VISUAL SUPPORT FOR THE ISLE SIMULATION ENVIRONMENT

BY

TINA ELIASSI-RAD

B.S., University of Wisconsin, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ACKNOWLEDGMENTS

I would like to thank my thesis adviser, Professor M. T. Harandi, for his direction, encouragement, and comments on this thesis.

Thanks also to my husband, Branden Fitelson, for his constant support and guidance during the writing process of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

All men by nature desire to know. An indication of this is the delight we take in our senses; for even apart from their usefulness they are loved for themselves; and above all others the sense of sight. For not only with a view to action, but even when we are not going to do anything, we prefer seeing to everything else. The reason is that this, most of all the senses, makes us know and brings many differences between things.

Aristotle
Metaphysics

1.1 Visual Support for Software Systems

Traditionally, the terms ‘visual programming’ and ‘program visualization’ have often been used interchangeably [10]. Recently, authors have been using these terms to distinguish different aspects of software engineering. For example, Myers [10] defines a visual programming system as “any system that allows the user to specify a program in two-(or more)-dimensional fashion”. He goes on to explain that traditional textual programs are not categorized as two dimensional because the compiler or interpreter treats them as a “long, one-dimensional stream”.

On the other hand, Myers says that program visualization is “an entirely different concept”. Specifically, Myers states that, in program visualization “…the program is specified in the conventional, textual manner, and the graphics is used to illustrate some aspect of the program…” Therefore, according to Myers, program visualization only involves *illustration* of some aspects of an existing program (which may, itself, have been specified in a conventional textual language); whereas, visual programming involves *development* of a program using two or more dimensional graphical tools.

Other authors have made a similar distinction. For instance, Chang [3] says that the defining characteristic of visual programming is its “use of visual expressions in the process of programming” (as opposed to conventional, textual expressions). He clarifies visualization as the “use of visual representations to illustrate data, program, the structure of a complex system, or the dynamic behavior of a complex system.”

Following Myers and Chang, we use the term *program visualization* to refer to any process of program manipulation and/or analysis that involves visualization. We reserve the term *visual programming* to refer to only those aspects of program development in which the user makes *changes* to a program using graphical tools. Moreover, we define the term *visual support system* as a computer system that supports both visual programming and program visualization. The present research involves a visual support system that is primarily concerned with program visualization.

Before we give a classification of visual support systems and techniques, we would like to answer the following two foundational questions about visual support:

- What is the *purpose* of a visual support system (*i.e.*, what is it designed to do)?
- What *advantages* do visual support systems have over traditional (non–visual) systems (*i.e.*, why were visual support systems invented in the first place)?

These two questions go hand-in-hand. It would be quite difficult to give an adequate answer to either of these questions without giving at least a partial answer to the other. Only after we have adequately answered both of these questions will we have a satisfactory understanding of what visual support systems *are* and why users[1] might prefer using them over traditional non–visual systems.

[1] We will be using the term ‘user’ in a very general way. The term ‘user’ will denote many different kinds of software users. For example, a ‘user’ might be a novice programmer, an instructor, a professional software engineer, *etc*. We want this term to be as generic as possible and still stay within the scope of software users who could potentially benefit from using visual support systems.

Below, we give a brief survey of how some experts in the field have tried to answer these important questions about visual support systems. Then, we will try to give some answers of our own. The following expert opinions appear in (what we take to be) increasing order of breadth and informativeness with respect to both of our central questions.

Brown *et. al.* [2] suggest that 'the' *purpose* of visual support systems is "to help programmers form clear and correct mental images of a program's structure and function." We doubt that *all* visual support systems have a *unique* purpose. In general, the purpose of a particular visual support will depend on the context in which it is used. In any case, the above statement of Brown *et. al.* concerning the purpose of visual support is too vague to provide a complete and satisfying answer to either of our two questions. What we need is a more informative and careful answer, one which touches on both questions in a more general way.

Grafton and Ichikawa [5] shed more light on *both* of our questions. They argue that visual support systems are useful tools because they provide clear and concise representations of both static and dynamic aspects of software using two or three dimensional graphics with coloring and highlighting. Consequently, users of visual support systems can cut through the complexity of software. We are starting to understand not only what visual support *is*; but, why users might *prefer* using visual support systems to traditional non-visual systems. Specifically, Grafton and Ichikawa give us reason to believe that visual support systems can (*via* graphical means) convey information about programs to users *in a more effective way* than traditional non-visual systems.

Raeder [12] gives a slightly more informative statement of this kind. He explains that traditional textual representations of complex programs cannot be readily understood by users. Such representations of programs do not make it clear to the user (at any given time) what the state of the program is. By maximizing the amount of information that can be simultaneously conveyed to the user on their computer screen, visual representations can convey more information about a program's structure than traditional textual representations.

A similar kind of sentiment has been expressed more elaborately by Myers [10]. He states that the use of visual tools makes the programming task easier than it would be using only traditional textual tools. Myers explains that graphical representations of programs (generally) tend to de-emphasize syntactical issues since they are at a higher level of abstraction than textual representations. Moreover, he explains that graphical representations tend to convey more information about the state of a program (*e.g.*, its current variables and data structures) than is feasible with just textual displays. As a result, Myers points out that (among other things) visual support systems can be especially useful for debugging programs.

There seem to be several common threads running through the above expert testimonials. The consensus seems to be that visual support systems generally share at least the following properties:

- Visual support systems graphically aide the user in understanding various aspects of programs. Depending on the context, visual support systems may include graphical representations of control flows, data flows, data structures, *etc*. Moreover, as visual support evolves, the set of program aspects captured and intuitively represented by visual support systems will inevitably grow.

- Because visual support systems use *graphical* representations, they are able to facilitate the user's understanding of (at least some) program aspects *in a more effective way* than traditional non-visual systems. As they say: "A picture is (at least sometimes) worth 1000 words." This is especially true in the areas of object-oriented programming (OOP) and parallel programming [12].[2]

• Because visual support systems often confer a better understanding of programs than their non-visual counterparts, they tend to provide greater support to the user *in all activities* (*i.e.*, the analysis, design, construction, test, and maintenance activities) of the software lifetime. For example, Brown *et. al.* [2] have identified a set of categories of visual support systems which span all phases of the software life cycle.

Stevlosky, Ackermann, and Conti [15] claim that "the programming process is a highly demanding intellectual activity *requiring* visual support at all stages of development". While this claim may seem a bit too strong (since it implies that some combination of program and visual programming is *necessary* for successful software engineering), our attempt to answer the two key questions of this section illustrates that there is *some* truth in it. Surely, properly understanding (thus, developing and analyzing) software systems is generally a daunting task. What makes visual support systems so attractive is their ability to efficiently and intuitively (*via* graphical means) convey necessary information to users; thereby, reducing the apparent complexity of programs.

Visual support systems have both visual programming and program visualization aspects. By describing program visualization and visual programming, we can gain a

[2] We can give a more precise and formal statement of the claims made in this bullet. Let $I(S_v)$ be the amount of information per unit time conveyed to the average user by a visual representation of some aspect of a program structure S; and, let $I(S_t)$ be the amount of information conveyed per unit time to the average user by a traditional (non-visual) representation. We are making two distinct claims here. The first claim is that, for some program structures S: $I(S_v) \gg I(S_t)$. This is a formal way of saying "A picture is (sometimes) worth 1000 words". The second claim we are making is that, for some pairs of program structures S, S*: $[I(S_v) - I(S_t)] \gg [I(S^*_v) - I(S^*_t)]$. In other words, some programs are *better suited* to visual representation than others. For example, an object-oriented program S (perhaps, written in C++) may be better suited to visual representation than a functionally equivalent non object oriented program S* (say, written in FORTRAN).

better understanding of the general nature of visual support systems. For the remainder of this section, we will give classification schemes for program visualization and visual programming systems, respectively. Since the present research is concerned primarily with program visualization, our discussion of program visualizations will be more complete than our subsequent discussion of visual programming.

Recently, a systematic approach for classifying and understanding program visualization systems has been introduced by Roman and Cox [13]. In their model, program visualization is understood as a "*mapping* from programs to graphical representations".[3] As the authors explain, the defining characteristics of a program visualization system are its graphical primitives, the semantics of the term 'program', the possible mappings between programs and graphical representations, and the ways in which these mappings are implemented.

In the Roman and Cox model, all program visualization systems involve interaction between the following three participants:[4]

- A *programmer*, who generates the program text.
- An *animator*, who facilitates the mapping from programs to visualizations.
- A *viewer*, who observes the graphical representations of the program text.

Roman and Cox [13] classify program visualization systems using the following five criteria:

[3] Although Roman and Cox have mainly applied their model of program visualization systems to classifying systems which visualize the *execution* of programs, we think that *all* program visualization systems fit somewhere in their taxonomy. Moreover, their taxonomy can be extended quite naturally to include visual programming systems. We discuss such an extension later in this section.

[4] Of course, these 'participants' need not be human beings. They may (and often will) be parts of computer systems themselves.

• Scope: “What aspect of the program is visualized?”

A *program* may be distinguished by its code, data and control states, and/or execution behavior. So, the *scope* of a program visualization system is determined by which aspects of a program it can visually capture. Depending on the program visualization system’s capabilities, it may visualize one or more aspects of a program.

• Abstraction: “At what level of abstraction is the information conveyed by the visualization?”

For instance, visualizations that consist of highlighted portions of the code of some object-oriented program *P* will generally be at a very *low* level of abstraction. Whereas, graphical diagrams of the relational structure of *P* will be at a rather *high* level of abstraction. In such diagrams, some information about the *low-level* structure of the program code will be hidden from view. Abstraction is necessary for managing the complexity of displayed material. After all, computer displays are only *so* big.

• Specification method: “What mechanisms does the animator use to construct the visualization?”

A system’s power and flexibility is determined by its specification method. Specification methods range from “hard-wired” mappings (which the animator has no control over) to arbitrary animator-defined mappings. Some program visualization systems emphasize mapping on program states, while others emphasize mapping on events. Moreover, depending on their specification methods, some systems may require modifications to the original code in order to generate visualizations.

• Interface: “What tools does the program visualization system provide for the visualization of programs?”

Generally, these tools are the graphical primitives and combinations of these primitives employed by the animator. The animator uses these tools to create the graphical representations. The viewer uses them to interact with the visualizations in order to investigate and understand the visualization. As Roman and Cox [13] explain: “The interface is what the viewer sees and how the viewer controls what the system shows.”

• Presentation: “What is the *meaning* of the visualization?”

Presentation is a cognitive (and not simply perceptual) aspect of program visualization. So many boxes, lines, circles, arrows, icons, *etc.* are meaningless to the viewer, unless they are presented in such a way that their semantics are made clear to the viewer.

The taxonomy of program visualization systems developed by Roman and Cox [13] is broad enough to classify all program visualization systems; and, it is informative enough to allow for meaningful and interesting comparisons between various program visualization systems. By adopting their framework, we are able to locate our present research (*qua* program visualization) on the ‘world map’ of program visualization systems, and obtain a deeper understanding of what we have accomplished.

A complete characterization of visual support systems requires an understanding of *both* visual programming *and* program visualization. The program visualization taxonomy of Roman and Cox [13] does not explicitly provide us with the requisite understanding of visual programming. However, their approach of conceptualizing program visualization as a mapping from programs to graphical representations can be ‘inverted’ to yield a parallel classification scheme for visual programming systems. In other words, we can think of visual programming as a mapping from graphical representations to programs. Along these lines, Grafton and Ichikawa [5] have suggested that an effective visual programming system must define a precise mapping of its graphical symbols into a syntactically correct formal structure; and, eventually, into efficient program code. Figure 1.1 provides a graphical representation of the relationship between program visualization and visual programming, conceptualized as inverse mappings between programs and graphical representations.

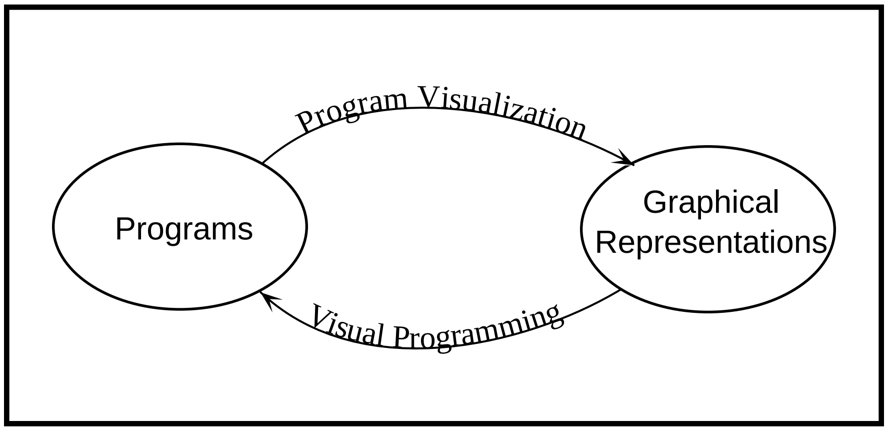


Figure 1.1: Program Visualization and Visual Programming as Inverse Mappings

1.2 Visual Support for a Simulation Environment

The core of a simulation environment is its underlying simulation language. Since a simulation environment will inevitably make use of components available in its simulation language, the nature of the underlying simulation language strongly influences the structure and form of the simulation environment [16]. Consequently, the choice of simulation language is of paramount importance.

General purpose programming languages (*e.g.*, PASCAL) are deficient for the purposes of simulation because they do not adequately support primitives necessary for representing many aspects of a simulation model including (1) its time dimension, (2) concurrency, (3) the creation of objects in the model, or (4) the handling of relationships between objects [8]. On the other hand, object-oriented languages *can* readily provide most of these primitives [8, 16]. In fact, it is now widely agreed upon that object-oriented methodologies are most conducive to simulation programming [8, 9, 16, 17]. As a result, most of the recent generations of simulation programming languages that have evolved out of traditional, general purpose languages are object-oriented in nature [8, 17].

Object models are typically specified (whether on paper, or on a computer screen) in a two-dimensional field using graphical primitives [14]. Hence, it makes sense to specify object-oriented programs in an analogous way. Moreover, analyzing the structure of an object-oriented program is most easily done graphically, simply by visually examining various aspects of the program's two-dimensional object model. It stands to reason, therefore, that visual support techniques (involving both visual programming and program visualization) are invaluable tools for the development and manipulation of object-oriented programs.

The purpose of this research is to provide visual support for the Integrated Simulation Language Environment (ISLE). ISLE is a project of the Advanced Simulation and Software Engineering Technology (ASSET) team of the U.S. Army Construction Engineering Research Laboratory (USACERL). ISLE is a simulation-based software engineering environment which integrates process-based, discrete-event simulation with object-oriented modeling methodologies, declarative programming, and persistent data storage. The underlying simulation language of ISLE is IMPORT which gives ISLE its requisite functionality. The CASE tool described in this thesis provides visual support for the development, manipulation, and analysis of IMPORT programs. In chapter 2, we place the present work in its proper context by describing ISLE and IMPORT in more detail. For the remainder of the thesis, we discuss the design, interface, and various distinctive features of our visual ISLE CASE tool.

CHAPTER 2

ISLE, IMPORT AND THE GRAPHICAL EDITOR[5]

2.1 The Integrated Simulation Language Environment (ISLE

The objective of ISLE is to provide a software engineering environment for the identification and development of technologies with a need for intelligence, collaboration, and simulation [18]. The ISLE environment includes: the IMPORT programming language, a persistent object-oriented database (called the *object repository*), and a set of computer aided software engineering (CASE) support tools.

The IMPORT programming language is the underlying simulation language of ISLE. IMPORT combines the main features of object-oriented modeling, process-based simulation, declarative programming, and persistent object storage into a single programming language.

The persistent object repository stores IMPORT objects in an object-oriented database. A set of generic access primitives have been defined to make ISLE independent of any particular database. ISLE has a collection of classes that model intermediate forms of IMPORT compilation structures. These classes set-up the basic data structures necessary to support the ISLE software engineering environment. They allow IMPORT abstract syntax tree storage in the object-oriented repository, and they

[5] In this chapter, we describe the relevant aspects of the ISLE system, borrowing figures and definitions from the on-line document describing the system [18].

provide a common access mechanism for the entire ISLE tool set. ISLE is designed to be a multi-user environment. Accordingly, a version control system has been implemented atop the object repository. The version control system provides the user with the version of the object repository he/she demands.

Each of ISLE's CASE tools interact with the object repository. The purpose of the ISLE CASE tools is to help ease the labor-intensive process of developing large scale software. The ISLE CASE tools are divided into two main groups: the *design support tools* and the *analysis support tools*. The design support tools aid the user in the development and manipulation of object specifications and other program artifacts. The design support tools include a graphical editor, a text editor, a parser, an unparser, a system-level framework selector, an automatic specification refiner, an object evolution manager, and a collection of dynamic collaborative agents [18].

The *graphical editor* (which is the main subject of this thesis) is a program visual support tool that helps the user develop, manipulate, and analyze IMPORT programs graphically. The *text editor* aides the user in the development and manipulation of IMPORT programs, textually. The *parser* takes an IMPORT program as input and places intermediate forms of its objects into the object repository. The *unparser* retrieves intermediate forms of IMPORT objects from the repository and translates them back into IMPORT source code for use by the editors. The *system-level framework selector* allows the user to select system-level frameworks directly from the object repository. The selected frameworks are composed of IMPORT object models. These frameworks are given to the graphical editor to manipulate. The *automatic specification refiner* assists the user in automated IMPORT specification and programming. The *object evolution manager* assists the user in making global modifications to objects in an IMPORT program. The collection of *dynamic collaborative agents* facilitates collaboration between IMPORT programs.

The analysis support tools are involved in the construction and maintenance of executable IMPORT programs. The analysis support tools include a type checker, a code generator, a debugger, an optimizer, and a compilation manager [18].

The *type checker* verifies that the type of each IMPORT construct is appropriate for its context. The *code generator* produces C++ code from an IMPORT program. The *debugger* aids in the identification and correction of IMPORT programs. The *optimizer* improves the C++ code produced by the code generator, so that it compiles into more efficient machine code. The *compilation manager* assembles an executable program from the IMPORT source program.

2.2 The IMPORT[6] Programming Language

The IMPORT programming language is the nucleus of ISLE. It has been designed to integrate object-oriented methodology, process-based discrete event simulation, logic programming, and persistent object storage into a single programming language. The significance of each of these four features, and the manner in which IMPORT provides them is described below:

- Object-Oriented Modeling Methodology

  One of the original motivations for the object-oriented paradigm was to provide a faithful (and intuitive) mapping between real-world objects and their implementation in computer models [18]. As such, it is not surprising that object-oriented methodologies are particularly well suited to simulation and modeling [14, 16]. Some have even maintained that *all* programs are simulations (this extreme position is known as *The Scandinavian View* [6]).

[6] The word IMPORT stands for Integrated Modular Persistent Object Representation Translator.

IMPORT takes advantage of this natural marriage between object-oriented methodologies and simulation. IMPORT follows the Rumbaugh Object Modeling Technique (OMT) [14]. Data abstraction in IMPORT is implemented by declaring an object to be of a certain class. An object class is a group of objects which share common attributes and behaviors. These object classes form modules which control scoping and visibility. IMPORT supports multiple inheritance and polymorphism. It also facilitates re-use by factoring common attributes and behaviors into abstract super-classes. Reuse, in turn, reduces the redundancy level in IMPORT programs.

- Process-Based Simulation

In the process-based approach to simulation, the dynamical behavior of a system during some period of time (or process) is described by a single process routine. As a result, the total history (including the simulated passage of time in the system) of each object as it moves through its corresponding (sub)process is captured by a corresponding (sub)process routine.

In an IMPORT simulation program, process-based simulation is readily accomplished by encoding the (sub)process routine for each object in one or more of its *methods* [18]. IMPORT's language constructs provide an asynchronous tasking mechanism, threads, and synchronization. The asynchronous tasking mechanism for concurrent method invocation provides process-based simulation for IMPORT. Execution of threads allows asynchronous processes of an object to proceed independent of other processes until synchronization occurs. Synchronization is based on either the global simulation time or the states of other processes.

- Declarative Programming

  One of the goals of ISLE is to simulate complex thought processes. Declarative programming languages (*e.g.*, PROLOG) have been effectively used to model reasoning, decision making and other complex behaviors. Such capabilities have been incorporated into IMPORT *via* the Declarative Object Manipulation Environment (DOME). Each object in an IMPORT program may have an associated knowledge-base. DOME facilitates the correspondence between each object and its knowledge-base. DOME (like PROLOG) uses a unification algorithm and horn-clause logic to prove theorems about objects embedded in simulations. The knowledge (and deductive engine) encoded *via* DOME in the objects of an IMPORT simulation program allows IMPORT to simulate complex, non-algorithmic behavior quickly and effectively [18].

- Persistent Data Storage

  Simulations produce a lot of data. Hence, scientists who use simulations must collect, organize, and analyze large amounts of simulation data. Ideally, a simulation environment would have the capability to *automatically* store simulation data (preferably in an organized fashion). This would greatly ease the collection, organization and analysis of simulation data.

  IMPORT facilitates persistency by allowing the user to label objects as *persistent*. Persistent objects are automatically stored (along with their corresponding simulation data) into an object-oriented database: *the object repository*. Once an object is placed into the object repository, its simulation data can be reported, (re)organized, and/or analyzed by scientists.

The historical/functional relationships between IMPORT and other basic programming languages is represented graphically in figure 2.1 below.[7]

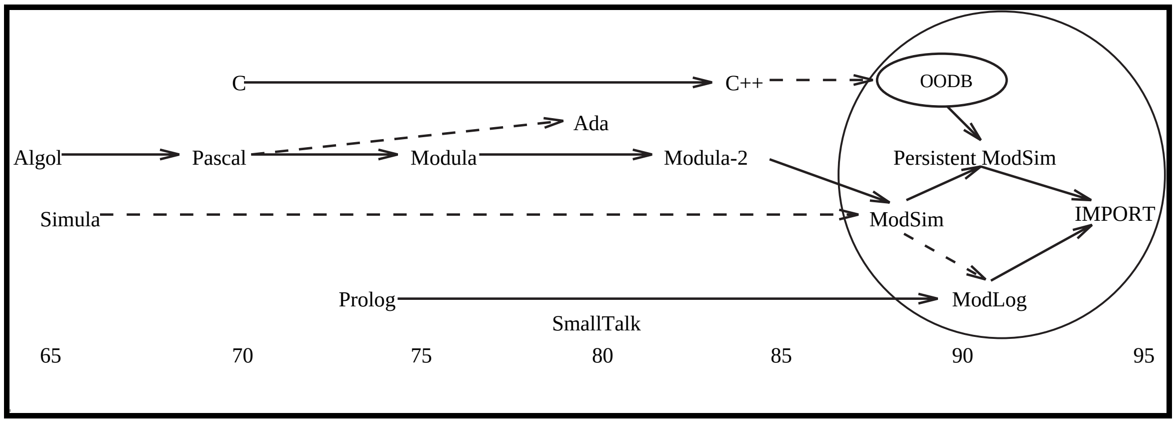


Figure 2.1: The Relationships between IMPORT and Other Languages

IMPORT has derived the concept of persistent object storage from C, its imperative programming capabilities from Algol, its discrete-event simulation features from Simula, and its declarative programming aspects from PROLOG.

2.3 The ISLE Designer Tool Set

The ISLE designer tool set is designed to provide the prospective ISLE user with the ability to access, manipulate, modify, and/or extend the contents of the ISLE object repository.

The ISLE designer tool set consists of several components. Figure 2.2 illustrates the architecture of the ISLE designer tool set and the relationship between its components.[8]

---

[7] In Figure 2.1, a solid arrow '$\longrightarrow$'means 'is a direct descendant of'; whereas, a dotted arrow '$\dashrightarrow$' means 'is an indirect descendant of'. IMPORT is a direct descendent of ModSim. See [7] for an informative survey of the languages depicted.

[8] In Figure 2.2, a solid arrow '$\longrightarrow$' means 'transmits data to'.

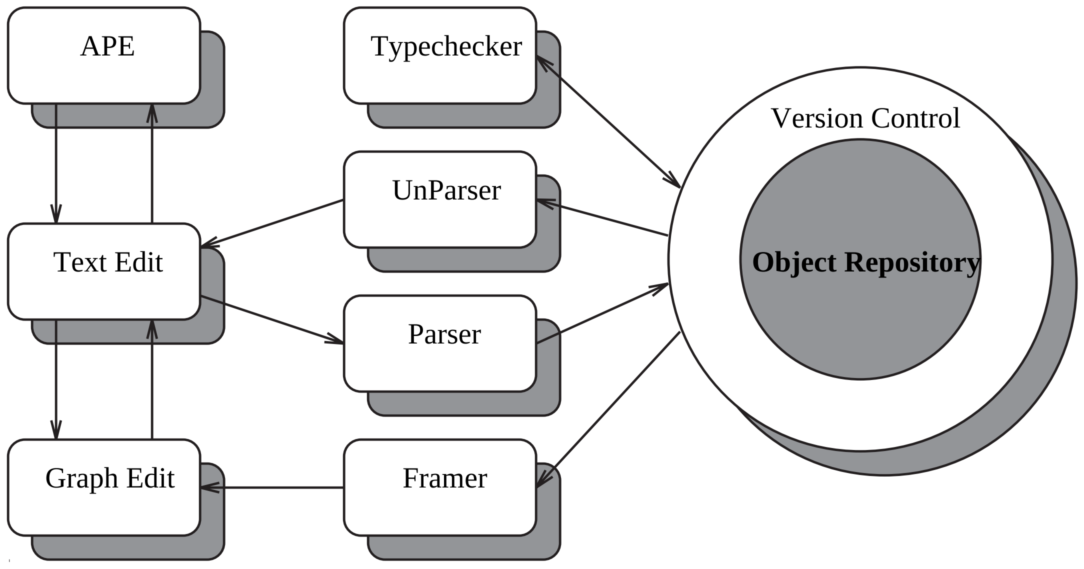


Figure 2.2: The Architecture of the ISLE Design Tool Set

The graphical editor—one of the tools in the ISLE designer tool set—is the topic of this thesis. The graphical editor has a very close relationship with two other components of the ISLE designer tool set: the text editor and the framer. The text editor aides the user in developing and manipulating IMPORT programs textually. Moreover, the text editor receives IMPORT programs from the object repository through the IMPORT unparser, and deposits IMPORT programs to the object repository *via* the IMPORT parser. The graphical editor is able to send its results to the object repository only through the text editor. The graphical editor receives IMPORT objects from both the text editor and the framer. The framer allows the user to directly select system-level frameworks from the object repository. The selected frameworks are composed of IMPORT object models. These frameworks are then given to the graphical editor to manipulate.

IMPORT is a strongly typed language (*i.e.*, all operators specify the types of their arguments). Because of this, ISLE comes equipped with a type checker which insures that the type of each IMPORT construct is appropriate for its context.

ISLE supports automated programming through the automated programming environment (APE). APE has a knowledge base and a program synthesizer which together assist the user is automated IMPORT programming.

2.4 The Graphical Editor: An Overview

The main objective behind this work has been to make the process of developing, manipulating, and analyzing software through the IMPORT programming language more user friendly by utilizing program visualization (and visual programming) techniques. To this end, we have designed a graphical editor with the following features:

• The capability to capture graphical representations of IMPORT objects and the relations between them.

Specifically, the graphical editor is based on the Rumbaugh [14] Object Modeling Technique (OMT), and it supports the entity relation (ER) model (*viz*. the object model). An ER model of an IMPORT program describes the static structure of the objects in an IMPORT program and the relationships between them.

• The capability to visually generate, explore, analyze, and reconstruct the object diagrams of IMPORT programs.

An object model of an IMPORT program consists of IMPORT program object diagrams. An object diagram of an IMPORT program is a graph whose vertices are IMPORT objects and whose edges represent relationships among IMPORT objects. Because the graphical editor allows the user to *both* (passively) *view* IMPORT program object models, *and* (actively) *specify* object diagrams of IMPORT programs, it serves jointly as an IMPORT program visualization tool and as an IMPORT visual programming tool.

• A user friendly GUI

The interface of the graphical editor is designed to be as user friendly as possible. To this end, it includes various on-screen and pop-up menus, a class browser, a relation matrix, and many other facilities including reverting, zooming in and out of diagrams, coloring, font changing, *etc*.

• The capability to allow the user to dynamically create and manipulate multiple graphical views.

Dynamical views correspond to underlying structures that represent IMPORT objects. With these views, the user is able to run queries on the object repository (as a whole) and see the results of their queries represented graphically, in 'real time'. This allows the user to observe the structure of an IMPORT program from multiple perspectives, simultaneously.

We can characterize our visual support tool (*i.e.*, the program visualization and visual programming aspects of our graphical editor) using the taxonomy of Roman and Cox [13] (described in section 1.1.2 above) as follows:

• The *scope* of our visual support tool is *code*. The graphical editor transforms IMPORT code into object diagrams (this is its program visualization aspect) and *vice versa* (this is its visual programming aspect).

• The *level of abstraction* of our visual support tool ranges from: *direct* to *structural representation*. In other words, the graphical editor allows the user to either look directly at the *low-level* textual code of an IMPORT program, or to focus on *higher-level*, *structural* aspects of the object model of an IMPORT program (*via* graphical representations).

• The *specification method* used by our visual support tool is: *predefinition*. The intermediate language of the graphical editor plays the role of the animator in the Roman and Cox model. It defines a mapping from IMPORT code to elements of an object diagram and *vice versa*.

• The *graphical vocabulary* of our visual support tool includes *simple objects*, *composite objects,* and *worlds*. The simple objects are the graphical primitives that the graphical editor uses as basic building blocks for ER diagrams of IMPORT programs (*e.g.*, objects, relations, *etc*.). Composite objects are incomplete (in the sense that they do not constitute a program's *entire* object diagram) collections of the graphical primitives. Worlds are *complete* object diagrams for IMPORT programs. The user *interacts* with the graphical editor's interface both *through controls* (*e.g.*, generating views of the object diagram) and *through the image* (*e.g.*, re-positioning a graphical entity in an object diagram, using the mouse).

• Our visual support tool makes use of *interpretation of graphics* (*e.g.*, on-screen annotation for various graphical entities), and *orchestration* (*e.g.*, allowing the user to generate *multiple views* into the object diagram of an IMPORT program) to help convey visual information to the viewer.

In short, the desired capabilities of the graphical editor are to allow the user to visually specify ER models of IMPORT applications (visual programming aspect), manipulate ER models of existing IMPORT applications, and dynamically create graphical views of IMPORT entities and structures through database queries (program visualization aspects). In the next few chapters we will discuss each of these capabilities in detail.

CHAPTER 3

THE GRAPHICAL EDITOR: ARCHITECTURE AND USER-INTERFACE

3.1 The Architecture of the Graphical Editor

The graphical editor has a structured design. The architectural design of the graphical editor has two main branches: the *data base manager* and the *interface manager*.

The data base manager maintains all of the IMPORT objects, their intermediate representations, and their corresponding graphical entities. The data base manager receives IMPORT objects from either the text editor or the framer and deposits them into the text editor (see Figure 2.2). An *intermediate language (IML)* provides the intermediate representations of the IMPORT objects. The IML was created to handle the translations between IMPORT programs and ER graphs (and *vice versa*) easier. It facilitates storage for the object-oriented aspects of both the IMPORT programs and the contents of the ER graphs. Intermediate representations of IMPORT objects are generated either by the *IMPORT* $\mapsto$ *IML translator* or by the *ER graph* $\mapsto$ *IML translator*, and passed on to the data base manager. The data base manager obtains graphical entities from either the palette (given directly as input by the user) or from the *graphing algorithm*. The graphing algorithm takes intermediate representation of IMPORT objects as input and produces corresponding graphical entities.

The interface manager oversees all of the viewing tasks concerning the diagrams, windows, and similar interface tools. One of the most important responsibilities of the interface manager is maintaining the query manager. The query manager handles the creation and maintenance of views of IMPORT graphic entities and structures. These views correspond to underlying structures that represent IMPORT objects.

3.1.1 The Database Manager

The purpose of the database manager is to sort all of the data (i.e. IMPORT objects, IML entities, and graphical entities) and send them to the appropriate sub-routines. The database manager breaks down into two parts: the *graph manager* and the *data manager* (figure 3.1). Since there are two different forms of data that our graphical editor manipulates, this division of the database manager is quite natural.

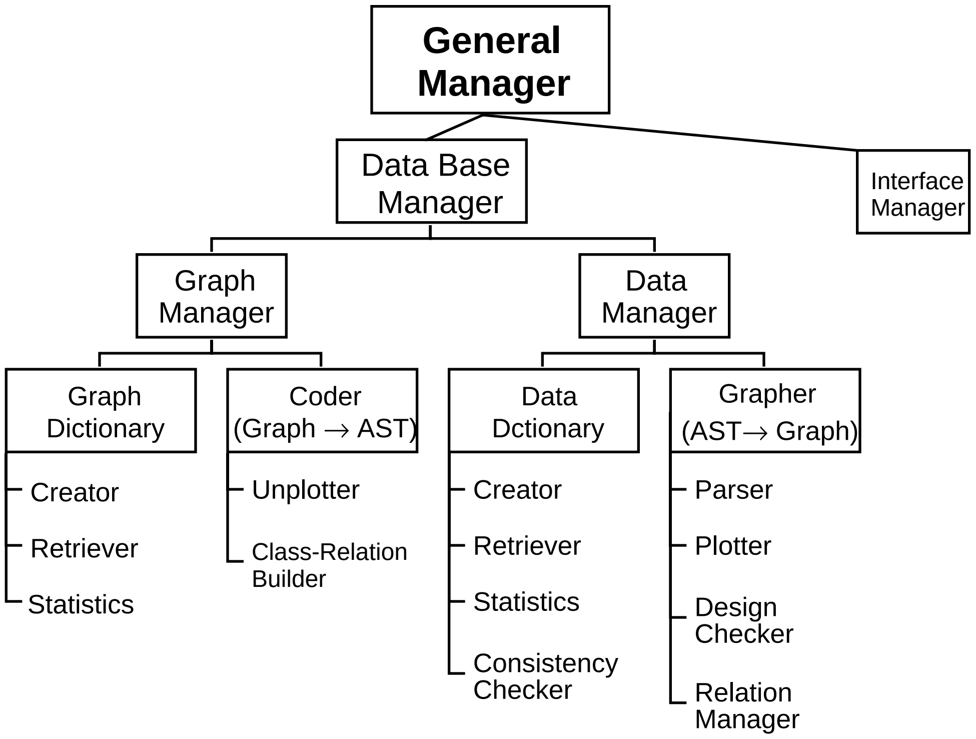


Figure 3.1: The Structured Design of the Database Manager

The graph manager has two parts: the *graph dictionary* and the *coder*. The graph dictionary keeps track of information concerning graph components. The coder performs a two step translation process. First, it translates an ER graph into IML text. Then, it translates the IML text into IMPORT abstract syntax trees (AST).

The data manager has two parts: the *data dictionary* and the *grapher*. The data dictionary keeps track of information concerning IMPORT objects . The data dictionary includes a facility for collecting statistics about IMPORT objects (*e.g.* how many classes are in an IMPORT program, *etc.*). The grapher performs a two step translation process. First, it translates IMPORT source code into IML text. Then, it translates the IML text into an ER graph. Finally, it checks the graph to insure appropriate design structure.

3.1.2 The Interface Manager

The interface manager performs all of the necessary interfacing tasks. These include zooming in/out of a window, navigating from one diagram to another, *etc*. The structured design of the interface manager is shown graphically in Figure 3.2.

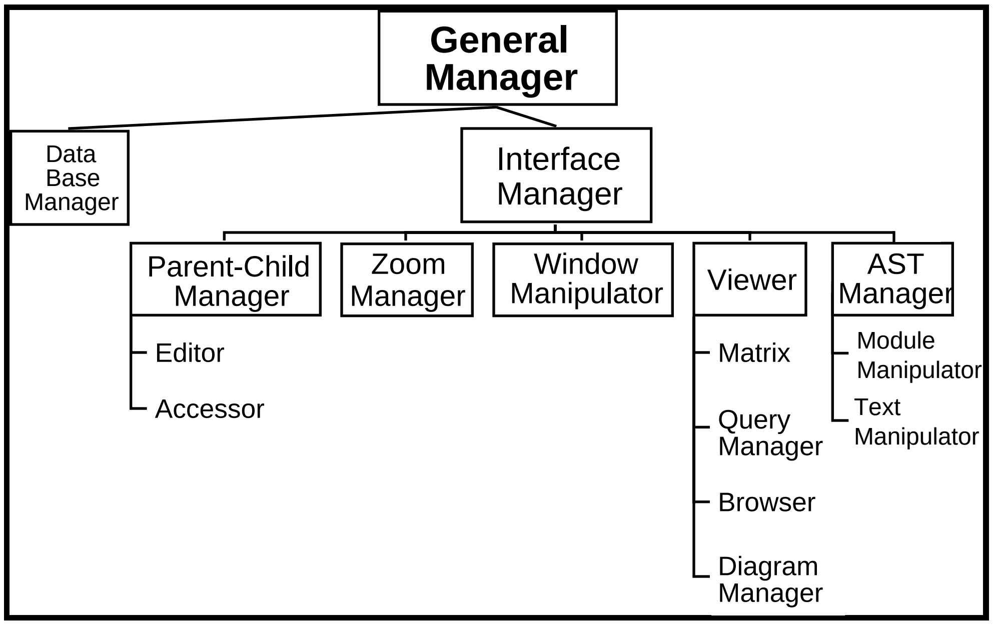


Figure 3.2: The Structured Design of the Interface Manager

The interface manager has the following five main components:

- A parent-child manager which facilitates parents accessing/manipulating their children; and, children accessing their parents. This includes parents adding/deleting/changing their children, and *vice versa*.

- A zoom manager which handles user controlled zooming in and out of diagrams/views. This is a very useful facility when dealing with large diagrams and multiple views.

- A window manipulator, where commands for manipulating windows are implemented. These commands include opening, deleting, focusing, raising, lowering, and resizing windows.

- A viewer where the relation matrix, the class browser, the query manager, and the diagram manager reside. The relation matrix is a square matrix whose rows and columns constitute the classes in the base diagram. The matrix itself holds the relation among the classes represented on the intersection of rows and columns. The class browser allows the user to browse through, add/delete/change the classes in the base diagram quickly. The query manager administers the multiple view diagrams. Last but not least is the diagram[9] manager, which manipulates diagrams in windows. This involves creating, opening, deleting, resizing, closing, and saving a diagram.

- An AST manager where module and text manipulation commands are implemented. These commands are the standard save, open, new, delete, *etc.* commands one finds in many text editors.

[9] Diagrams are part of windows. However, manipulating them does not necessary result in manipulation of the window which contains them.

3.2 The User Interface of the Graphical Editor

The user interface of the graphical editor includes an ER palette and a main window. The palette consists of a set of graphical primitives. These graphical primitives are the building blocks of the object diagrams displayed on the window. The palette is static (*i.e.* different palettes cannot be obtained from it). The main window (*viz.* the editor window) contains the base diagram and a menu bar. The main screen of the graphical editor's user interface is shown in Figure 3.3.

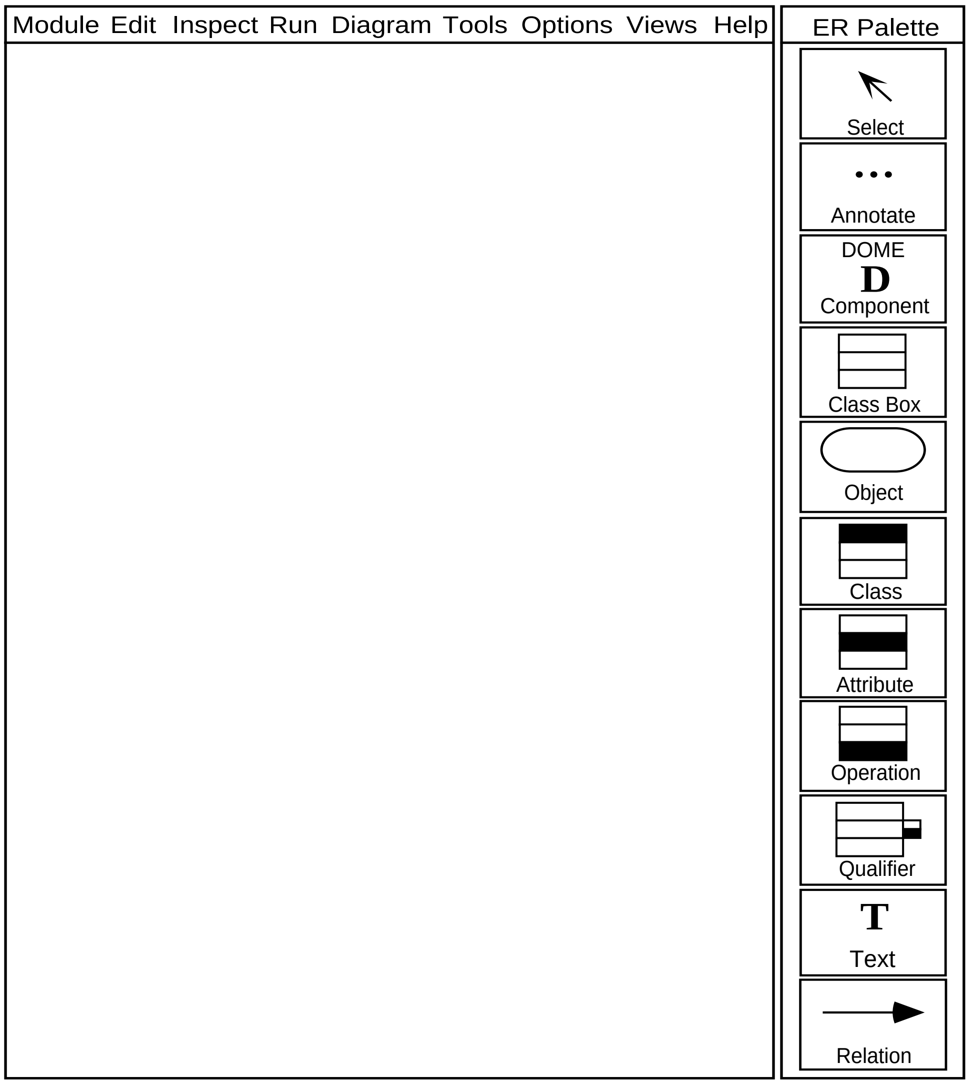


Figure 3.3: The Graphical Editor's Main Screen (ER Model)

The structure of the components of the graphical editor's screen is depicted in greater detail in Figure 3.4.

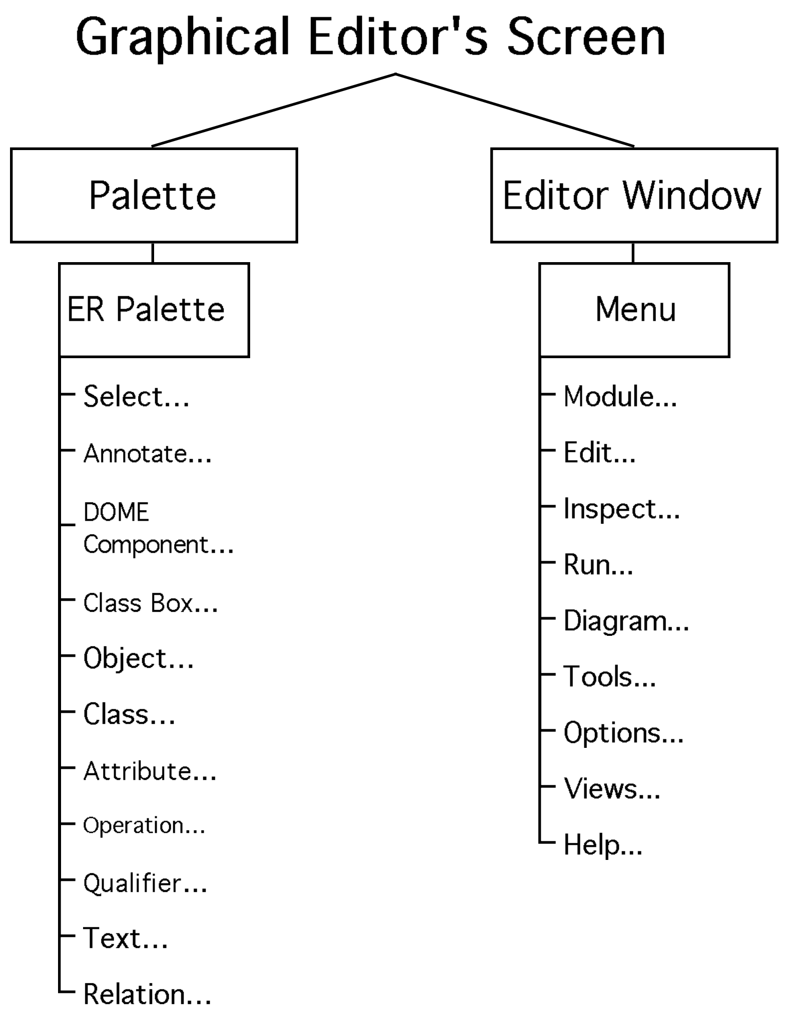


Figure 3.4: Components of the Graphical Editor's Main Screen

The ER Palette is the main graphical tool set and has the following tools.[10]

- Select: A tool for selecting elements of a diagram

- Annotate: A tool for adding comments to elements of a diagram

[10] The definitions of *Object*, *Class*, *Attribute*, *Operation*, and *Qualifier* were taken from [14].

• DOME Component: A tool for labeling classes which have DOME clauses in their definition

• Class Box: A tool for creating a class box. A class box is a box consisting of the name of a class, its attributes, and its operations

• Object: A tool for creating an object. An object is an instance of a class.

• Class: A tool for creating a class. A class is a description of a group of objects with similar properties, common behavior, common relationships, and common semantics.

• Attribute: A tool for creating an attribute. An attribute is a named property of a class describing a data value held by each object of the class.

• Operation: A tool for creating an operation. An operation is a function or transformation that may be applied to objects in a class.

• Qualifier: A tool for creating a qualifier. A qualifier is an attribute of an object that distinguishes among the set of objects at the 'many' end of an association.

• Text: A tool for placing text anywhere in a diagram.

• Relation: A tool for building relations (*e.g.*, generalization, association, aggregation, *etc.*) between classes. The user may define his/her own relations with various properties.

The editor window's menu bar consists of the following elements:

• A Module menu with commands for modifying and/or manipulating modules

• An Edit menu with commands for performing basic editing tasks

- An Inspect menu with commands for scanning diagrams in several ways
- A Run menu which has commands that allow the user to generate IMPORT code from diagrams, and *vice versa*.[11] It also has commands that allow the user to switch between operational modes and run reports and checks on the objects
- A Diagram menu which has commands to manipulate diagrams
- A Tools menu which has commands to make diagrams more organized and readable
- An Options menu which has commands to change the window's format
- A Views menu with commands that allow the user to define, modify and access various views of the base diagram.
- A Help menu which has commands to give the user help concerning the tool with which he/she is working

[11] Both the abstract syntax tree (AST) code and the graphical diagram are generated incrementally.

CHAPTER 4

TEXT AND GRAPH TRANSLATIONS

4.1 The Intermediate Language

One of the main functions of the graphical editor is to translate IMPORT text into ER graphs and *vice versa*. We have created an *intermediate language (IML)* to perform these translations.[12] IML builds a bridge between the object-oriented semantics of an IMPORT program and the structure of an ER graph. IML establishes semantics for the entities of ER graphs by providing one-to-one correspondences between IMPORT objects and ER objects, IMPORT object attributes and ER object attributes, IMPORT object methods and ER object operations, and IMPORT object relations and ER object relations. As a result, IML facilitates the storing of both the object-oriented aspects of an IMPORT program and the contents of its ER graph.

IML represents information about IMPORT objects and graphical entities as a set of *records* (referred to as *nodes*), each of which is composed of *objects*, *attributes*, *operations* and *connections*. The objects correspond to IMPORT objects; the attributes are properties or characteristics of IMPORT objects; the operations are methods of IMPORT objects; and, the connections are relations between IMPORT objects. There are two kinds of connections: *is-a* and *assoc*. The is-a connections depict inheritance relations between classes. The assoc connections capture any relations between objects other than the inheritance relations.

[12] The user of the graphical editor never actually comes into direct contact with the intermediate language.

The BNF for IML is given in Appendix B, section B.1. Figure 4.1 shows an example of an IML node.

```
node
   object  pirana;
   attributes
      mySex=female;
      myHungerThreshold=5;
   operations
      amHungry=ASK METHOD amHungry () : BOOLEAN
               BEGIN
               RETURN (myFood<myHungerThreshold);
               ENDMETHOD;
   connections
      isa      fish;
      assoc   (eats,goldfish);
end node
```

Figure 4.1: An Intermediate Language Node

4.2 IMPORT Text to ER Graph Translations

The IMPORT text to ER graph translation process is a composition of two translation mappings:

- The first mapping is from IMPORT text to IML text. The *IMPORT $\mapsto$ IML translator* induces this mapping by: (1) extracting the relevant object-oriented aspects of the given IMPORT program, and (2) structuring the extracted information into the appropriate format for the second stage of the translation process. The translator is given the AST (abstract syntax tree) of an IMPORT program as a list and returns a list of IML records. The *IMPORT $\mapsto$ IML translator* goes through the following main steps to accomplish its objective:

– It traverses the given AST list searching for IMPORT OBJECT and ASSOCIATION declarations.

– For each OBJECT declaration, it checks the output queue for a previous declaration of the object. If no such declaration is found, it creates an IML record with the OBJECT information; otherwise, it reports an error. The translator knows that there cannot be two objects with the same name in the same program.

– For each ASSOCIATION declaration, it checks the output queue for the objects involved in the association. If such objects exist, it puts the association in the `assoc` sub-field of the `connections` field of the first object in the ASSOCIATION declaration; otherwise, it reports an error. The translator knows that there cannot be an association among objects that have not been declared.

– It places the intermediate language record on the output queue, and continues to do the above steps until it reaches the end of the AST list.

• The second phase is the mapping from IML text to an ER graph. This mapping is induced by the *graphing algorithm*. The graphing algorithm gets a list of IML records and draws a graph on the screen. The vertices of this graph correspond to the nodes in the IML records and its edges correspond to the connections in the IML records. We have chosen to use a variation of Ning's [11] graphing algorithm. He uses a set of drawing algorithms, called *Quick-and-Dirty* algorithms to draw data flow diagrams on a computer screen.[13] Our graphing algorithm uses the same main steps as Ning's *Quick-and-Dirty* algorithms;

[13] See Ning [11] for a detailed description of this kind of algorithms.

however, it switches the places of *x* and *y* coordinates, since we want the diagram to grow downward and not from left to right. Also, since we are dealing with ER diagrams, we do not enforce the strict left to right ordering of nodes that Ning enforces. The main steps in our graphing algorithm are as follows:

– Assign a non-negative integer (called a label) to each node. These labels represent the horizontal position of the node. Nodes with connections between them do not receive the same label.

– Place the nodes with the same label into a single set. A combination of this step and the above step, creates a collection of maximal independent sets[14] of the graph to be drawn.

– Arrange the above collection of uniformly along the vertical axis, so that the nodes in the same set possess the same *y* coordinate.

– Assign *x* coordinates to nodes in the first set; and, arrange them uniformly along the horizontal axis.

– Compute *x* coordinates of the nodes in the other sets by taking the mean of the *x* coordinates of their parents. This situates the neighboring nodes at the same (or nearby) vertical level so that the edges in the graph between these nodes will be parallel to each other, hence reducing edge intersections. Moreover, the neighboring nodes will be located near each other.

– Sort the nodes in each set in ascending order of their *x* coordinates.

[14] An *independent set* of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. A *maximal independent set* is an independent set V' such that for all vertices $v \in V - V'$, the set $V' \cup \{v\}$ is *not* independent—every vertex not in V' is adjacent to some vertex in V' [4].

– Change the *x* coordinates of any nodes in the same set that cluster together.

– Repeatedly remove overlaps by moving the nodes along the horizontal axis. This is done by increasing and decreasing the *x* coordinates of the overlapped nodes. By changing the *y* coordinates only, it is assured that the adjustments do not create new overlaps between nodes in different sets.

– Compute the display coordinates of the nodes and the edges. The starting and ending points of the edges are computed by using the coordinates of the source and the target nodes corresponding to each edge.

In short, for every object in the IMPORT text, an IML record is created by the $\textit{IMPORT} \mapsto \textit{IML translator}$. The sequence of records generated by this first mapping is then passed on to the graphing algorithm, which converts them into an ER graph. See appendix A for a concrete example of the above translation process.

4.3 ER Graph to IMPORT Text Translations

The ER graph to IMPORT text translation process is also a composition of two mappings:

- The first mapping is from ER graph to IML text. The $\textit{ER graph} \mapsto \textit{IML}$ *translator* induces this mapping by transforming the ER graph into IML records, so that it can later be translated into an IMPORT program. The translator is given the ER graph and returns a list containing IML records. The $\textit{ER graph} \mapsto \textit{IML}$ *translator* goes through the following main steps to accomplish its objective:

  – It traverses the given ER graph by using depth-first search [4]. In depth-first search, the unexplored edges leaving the most recently visited vertex *v* are traversed. When all of *v*'s edges have been traversed, the search backtracks to the vertex form which *v* was originally discovered and continues exploring the untraversed edges of that vertex. This process is repeated until all the vertices that are reachable from the original source vertex are visited. If any unvisited vertices remain in the graph, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices in the graph are discovered.

  – For each vertex in the ER graph, it creates an IML node which contains all the information about the vertex and its edges.

  – It arranges these nodes into a list. This list is given as input to the next step of the process after the complete traversal of the ER graph.

• The second mapping is from IML text to an IMPORT program. The $\mathit{IML} \mapsto \mathit{IMPORT\ translator}$ induces this mapping. It converts a list of IML nodes into an IMPORT program. The $\mathit{IML} \mapsto \mathit{IMPORT\ translator}$ goes through the following main steps to accomplish its objective:

  – It parses the given list of IML nodes.

  – For every node on the list, it creates an IMPORT OBJECT. If the node has an `assoc` connection, it will also create an IMPORT ASSOCIATION corresponding to the association relation that obtains between the two IMPORT objects.

  – It stores the IMPORT objects and associations in an output file.

In summary, for every object in the ER graph, an IML node is created by the $\mathit{ER\ graph} \mapsto \mathit{IML\ translator}$. The sequence of nodes generated by the this first mapping is then passed on to the $\mathit{IML} \mapsto \mathit{IMPORT\ translator}$ which converts it into IMPORT text. The IMPORT text generated by the above process will be a *skeleton* which captures the basic object-oriented structure of an IMPORT program. This program skeleton must then be embellished using the text editor to yield a completed IMPORT program. See appendix A for a concrete example of the ER graph to IMPORT text translation process.

CHAPTER 5

VIEWS

5.1 Views: What and Why

In general, a *view* (of a program) is a representation of some component, feature, or aspect of a program. In visual support systems, such representations are naturally depicted in graphical forms. As such, graphical views are an integral part of any visual support system.

Software systems (especially large ones) contain a tremendous amount of information. By providing graphical views of programs (at varying levels of abstraction), visual support systems allow users to comprehend large program structures by focusing on graphical representations of individual features of a program. Consequently, views enable visual support systems to aid the user (1) in the analysis of a program at hand (*i.e.*, program visualization) and (2) in the development of other programs (*i.e.*, visual programming).

5.2 IMPORT Views

The graphical editor allows its user to dynamically create views of IMPORT programs (we call such views *IMPORT views*). An IMPORT view depicts some aspect of the object model of an IMPORT program. For example, an IMPORT view may illustrate a set of objects in an IMPORT program that satisfy some predicate or relation. By allowing the user to graphically focus on particular object-oriented components of an

IMPORT program, the graphical editor gives the user the ability to cut through the inevitable complexities involved in analyzing and developing large IMPORT programs.

IMPORT views are generated by running queries on the object database of an IMPORT program.[15] The results of such queries are represented graphically[16] on the computer screen—in 'real time'. The user is able to *define* IMPORT views: either (1) *textually*, by specifying a query in the graphical editor's query language, or (2) *graphically*, by selecting a portion of a graph or a program on the computer screen using the graphical editor's Select tool.

The graphical editor comes equipped with a default set of *pre–defined* IMPORT views. In addition to the default set of pre-defined IMPORT views, we provide the user with the ability to store customized sets of *pre–defined* IMPORT views, so that he/she can have a record of his/her most frequently used queries. Moreover, the user may perform queries on existing dynamical IMPORT views (yielding what we call *sub-views* of an IMPORT view)—we call this functionality *modifying* an IMPORT view.

5.3 Techniques for Generating IMPORT Views

5.3.1 Generating IMPORT Views Using the Query Language

The graphical editor's query language aides the user in generating multiple views of IMPORT ER diagrams. This query language is an embellished structured query language (SQL). It allows the user to perform queries involving:

[15] IMPORT programs are stored as abstract syntax trees (AST) in an object repository. When using the graphical editor, the user does not come into direct contact with the AST. Instead, he/she views various object-oriented graphical representations of its structure.

[16] Not all views of IMPORT programs are *graphical* views. The user can specify whether they want to see the result of their query *as a list* or *as a graph*. We emphasize graphical views because they tend to be more informative.

- Boolean set operations

  For example, if the user wants to define a view containing the *union* of the set of classes with an attribute called ‘sex’, and the set of classes with an attribute called ‘age’, then the user would enter the following query:

```
(find all x: class such that attribute(x,”sex”)) union
(find all x: class such that attribute(x,”age”))
```

- First Order Predicate logic (FOPL)

  All queries in the query language involve statements of FOPL. Every individual query begins with some *quantifier*. Specifically, each query will exemplify exactly one of the following forms:

```
(find all …)
(find any …)
(find …)
```

  The latter being a short-hand for “find any”.

  The query language also comes equipped with the standard FOPL *connectives*. For instance, the following query will generate a view of some class which has an attribute called ‘age’; *and*, does *not* have an attribute called ‘sex’.

```
(find any x: class such that
    (not(attribute(x,”sex”)) and attribute(x,”age”)))
```

- Relational comparisons

  The query language supports the standard suite of relational comparisons: (>, <, >=, <=, =, !=). For instance, the following query will generate a view of some instance *x* which is taller than some other instance *y*.

```
(find any x,y: instance such that
    attribute(x,"height") > attribute(y,"height"))
```

- Wild cards

  We accomplish this feature using the '`*`' operand. For example, consider the following query:

  ```
  (find all x,y: class such that assoc(x,y,*))
  ```

  This query defines a view of all pairs of classes *x*, *y* such that *x* is associated with *y—in some way or other*.

The BNF for the query language is depicted in Appendix B, section B.2. Also, see Appendix A for more examples of queries and views.

5.3.2 Generating IMPORT Views Using Graphical/Program Tools

To generate a view, the user of the graphical editor may select a portion of a graph (or a program) using the Select tool of the graphical editor. A menu will 'pop-up' allowing the user to choose one of the appropriate *pre-defined views* for the type of component(s) selected. This will automatically perform the desired query on the database, and return the result to the user.

Pre-defined views allow the user to automatically perform the most common queries. This saves the user time and effort because it eliminates the need to repeatedly remember and manually type-in their favorite queries (for the task at hand).

Of course, not all users share the same set of 'most commonly used queries'. Moreover, different programs may be conducive to different query patterns. For these reasons, we have provided the user with the capability to store *custom sets* of pre-defined queries. This allows the user to customize the graphical editor's environment to most effectively suit their needs. The *default* set of pre-defined views (which comes pre-installed with the graphical editor), is the set described in Table 5.1 below.

Item Selected by User	Default Pre-Defined Views Provided
Some class: *C*	All/any subclasses of *C* All/any superclasses of *C* All/any relations involving *C*
Some instance: *i*	All/any classes *C* such that $i \in C$
An attribute: *A*	All/any classes/instances with *A*
An operation: *O*	All/any classes/instances with *O*
A qualifier: *Q*	All/any classes/instances with *Q*
An inheritance (is-a) relation: *I*	All/any pairs of classes satisfying *I*
An association (assoc) relation: *R*	All/any pairs of classes satisfying *R*

Table 5.1: The Default Set of Pre-Defined Views

5.4 Multiple Views and Viewing Modes

Ideally, visual support systems should make *multiple* views of a program available to the user at any given time. As Grady Booch has testified: “It is impossible to capture all of the subtle details of a complex software system in just one view” [1]. Visual support systems with multiple-view capability have several advantages over systems that can only display one view of a program at a time. A combination of two (non-equivalent) views will always contain *more* information than either of the individual views. Moreover, different *kinds* of views—with varying levels of abstraction—can be combined in a single display. In this way, the user of a multiple-view visual support system can not only obtain more information about a program; but, they can also obtain several different kinds of information about a program—all at the same time. Because multiple-view capability is essential for optimal visual support, the graphical editor allows the user to generate, and concurrently observe or manipulate *multiple* IMPORT views.

To distinguish between the program visualization and visual programming aspects of our visual support tool, we have incorporated two distinct operational modes into the graphical editor: *analytical* mode and *editorial* mode. The user is in the *analytical* mode of the graphical editor when he/she is viewing one of the dynamically defined IMPORT views. He/she is in the *editorial* mode of the graphical editor when he/she is developing/modifying programs in the base diagram of the IMPORT program. The user cannot edit IMPORT views (*i.e.*, IMPORT views are *read-only*; whereas, the base diagram of an IMPORT program is *modifiable*). While the user is observing an IMPORT view, its components will be shown in distinct colors on the base diagram. The user can switch from an IMPORT view to the base diagram of the IMPORT program at any time, in order to perform modifications to the corresponding part of the IMPORT program's ER graph.[17] As the user makes changes to the object-oriented structure of an IMPORT program, these changes are automatically (and in real time) updated and reflected graphically in the display of each of the current IMPORT views.

[17] This is technically feasible, since the same reference names are used for the view components throughout the diagram and its dynamical views.

CHAPTER 6

CONCLUSION AND EXTENSIONS

This manuscript has presented a visual support system for the ISLE simulation environment. The ISLE simulation environment integrates characteristics of object-oriented modeling, process-based simulation, declarative programming, and persistent object storage into one software engineering environment. The IMPORT programming language is the underlying simulation language of ISLE. It satisfies the requisite functionality of ISLE.

The ISLE visual support system was designed to provide the IMPORT programming language with a graphical user interface (GUI) development and analysis tool. The ISLE visual support system combines visual programming and program visualization techniques to ease the processes of IMPORT code development and manipulation. Because IMPORT programs can be very complex, understanding their structure can be a daunting task. ISLE's GUI visual support tool allows the IMPORT user to cut through the complexity of IMPORT programs. Consequently, the graphical editor supports the IMPORT user in all activities of the software life cycle.

The graphical editor described in this thesis defines a visual programming mapping from graphical representations of the object-oriented structures of IMPORT programs into IMPORT code. Inversely, the graphical editor defines a program visualization mapping from IMPORT programs into graphical representations of their object-oriented structures. The graphical representations of the object-oriented structures of IMPORT programs are based on Rumbaugh's [8] object model. They capture graphical representations of IMPORT objects and the relations between them.

An intermediate language (IML) has been developed to perform translations from IMPORT code to ER graph components and *vice versa*. The IML builds a bridge between the object-oriented semantics of an IMPORT program and the structure of an ER graph. The IML establishes semantics for the entities of an ER graph by providing a one-to-one correspondence between the components of an IMPORT program's object model and the components of its ER diagram. As a result, IML facilitates the storing of both the object-oriented aspects of an IMPORT program and the contents of its ER graph.

An important feature of this visual support tool is its ability to dynamically create multiple views of IMPORT entities and structures *via* database queries. An IMPORT view depicts some aspects of the object model of an IMPORT program. The graphical editor allows its user to concurrently analyze and manipulate multiple IMPORT views. Hence, the user of the graphical editor can simultaneously perceive the structure of IMPORT programs from a variety of different perspectives and levels of abstraction.

The graphical editor's multiple-view capability is facilitated either textually, by specifying a query in the graphical editor's query language, or graphically, by selecting a portion of a graph or a program on the computer screen. The graphical editor's query language is an embellished, structured query language. The query language allows the user of the graphical editor to define and modify IMPORT views using Boolean set operations, first order predicate logic, relational comparisons, and/or 'wildcards'. Consequently, the user of the graphical editor can view the object-oriented structure of an IMPORT program from many distinct points of view.

Currently, our visual support tool is only capable of capturing the ER model of an IMPORT program. Since the IMPORT programming language combines the features of several programming languages, the next logical step would be to generalize the graphical editor so that it would be able to capture *other* models of IMPORT programs. Specifically, IMPORT is not only *object-oriented*; but, it is also *imperative*, *declarative*, and *concurrent*. Accordingly, a more complete visual support system for ISLE would be able to visually capture these other important aspects of IMPORT programs. For instance, such a system might have the capability to provide visual representations of *state transition models*, *data flow models* and *concurrency models* of IMPORT programs. An extended visual support system of this kind would require the appropriate palettes, graphical primitives, and visual programming/program visualization mappings for each of the different models it supports.

Finally, the present graphical editor only allows the user to visualize one IMPORT program abstraction at a time. It would be desirable for future versions of the graphical editor to facilitate visual abstractions of *multiple* IMPORT programs, simultaneously. By visualizing more than one IMPORT program abstraction at a time, the graphical editor would aide the user in creating new IMPORT programs from existing ones. This would greatly enhance the graphical editor user's ability to reuse IMPORT structures.

APPENDIX A

THE GRAPHICAL EDITOR: AN EXAMPLE

A.1 The Aquarium Simulation

The ISLE aquarium simulation models a small, fish-tank environment as a predator-prey system [18]. Aquarium systems include predator fish (*viz*., piranha), prey fish (*viz*., goldfish), and prey food (*viz*., goldfish food). Both predators and prey exhibit complex behaviors such as hunger, exhaustion, life span, sex drive, and roaming. The simulation models the behavior of the aquarium's inhabitants by keeping track of their hunger level, energy level, location, speed, and several other bits of information.

Fish in the model aquarium can die from lack of food, old age, or by becoming fish food. Living in such an environment often requires complex decision making skills. The inhabitants of the tank must constantly weigh the costs and benefits of potential actions. Each fish in the model aquarium is equipped with its own expert system to facilitate the decision making processes necessary for its survival. By altering the knowledge bases of predators and prey, one can use the aquarium simulation to examine how different strategies will affect the fitness of various inhabitants of an aquarium environment. Figure A.1 depicts a sample screen from an aquarium simulation.[18]

[18] This figure was generated by running the ISLE aquarium demo described in appendix A of [18].

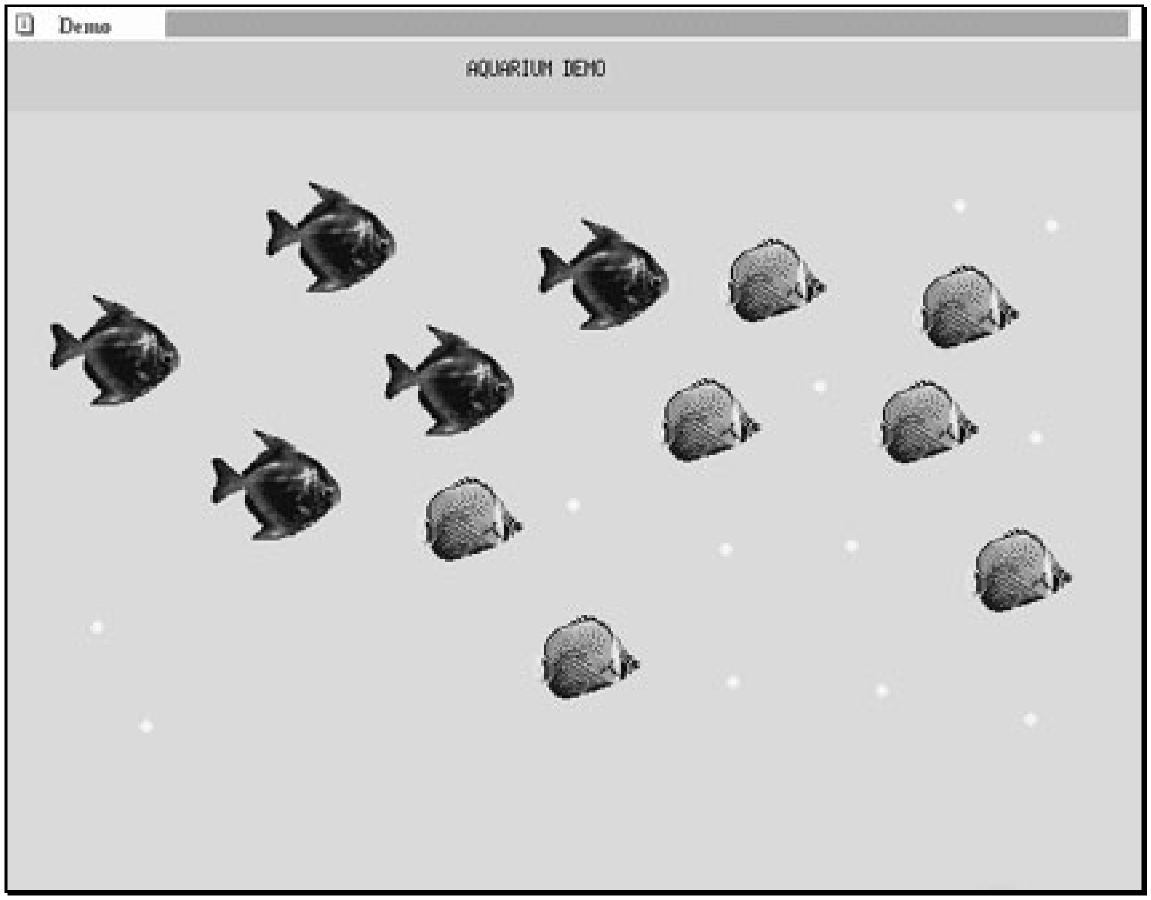
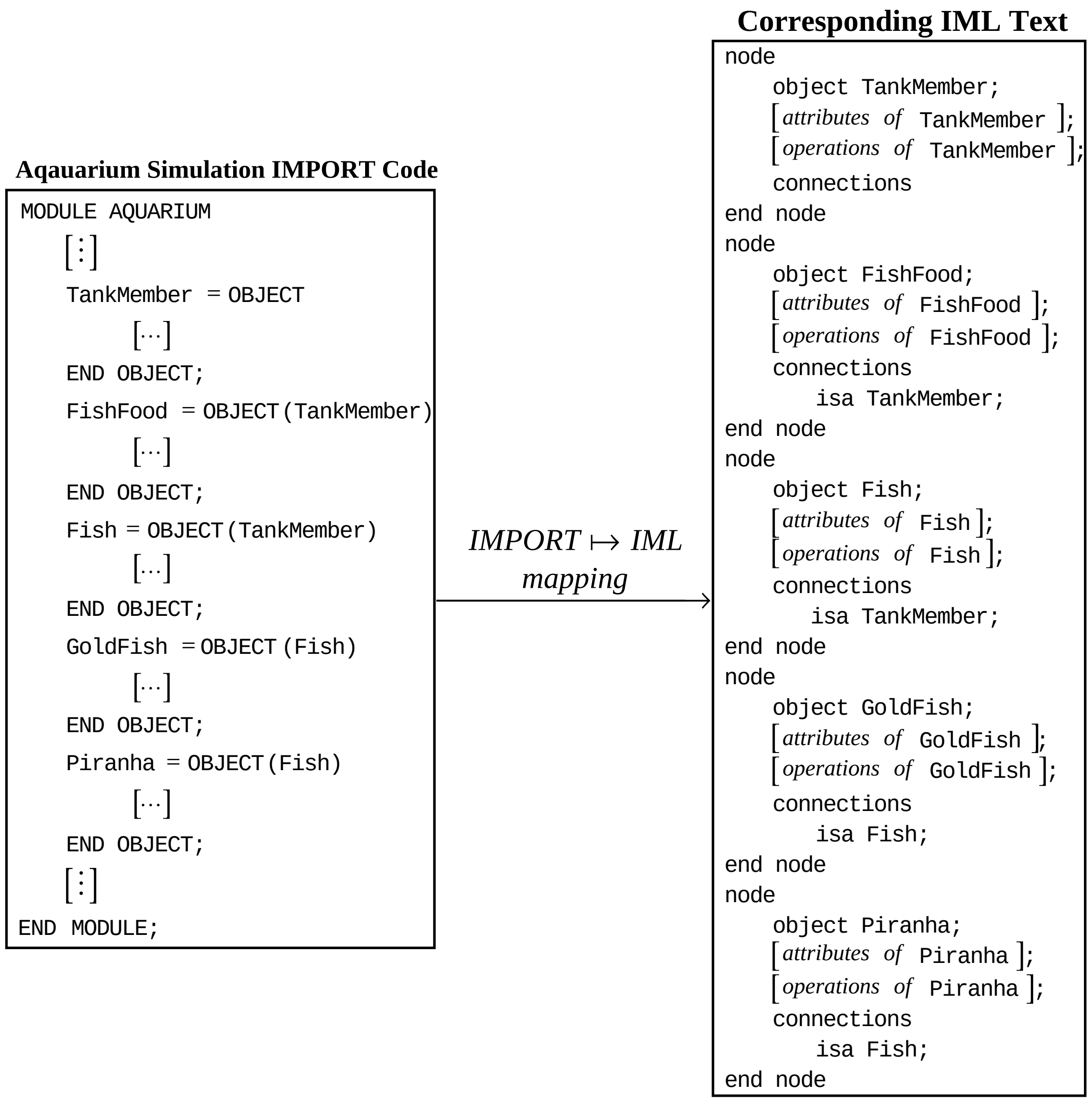


Figure A.1: A Sample Screen from an Aquarium Simulation

A.2 IMPORT Text to ER Graph Translation in the Aquarium Simulation

As discussed in section 4.2, the IMPORT text to ER graph translation process consists of two mappings. The first mapping is from IMPORT text to IML text, and it is facilitated by the $IMPORT \mapsto IML \;\; translator$. The second mapping is from IML text to ER graph, and it is facilitated by the *graphing algorithm*.

We have extracted a portion of IMPORT text from the aquarium simulation source code for the purpose of illustrating the process of translating IMPORT text into an ER graph.[19] The IMPORT text to IML text mapping is shown in figure A.2; and, the IML text to ER graph mapping is shown in figure A.3.



Aqauarium Simulation IMPORT Code

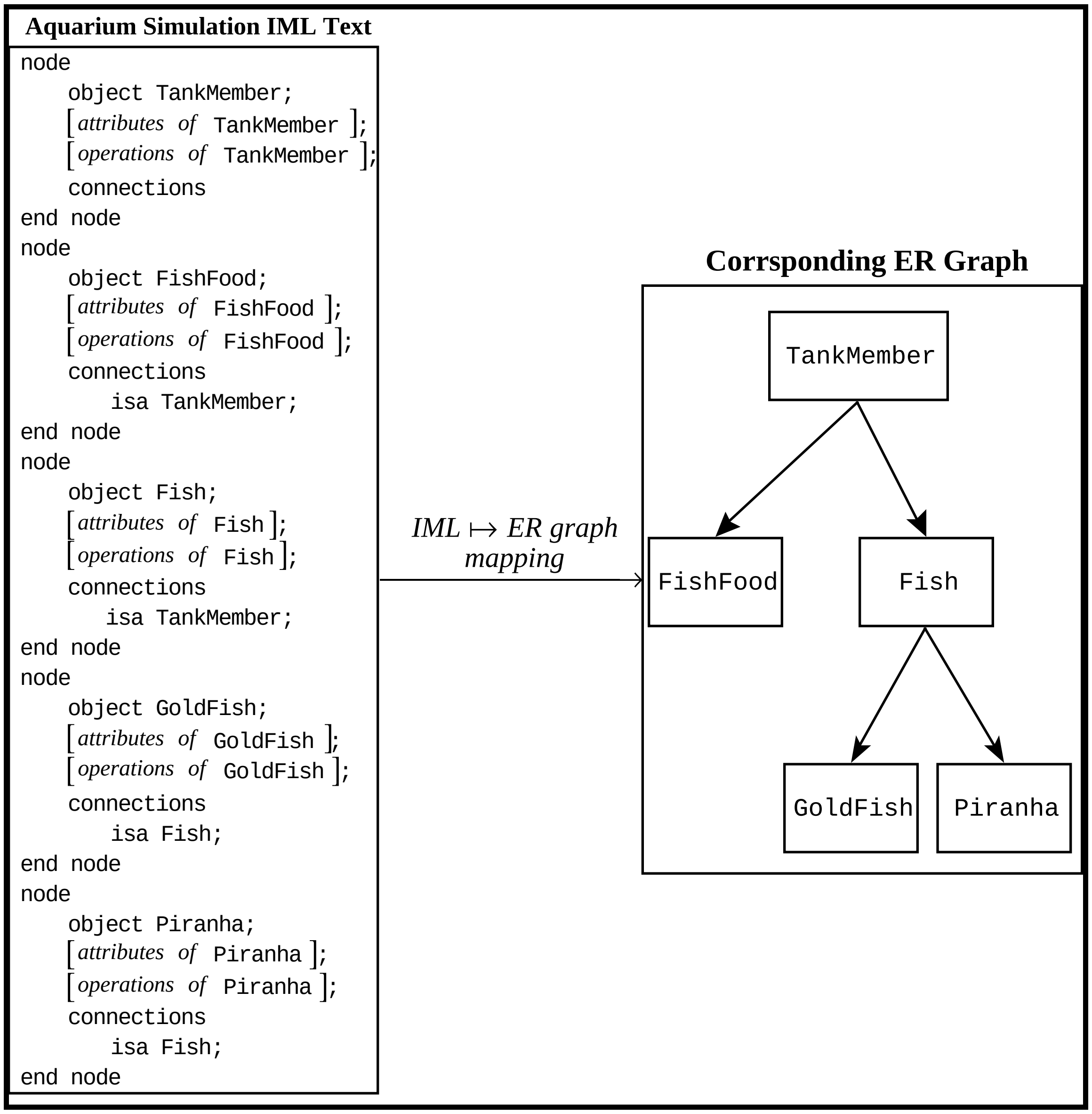
```
MODULE AQUARIUM
    [⋮]
    TankMember = OBJECT
          [⋯]
    END OBJECT;
    FishFood = OBJECT(TankMember)
          [⋯]
    END OBJECT;
    Fish = OBJECT(TankMember)
          [⋯]
    END OBJECT;
    GoldFish = OBJECT (Fish)
          [⋯]
    END OBJECT;
    Piranha = OBJECT(Fish)
          [⋯]
    END OBJECT;
    [⋮]
END MODULE;
```

Corresponding IML Text

```
node
    object TankMember;
    [attributes of TankMember];
    [operations of TankMember];
    connections
end node
node
    object FishFood;
    [attributes of FishFood];
    [operations of FishFood];
    connections
        isa TankMember;
end node
node
    object Fish;
    [attributes of Fish];
    [operations of Fish];
    connections
        isa TankMember;
end node
node
    object GoldFish;
    [attributes of GoldFish];
    [operations of GoldFish];
    connections
        isa Fish;
end node
node
    object Piranha;
    [attributes of Piranha];
    [operations of Piranha];
    connections
        isa Fish;
end node
```

Figure A.2: Translation of Aquarium Simulation IMPORT Code into IML Text

[19] Specifically, this portion of the aquarium simulation source code describes the following IMPORT objects: `TankMember`, `FishFood`, `Fish`, `Piranha`, and `GoldFish`.



```
node
    object TankMember;
    [attributes of TankMember];
    [operations of TankMember];
    connections
end node
node
    object FishFood;
    [attributes of FishFood];
    [operations of FishFood];
    connections
       isa TankMember;
end node
node
    object Fish;
    [attributes of Fish];
    [operations of Fish];
    connections
       isa TankMember;
end node
node
    object GoldFish;
    [attributes of GoldFish];
    [operations of GoldFish];
    connections
       isa Fish;
end node
node
    object Piranha;
    [attributes of Piranha];
    [operations of Piranha];
    connections
       isa Fish;
end node
```

Figure A.3: Translation of Aquarium Simulation IML Text into an ER Graph

After this pair of translations is performed on the extracted portion of the aquarium simulation's IMPORT source code, a visualization of its object-oriented structure is obtained. Hence, we have concrete example of the *program visualization* capabilities of the graphical editor.

A.3 ER Graph to IMPORT Text Translation in the Aquarium Simulation

As discussed in section 4.3, the ER graph to IMPORT text translation process consists of two mappings. The first mapping is from ER graph to IML text, and it is facilitated by the $\textit{ER graph} \mapsto \textit{IML translator}$. The second mapping is from IML text to IMPORT text, and it is facilitated by the $\textit{IML} \mapsto \textit{IMPORT translator}$.

Suppose we want to extend the aquarium simulation model (of the previous section) to include the following two new classes of IMPORT objects: `Plant` and `ToySubmarine`. We would begin by using the appropriate tools from the ER palette to add the new classes to the existing ER diagram of the aquarium simulation model (depicted in figure A.3). The new ER diagram (figure A.4) is then passed on to the $\textit{ER graph} \mapsto \textit{IML translator}$ which produces the corresponding IML text. Finally, the $\textit{IML} \mapsto \textit{IMPORT translator}$ transforms the IML text into IMPORT text (figure A.5). The IMPORT text generated by the aforementioned process is only an IMPORT program *skeleton* (*i.e.*, it only captures the basic object-oriented structure of an IMPORT program). In order to obtain a full-fledged IMPORT program, we must employ the text editor to properly embellish the IMPORT skeleton.

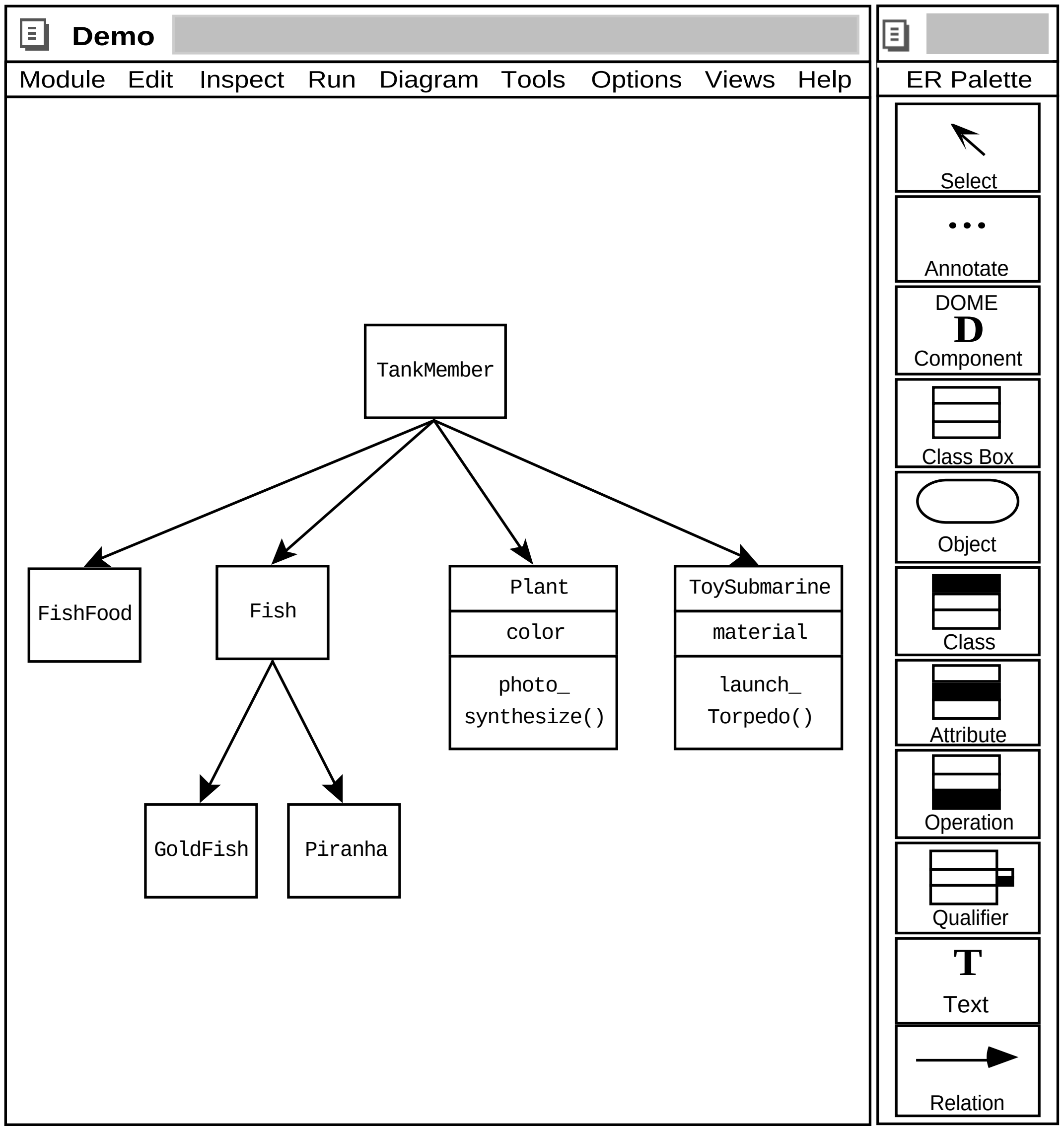
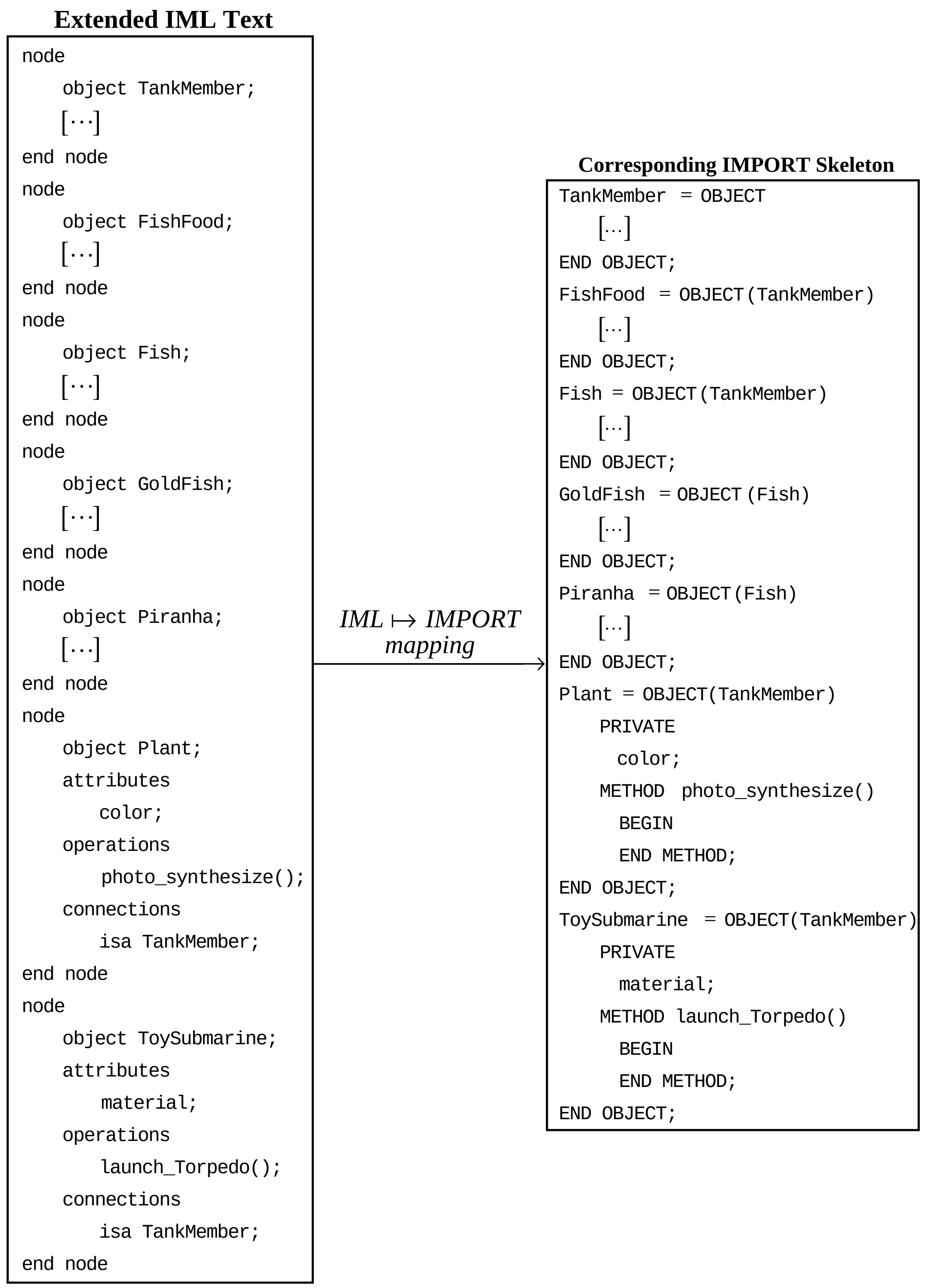


Figure A.4: Extending the ER Graph of the Aquarium Simulation[20]

By allowing the user to visually specify ER diagrams for IMPORT programs (as above), the graphical editor facilitates visual IMPORT programming.

[20] This screen is only a 'mock-up' of an actual graphical editor screen.



```
node
    object TankMember;
    [···]
end node
node
    object FishFood;
    [···]
end node
node
    object Fish;
    [···]
end node
node
    object GoldFish;
    [···]
end node
node
    object Piranha;
    [···]
end node
node
    object Plant;
    attributes
        color;
    operations
        photo_synthesize();
    connections
        isa TankMember;
end node
node
    object ToySubmarine;
    attributes
        material;
    operations
        launch_Torpedo();
    connections
        isa TankMember;
end node
```

```
TankMember  = OBJECT
    [···]
END OBJECT;
FishFood  = OBJECT(TankMember)
    [···]
END OBJECT;
Fish = OBJECT(TankMember)
    [···]
END OBJECT;
GoldFish  = OBJECT (Fish)
    [···]
END OBJECT;
Piranha  = OBJECT(Fish)
    [···]
END OBJECT;
Plant = OBJECT(TankMember)
    PRIVATE
      color;
    METHOD  photo_synthesize()
      BEGIN
      END METHOD;
END OBJECT;
ToySubmarine  = OBJECT(TankMember)
    PRIVATE
      material;
    METHOD launch_Torpedo()
      BEGIN
      END METHOD;
END OBJECT;
```

Figure A.5: Translation of the Extended IML Text into an IMPORT Skeleton

A.4 Views of the Aquarium Simulation

As was discussed in section 5.3.1, we may generate views of IMPORT programs (called *IMPORT views*) using the query language. We can illustrate this way of generating IMPORT views using the aquarium simulation. For example, assume that we have created 10 instances of the IMPORT object `GoldFish`. To generate an IMPORT view consisting of any `GoldFish` that was assigned a maximum speed of 5, we may perform the following query:

```
(find x: instance such that
     isa(x, “GoldFish”) and (attribute(x, mySpeed) <= 5))
```

To create an IMPORT view containing the intersection of the set of all IMPORT objects with an operation called `reproduce`, and the set of all IMPORT objects with an operation called `die`, we may perform the following query:

```
(find all x: class such that operation(x, “reproduce”))
     intersection
(find all x: class such that operation(x, “die”))
```

The resulting IMPORT view is depicted in figure A.6.

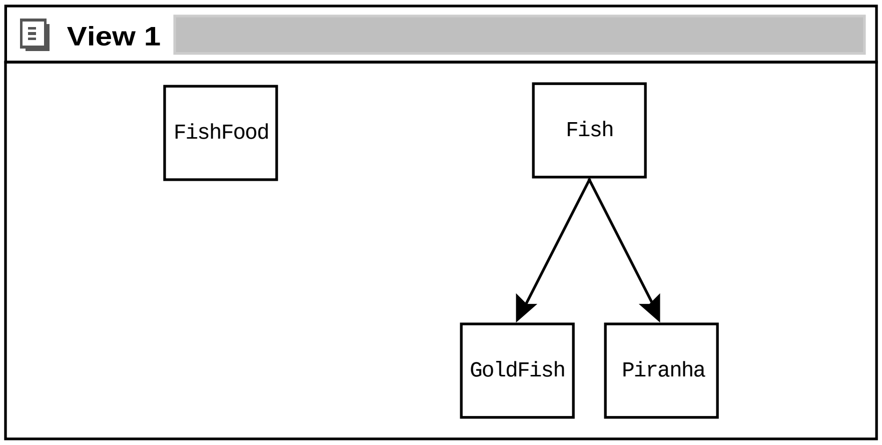


Figure A.6: An IMPORT View of the Aquarium Simulation

To create an IMPORT view containing all classes that have no sub-classes, we may perform the following query:

```
(find all x: class such that not(isa(*,x)))
```

The resulting IMPORT view is depicted in figure A.7.

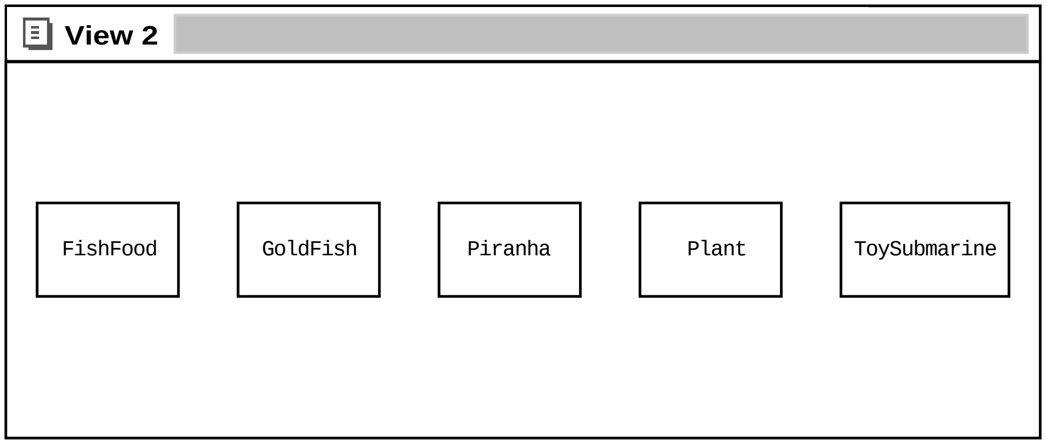


Figure A.7: Another IMPORT View of the Aquarium Simulation

When developing and analyzing large IMPORT programs, it helps to be able to simultaneously observe their structure from multiple perspectives. As the above examples illustrate, the graphical editor allows its user to generate and concurrently analyze multiple IMPORT views.

APPENDIX B

LANGUAGE CONSTRUCTS

In this appendix, we will introduce the BNF language constructs for the intermediate language and the query language. The intermediate language and the query language were discussed in detail in section 4.1 and chapter 5 respectively.

B.1 The BNF for the Intermediate Language

Figure B.1 shows the BNF for the intermediate language which was discussed in section 4.1.[21]

⟨node list⟩ := ⟨node⟩ | ⟨node list⟩⟨node⟩

⟨node⟩ := **node**⟨object name⟩[**attributes**⟨attribute list⟩] [**operations**⟨operation list⟩]
    [**connections**⟨connection list⟩] **end node**

⟨object name⟩ := **object**⟨identifier⟩;

⟨attribute list⟩ := ⟨attribute⟩ | ⟨attribute list⟩⟨attribute⟩

⟨attribute⟩ := ⟨identifier⟩[**=**⟨attribute expr⟩]**;**

⟨operation list⟩ := ⟨operation⟩ | ⟨operation list⟩⟨operation⟩

⟨operation⟩ := ⟨identifier⟩**=**⟨operation expr⟩**;**

⟨connection list⟩ := [**isa**⟨isa list⟩]**;**[**assoc**⟨assoc list⟩]

⟨isa list⟩ := ⟨isa⟩ | ⟨isa list⟩⟨isa⟩

⟨isa⟩ := ⟨identifier⟩**;**

⟨assoc list⟩ := ⟨assoc⟩ | ⟨assoc list⟩⟨assoc⟩

⟨assoc⟩ := **(**⟨identifier⟩**,**⟨identifier⟩$^+$**)**;

⟨attribute expr⟩ := ⟨identifier⟩ | ⟨IMPORT enum⟩

⟨operation expr⟩ := ⟨identifier⟩ | ⟨IMPORT function⟩

⟨IMPORT function⟩ := ⟨function kind⟩**METHOD**⟨identifier⟩**(**⟨identifier⟩***)**
    ⟨return type⟩⟨function body⟩**;**

⟨identifier⟩ := ⟨letter⟩$^+$ ⟨number⟩*

⟨letter⟩ := **A** | **B** | **C** | ⋯ | **X** | **Y** | **Z** | **a** | **b** | **c** | ⋯ | **x** | **y** | **z**

⟨number⟩ := **0** | **1** | **2** | ⋯ | **7** | **8** | **9**

Figure B.1: BNF for the Intermediate Language

[21] For further information on the constructs ⟨IMPORT enum⟩ and ⟨IMPORT function⟩ see the IMPORT language reference documentation [18].

B.2 The BNF for the Query Language

Figure B.2 below depicts the BNF for the query language discussed in chapter 5.

⟨expr⟩ := ⟨expr⟩ (⟨set operation⟩⟨expr⟩)*

⟨expr⟩ := (**find**⟨selector⟩ **such that** ⟨clause⟩)

⟨selector⟩ :=⟨select list⟩

⟨select list⟩ := ⟨select item⟩ | ⟨select list⟩,⟨select item⟩

⟨select item⟩ := [⟨qualifier⟩]⟨variable list⟩* **:** (**class** | **instance** | **assoc**)

⟨clause⟩ := [**not**]⟨sub clause⟩(⟨logical connective⟩⟨sub clause⟩)$^{+}$

⟨sub clause⟩ := ⟨sub clause⟩(⟨relational comparison⟩⟨sub clause⟩)$^{+}$

⟨sub clause⟩ := ⟨operator⟩⟨operands⟩

⟨operator⟩ := **class** | **instance** | **attribute** | **operation** | **quantifier** | **isa** | **assoc**

⟨operands⟩ := (⟨variable⟩$^{+}$**,**⟨operand⟩)

⟨operand⟩ := ⟨variable⟩ | **“**⟨variable⟩**”**

⟨set operation⟩ := **union** | **intersection** | **difference**

⟨logical connective⟩ := **and** | **or**

⟨relational comparison⟩ := **>** | **<** | **>=** | **<=** | **=** | **!=**

⟨variable list⟩ := ⟨variable⟩ | ⟨variable list⟩**,**⟨variable⟩

⟨variable⟩ := ⟨alphanumeric⟩ | *****

⟨alphanumeric⟩ := ⟨letter⟩$^{+}$ ⟨number⟩*

⟨letter⟩ := **A** | **B** | **C** | ⋯ |**X** | **Y** | **Z** | **a** | **b** | **c** | ⋯ | **x** | **y** | **z**

⟨number⟩ := **0** | **1** | **2** | ⋯ | **7** | **8** | **9**

⟨quantifier⟩ := **all** | **any**

Figure B.2: BNF for the Query Language

REFERENCES

1. Booch, G., 1994, *Object-Oriented Analysis and Design*, Second Edition, Benjamin/Cummings, New York.

2. Brown, G. P., *et. al.*, August 1985, “Program Visualization: Graphical Support for Software Development”, *IEEE Computer*, **18**(8) 27-35.

3. Chang, S. K., 1990, “Visual Reasoning for Information Retrieval from Very Large Databases”, *Journal of Visual Languages and Computing*, **1**(1) 41-58.

4. Cormen, T. H., Leiserson, C. E., and Rivest, R. L., 1990, *Introduction to Algorithms*, McGraw-Hill, New York.

5. Grafton, R. B. and Ichikawa, T., August 1985, “Guest Editors’ Introduction in the Special Issue on Visual Programming”, *IEEE Computer*. **18**(8) 6-9.

6. Johnson, R. E., Spring 1994, Lecture Notes for Computer Science 397 at the University of Illinois Urbana-Champaign.

7. Kamin, S. N., 1990, *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, New York.

8. Kreutzer, W., 1986, *System Simulation Programming Styles and Languages*, International Computer Science Series, Addison-Wesley, Sydney.

9. Mollamustafaoglu, L., Gurkan, G., and Ozge, A. Y., January 1993, “Object-Oriented Design of Output Analysis Tools for Simulation Languages”, *Simulation*, **60**(1) 6-16.

10. Myers, B. A., 1990, “Taxonomies of Visual Programming and Program Visualization”, *Journal of Visual Languages and Computing*, **1**(1) 97-123.

11. Ning, Q., 1984, “Graphical Representation of Dataflow Diagrams: Design and Implementation”, CS Masters Thesis, UIUC.

12. Raeder, G., August 1985, “A Survey of Current Graphical Programming Techniques”, *IEEE Computer,* **18**(8) 11-25.

13. Roman, G. C. and Cox, K. C., December 1993, “A Taxonomy of Program Visualization Systems”, *IEEE Computer*, **26** 11-24.

14. Rumbaugh, J., *et. al.*, 1991, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey.

15. Stelovsky, J., Ackermann, D., and Conti, P., May 1986, “Visualizing of Program Structures: Support Concepts and Implementation”, in the Selected Contributions of Visualization in Programming 5th Interdisciplinary Workshop in Informatics and Psychology, *LNCS*, **282** 37-52.

16. Tanir, O. and Sevinc, S., February 1994, “Defining Requirements for a Standard Simulation Environment”, *IEEE Computer*, **27**(2) 28-34.

17. Whitehurst, R. A., *et. al.*, 1993, “Integrated Object Technologies for General Purpose Simulation”, *OOPSLA ‘93*.

18. Whitehurst, R. A., *et. al.*, 1995, *Integrated Simulation Language Environment*, on-line manuscript.