Each line in the file contains the following data fields (in the specified order) separated by colons: MP3 file name, track name, artist name, album name, genre, track number, and track duration in seconds.
   3.2. Edit the two files named MP3Player.java (these files are not the same) located in `representation\mp3player\pea` and `representation\mp3player\uml`
3.2.1. At the top of the class, edit the variable musicBase to indicate the location of the MP3 files. The location may be on a local disk or it may be on a web site. Examples of both are provided in source code. If the MP3 files are located on a web site, the simulations require that both the client and the server are connected to the Internet via a broadband connection. The simulations work well with a 1.5Mbps or greater connection but may stall intermittently or may fail to play during times of high Internet traffic.

3.2.2. Also at the top of the class, edit the variable tracksFile to indicate the name of the file containing the track data. The current file name is tracks.txt (see 3.1 above) but the name may be changed.

4. Build the complete system with the make utility (4.1) or the ant utility (4.2).

4.1. The top-level makefile, named Makefile, supports three build operations and requires that the Java classpath be initialized before the make process begins.

4.1.1. Edit the file named setpath.bat to update the environment variable build_home to indicate the full path name of the directory containing the source code files extracted from the jar file.

4.1.2. Execute setpath from the same command prompt window that will be used to run the following make commands. This step must be repeated whenever a new command prompt window is used in the construction process.

4.1.3. make: Compiles all of the source code, builds the documentation (not all packages are currently documented), makes the library and CBI jar files.

4.1.4. make clean: Removes all of the intermediate files.
4.1.5. *make clobber*: Removes all files created by the build process leaving only
the source code files.

4.2. The top-level ant build file, named build.xml, supports four build operations.

(Please note that the make and ant build processes are not compatible. If you
build any files with make, you must clobber them before running ant.)

4.2.1. *ant*. Compiles all of the source code and makes the library and CBI jar
files.

4.2.2. *ant docs*. Builds the documentation for the documented packages (not all
packages are currently documented).

4.2.3. *ant clean*. Removes all of the intermediate files.

4.2.4. *ant clobber*. Removes all files created by the build process leaving only
the source code files.

4.2.5. The ant build process is not compatible with the make build process. If
the build was started with make, you must do a make clobber before
proceeding with an ant build.

5. Edit the file named launch.bat by updating the environment variable build_home to
indicate the full path name of the directory containing the generated jar files.

6. The batch file launch.bat includes the commands to launch all of the examples
described in the dissertation. Execute the program launch to display a menu of and to
launch each example program.
Appendix B. Canal Lock PEAnet Diagrams
<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boat</td>
<td>location</td>
<td>in canal / in lock / in lake / canal_to_lock / lock_to_lake / lake_to_lock / lock_to_canal</td>
</tr>
<tr>
<td></td>
<td>direction</td>
<td>right / left</td>
</tr>
<tr>
<td></td>
<td>level</td>
<td>low / high</td>
</tr>
<tr>
<td>Valve</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Lower Gate</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Upper Gate</td>
<td>position</td>
<td>closed / opened / opening / closing</td>
</tr>
<tr>
<td>Lock</td>
<td>level</td>
<td>full / empty / draining / filling</td>
</tr>
</tbody>
</table>
Appendix C. Canal Lock UML Diagrams
Appendix D. MP3 Player CBI PEAnet Diagrams
## MP3 Player Simulation PEAnet Diagrams

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3Player</td>
<td>mode</td>
<td>off / loading / playing / paused / menu / stopping</td>
</tr>
<tr>
<td>light</td>
<td></td>
<td>off / on</td>
</tr>
<tr>
<td>display</td>
<td></td>
<td>blank / playing / menu / play music / modes / settings / extras / information / Play All / Artist / Album / Songs / Genre / Year / Audio Playback / FM Radio / Playback / Sound Effect / Power / Display / Set Time / Restore Default / Stop Watch / Equalizer / SRS WOW / Focus / Tru Base / SRS 3D / Auto Power Off / Sleep / Backlight / Contrast / Shutdown / Hold</td>
</tr>
<tr>
<td>item</td>
<td></td>
<td>void / play music / modes / settings / extras / information / Play All / Artist / Album / Songs / Genre / Year / Audio Playback / FM Radio / Playback / Repeat / Shuffle / Sound Effect / Equalizer / SRS WOW / WOW / Focus / Tru Base / SRS 3D / Power / Display / Set Time / Restore Default / Stop Watch / Auto Power Off / Sleep / Backlight / Contrast / low / mid / high / ST year / ST mon / ST day / ST hour / ST min / ST sec /</td>
</tr>
<tr>
<td>selecting</td>
<td></td>
<td>off / false / true</td>
</tr>
<tr>
<td>repeat mode</td>
<td></td>
<td>off / one / all</td>
</tr>
<tr>
<td>shuffle mode</td>
<td></td>
<td>off / on</td>
</tr>
<tr>
<td>menu</td>
<td></td>
<td>off / on</td>
</tr>
<tr>
<td>focus</td>
<td></td>
<td>low / mid / high</td>
</tr>
<tr>
<td>WOW mode</td>
<td></td>
<td>off / on</td>
</tr>
<tr>
<td>power</td>
<td></td>
<td>off / on</td>
</tr>
<tr>
<td>stop watch</td>
<td></td>
<td>stopped / running</td>
</tr>
<tr>
<td>battery charge</td>
<td></td>
<td>100 / 87.5 / 75 / 50 / 25 / 0</td>
</tr>
<tr>
<td>Hold Switch</td>
<td>position</td>
<td>release / hold</td>
</tr>
<tr>
<td>PlayDisplay</td>
<td>icon</td>
<td>pause icon / play icon</td>
</tr>
<tr>
<td></td>
<td>meter</td>
<td>time / volume</td>
</tr>
<tr>
<td></td>
<td>hold</td>
<td>off / on</td>
</tr>
<tr>
<td></td>
<td>information</td>
<td>off / on</td>
</tr>
<tr>
<td>Activity</td>
<td>Trigger</td>
<td>Process</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>AN: item = Audio Playback</td>
<td>MV, F, AO</td>
<td></td>
</tr>
<tr>
<td>AO: item = FM Radio</td>
<td>MV, F, AN</td>
<td></td>
</tr>
<tr>
<td>AP: item = PlayBack</td>
<td>MV, F, AQ</td>
<td></td>
</tr>
<tr>
<td>AQ: item = Sound Effect</td>
<td>MV, F, AR</td>
<td></td>
</tr>
<tr>
<td>AR: item = Power</td>
<td>MV, F, AS</td>
<td></td>
</tr>
<tr>
<td>AS: item = Display</td>
<td>MV, F, AT</td>
<td></td>
</tr>
<tr>
<td>AT: item = Set Time</td>
<td>MV, F, AU</td>
<td></td>
</tr>
<tr>
<td>AU: item = Restore Default</td>
<td>MV, F, AP</td>
<td></td>
</tr>
<tr>
<td>AV: item = Auto Power Off</td>
<td>MV, KW, AW</td>
<td></td>
</tr>
<tr>
<td>AW: item = Sleep</td>
<td>MV, KW, AV</td>
<td></td>
</tr>
<tr>
<td>AX: item = Backlight</td>
<td>MV, KK, AY</td>
<td></td>
</tr>
<tr>
<td>AY: item = Contrast</td>
<td>MV, KK, AX</td>
<td></td>
</tr>
<tr>
<td>AZ: item = Play All</td>
<td>MV, DQ, BA</td>
<td></td>
</tr>
<tr>
<td>BA: item = Artist</td>
<td>MV, DQ, BB</td>
<td></td>
</tr>
<tr>
<td>BB: item = Album</td>
<td>MV, DQ, BC</td>
<td></td>
</tr>
<tr>
<td>BC: item = Songs</td>
<td>MV, DQ, BD</td>
<td></td>
</tr>
<tr>
<td>BD: item = Genre</td>
<td>MV, DQ, BE</td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>BE: item = Year</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF: item = Repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG: item = Shuffle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH: item = Equalizer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bi: item = SRS WOW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BJ: item = WOW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BK: item = Focus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL: item = Tru Base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM: item = SRS 3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BN: item = low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BO: item = mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP: item = high</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MP3Player

previousMenuItem() MV, KI
previousMenuItem() MV, KJ
previousMenuItem() MV, KL
previousMenuItem() MV, KP
previousMenuItem() MV, KQ
<table>
<thead>
<tr>
<th>Activity</th>
<th>trigger</th>
<th>Process</th>
<th>change</th>
<th>Consequence</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA:</td>
<td>repeat mode = all</td>
<td>I, MV, BZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CB:</td>
<td>repeat mode = off</td>
<td>I, MV, CA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC:</td>
<td>item = play music</td>
<td>MV, F, AM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CD:</td>
<td>item = modes</td>
<td>MV, F, AI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CE:</td>
<td>item = settings</td>
<td>MV, F, AJ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CI:</td>
<td>item = extras</td>
<td>MV, F, AK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG:</td>
<td>item = information</td>
<td>MV, F, AL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH:</td>
<td>item = PlayBack</td>
<td>MV, F, CO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CJ:</td>
<td>item = Audio playback</td>
<td>MV, F, CJ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CK:</td>
<td>item = Sound Effect</td>
<td>MV, F, CH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL:</td>
<td>item = Power</td>
<td>MV, F, CK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM:</td>
<td>item = Display</td>
<td>MV, F, CL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CN:</td>
<td>item = Set Time</td>
<td>MV, F, CM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO:</td>
<td>item = Restore Default</td>
<td>F, CN, MV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP:</td>
<td>item = Auto Power Off</td>
<td>MV, KW, CQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CQ:</td>
<td>item = Sleep</td>
<td>MV, KW, CP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
<td>chance</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>CR: item = Backlight</td>
<td></td>
<td></td>
<td></td>
<td>MV, KX, CS</td>
<td></td>
</tr>
<tr>
<td>CS: item = Contrast</td>
<td></td>
<td></td>
<td></td>
<td>MV, KX, CR</td>
<td></td>
</tr>
<tr>
<td>CT: item = Play All</td>
<td></td>
<td></td>
<td></td>
<td>MV, DQ, CY</td>
<td></td>
</tr>
<tr>
<td>CU: item = Artist</td>
<td></td>
<td></td>
<td></td>
<td>MV, DQ, CT</td>
<td></td>
</tr>
<tr>
<td>CV: item = Album</td>
<td></td>
<td></td>
<td></td>
<td>MV, DQ, CU</td>
<td></td>
</tr>
<tr>
<td>CV: item = Songs</td>
<td></td>
<td></td>
<td></td>
<td>MV, UJ, LV</td>
<td></td>
</tr>
<tr>
<td>CX: item = Genre</td>
<td></td>
<td></td>
<td></td>
<td>MV, DQ, CW</td>
<td></td>
</tr>
<tr>
<td>CY: item = Year</td>
<td></td>
<td></td>
<td></td>
<td>MV, DQ, CX</td>
<td></td>
</tr>
<tr>
<td>CZ: item = Shuffle</td>
<td></td>
<td></td>
<td></td>
<td>MV, DA</td>
<td></td>
</tr>
<tr>
<td>DA: item = Repeat</td>
<td></td>
<td></td>
<td></td>
<td>MV, CZ</td>
<td></td>
</tr>
<tr>
<td>DB: item = Equalizer</td>
<td></td>
<td></td>
<td></td>
<td>MV, DC</td>
<td></td>
</tr>
<tr>
<td>DC: item = SRS WOW</td>
<td></td>
<td></td>
<td></td>
<td>MV, DB</td>
<td></td>
</tr>
<tr>
<td>DD: item = WOW</td>
<td></td>
<td></td>
<td></td>
<td>MV, DG, EE</td>
<td></td>
</tr>
<tr>
<td>DE: item = Focus</td>
<td></td>
<td></td>
<td></td>
<td>MV, DD, EE</td>
<td></td>
</tr>
<tr>
<td>DF: item = Tru Base</td>
<td></td>
<td></td>
<td></td>
<td>MV, DE, EE</td>
<td></td>
</tr>
<tr>
<td>DG: item = SRS 3D</td>
<td></td>
<td></td>
<td></td>
<td>MV, DF, EE</td>
<td></td>
</tr>
<tr>
<td>DH: item = low</td>
<td></td>
<td></td>
<td></td>
<td>MV, KD, DJ</td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
<td>change</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------</td>
<td>----------------------</td>
<td>--------</td>
<td>-------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>MP3Player Play All</td>
<td></td>
<td></td>
<td></td>
<td>DM: mode = playing</td>
<td>MV, AZ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Next Meter Value</td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, DZ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, EA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, EB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, EC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, EK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nextMeterValue()</td>
<td>MV, KF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Set Focus</td>
<td></td>
<td>DN: focus = low</td>
<td>MV, KD, BN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DO: focus = mid</td>
<td>MV, KD, BO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DP: focus = high</td>
<td>MV, KD, BP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set MP3Player Display</td>
<td></td>
<td>DQ: display = play music</td>
<td>MV, K, AI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DR: display = modes</td>
<td>MV, K, AJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DS: display = settings</td>
<td>MV, K, AK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DT: display = extras</td>
<td>MV, K, AL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DU: display = information</td>
<td>MV, K, AM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DV: display = playing</td>
<td>MV, KD</td>
</tr>
<tr>
<td>Activity</td>
<td>Trigger</td>
<td>Process</td>
<td>Change</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------------</td>
<td>----------------</td>
<td>---------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>MP3Player Next Item</td>
<td></td>
<td></td>
<td></td>
<td>GE: item = ST year</td>
<td>MV, KG, GJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GF: item = ST mon</td>
<td>MV, KG, GE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GG: item = ST day</td>
<td>MV, KG, GF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GH: item = ST hour</td>
<td>MV, KG, GG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GI: item = ST min</td>
<td>MV, KG, GH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GJ: item = ST sec</td>
<td>MV, KG, GJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Push Display</td>
<td></td>
<td>GK: display = Hold</td>
<td>MS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player</td>
<td>setHoldTimer()</td>
<td></td>
<td>MS</td>
</tr>
<tr>
<td>Press Previous Button</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Light On</td>
<td></td>
<td>GL: light = on</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player</td>
<td>setPowerOff()</td>
<td></td>
<td>MV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Stop</td>
<td>stop()</td>
<td></td>
<td>MV, I, BT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Rewind</td>
<td>rewind()</td>
<td></td>
<td>MV, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Play Or Pause</td>
<td>GM: mode = playing</td>
<td></td>
<td>MV, I, BT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Player Back Meter Value</td>
<td>backMeterValue()</td>
<td></td>
<td>MV, DZ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>backMeterValue()</td>
<td>MV, EA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>backMeterValue()</td>
<td>MV, EB</td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
<td>change</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JD: focus = high</td>
<td>MV, BP, KD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JE: repeat mode = one</td>
<td>MV, JG, BF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JF: repeat mode = all</td>
<td>MV, JE, BF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JG: repeat mode = off</td>
<td>MV, JF, BF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JH: shuffle mode = on</td>
<td>MV, JI, BG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JJ: shuffle mode = off</td>
<td>MV, JH, BG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JJ: selecting = true</td>
<td>MV, JW, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JK: selecting = true</td>
<td>MV, JX, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JL: selecting = true</td>
<td>MV, JY, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JM: selecting = true</td>
<td>MV, JZ, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JN: selecting = true</td>
<td>MV, KE, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JO: selecting = true</td>
<td>MV, KF, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JP: selecting = true</td>
<td>MV, KG, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JQ: selecting = true</td>
<td>MV, KB, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JR: selecting = true</td>
<td>MV, LB, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JS: selecting = true</td>
<td>MV, KC, BK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>resetTrackList()</td>
<td>MW, AZ, DQ</td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
<td>change</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
<td>-------------------</td>
<td>------------</td>
</tr>
<tr>
<td>KE: display = Tru Base</td>
<td>MV, MG, KC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KF: display = SRS 3D</td>
<td>MV, MH, KC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KG: display = Set Time</td>
<td>MV, DS, AT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KH: display = Stop Watch</td>
<td>MV, MB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3Player Push Display</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KL: display = Artist</td>
<td>MV, DQ, CU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KJ: display = Album</td>
<td>MV, UJ, LV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KK: display = Album</td>
<td>MV, KI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KL: display = Songs</td>
<td>MV, DQ, CW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KM: display = Songs</td>
<td>MV, KJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KN: display = Songs</td>
<td>MV, KQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KO: display = Songs</td>
<td>MV, KP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KP: display = Genre</td>
<td>MV, DQ, CX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQ: display = Year</td>
<td>MV, DQ, CY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set MP3Player Display</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KR: display = play music</td>
<td>MV, K, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS: display = modes</td>
<td>MV, K, AJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KT: display = settings</td>
<td>MV, K, AK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KU: display = extras</td>
<td>MV, K, AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
<td>change</td>
<td>Consequence</td>
<td>Conditions</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>MC: item = play music</td>
<td></td>
<td></td>
<td></td>
<td>MV, F, DV</td>
<td></td>
</tr>
<tr>
<td>MD: item = Repeat</td>
<td></td>
<td></td>
<td></td>
<td>MV, AP, F</td>
<td></td>
</tr>
<tr>
<td>ME: item = Equalizer</td>
<td></td>
<td></td>
<td></td>
<td>MV, AQ, F</td>
<td></td>
</tr>
<tr>
<td>MF: item = Equalizer</td>
<td></td>
<td></td>
<td></td>
<td>MV, KY</td>
<td></td>
</tr>
<tr>
<td>MG: item = Tru Base</td>
<td></td>
<td></td>
<td></td>
<td>MV, F, KY</td>
<td></td>
</tr>
<tr>
<td>MA: item = MRS 3U</td>
<td></td>
<td></td>
<td></td>
<td>MV, F, KY</td>
<td></td>
</tr>
<tr>
<td>MI: item = WOW</td>
<td></td>
<td></td>
<td></td>
<td>MV, KZ</td>
<td></td>
</tr>
<tr>
<td>MJ: item = low</td>
<td></td>
<td></td>
<td></td>
<td>MV, KD</td>
<td></td>
</tr>
<tr>
<td>MK: item = ST year</td>
<td></td>
<td></td>
<td></td>
<td>MV, KG</td>
<td></td>
</tr>
<tr>
<td>ML: item = Restore Default</td>
<td></td>
<td></td>
<td></td>
<td>MV, ML</td>
<td></td>
</tr>
<tr>
<td>MM: item = Set Time</td>
<td></td>
<td></td>
<td></td>
<td>MV, GE</td>
<td></td>
</tr>
<tr>
<td>MN: item = void</td>
<td></td>
<td></td>
<td></td>
<td>MV, I</td>
<td></td>
</tr>
<tr>
<td>Set Play/Display Icon</td>
<td></td>
<td></td>
<td></td>
<td>MV: icon = play icon</td>
<td>MV, GS, MQ</td>
</tr>
<tr>
<td>MP3Player Push Display</td>
<td></td>
<td></td>
<td></td>
<td>MP: display = Hold</td>
<td>MS</td>
</tr>
<tr>
<td>MP3Player</td>
<td></td>
<td></td>
<td></td>
<td>setHoldTimer()</td>
<td>MS</td>
</tr>
<tr>
<td><em>Reset Info Display</em></td>
<td></td>
<td></td>
<td></td>
<td>MQ: information = off</td>
<td></td>
</tr>
<tr>
<td>Reset Play/Display</td>
<td></td>
<td></td>
<td></td>
<td>MR: light = on</td>
<td>N</td>
</tr>
<tr>
<td>Slide Hold Switch Down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### CBI Manager PEAnet

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3Control</td>
<td>shell</td>
<td>introduction / setup0 / setup1 /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>controls0 / controls1 / controls2 /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>test / play / menu / menu1 / info1 /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>info2 / instruct / seltest1 /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>seltest2 / swatch / sleep / geq /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td><em>NEXT</em></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td><em>TO_LEFT</em></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td><em>TO_RIGHT</em></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td><em>TO_FRONT</em></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control displayShell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Control</td>
</tr>
</tbody>
</table>
## CBI Controls Test PEAnet

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3Tester</td>
<td>control</td>
<td>start / press menu / hold menu / play / next / repeat / previous / double previous / select / volume / hold up / hold down / volume up / volume down</td>
</tr>
<tr>
<td>Hold Switch</td>
<td>position</td>
<td>start / release / hold</td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
<td>------------------</td>
</tr>
<tr>
<td><strong>_START</strong></td>
<td></td>
<td>MP3Tester Set Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hold Switch Set Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td><strong>Slide Hold Switch Down</strong></td>
<td></td>
<td>Set Hold Switch On</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td><strong>Slide Hold Switch Up</strong></td>
<td></td>
<td>Set Hold Switch Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td><strong>Press Menu Button</strong></td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td>Activity</td>
<td>Trigger</td>
<td>Process</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Press Play Button</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Press Repeat Button</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Press Next Button</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Press Previous Button</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>trigger</td>
<td>Process</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>_Break</td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td>MP3Tester</td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td>MP3Tester</td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td>Double Click Previous</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td>Press Select Button</td>
<td></td>
<td>MP3Tester Activate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Break</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3Tester</td>
</tr>
</tbody>
</table>
Appendix E. MP3 Player UML Diagrams
MP3 Player Simulation UML Diagrams
Shutdown Timer

- **Shutdown Timer Off**
  - menu pressed
  - menu released
  - after(500u) / startShutdown()

- **Shutdown Started**
  - after(250u) [counter > 0] / countDown()

- **Shutdown Timer Countdown**
  - [counter == 0] / shutdown()
  - menu released / stopShutdown()
CBI Manager UML Diagrams
CBI Controls Test UML Diagrams
CURRICULUM VITAE

Delroy A. Brinkerhoff
(April 2010)

EDUCATION
2010 Ph.D. in Instructional Technology and Learning Sciences, Utah State University
1999 Java Platform 2 Programmer Certification, Sun Microsystems
1996 M.S. in Computer Science, Utah State University
1984 B.S. in Computer Science and Computational Mathematics, Physics minor, Brigham Young University

TEACHING EXPERIENCE
2003 - Present Associate Professor, Weber State University, Dept. of Computer Science
1998 - 2003 Assistant Professor, Weber State University, Dept. of Computer Science
1996 - 1998 Instructor, Weber State University, Dept. of Computer Science
1996 Part Time Contract Instructor, Productivity Point International
1996 Internal Instructor at Unisys Corporation
1989 Adjunct Instructor, Salt Lake Community College
1995 - 1996 Adjunct Instructor, Salt Lake Community College
1986 Internal Instructor, National Applied Computer Technologies
1985 Adjunct Instructor, Utah Valley Community College (now UVU)
1978 - 1984 Teaching Assistant, Brigham Young University, Dept. of Physics and Dept. of Computer Science

PUBLICATIONS AND PRESENTATIONS
2007 Instructional Simulations: Extending Experiential Learning to Abstract Worlds, Weber State University Faculty Forum
2002 Revitalizing ISD: A Cross-Discipline View of Model Building, AECT Annual Conference
1996 Second author of A Stable Distributed Tuple Space presented at the International Conference on System Sciences (w/Scott R. Cannon)
1995 First author and presenter of A Stand-alone Remote Thread Library for Transputer Systems at the International Conference on Parallel and Distributed Processing Techniques and Applications (w/Scott R. Cannon)
1995 Workshop at Software Development’95 West (w/Scott J. Cima)
CLASSES TAUGHT
  Connecting with Computer Science
  Introduction to UNIX
  Introduction to C
  Fundaments of Programming (Problem Solving with Java)
  Object-Oriented Programming with C++
  Structured Computing in C
  Object-Oriented Analysis and Design (both OMT- and UML-based)
  Operating Systems
  Internet and Multimedia (advanced Java)
  Independent Projects
  Cooperative Experience
  Introduction to Computers
  UNIX System Administration
  UNIX System Programming
  Using UNIX
  Introduction to C++
  Advanced C++
  C++ for C Programmers
  Introduction to Java
  Data Structures and Algorithms
  Computer Literacy
  Intermediate Algebra
  Engineering Mathematics (Numerical Analysis using FORTRAN-77)
  Programming in Pascal
  Physics of Light and Photography (T/A)
  Computer Architecture (T/A)
  Microcomputer Construction (T/A)

ENGINEERING EXPERIENCE
  1986 - 1996  Senior Software Engineer, Unisys Unix Group
  1986          Programmer, UNIX Systems Administrator, Internal Instructor,
                Nation Applied Computer Technologies
  1984 - 1986  National Software Support Engineer, WICAT Systems