Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

# Development of a Client Interface for a Methodology Independent Object-Oriented CASE Tool

A thesis presented in partial fulfilment
of the requirements for the degree
of Master of Science in Computer Science
at Massey University, New Zealand

**Steven Kevin Adams**

**1998**

# ABSTRACT

The overall aim of the research presented in this thesis is the development of a prototype CASE Tool user interface that supports the use of arbitrary methodology notations for the construction of small-scale diagrams. This research is part of the larger CASE Tool project, MOOT (Massey's Object Oriented Tool). MOOT is a meta-system with a client-server architecture that provides a framework within which the semantics and syntax of methodologies can be described.

The CASE Tool user interface is implemented in Java so it is as portable as possible and has a consistent look and feel. It has been designed as a client to the rest of the MOOT system (which acts as a server). A communications protocol has been designed to support the interaction between the CASE Tool client and a MOOT server.

The user interface design of MOOT must support all possible graphical notations. No assumptions about the types of notations that a software engineer may use can be made. MOOT therefore provides a specification language called NDL for the definition of a methodology's syntax. Hence, the MOOT CASE Tool client described in this thesis is a shell that is parameterised by NDL specifications.

The flexibility provided by such a high level of abstraction presents significant challenges in terms of designing effective human-computer interaction mechanisms for the MOOT user interface. Functional and non-functional requirements of the client user interface have been identified and applied during the construction of the prototype. A notation specification that defines the syntax for Coad and Yourdon OOA/OOD has been written in NDL and used as a test case. The thesis includes the iterative evaluation and extension of NDL resulting from the prototype development.

The prototype has shown that the current approach to NDL is efficacious, and that the syntax and semantics of a methodology description can successfully be separated. The developed prototype has shown that it is possible to build a simple, non-intrusive, and efficient, yet flexible, useable, and helpful interface for meta-CASE tools. The development of the CASE Tool client, through its generic, methodology independent design, has provided a pilot with which future ideas may be explored.

# ACKNOWLEDGEMENTS

This thesis is deservedly dedicated to Miriam, one who knows self-control much better than I, and who always steered me back toward my work when it needed to be done. Seeing this thesis finally reaching fruition means so much more to me because of her and her unselfish and unconditional love, support, and encouragement.

Thanks must also go to the following people who have played a part in my research efforts:

- Daniela Mehandjiska-Stavreva, for being my supervisor, and giving me the opportunity to work in a field I enjoy;
- Chris Phillips, for the co-supervision and invaluable input you have given to me in your field of expertise;
- David Page, for checking my work, and for being the one who knew what they were talking about when no-one else seemed to;
- The other members of the MOOT research team, past and present, whom I have had the pleasure of working with.

# CONTENTS

# TABLE OF FIGURES

# Chapter 1

# INTRODUCTION

## 1.1. Object-Oriented Development

Over the past decade, object-oriented (OO) technology has moved into the mainstream of industrial-strength software development. Object-oriented languages in particular were developed in response to a need for programming languages with semantics that captured more meaning from the problem domain, rather than from the artefacts of computer hardware (Collins, 1995). The evolution of software development methods from structured analysis, design, and implementation to object-oriented approaches has revolutionised the way that software is built (Sommerville, 1996). Indeed, OO modelling techniques have changed the way that we think about enterprises and the way we design related business processes (Martin, 1993).

Object-oriented software development is characterised by four main features: information hiding (encapsulation), data abstraction, inheritance, and dynamic binding. Object-oriented modelling techniques focus software development on data (ie. objects) and the interfaces to it, rather than on the tools that are available for system construction. Encapsulation and data abstraction allow a clear separation between the specification of data and how it may be manipulated, and the actual implementation of object interfaces. Inheritance allows new classes to be defined in terms of existing classes, thereby improving and reinforcing reuseability. Dynamic binding allows different but related classes of objects to be dynamically substituted in place of a common class, which supports a higher level of generalisation than could have previously be obtained.

The acceptance of OO modelling techniques as an effective software development strategy has led to the development of numerous OO methodologies (over 50 at the time of writing). Each OO methodology prescribes a particular process for one or more phases of the software development lifecycle including requirements gathering, analysis, design, implementation, testing, and maintenance. Each OO methodology

uses its own set of models that are used to describe a software artefact. Construction of these models is undertaken using a methodology's own particular set of notations.

Three generations of OO methodologies have been defined over the past decade. First generation methodologies were developed in the late 1980s and early 1990s. These included Wirfs-Brock's responsibility driven design (Wirfs-Brock, 1990), Booch's OOD (Booch, 1991), Rumbaugh's OMT (Object Modelling Technique) (Rumbaugh, 1991), Coad and Yourdon OOA/OOD (Coad and Yourdon, 1991a, 1991b), Shlaer and Mellor's OOA (Shlaer and Mellor, 1991), and Jacobson's Objectory (Object Factory for Software Development) (Jacobson *et al*, 1994).

The first generation techniques were applied and evaluated, resulting in second generation methodologies. These included Booch's OOA/OOD (Booch, 1994), Graham's SOMA (Semantic Object-Oriented Modelling Approach) (Graham, 1994), Henderson-Sellers' MOSES (Methodology for Object Oriented Software Engineering Systems) (Henderson-Sellers *et al*, 1994), Martin and Odell's Advanced Object Modelling (Martin and Odell, 1995), Coleman's Fusion method (Coleman *et al*, 1993) and Rumbaugh's second generation OMT (Rumbaugh, 1995a, 1995b).

To address the diversity of first and second generation methodologies, the OO community has started looking at the possible standardisation of third generation methodologies. The Unified Modelling Language (UML) (Booch, 1994; Rumbaugh, 1995b; Jacobson *et al*, 1994) and the OPEN Modelling Language (OML) (Henderson-Sellers *et al*, 1996) have been defined. UML is a convergent modelling language comprising Booch, Rumbaugh's OMT, and Jacobson's Objectory. OML is proposed by Brian Henderson-Sellers, Ian Graham, and Donald Firesmith, with input from a Consortium of methodology researchers including Larry Constantine, Meilir Page-Jones, Bertrand Meyer, Rebecca Wirfs-Brock, and James Odell. UML provides only a notation, whilst OML also has a defined process.

## 1.2. CASE Technology

The diversity of OO software development methodologies was reflected by the creation of several generations of CASE (Computer Aided Software Engineering) tools. The main objective of CASE tools is to support software engineers in some or all phases of the software development lifecycle, with the ultimate aim of enhancing productivity and producing low defect solutions. First generation CASE tools addressed mostly form and representation issues of software development methodologies (Sorenson, 1988a).

Program support tools such as translators, compilers, assemblers, linkers, and loaders were developed. Later, the range of support tools began to expand with the development of program editors, debuggers, code analysers, and so on. (Page *et al*, 1998)

Large-scale software development, however, demanded enhanced support for the entire software development process from CASE tool developers (Sumner, 1992). Assistance was required for the requirements definition, design, and implementation phases of the software development lifecycle. Testing, documentation and version control support was also required. The evolution of CASE tools split into two broad domains. Front-end or upper-CASE tools were concerned with the early phases of the software development lifecycle (such as requirements definition and design support tools). Those tools used later in the lifecycle (such as compilers and testing tools) were referred to as back-end or lower-CASE tools.

First generation CASE tools aided the user in creating system analysis and design diagrams, and detailed textual-based specifications. They performed consistency, completeness, and correctness checking, and some provided a primitive form of code generation. Their main disadvantages were inadequate methodology support, no customisation facilities, lack of support for reverse engineering, and an inability to integrate the different CASE tools used at various stages of software development (Page *et al*, 1998).

Second generation CASE tools attempted to address some of the problems of first generation tools. Integration was achieved by sharing the definitions of objects and relationships described in a common dictionary. Methodology support was improved by the production of tool sets supporting customisation using a meta-system approach (Brough, 1992; Rossi *et al*, 1992; Smolander *et al*, 1991; Sorenson, 1988b). However, second generation CASE tools were still deficient in a number of important areas. They lacked support for defining new methodologies (Nilsson, 1990; Papahristos, 1991), and they did not provide information interchange of analysis and design results expressed in different methodologies. Meta-system support for the description of the semantics of more than one methodology was also limited (Mehandjiska *et al*, 1996a). From a useability perspective, the tools did not facilitate the navigation of complex structures of data. (Page *et al*, 1998)

Current research into CASE technology has been concentrated in two main areas. The first addresses the development of software environments with an open architecture,

aiming at the integration of independently developed CASE tools (Lang, 1991; Nilsson, 1990; Sorenson, 1988b; Papahristos *et al*, 1991). Attempts have been made to create an open environment in which different methodologies and their supporting CASE tools coexist. Such environments would provide multiple views of evolving models in both graphical and textual forms. To support the user, all views within an environment would be kept consistent with one another in as automatic and transparent a fashion as possible (Grundy *et al*, 1995). The benefit to users of such integrated environments is that the interaction model with the tool is consistent across all phases of software development. This approach has increased the reuseability of information. For example, communication among diverse methodologies has been addressed by a common data dictionary in the proposed Federated CASE Environment (Papahristos *et al*, 1991). Unfortunately, however, these environments are typically restricted to particular methodologies, and cannot be significantly extended or customised to meet specific user requirements.

The second area of research addresses the methodology dependence of CASE tools. A meta-modelling approach has been utilised to allow the generation of customised software environments. The goal of a meta-system is to (semi-)automatically generate the software necessary for a specific environment. Research prototypes adopting this approach include Metaview, MetaEdit, MetaPlex, and RAMATIC (Smolander *et al*, 1991, Sorenson *et al*, 1988b). The meta-system approach allows the environment for a given methodology to be specified in two parts: a conceptual definition, and a graphical definition. Conceptual definitions can be based on different data models. For example, MetaEdit (Smolander *et al*, 1991) is based on the Object-Property-Role-Relationship (OPRR) model, Metaview (Sorenson *et al*, 1988b) is based on Entity-Aggregate-Relationship-Attribute (EARA) model, and RAMATIC is based on the set-oriented data model. The developed prototypes support mechanisms to express the mapping between the meta-modelling concepts and the corresponding graphical representations.

The developed meta-tools have several deficiencies. In general, none of these systems are aimed purely at OO software development. The underlying models of the tools (eg. EARA, OPRR, etc.) do not directly support the object-oriented concepts of inheritance and message passing. In addition, the developed research prototypes also do not address the important human-computer interaction issues.

## 1.3. Industry Adoption of CASE Technology

Due to the vast number of OO software development methodologies, an equally large number of OO CASE tool products are available for use in the software industry. Each product offers support for specific phases of the software development lifecycle, using any manner of methodologies.

Unfortunately, many of the current OO CASE tools suffer from generic problems. One of the fundamental problems is the lack of flexibility (Phillips *et al*, 1998). Because of their methodology dependence, current CASE tools often cannot meet the needs of different users, and many CASE environments provide too fixed a variety of techniques (Marttiin 1994). In one study conducted on the adoption of CASE tools in industry (Iivari, 1996) it was found that although CASE tools improved development procedures and standardisation to a degree, in many cases an increase in productivity was not forthcoming. This may be due to the lack of CASE tool functionality being properly identified. Identifying and standardising CASE tool interfaces is crucial for the success of open and customisable CASE environments (Lang 1991).

The software industry has been very slow to adopt CASE technology for many other reasons:

- The support of a methodology that is provided by a CASE tool is often limited to a collection of diagram editors that correspond to the various models a methodology provides. The underlying process and the actual methods are often ignored.
- Many firms utilise in-house processes or methodologies. Their means of work may also be a modification or extension of a popular, accepted methodology. Neither of these situations are supported very well by current CASE technology as the majority of OO CASE tools do not allow customisation.
- CASE tools that support the exchange of information between individual components of the CASE environment do so at the expense of effective exchange of information between the software engineers who need to work together on a project (Churcher *et al*, 1996). Often users of such tools are given the impression that they are the only user of the system.
- Many CASE tools do not integrate well into the existing operation of an organisation. This means that changes are required to adopt a new tool. People in general are resistant to change.

- CASE tools do not provide support for reuse of analysis and design models between different projects. Whilst OO technology does not guarantee reuse, it is generally accepted that one of the principle objectives of OO technology is to support reuse.
- Some people feel that CASE tools will 'de-skill' and 'constrain' them rather than enhance their productivity.

The reasons for lack of proliferation of CASE technology in the software industry can be classified into limitations concerning flexibility, functionality, and useability of the available CASE tools.

## 1.4. Meta-CASE Tool Interfaces

Research in the area of meta-CASE technology has focused almost entirely on the underlying meta-models of such tools and the application of these meta-models to describing the semantics of methodologies. The evaluation of several well-known meta-CASE tools (Graphical Designer, Meta Edit+, Rational Rose, WithClass, and OOTher) (Phillips *et al*, 1998) suggests that very little research has been conducted on the user interface requirements of such tools. The evaluation framework described in the paper identifies criteria of a user interface that relate to usability. In reference to useability, it was found that the meta-CASE tools examined were inflexible, supporting the view that current CASE tools provide a rigid environment in which user actions are constrained. Also of concern was that none of the tools were considered particularly robust, in that support for the achievement of user goals (such as error prevention and recovery) was potentially lacking.

The results of this evaluation are not surprising. The design of the user interface of meta-CASE tools is a much more difficult task than for a traditional piece of software. Meta-CASE tools are designed to support multiple software development methodologies, and hence the user interfaces to them must be designed at a very high level of abstraction. Features specific to a subset of the available methodologies typically cannot be supported without the tool becoming more specialised toward that subset. The user interface of a meta-CASE tool would need to either support only the subset of user interface features common to different methodologies, or support some method of parameterisation that allows the interface to be customised to arbitrary methodologies.

Many CASE tool environments are unnecessarily complex. For example, consider Figure 1-1. This shows the user interface of the Paradigm Plus CASE tool, and is a typical example of the traditional direct manipulation, tool-based interface. This interface appears large and complex, and the diagram being edited is overwhelmed by the interface[1]. Such an interface can be quite difficult and slow to use, mainly because it is based on modes and selections. A user interface that was much leaner in design, and provided more generic methods of operation that could be supported easily across a wide range of methodologies, would be significantly quicker to learn, easier to use, and reduce the net amount of errors and error-recovery mechanisms required.

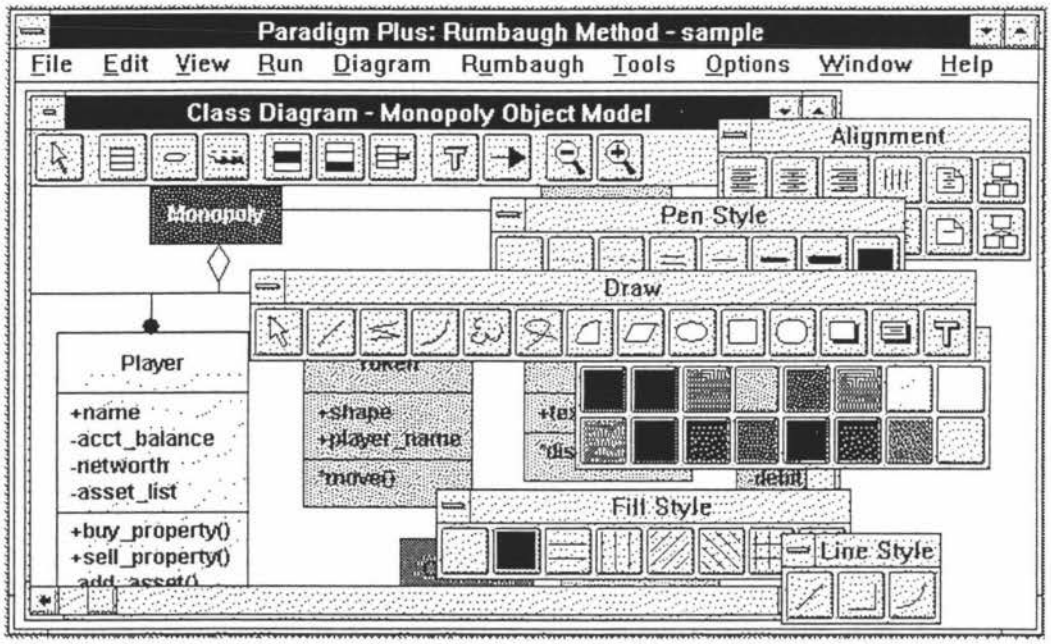



**Figure 1-1 – User interface of the Paradigm Plus CASE Tool (Noble, 1996)**

## *1.5. MOOT – A New CASE Tool*

Research to address the deficiencies of existing CASE technology has been undertaken through the development of a new CASE tool, MOOT (Massey's Object-Oriented Tool) (Mehandjiska *et al,* 1995, 1996a; Page *et al*, 1998). The research aim is to construct a useable, customisable CASE tool which provides a framework within which OO methodologies can be described.

The initial focus of the research was the development of a CASE tool which supports only OO methodologies. However, further consideration of existing OO methodologies

[1] It should be noted that the image is from promotional material for Paradigm Plus, and hence the figure appears more congested than it would in normal use.

indicated that some of them support models adopted from conventional structured analysis/structured design and information engineering methodologies (eg. Rumbaugh, Martin and Odell, UML). In addition, future developments of OO technology may result in new methodologies with different perceptions of the OO paradigm and consequently new requirements for the supporting tools. These future developments cannot be predicted. This means that the new methodology independent CASE tool MOOT must be flexible enough to allow description of such methodologies.

Methodologies are defined in terms of a process with which a software artefact is developed. The process involves the construction of a number of models that describe the artefact. These models have semantic meaning from which information about the artefact can be ascertained. Models typically consist of graphical structures that are built using a predefined set of symbols. These symbols form the syntax with which models may be expressed.

To allow high levels of customisation and flexibility, MOOT utilises two methodology specification languages: Semantic Specification Language (SSL) and Notation Definition Language (NDL). These languages support the definition of the semantics and syntax of a methodology, respectively. The logical and physical separation of the two languages is a fundamental design decision to promote reuse of semantic and syntactic methodology components. For example, an SSL description of a methodology may be associated with more than one NDL definition.

The underlying meta-modelling approach adopted by MOOT breaks away from traditional methods used in existing meta-CASE tools. Instead of extending the conventional models to permit advanced semantic-based data modelling (eg. aggregation, generalisation, and classification), the MOOT approach is to use the object meta-model which naturally and directly supports all these concepts. To this end, MOOT is based on the object-oriented concepts of objects, classes, inheritance, and message passing. MOOT has a common methodology knowledge base which models the core (generic) OO concepts. Non-generic features of OO methodologies require the use of specialised knowledge bases to allow the complete definition of an OO methodology. The common methodology knowledge base serves as a basis for achieving migration of analysis and design results between different methodologies.

## 1.6. Architecture of the MOOT CASE Tool

The MOOT environment is divided into two logical sub-systems: the CASE Tool sub-system, and the Methodology Development sub-system. These sub-systems support the two types of users of MOOT. The first is the software engineer who interacts with the CASE Tool sub-system to build descriptions of software artefacts (referred to as a user project). The second is the methodology engineer who interacts with the Methodology Development sub-system to build and modify descriptions of methodologies. The research presented in this thesis relates only to the MOOT CASE Tool sub-system.

### 1.6.1. The MOOT CASE Tool Sub-System

The CASE Tool sub-system is the CASE of the MOOT environment. It is an integrated tool-set that allows software engineers to develop software by applying methodologies described using the Methodology Development sub-system. The behaviour of the CASE Tool sub-system is completely determined by the methodology is use. Each user project is an instance of the methodology the software engineers use to define it.

The CASE Tool sub-system supports a client-server architecture, as shown in Figure 1-2. Multiple clients may interact with the CASE Tool server via the Tool Manager of the MOOT Core. The Tool Manager functions as a server, processing one thread of control for each CASE Tool client. The Tool Manager maintains an instance of a Methodology Interpreter for each user project that is open in each client. The Tool Manager and the Methodology Interpreters are in turn clients of the Persistent Store. The Persistent Store is a shared repository that facilitates storage of methodology descriptions, user projects, individual user environment details, and so on.

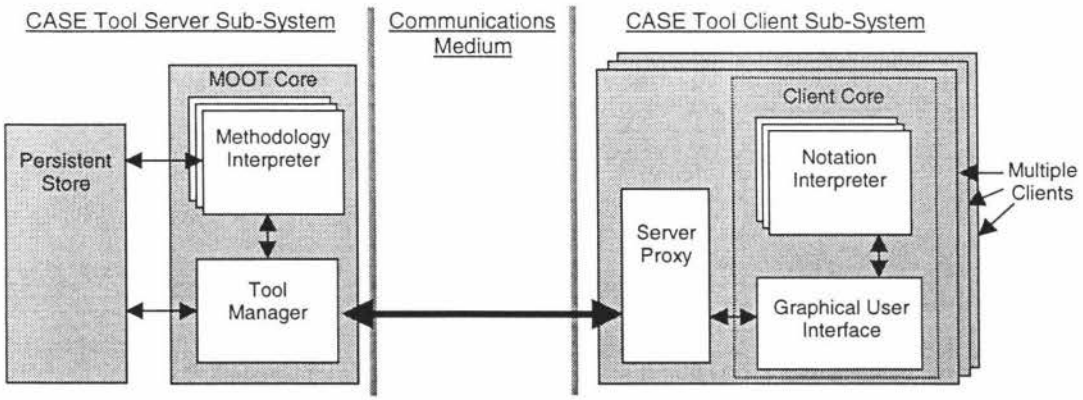



**Figure 1-2 – Architecture of the CASE Tool Sub-system of MOOT**

**CASE Tool Clients**

Each CASE Tool client is only responsible for the presentation of, and user interaction with, a methodology. The only methodology specific information maintained by a client is an NDL description of the methodology syntax. A Notation Interpreter is used in the client to provide the syntactic descriptions of a methodology and the user interactions that may occur with these descriptions to the graphical user interface. The semantics of methodology descriptions are managed by unique corresponding instances of a Methodology Interpreter in the server. Each client is responsible for mapping physical user input to equivalent logical actions for the CASE Tool server. Only actions that have an effect on the meaning of the model being described are propagated to the server (eg. the creation or deletion of a concept or connection). The Methodology Interpreter corresponding to the methodology in use applies the description of that methodology, specified using SSL, to create user software projects.

**Server Proxy**

The communication between the client and server sub-systems is supported by a Server Proxy defined in the client. This proxy acts as a communication interface between the client's graphical user interface and the Tool Manager. Requests for semantic changes to a model are generated by various user interactions with the graphical user interface. The Server Proxy is responsible for assembling these requests into a suitable form for transmission to the server. The Server Proxy is also responsible for receiving requests from the server, and delivering the request details to the appropriate target in the client. Only one instance of a Server Proxy is created in each instance of a client.

**Tool Manager**

The Tool Manager facilitates communication between CASE Tool clients and the other components of the server. The Tool Manager is responsible for coordinating access to shared resources, and for monitoring the system's operation. Only one instance of the Tool Manager is operating in each instance of the MOOT CASE Tool sub-system. The Tool Manager is responsible for maintaining details on the user environments specific to each client, such as personal preferences, the methodology in use, the projects that are open, and so on. The Tool Manager is also responsible for maintaining corresponding instances of Methodology Interpreters for each project that is open in each CASE Tool client.

### 1.6.2. Methodology Descriptions

A methodology description in MOOT is composed of three parts: a description of the semantics of the methodology, a description of the visual syntax, and a table describing the mapping between the two descriptions (a Notation-Semantic Mapping (NSM) Table). Two methodology specification languages have been developed to allow the definition of the semantics and syntax of a methodology in the MOOT system. Respectively these are SSL (Semantic Specification Language) and NDL (Notation Definition Language).

**SSL**

SSL is an object-oriented language used to define the semantics of a methodology (Page *et al*, 1997, 1998). The semantic description includes the models supported by the methodology, the underlying process, and the various documents that are produced by application of the methodology. A semantic description of a methodology consists of a collection of SSL classes. SSL classes are compiled into a platform-independent, binary byte-code representation that is interpreted by an SSL virtual machine. Each Methodology Interpreter contains an instance of an SSL virtual machine (Page *et al*, 1997).

Each SSL class may have many instances. For example, an SSL class that represents a particular methodology model will have a corresponding SSL object instance created for each new model of that type that is created. A software project, developed with a particular methodology, consists of a collection of SSL objects.

**NDL**

NDL is a scripting specification language used to define the notation of methodology models. Notations are described in an NDL specification as a collection of NDL templates. NDL templates describe how the symbols and connections that may appear in the individual diagrams of a model are rendered onto a computer display. NDL provides facilities for binding user actions (such as text area updates) to symbols and connections. Logical distortion (the reshaping of symbols to show more, less, or just different information) is also supported in NDL.

A rendered image that is generated from an NDL template is called an NDL view. A new NDL view is created every time a property of the view (such as the contents of a text area) is modified. Many NDL views may be created from a single NDL template. For example, every view of a class symbol that is rendered in a diagram will be an instance of a single NDL template that describes the appearance of such a symbol.

Figure 1-3 shows the relationship between the description languages (SSL and NDL), a description of a particular methodology in MOOT, and a corresponding software project. A methodology is described by a collection of SSL classes and NDL scripts. Software projects in MOOT consist of a collection of SSL objects and NDL views. A software project in MOOT is an instance of a methodology description that is defined in SSL and NDL.

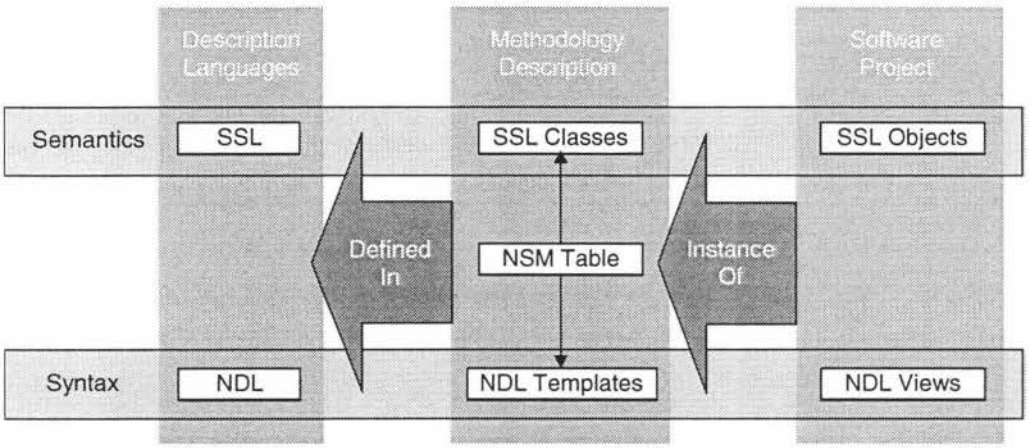



**Figure 1-3 – Relationship between software projects, methodology descriptions, and the description languages**

**NSM Table**

Each methodology description also defines exactly one Notation-Semantic Mapping (NSM) table. NSM tables (which exist in the Tool Manager of the CASE Tool server sub-system) contain the mapping necessary to translate logical actions at the user interface (such as the creation or deletion of a connection) to the corresponding equivalent semantic action. This means the logical action is transformed into a message to an SSL object which responds to the action. In the process of executing a semantic action, an SSL object may generate other semantic actions as a side effect. If these knock-on actions affect the syntactic representation of a model, then the user interface needs to be informed. Therefore, the NSM table is also be used to transform semantic actions back into the equivalent logical actions that the user interface can deal with.

### 1.6.3. Notation-Semantic Mapping

NDL views are visual representations of the semantic information described by SSL objects (for example, a particular class or object). An SSL object may take part in more than one model in a project (for example, an object may appear in sequence and class diagrams in UML). Thus more than one view for any SSL object will often exist (Figure 1-4). The different views may exist in different contexts (ie. different models)

but may also appear in the same context (for example the same external entity may appear more than once on a data flow diagram).

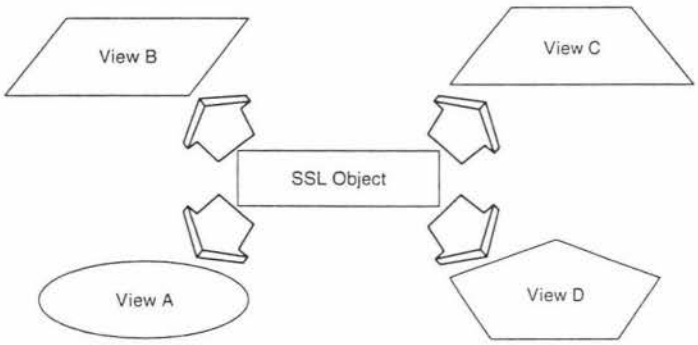



**Figure 1-4 – Multiple views of an SSL object**

An NDL view is a container of visual syntax information and is derived from an NDL template. De-coupling as much as possible an NDL view from the SSL object that it represents is one of the requirements of MOOT.

An SSL object proxy is used in MOOT to de-couple NDL views and SSL objects. An SSL object proxy, termed a *viewable thing*, is a container of the values of the attributes of an SSL object, and provides the values that appear in the text areas in a corresponding NDL view[2]. Attributes of SSL objects are typed (integers, strings, collections, and so on), while properties of viewable things (ie. viewable properties) are only strings. The purpose of this de-coupling mechanism is to maximise the separation between the NDL and SSL descriptions. The use of strings in a viewable thing has been also been adopted by UML, as stated in the UML notation guide (Rational, 1997):

> "Strings represent various kinds of information in an 'unparsed' form. UML assumes that each usage of a string in the notation has a syntax by which it can be parsed into underlying model information. For example, syntaxes are given for attributes, operations, and transitions. These syntaxes are subject to extension by tools as a presentation option."

Each property that an SSL class defines has a type, and an ID number that is unique within the context of the MOOT system. Viewable properties that relate to the attributes of SSL objects are all strings, and have an ID number that is unique within the context of a particular notation. NDL templates are written in terms of viewable

[2] A viewable thing is actually a container of all the syntactic and semantic properties of view. Only the properties that relate to SSL objects relevant to the notation description are discussed in this section.

property ID numbers. The mapping between SSL ID numbers and viewable property ID numbers is defined in a Notation-Semantic Mapping table (Figure 1-5).

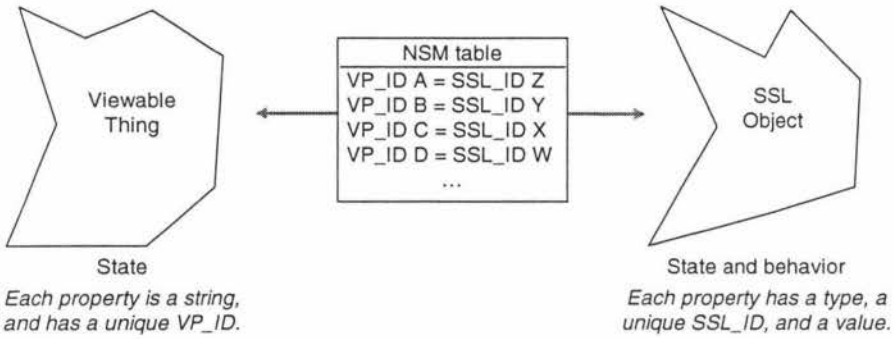



**Figure 1-5 – Notation-Semantic Mapping**

This notation-semantic mapping mechanism effectively isolates the syntactic and semantic descriptions of a methodology to the extent that different NDL descriptions may be associated with different SSL descriptions. By modifying the NSM table, a single notation can be associated with completely different semantic definitions. Alternatively, an SSL semantic description may be able to be expressed using different notations. This support for reuse in MOOT is fundamentally different to that of other meta-CASE environments which only provide reuse by duplication (such as MetaEdit+). The reuse strategy of MOOT is a reflection of the underlying object meta-model.

### 1.6.4. CASE Tool Clients

The CASE Tool clients of the MOOT system encapsulate all the information on how to display, manipulate, and control the interface that software engineers use in the description of software artefacts. The CASE Tool client sub-system provides support for software engineers to create user projects using software engineering methodologies that have been previously defined. User projects typically consist of a number of models supported by the methodology. Each model may contain one or more diagrams. In most instances there is a one-to-one mapping between models and diagrams (ie. a model generally contains only one diagram), however multiple diagrams may be permitted where a methodology definition supports it.

A MOOT CASE Tool client is essentially a graphical user interface (GUI) shell that is parameterised by NDL specifications. Each specification defines the syntax of a set of symbols and connections (notation elements) that may exist in the diagrams of a model. The GUI provides a set of drawing tools that allows a software engineer to construct

diagrams using these notation elements. The set of drawing tools available for a particular model is based on a standard set of generic tools (applicable to the construction of any diagram) and the notation elements that are defined in the NDL specification for that model.

A software engineer creates a diagram by selecting drawing tools that represent notation elements and by placing instances of these onto a drawing canvas. Each notation element that appears in a diagram is an instance of one or more NDL template (an NDL view). Each NDL view encapsulates a viewable thing that contains the viewable properties associated with the view.

Figure 1-6 illustrates the relationship between a viewable thing, an NDL template, and a generated NDL view for an arbitrary notation. The template contains a definition of the view in terms of graphical components (and other primitives). A Notation Interpreter creates a view of a template when it is provided with a viewable thing. The Notation Interpreter requests information from the viewable thing as it generates the view. The size and position of the graphical components for a view may depend on the information stored in the corresponding viewable thing. For instance, if additional attribute items were defined in the viewable thing shown in the figure, the size of the corresponding view would increase, and the text describing the operations would be repositioned in order to accommodate the new information.

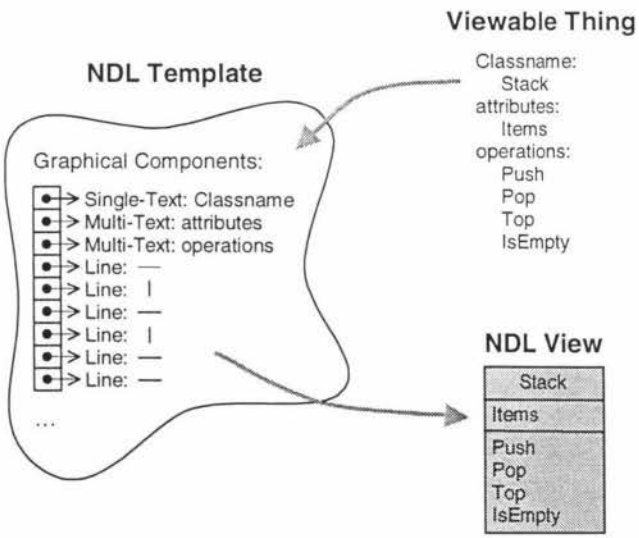



**Figure 1-6 – The relationship between Viewable Thing, NDL Template, and NDL View**

The CASE Tool client communicates with the server whenever semantic changes to a user project take place. For example, the creation of a new model, or the updating of text in a view, is a semantic change. User interactions that do not cause semantic changes, such as the repositioning of a notation element, are handled entirely by the client.

## 1.7. Aspects of MOOT Related to the Thesis

The focus of this thesis is on the representation and interpretation of methodology notation descriptions by the MOOT CASE Tool. The overall aim is to develop a prototype MOOT CASE Tool client that supports the use of arbitrary methodology notations in the construction of small-scale diagrams. Research has been conducted in the following areas:

**A.** An analysis and review of existing methodology notations for the purposes of defining the requirements of NDL.

**B.** The development of an abstract notation definition language (NDL) that allows the specification of the syntax of arbitrary methodologies, and the design of a notation interpreter that allows sentences defined in NDL to be subsequently interpreted and executed.

**C.** The analysis, design and implementation of a MOOT CASE Tool client that supports the interpretation of NDL specifications for creating and modifying methodology model diagrams. This includes the analysis of the specific requirements of the graphical user interface, and the definition of a protocol for the communication of information between the client and server CASE Tool sub-systems.

## 1.8. Structure of the Thesis

The thesis is structured into nine chapters. The thesis begins with the definition and specification of NDL, and proceeds to the analysis, design and implementation of the CASE Tool client.

Chapters Two to Four specifically cover NDL in detail. In Chapter Two an extensive analysis of existing methodology notations is performed. This culminates in the requirements definition of NDL as it will be developed in this thesis. A review of

previous research that the current research succeeds is also conducted at the end of Chapter Two. Chapter Three describes the set of basic notation primitives that can be defined in NDL. These primitives can be utilised to construct NDL templates in a notation specification, as described in Chapter Four. The design of the NDL Interpreter that is used to construct views from NDL templates is also described in Chapter Four.

Chapters Five to Eight describe the CASE Tool client. In Chapter Five an overview of the design of the client is presented, with details about how a notation specification and the NDL Interpreter are used to construct diagrams. In Chapter Six the requirements of the graphical user interface of the client are analysed and presented. This is followed by a description of the subsequent design and implementation of the graphical user interface. The communications protocol between the client and server is examined in Chapter Seven. Chapter Eight describes the eventual implementation of the prototype CASE Tool client as a platform-independent graphical user interface shell to the MOOT CASE Tool sub-system. The prototype is implemented in Java (Sun, 1998).

The conclusions that have been drawn from the application of this research are presented in Chapter 9. Proposals for future enhancements and extensions are also considered in this chapter.

The design and implementation of NDL and the MOOT CASE Tool client has been scaled down during the course of this research due to time constraints. NDL supports a minimal subset of graphical primitives (lines, arcs, and text boxes) to generate template views. This subset has been chosen as it is sufficient for constructing typical views and determining the efficacy of the proposed approach to defining the syntax of a methodology. Design and implementation of the client GUI is focused specifically toward the diagram editor that allows the basic construction and manipulation of diagrams. Supporting elements, such as project managers and advanced GUI features (eg. group selections, cut/copy/paste operations, etc) have been considered however they have yet to be incorporated into the design. Other constraints that have been imposed that relate to specific areas of the research are documented in the thesis where relevant.

# Chapter 2

# Requirements Definition for a Notation Definition Language

Each software engineering methodology supports a different set of models to describe a particular system. For example, the Coad and Yourdon OOA/OOD method (Coad and Yourdon, 1991a, 1991b) supports class diagrams, while the UML notation (Rational, 1997) supports class, use-case, sequence, collaboration, state, activity and implementation diagrams[1]. A notation is the visual syntax that is used to describe a model. This chapter examines various notations to determine the requirements for an abstract language that can be used to describe any notation. The language is called the Notation Definition Language (NDL).

## 2.1. Analysis of Notations

Models supported by a software engineering methodology are used by a software engineer to express and investigate the relevant abstractions of a problem domain. A methodology may support a range of different types of models that each express various aspects (such as entity relationship, use case, state transition, etc) of the problem domain.

A large proportion of the models supported by software engineering methodologies are expressed as pictorial graphs, as humans can generally analyse and express complex systems more easily using pictures than by purely textual means. The graphs are composed of nodes (object symbols) that are connected together by paths (connections). The object symbols represent semantic concepts of the problem domain, such as a class or process. Examples of object symbols include Coad and Yourdon's Class-&-Object, Booch's Bubble, and Jacobson's Actor. Connections express the semantic relationships between concepts, such as inheritance, message invocation, or state transition. Examples of connections include the Whole-Part relationship of Coad and Yourdon, the

[1] While methodologies describe these as "diagrams", in the context of this thesis they are termed "models". A model is a container of one or more diagrams. Collectively the diagrams describe the model.

Using relationship of Booch, and the Use Case relationship of Jacobson (Jacobson, 1994). The placement of connections is often constrained to certain parts of an object symbol. For example, in Coad and Yourdon, connections cannot be attached to the round corners of the Class and Class-&-Object object symbols. Text strings also can frequently be associated to object symbols and connections (notation elements). Properties such as the name of an object symbol or the cardinality of a connection can be described using text.

A notation is the visual syntax that is used to describe a model. A notation definition language that supports that definition of arbitrary notations must support all of the features identified in the previous paragraph. In order to determine the properties and behaviour of notation elements, various existing notations and models used in software engineering must be examined. Due to the wide range of models that exist and the time frame available, effort is focussed mainly towards the description of models depicting class diagrams. In particular, the methodology notations of Coad and Yourdon's OOA model (Coad and Yourdon, 1991a), Rumbaugh's Object Model (Rumbaugh, 1991), Booch's Object Model (Booch, 1994), and UML's Class Diagram (Fowler *et al*, 1997) are examined. During the analysis process, however, consideration is given to other notations in an effort to keep the notation definition language as generic as possible.

The analysis of notations is focused solely on the visual syntax, and not on the semantics or meaning that the models are capable of expressing. It is important to avoid a simple static view of notations during the analysis, and note that they are not restricted to simply being "pictures". For example, a software engineer may need to interact with notation elements to view or update their properties. The potential for logical distortion (the reshaping of a symbol or connection to reveal more, less, or just different properties) must also be considered. Only two-dimensional notations are considered in this analysis as three-dimensional layout and navigation is not practical on current desktop machines. The UML notation guide (Rational, 1997) stresses this limitation:

> "Note that [the] UML notation is basic[ally] 2-dimensional. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes) but they are still rendered as icons on a 2-dimensional surface. In the near future true 3-dimension layout and navigation may be possible on desktop machines but it is not currently practical."

Figure 2-1 depicts an example of a class diagram that appears in Martin and Odell (1995). The diagram models a simple customer ordering system. Three types of

Figure 2-1 depicts an example of a class diagram that appears in Martin and Odell (1995). The diagram models a simple customer ordering system. Three types of customers are represented. Customers may place orders for products. Orders are composed of order lines, each order line being associated with a single product. The model is presented in four differing notations for examination and comparison of features.

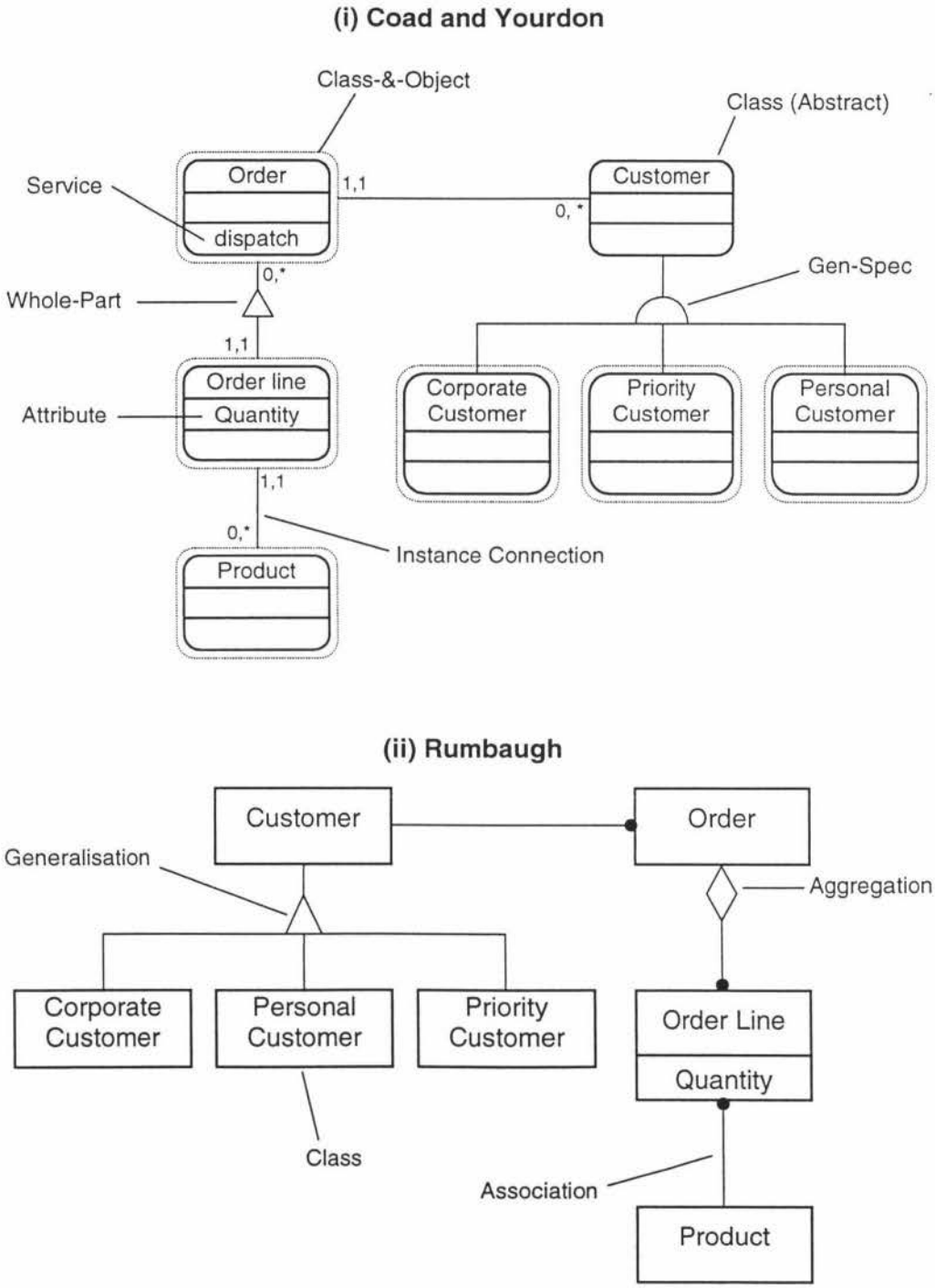



**Figure 2-1(a) – Comparative examples of notations: (i) Coad and Yourdon OOA model; (ii) Rumbaugh object model**

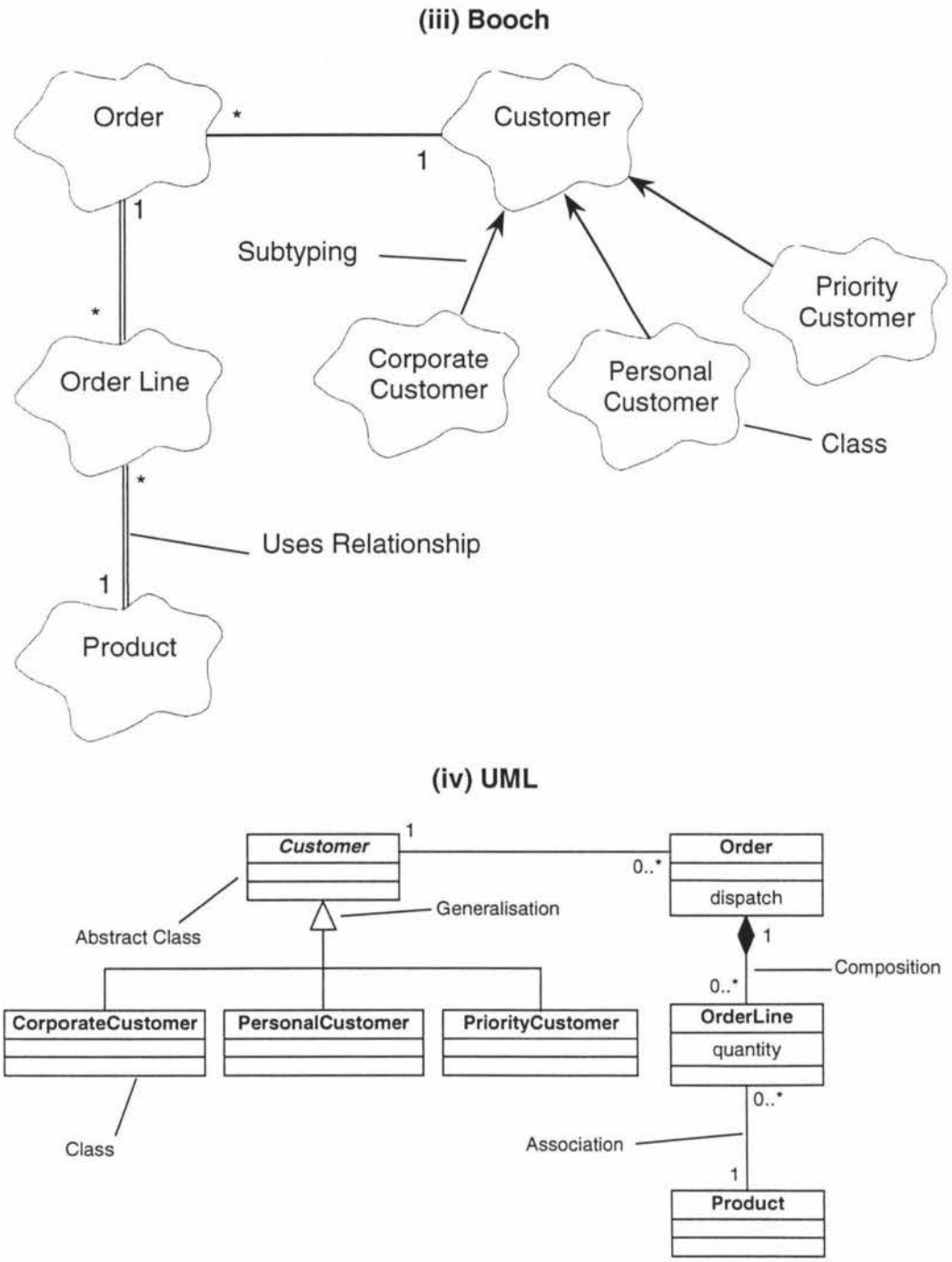



**Figure 2-1(b) – Comparative examples of notations: (iii) Booch (1991) class diagram; (iv) UML class diagram[2]**

[2] Figure 2-1(i), (ii), and (iii) are extracted from Martin and Odell (1995).

### 2.1.1. Object Symbols

Object symbols (eg. Coad and Yourdon Class-&-Object, UML Abstract Class) are composed of a number of lines and/or arcs that enclose text areas. The text areas describe properties of the concept that the object symbol represents. The height and width of object symbols may vary in order to contain the text associated with them. Object symbols may also change in size to accommodate additional information such as additional text or other symbols.

Many object symbols are divided into compartments. Each compartment generally defines different properties of the concept (although this is not a rule). For example, Figure 2-2 shows three different UML class symbols, each with three compartments. The first compartment is a single line of text that names the class. The other compartments are lists of text that describe the attributes and operations of the class.

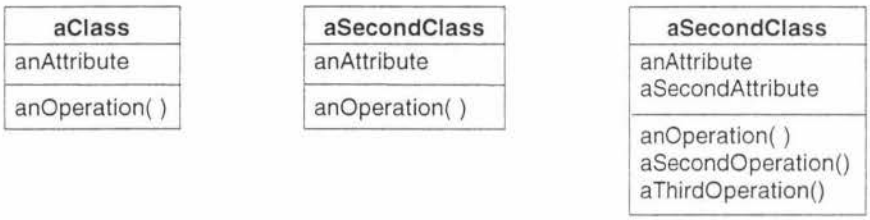



**Figure 2-2 – UML class symbols**

The overall size (width and height) of a UML class symbol is related to the size of the text that it encloses. The size and position of each compartment depends on the enclosed text and the relative size and position of the other compartments of the symbol. A UML class symbol can be described topologically as shown in Figure 2-3.

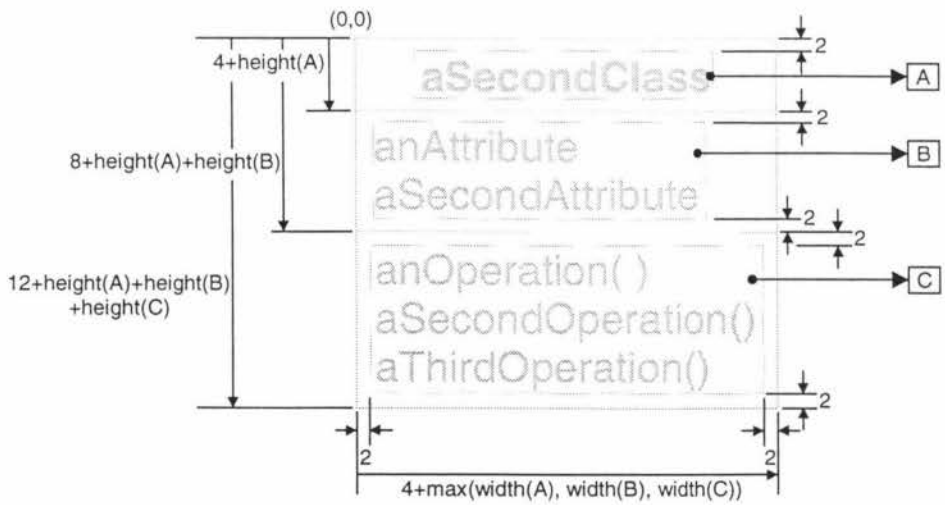



**Figure 2-3 – Topological description of a UML class symbol**

The start and end points of the various line segments that describe the shape of the class symbol are all functions of the text areas A, B, and C, which enclose the class name, attributes, and operations properties respectively. For example, the overall height of the class symbol is 12+height(A)+height(B)+height(C) drawing units. The width of the class symbol is 4+max(width(A), width(B), width(C)) drawing units.

A notation may contain a number of object symbols with common sub-components. For example, the Coad and Yourdon Class and Class-&-Object symbols shown in Figure 2-4 are distinctly similar, only the latter has an additional bounding round rectangle. Both of these object symbols could be defined in terms of the common shape (which includes the text areas). The Class-&-Object symbol need only define an additional round rectangle, the size of which would be based on the size of the common shape.


**Figure 2-4 – Common sub-components in Coad and Yourdon: (i) Class symbol; (ii) Class-&-Object symbol**

Object symbols may also be composites of other notation elements. An example is the subject area in Coad and Yourdon. A subject area may be drawn as a border around the members of the subject, or collapsed into a single object symbol that names the subject. Other similar examples include class categories (Booch) and packages (UML). Composite symbols are unique in that their size and shape depends on one or more other notation elements (each of which is dependent on its own properties) as well as properties of its own.

### 2.1.2. Connections

Connections are typically composed of simple line segments and arcs. The individual segments of a connection have no semantic significance and occur only for visual presentation. Some notations specify that connections may only exist in horizontal and vertical directions (eg. Coad and Yourdon), while others may not have any such constraints at all (eg. Booch, UML). Connections may have other symbols as annotations, such as the semicircle in the Gen-Spec relationship of Coad and Yourdon, or the filled diamond in the UML Composition relationship. Text may also be

associated with a connection to describe properties such as cardinality or role names. Connections cannot exist in isolation and must attach to object symbols at all endpoints. This definition of a connection does not constrain the manner in which they may be constructed or drawn. UML has embraced a similar understanding of a connection, as stated in the UML notation guide (Rational, 1997):

> "Paths[3] are sequences of line segments whose endpoints are attached. Conceptually a path is a single topological entity, although its segments may be manipulated graphically. A segment may not exist apart from its path. Paths are always attached to other graphic symbols at both ends (no dangling lines). Paths may have *terminators*, that is, icons that appear in some sequence on the end of the path and that qualify the meaning of the path symbol."

Furthermore:

> "[A] path may consist of one or more connected segments. The individual segments have no semantic significance but may be graphically meaningful to a tool in dragging or resizing an association symbol."

Some notations support the grouping of connections. Figure 2-5 is a fragment of Figure 2-1, showing only the inheritance connections of UML and Booch. In Figure 2-5(i), the single UML tree-like inheritance structure actually represents three conceptually separate inheritance relationships. The fact that these are indeed conceptually separate is shown more clearly by the Booch notation for the same model (Figure 2-5(ii)). Other

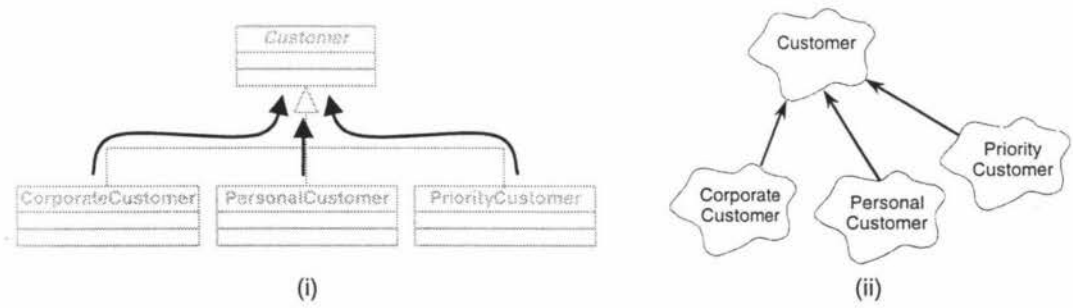



**Figure 2-5 – Conceptual relationships: (i) Grouping of three conceptually separate inheritance relationships in UML; (ii) Individually separate relationships in the Booch notation**

[3] Path is the equivalent UML term for a connection.

examples where grouping may occur include the inheritance connection in Coad and Yourdon, and the aggregation connection in UML.

The grouping of connections is only a presentation issue. This view is also supported by the UML notation guide (Rational, 1997):

> "In some relationships (such as aggregation and generalization) several paths of the same kind may connect to a single symbol. In some circumstances (described for the particular relationship) the line segments connected to the symbol can be combined into a single line segment, so that the path from that symbol branches into several paths in a kind of tree. The is purely a graphical presentation option; conceptually the individual paths are distinct."

The structure of a connection can be decomposed into several parts: a series of interconnecting line segments, an optional special connection symbol, and connection terminators (Figure 2-6). The connection symbol and the connection terminators identify the actual type of relationship. Each component of a connection may have one or more associated properties, such as a text area describing cardinality.

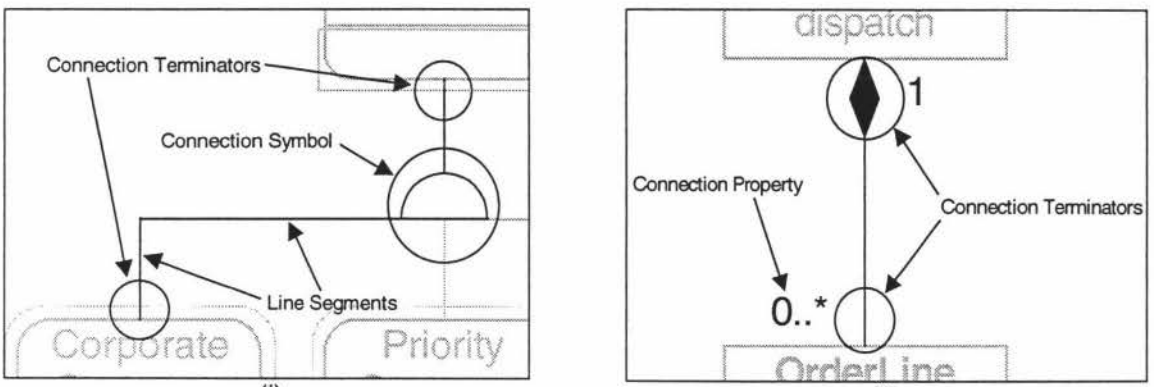



**Figure 2-6 – Composition of connections**

### 2.1.3. Docking Areas

A connection attaches to object symbols at particular connection points or *docking areas*. The UML notation guide (Rational, 1997) states that:

> "Paths are connected to two-dimension symbols by terminating the path on the boundary of the symbol."

This description of a docking area is insufficient for the specification of many other notations. For example, in the Coad and Yourdon notation, a Class-&-Object symbol is represented as a Class symbol with an additional bordering round rectangle. Whole-Part and Instance connections attach to the outer round rectangle to represent that the relationship is between instances of classes[4]. Inheritance connections attach to the inner round rectangle of the Class symbol, to represent that the relationship is between classes. None of these connections may attach to the curved parts of the symbols, and connections are always rectilinear and orthogonal. Figure 2-7 illustrates some examples of the docking areas on Coad and Yourdon object symbols.

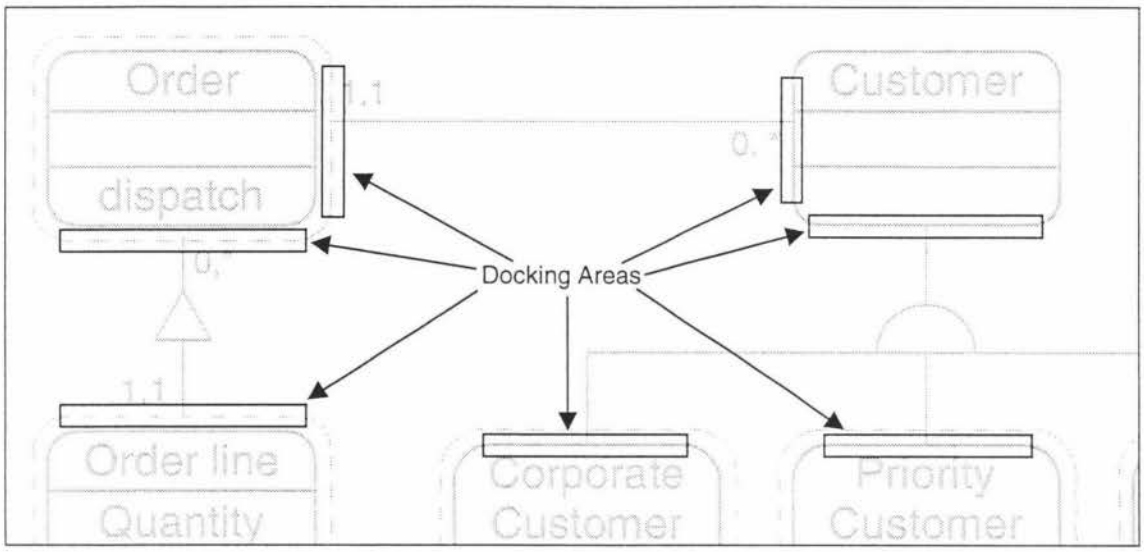



**Figure 2-7 – Docking areas on Coad and Yourdon object symbols**

Other notations may have different constraints in the construction of connections. Figure 2-8 shows an example of a UML sequence diagram that appears in the UML notation guide (Rational, 1997). Connections are generally constrained to being horizontal except for special cases such as conditional control and messages to self. The sequence diagram is an example where the coupling between the syntax and semantics is very high. The diagram represents a two-dimensional graph where the vertical axis represents time. Meaning is associated with the relative vertical positions of the message invocations. Modifying the relative positions of connections can significantly alter the meaning of the diagram. The sequence numbers appearing on message invocations indicate the chronological order of invocations. Rather than being a property of an invocation, a sequence number is more a property of relative position, as one would expect a sequence to be renumbered if the timing sequence was reordered.

[4] These connections may also adhere to class symbols that represent abstract classes. An example of this is shown in Figure 2-7.

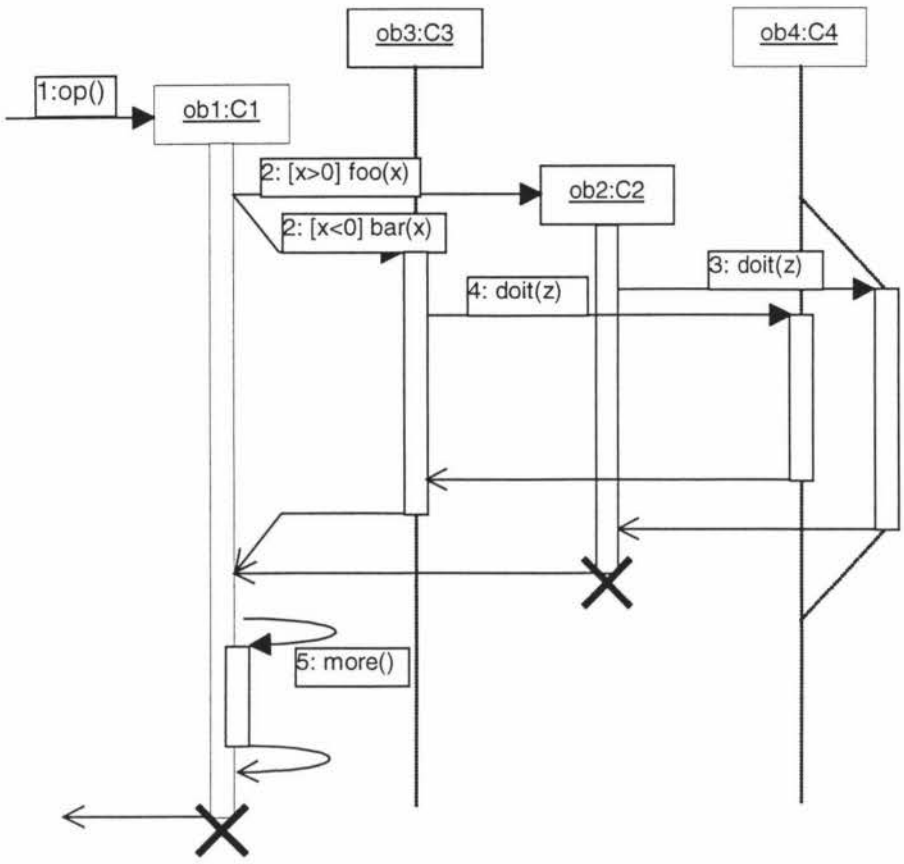



**Figure 2-8 – UML Sequence Diagram [Rational, 1997]**

### 2.1.4. Presentation Issues

A common problem associated with any computer representation of large-scale systems is the relatively small window through which details can be viewed. CASE tools, which generally provide multiple related representations of the system being developed, share this problem. The small window effect alludes to difficulties in locating a particular item (navigation), in interpreting that item once it has been located, and in relating that item to others, if that item cannot be seen in its full context (Plaisant *et al*, 1995).

To overcome some of these difficulties, a range of distortion-oriented presentation techniques has evolved (Leung and Apperley, 1993). The common feature of these techniques is to allow a user to examine a local area in detail, whilst presenting a global view of the system, to provide an overall context and to aid in subsequent navigation. There are two main categories of distortion: perspective distortion and logical distortion. Perspective distortion typically involves a bifocal display where an entire model is presented in a compressed view, and the area around a user point of focus is

enlarged to show detail[5]. Logical distortion permits unimportant portions of a diagram to be hidden from view, leaving only those portions of interest in focus (Apperley and Chester, 1996). Many current CASE tools support logical distortion by allowing portions of diagrams or symbols to be hidden, thus reducing the overall space taken up by the model. For example, Rational Rose (Rational, 1998) allows various levels of detail to be shown in the compartments of object symbols (Figure 2-9).

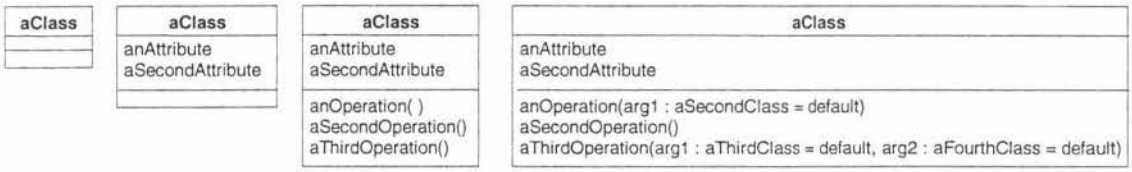



**Figure 2-9 – A UML class symbol showing various levels of detail**

The provision for logical distortion is not normally a part of the definition of a model notation. Typically these features would be dependent on the particular implementation of a CASE tool. UML is a notable exception to this precept, as it includes a description on presentation options for every notation element in the notation. The UML notation guide (Rational, 1997) discusses presentation options as follows:

> "**Presentation options**: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of showing information within a tool. Dynamic tools need the freedom to present information in various ways and we do not want to restrict this excessively. In some sense, we are defining the 'paper notation' that printed documents show, rather than the 'screen notation'. ... Note that a tool is not supposed to pick one of the presentation options and implement it; tools should give the users the options of selecting among various presentation options, including some that are not described in this document."

This explicit statement about presentation options by UML is a significant development in the evolution of CASE technology. It signals that methodologists are finally recognising the importance of human-computer interaction issues when dealing with very large-scale systems. Clearly one could not examine such a system in all its detail

[5] A discussion on perspective distortion techniques in the display of large-scale models is beyond the scope of this thesis.

at once, so the elicitation of important information, in the context of the overall system (or subsystem), is important.

## 2.2. Requirements of NDL

NDL must provide the necessary facilities to describe how notation elements may be rendered and manipulated in order to describe the types of notations examined in section 2.1. The following requirements must be supported by NDL:

1. Graphical primitives, such as lines, arcs, and text areas. These are the building blocks for describing a view of an object symbol or connection component (ie. connection symbol or connection terminator).
2. Relationships between sub-parts of a view. For example, the length of a line segment may depend on the amount of text in a text area.
3. Docking areas, for the specification of where particular types of connections may be attached to a view.
4. Support for user interactability with views. For example, a user should be able to update the text in designated text areas.
5. Logical distortion, to allow information to be expressed at various levels of detail.
6. Templates, for the grouping of the above requirements into a single unified entity that describes an object symbol or connection component.

NDL should also be efficient. This means that potentially repetitive definitions should only have to be defined once, and then reused as required. For example, if several object symbols are based around a common shape, that shape should only have to be defined once[6].

The purpose of NDL is purely to provide a syntactic representation of a notation. It is not intended and must not allow any expression of the semantics that a notation represents. The overall design philosophy of MOOT is to separate that syntax and semantics of a methodology as much as possible. NDL is intended only to define methodology syntax, while the semantics of a model are defined by SSL.

[6] The reuseability aspects of NDL comply with one of the underlying philosophies of MOOT to fully support reuse wherever possible.

## 2.3. Previous Research on an Abstract Notation Definition Language

The research undertaken toward the development of NDL, as described in this thesis, continues from the efforts of two other researches. The specification for a language that supports the abstract definition of graphical notations, and the human-computer interaction with them, was originally developed by Clark (Clark, 1994, also described in Page *et al*, 1994). Subsequent to this, the language was revised and restructured by Page (Page, to appear) as part of the research toward methodology engineering. The goal of this research was the separation of syntax and semantics of a methodology description. Sections 2.3.1 through 2.3.5 critically examine the previous research on developing an abstract notation definition language performed by Clark and Page, beginning with a brief overview. The deficiencies that were identified and addressed in the development of NDL as it appears in this thesis are described. Modifications, extensions, and improvements to the initial prototype built by the previous researchers in 1995 are presented.

### 2.3.1. Overview

The research undertaken by Clark (as described in Clark, 1994) was the first stage in the development of an intelligent methodology-independent OO CASE tool (MOOT) (Mehandjiska *et al*, 1994). Clark's overall focus was on the development of a customisable graphical user interface that utilised an abstract notation definition language. At that time, the architecture of MOOT was remarkably different to its present form (described in Page *et al*, 1997). In particular, the specification languages SSL and NDL had not been defined in concept as a means to separate the semantics and syntax (respectively) of a software development methodology.

The language developed by Clark was a "first attempt" at providing an abstract description of graphical notations. The approach taken was very broad, in that Clark implemented many features that only served to distract from research into the more important aspects of the language. For example, extensive support was provided for line and text colours, text styles, fills, fill patterns, and so on. These features simply affect the presentation of notation elements in a diagram, and do not focus on other more important requirements of a notation definition language. The implementation of connections in Clark's research, for example, and how they may interact with object symbols, is rather obtuse. A particularly noticeable flaw is that Clark does not appear to have performed any significant analysis of existing methodology notations in order to

determine what the particular requirements of an abstract notation definition language are.

The research performed by Page on a notation definition language begun to address some of the problems identified in Clark's approach. Significant changes were made to the structure and syntax of the language. Changes were also made to the terminology that had been used to describe some of the language aspects. Features such as colour and text formatting were removed by Page as a result of an analysis of existing notations focussing attention to more important areas of the language development. Support for an abstract syntax tree representation of a notation specification was developed. This involved a redesign of the class hierarchies developed by Clark that represented a notation specification. Improvements were made to the template definitions of object symbols, after which the research for this thesis commenced, and Page abandoned further effort in the development of that language. The subsequent improvement in the description of connections, and intelligent support for them in a graphical user interface, had particular emphasis during the course of the current author's research.

### 2.3.2. Basic Language Philosophy

The basic philosophy behind Clark's implementation of a notation definition language was that a visual construction tool would typically be used to generate a notation specification. Such a tool (for example, Ham, 1994) would allow the abstract definition of notation elements, and the automatic generation of the equivalent notation language specification would follow. As a consequence, references made in a notation specification to other parts of the specification (such as group templates, text areas, common expressions, etc) were achieved via numerical IDs. The IDs were sequential, and based on the relative order of definition.

The improvements made by Page and the current author were based around the assumption that no new or updated visual construction tool was available (as was the case). In consideration that notation specifications would be coded by hand during the research and development process, additional effort was put into improving the referencing method of Clark to be more forgiving of accidental errors. The numerical IDs, which have no real meaning to a human coding a specification, and hence could be easily confused, were replaced with textual identifiers that could express meaning. The syntax and structure of a notation specification were also improved to enhance human readability and interpretation. It was considered important that a notation specification

should be clear in intent to its author (or others examining it), and not be cluttered with complex statements or confusing symbols.

### 2.3.3. Definition of Language Primitives

Clark had identified that common expressions in a template definition could be factored out so that they were only evaluated once. A basic set of graphical primitives (lines, arcs, and text areas) had been defined by Clark, as had the concept of active areas supporting user interaction with template views. The concept of docking areas for connections had been also been identified and investigated very briefly, but not implemented. Page had further investigated the need for docking areas, however the current author's research, which continued Page's efforts, was begun before a concrete design approach and implementation of connections, and their association with object symbols, had commenced.

An examination of the previous researchers' efforts in the description of the basic notation language primitives, and the areas that were identified for improvement during the course of the research described in this thesis, are described in the following sections.

**Expressions**

Clark supported the basic set of mathematical operations (eg. addition, subtraction, etc), a set of functions that operated on text areas (text-width and text-height), and a set of functions that operated on other expressions (minimum and maximum). The order of expression definitions was infix, so parentheses were supported in a definition to force evaluation precedence where necessary. While support was provided in a notation specification for the definition of common groups, no support for the calculation of the size of individual groups was provided. This made it impossible to define the position of primitives in a template relative to the size of any embedded groups.

Page addressed this problem by introducing a new set of functions that operated on groups (group-width and group-height). Syntactic changes to the grammar were also performed by Page to improve readability. The class hierarchy that represented the different types of expressions was completely remodelled to better represent the inheritance relationships between the different types of expressions and functions.

Further improvements to be made in the current research include the introduction of Reverse Polish Notation in the syntax of expressions. This eliminates the need for

explicit statements of evaluation precedence, and hence makes it easier for a notation parser to construct expression trees from a specification. The expression class hierarchy developed by Page also needs extensions to include support for references to previously defined expressions. Up to this point, common expressions that had been factored out of other expressions in a notation specification were substituted back in to those expressions in the abstract syntax tree. This removed the benefit that they were originally designed for (ie. to eliminate repetitive calculations).

**Graphical Components**

Clark defined the basic set of graphical components that still currently exists. These are lines, arcs, single-line text areas, multi-line text areas, and group references (for the reuse of previous group definitions). All the facilities for colour, line styles, text fonts, text styles, fills, and fill patterns have been eliminated, as these are features that need only be made once an ideal notation definition language has been developed. Other features that have been removed in the interest of simplicity include the minimum or maximum size restrictions placed on text areas. Text alignment (ie. left, centre, or right alignment), originally supported by Clark as a property of a text area, can be supported indirectly through positioning the text area with appropriate expressions. This simplifies further the definition of text areas. To reduce the possibility of human error during the scripting of a notation specification, text areas are also now identified by textual names, rather than numerical IDs which can be easily confused.

The mechanism for using groups in template definitions is flawed in both Clark's and Page's implementation. A necessary level of indirection is missing between the functions that can be applied to a group, and the definition of the group itself. This indirection is necessary as different groups may be used in the construction of a view at different times. Group functions needs to be applied dynamically to the group that is currently being used to construct the view, rather than statically to one particular group definition (as is implemented by Clark and Page). The indirect link, a *group reference*, can be used to supply the name of a group that is currently being used in a view to a group function.

**Active Areas**

Clark supported active areas (called regions in his implementation). Many different actions were apparently defined in Clark's implementation (the exact number is unclear), including actions that activated pop-up menus, a change of drawing window, the opening of dialog boxes, and so on. This was too many, and much too complicated, when the emphasis behind the user interface was to keep it as simple as practical. Page

recognised this, and eliminated the unnecessary actions, leaving only two actions in his implementation (text area updates and group transitions). The direction of the current research does not add or remove further actions from Page's implementation.

**Docking Areas**

The implementation of docking areas as a notation primitive is a new inception in the evolution of the notation definition language. Clark had identified where docking areas may have been of use, but had not extensively investigated their application. Instead, Clark attached connections to the border of object symbols "plus or minus" some value. While this allowed for connections that may attach to points inside (or outside, as a matter of point) an object symbol, it did not constrain exactly at which locations around an object symbol connections could be made.

Page's research on the notation definition language had not taken him as far as seriously considering docking areas before the current author took over his research in this area. As was seen in the analysis of methodology notations (section 2.1), the concept of docking areas is an absolute necessity for describing where connections may attach to object symbols.

### 2.3.4. Definition of Language Templates

Both Clark and Page had identified three types of template for the construction of views: object templates, connection templates, and composite templates. The basic definition and structure of object templates has remained static in the evolution of the notation definition language. Composite templates were identified by both Clark and Page, however their implementation was considered beyond the scope of their research. This view is continued in this thesis, in consideration of the time frame available and the low priority given to a full investigation into the support for composite symbols.

The analysis of notations performed in section 2.1 identified a connection as being composed of several parts: a series of line segments, an optional connection symbol, and connection terminators. Connection symbols and connection terminators appear to each require their own template definition, as they may be complex in their description (and/or behaviour). This is remarkably different to the method used by Clark to describe connections. Clark described connections using a single connection template. Connections consisted of one or more line segments that could be annotated with other graphical components. These annotations were positioned on a connection by various means such as a percentage distance along the connection, or other relative offsets. In

practice, it seemed very difficult to understand where an annotation might end up when it was placed at "Start plus 50 percent minus 10 offset –10"[7]. Each definition of a connection also indicated what object symbols it could attach to. Rather obscure was that a connection could be defined to have a completely different appearance depending on the combination of object symbols it was attached between. This ability is rather unnecessary, as a simple analysis reveals that methodology notations typically define a particular type of connection to always appear one way only.

Connection types other than simple binary connections were not supported in Clark's implementation of connections. Ternary or other higher-order relationships could not be defined, and the grouping of multiple connections was not supported either. The provision of connection symbol templates in the notation definition language described in this thesis has the potential to overcome these limitations, as grouping and higher-order relationships are often centred about a connection symbol.

The separating of connections into individual line segments and annotation templates can also improve the user interaction possible with a connection. Clark's approach appeared to consider a connection as a single manipulable entity. This became obvious while examining his investigation into the support for the grouping of connections. The design supported by the analysis performed in section 2.1 can regard each line segment and annotation of a connection as an individually selectable and manipulable entity in a graphical user interface. Supporting this means that an extra responsibility of the graphical user interface to ensure that a connection always remained "interconnected" along its length would be required.

### 2.3.5. Notation-Semantic Mapping

A deficiency found in previous research was that no serious investigation was undertaken into how a complete diagram could be mapped onto its underlying semantics. A diagram, described by a collection of interrelated notation elements, has an underlying semantic description that identifies what the diagram actually means. Support for the mapping between the contents of a diagram and its underlying semantics stored in the CASE Tool server needed to be addressed during the course of research for this thesis. Included with this was the identification of the client-server communication necessary to convey important semantic changes made in a diagram to the server.

[7] This example actually appears in Clark, 1994.

# Chapter 3

# NOTATION DEFINITION LANGUAGE PRIMITIVES

The notation definition language (called NDL) that has been developed for this thesis prescribes a set of primitive building blocks or *segment templates*. Segment templates are used to describe how the notation elements that describe a notation may be rendered and manipulated. Segment templates are blueprints for creating segments. Segments are instantiations of segment templates in a particular context. Segment template primitives are grouped into the following categories: graphical components, groups, active areas, and docking areas. Each segment template is defined in NDL by a series of expressions that determine the template's position and shape.

This chapter describes the primitives that NDL supports, along with a formal language grammar for their specification[1]. The grammar is pseudo-English in style. NDL has been designed to allow automatic generation by visual programming tools while still being readable in printed form. At this time no visual construction tool such as that developed by Ham (1994) has been developed to accommodate the new language features described in the thesis. Notation definitions must currently be coded by hand by a developer familiar with the language.

Class hierarchies that model the NDL primitives are also presented (using the UML class diagram notation). Objects of these classes are instantiated to contribute towards the construction of an abstract syntax tree for a notation.

## 3.1. Describing Notation Elements in NDL

Each notation element is constructed from a set of graphical components. A topological description of a UML class symbol is shown in Figure 3-1. The class symbol is composed of a number of line segments that together describe three compartments. Each compartment contains an area (*classname*, *attributes*, and *operations*) where text

[1] The grammar is expressed in Extended Backus-Nauer Form (EBNF). For an explanation of EBNF, refer to Appendix A. For a complete list of the NDL grammar, refer to Appendix B.

may appear. The height and width of these text areas depend on the text they hold, and are a property of the text area. The height and width of the class symbol and the positioning of the compartment dividing line segments are dependent on the height and width of the text areas.

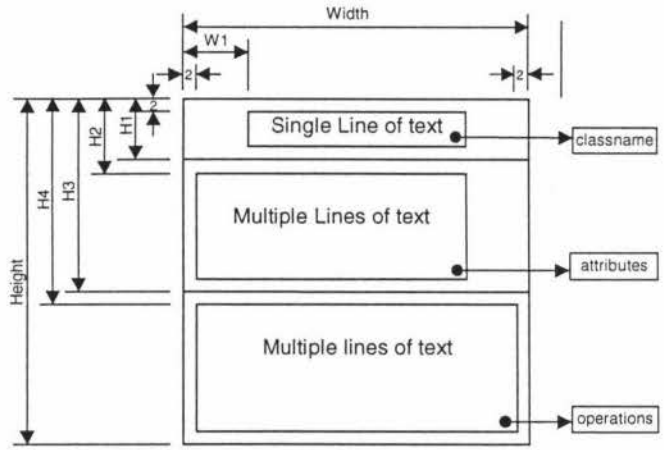



**Figure 3-1 – Topological description of a UML class symbol**

Various dimensions that denote the size and positioning of the lines and text areas are drawn in the figure (eg. H1, H2, Height, Width). These dimensions are defined in NDL by equations. Equations are composed of various types of expressions. NDL provides a range of expression types which fully support dynamically resizable notation elements. Equations and expressions are described in section 3.2. The graphical components that compose notation elements are described in section 3.3.

Some notation elements may contain common sub-components. For example, in Figure 3-2(i) the Coad and Yourdon Class and Class-&-Object symbols are distinctively similar in appearance, only the latter has an additional bounding round rectangle. Similarly, the Booch Class, Parameterised Class and Instantiated Class shown in Figure 3-2(ii) all have the Booch bubble in common. To reflect the existence of common sub-components and to support reuse, NDL supports the composition of a group of segment templates into a single entity called a group template. Group templates can be reused as necessary in the guise of a primitive component. Section 3.4 discusses group templates in detail.

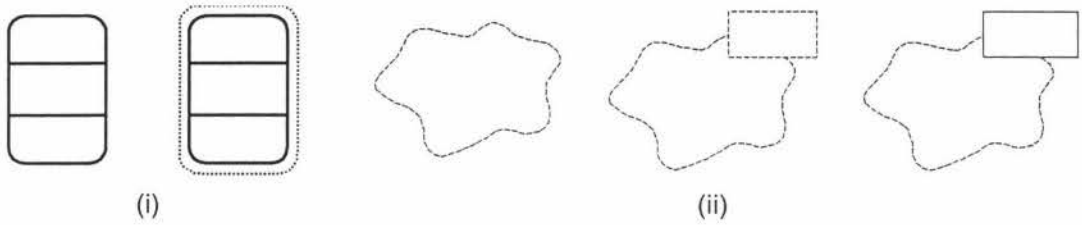



**Figure 3-2 – Common sub-components: (i) Coad and Yourdon Class and Class-&-Object; (ii) Booch Class, Parameterised Class, and Instantiated Class**

A notation element may also be adorned with a number of active areas and docking areas. Figure 3-3 illustrates extended topological descriptions of the UML class symbol that was shown in Figure 3-1, showing active areas and docking areas. Active areas are described in section 3.5. Docking areas are described in section 3.6.

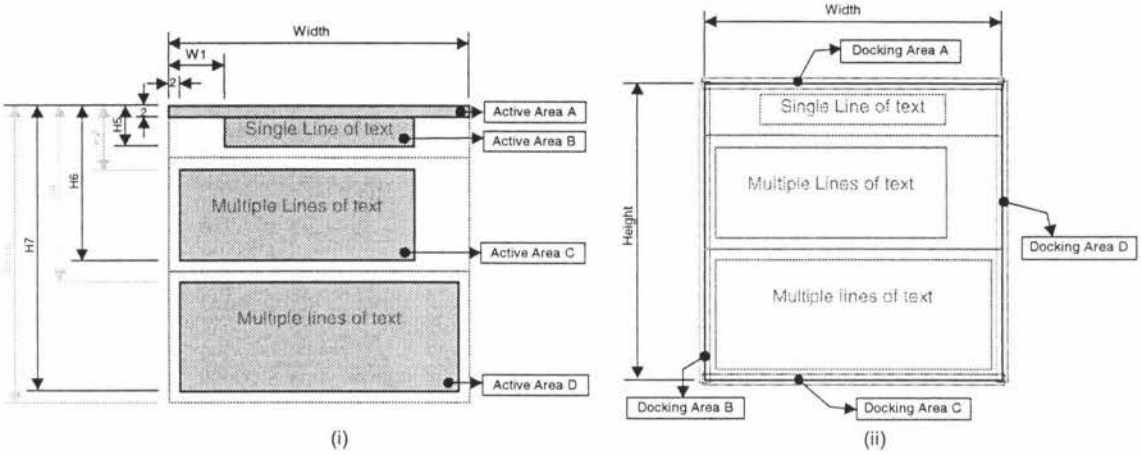



**Figure 3-3 – A UML class symbol with (i) active areas and (ii) docking areas**

All notation elements have a finite size. For example, the UML class symbol illustrated in Figure 3-1 is defined in size by the expressions Width and Height. The width and height of a notation element define the size of its bounding region. The bounding region of a notation element represents the smallest area that a view of the element will occupy. Bounding regions are discussed in section 3.7.

## *3.2. Equations and Expressions*

NDL allows for the declaration of equations or predefined expressions that can be used in the later construction of segment templates. The primary use for these declarations is to avoid unnecessary repetitive computation. A group of expressions in a template definition may contain common sub-expressions. By declaring the common sub-expressions as separate equations, the sub-expressions are evaluated only once, rather

than for each time they would have appeared. Equations can be composed with the following types of expressions:

- numerical constants;
- equation reference;
- compound expressions;
- text-related functions;
- group-related functions;
- expression-related functions.

*Numerical constants* are the simplest expression type. Both signed integers and floating point numbers are supported in NDL, although integers are type converted to floating point numbers during the evaluation of an expression. NDL specifies that integers are 32 bits wide, giving a valid range of approximately ±2 billion. Floating-point numbers are double precision, 64-bit quantities (following the IEEE 754 specification), with a range of approximately ±1.8x10±308 and an accuracy of about 12 decimal places.

*Equation references* are used to include the value obtained from a previously defined equation into the current expression. Referenced equations are only re-evaluated if the values that they depend on have changed. Defining a common sub-expression as a separate equation and referencing it can reduce the overall computational time of other equations that depend on the value of the sub-expression.

A *compound expression* is the application of the addition, subtraction, multiplication, or division operators to two sub-expressions. The usual mathematical symbols +, -, *, and / are used respectively. Compound expressions are written in Reverse Polish Notation, eliminating the need for operator precedence.

Two *text-related functions* have been identified: text width and text height. These functions are used to calculate the size of a text area. The size of a text area depends on the text string(s) contained in it and the particular context that the text is viewed in (eg. font, font size, etc). Therefore these functions can only be evaluated at run-time.

*Group–related functions* are similar in nature to the text-related functions, with the difference that they operate on an entire group template. Only two functions have been identified: group width and group height. These functions are also evaluated at run-time, as the dimensions of a group may not be constant.

*Expression-related functions* provide for the application of a function to one or more expressions. Currently only two functions have been implemented: maximum and minimum. Many other functions could be included, such as the trigonometric functions sine and cosine, however their use is outside the scope of this thesis.

Expressions are written in NDL using the following EBNF grammar:

```
<Expression>              ::= '+' <Expression> <Expression> |
                              '-' <Expression> <Expression> |
                              '*' <Expression> <Expression> |
                              '/' <Expression> <Expression> |
                              <Term>
<Term>                    ::= <Number> | <Identifier> | <Function>

<Function>                ::= <Text_Function> | <Group_Function> |
                              <Expression_Function>
<Text_Function>           ::= <Text_Width_Function> | <Text_Height_Function>
<Group_Function>          ::= <Group_Width_Function> | <Group_Height_Function>
<Expression_Function>     ::= <Max_Function> | <Min_Function>

<Text_Width_Function>     ::= TEXTWIDTH '(' <Identifier> ')'
<Text_Height_Function>    ::= TEXTHEIGHT '(' <Identifier> ')'
<Group_Height_Function>   ::= GROUPHEIGHT '(' <Identifier> ')'
<Group_Width_Function>    ::= GROUPWIDTH '(' <Identifier> ')'
<Max_Function>            ::= MAX '(' <Expression> { ',' <Expression> } ')'
<Min_Function>            ::= MIN '(' <Expression> { ',' <Expression> } ')'
```

The use of Reverse Polish Notation needs attention when using a minus sign in front of numerical constants when authoring expressions. Depending on the context a minus sign appears in, it can have two meanings. A minus sign placed flush in front of a number is interpreted by the notation parser as negating that number. However, if any white space appears between the minus sign and the number, the sign is interpreted as the beginning of a compound expression representing subtraction. Lack of attention to this detail will cause either syntax errors during parsing (invalid expression structure), or erroneous results to be calculated (valid structure, but incorrect meaning).

An NDL equation is specified by the assignment of an expression to a variable identifier:

```
<Assignment>          ::= <Identifier> '=' <Expression> ';'
```

Equations are grouped together in the template definition of notation elements:

```
<Equations>           ::= EQUATIONS '{' { <Assignment> } '}'
```

Some examples illustrating the use of equations and expressions follow. The equations are based on the values of the dimensions of the UML class symbol shown in Figure 3-1.

```
EQUATIONS {
    H1 = + 4 TEXTHEIGHT(classname);
    H2 = + 2 H1;
    H3 = + 2 + H2 TEXTHEIGHT(attributes);
    ...
    Width = + 4 MAX(TEXTWIDTH(classname), TEXTWIDTH(attributes),
                TEXTWIDTH(operations));
    W1 = / - Width TEXTWIDTH(classname) 2;
}
```

## 3.3. Graphical Components

The physical appearance of a notation element is described in NDL by a number of primitive graphical component segment templates. Four of these primitives have been identified: lines, arcs, single-line text areas, and multi-line text areas. A fifth graphical component segment template that has been identified is a group reference. A group reference allows a predefined group template to be included as a component of the current notation element. Although a group reference is not strictly a primitive component, it is treated as such for the purposes of describing the shape of a notation element. Group reference segment templates are described further in section 3.4.

> The graphical component segment templates are the only type of segment templates that produce a visible image on the display.

The graphical component segment templates are grouped into an inheritance class hierarchy and are included in the abstract syntax tree of a notation. This hierarchy is itself a part of a larger class hierarchy that describes all segment templates. The class Segment Template is the superclass to all segment templates and is placed at the top of the hierarchy. Figure 3-4 shows the branch of the Segment Template class hierarchy related to graphical components. Each graphical component segment template is described by a number of expressions that define its shape and position.

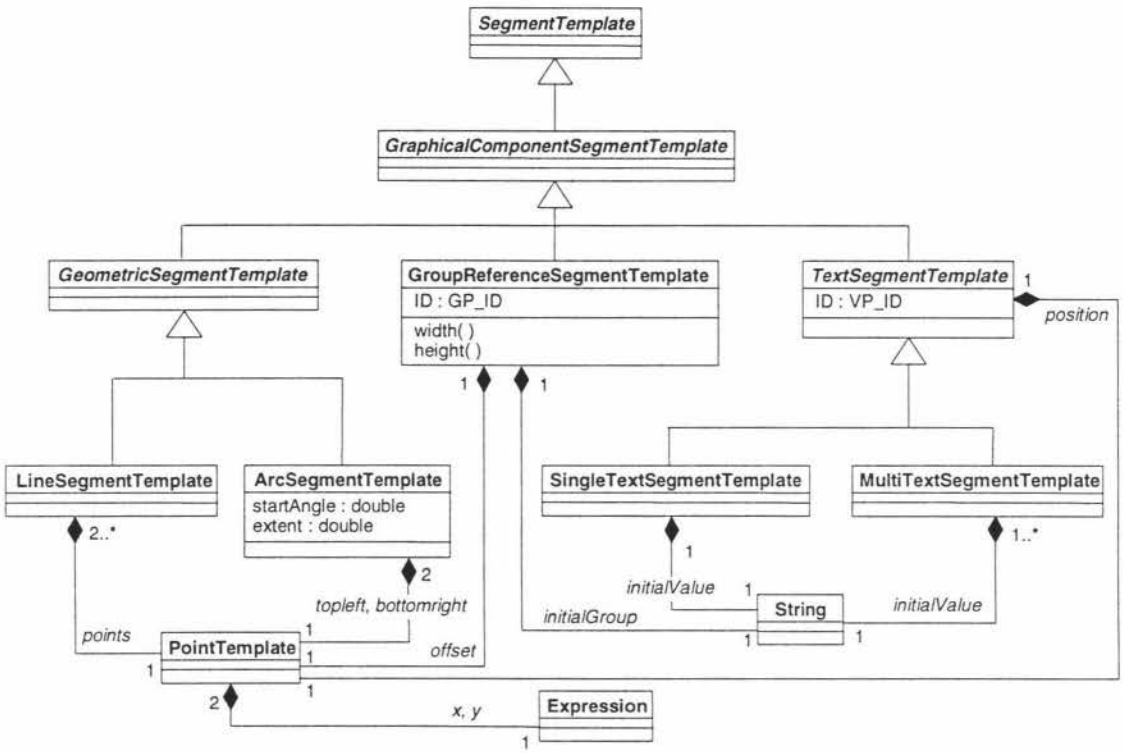



**Figure 3-4 – The Graphical Component class hierarchy**

Within the NDL specification of a notation element, graphical component segment templates are grouped together into a single section, given by the grammar:

<Graphic_Components> ::= COMPONENTS '{' { <Component_Template> } '}'
<Component_Template> ::= <Line_Template> | <Arc_Template> |
<Single_Text_Template> | <Multi_Text_Template> |
<Group_Reference>

### Line Segment Templates

A line segment template is used to describe a series of one or more connected line segments. The segment template specifies two or more points that are joined together by straight lines in a consecutive manner. The NDL grammar necessary to define line segment templates follows:

```
<Line_Template>    ::= LINE <Point> TO <Point> { TO <Point> } ';'
<Point>            ::= '[' <Expression> ',' <Expression> ']'
```

An example of the use of line segment templates follows. The example draws the basic shape of the UML class symbol shown in Figure 3-1.

```
COMPONENTS {
    ...
    LINE [0, 0] TO [Width, 0] TO [Width, Height] TO [0, Height] TO [0, 0];
    LINE [0, H1] TO [Width, H1];
    LINE [0, H3] TO [Width, H3];
    ...
}
```

### Arc Segment Templates

Arc segment templates can be used to describe anything from a small arc through to a complete circle or ellipse. An arc is specified by two angles called the *start angle* and *extent*. The start angle identifies the angle where drawing begins, while the extent signifies the angle that the arc subtends. Angle measurement starts from the positive-$x$ axis, or the eastward direction, and is measured counter-clockwise. Both measurements have a valid range from zero to 360 degrees inclusive. The start angle of an arc may also be expressed as one of the eight textual compass directions (*north*, *south*, etc). Figure 3-5 illustrates an example of the calculation of the start angle and extent of an arc.

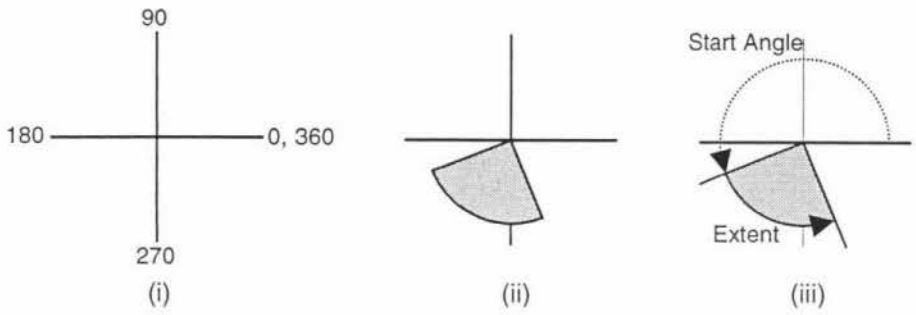



**Figure 3-5 – Calculation of an arc's start angle and extent: (i) the measurement axes; (ii) an arc; (iii) the start angle and extent**

Arcs are further specified by a bounding rectangle that would encompass the entire circle or ellipse. The aspect ratio of the bounding rectangle governs whether the final arc is circular or elliptical in shape. Figure 3-6 illustrates some examples of arcs that can be described. In this figure, the short lines crossing the bounding rectangle signify the start and end points of the arcs.

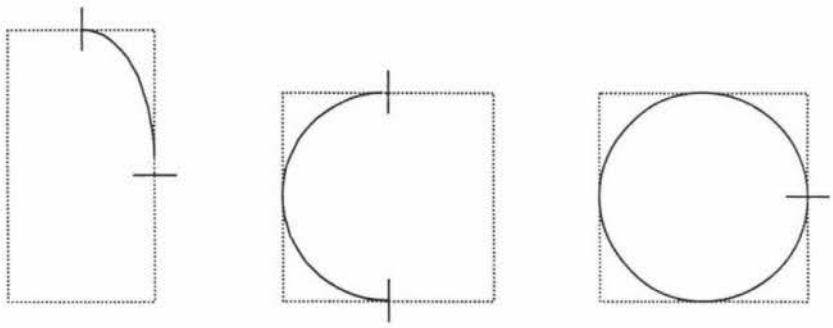



**Figure 3-6 – Arcs defined by bounding rectangle, start angle, and extent**

The grammar for arc segment templates follows:

```
<Arc_Template>        ::= ARC IN BOX <Point> TO <Point>
                          STARTING <Direction> EXTENDING <Number> ';'
<Point>               ::= '[' <Expression> ',' <Expression> ']'
<Direction>           ::= <Compass_Direction> | <Number>
<Compass_Direction>   ::= NORTH | SOUTH | EAST | WEST | NORTHEAST |
                          NORTHWEST | SOUTHEAST | SOUTHWEST
```

The following example illustrates the use of an arc segment template in the construction of the Coad and Yourdon inheritance connection symbol shown here:

The connection symbol is composed of an arc and a line:

```
COMPONENTS {
    ...
    ARC IN BOX [0, 0] TO [40, 40] STARTING 0 EXTENDING 180;
    LINE [0, 20] TO [40, 20];
}
```

**Single-Text and Multi-Text Segment Templates**

The single-text and multi-text segment template types are used for the rendering of text strings on the display. The single-text segment template allows for the specification of a single line of text, eg. the name of a class. The multi-text template is a variation that allows a vertical list of strings to be specified, useful for example for defining attribute or operation fields in a class symbol. There is no restriction on the length of any individual string.

The specification of a text segment template provides the segment template with an identifying name. This identifier is used by other segment template definitions to reference the text area. Text segment templates are positioned by a pair of ordinates representing the top-left corner of a rectangle that would encompass the text.

> Text areas do not conform to any size restrictions, and will grow or shrink as appropriate to maintain the minimum size necessary to fully contain the text they hold.

Text segment templates can be initialised to a default value, if required, by specifying this value after the usual definition of the segment template. For multi-text segment templates, a list of values is permitted, and each value will appear on a new line within the text area. No limit is placed on the maximum number of lines that can be specified for multi-line text areas.

The grammar for single-text and multi-text segment templates follows:

```
<Single_Text_Template>  ::= SINGLETEXT <Identifier> <Point> [ <String> ] ';'
<Multi_Text_Template>   ::= MULTITEXT <Identifier> <Point>
                            [ <String> { ',' <String> } ] ';'
<Point>                 ::= '[' <Expression> ',' <Expression> ']'
```

The following example describes the text areas present in the UML class symbol shown in Figure 3-1.

```
COMPONENTS {
    ...
    SINGLETEXT classname [W1, 2];
    MULTITEXT attributes [2, H2];
    MULTITEXT operations [2, H4];
    ...
}
```

## 3.4. Group Templates

NDL supports reuse of a group of common segment templates via group templates. The group template aims to reduce the redundancy that is created through repeatedly describing a collection of segment templates. The segment templates can be described once, defined as a group template, then reused as many times as required in the definition of other notation elements. Group templates may also be defined in terms of other group templates. This allows specifications to be progressively built to enhance the appearance or functionality of smaller components. A group template is referenced in a group reference segment template by an identifying name that is given to the group template when it is defined in the notation specification. This name must be unique within the scope of the specification.

Group templates are regarded as graphical component type when used in the specification of other notation elements, primarily as they are a container of other graphical components. While the most common use of group templates is to describe common graphical shapes, a group template may also adorn the shape it describes with active area segment templates. Active area segment templates permit the binding of actions to particular areas of the shape. The shapes described by group templates also have a finite size. A group template defines its size in terms of a bounding region.

Group templates are defined by the class Group Segment Template, which is a subclass of the Segment Template class, as shown in Figure 3-7. The Group Segment Template class is a container of common expressions, graphical components, active areas, and a bounding region. The equations are a set of common expressions that may be used in the definition of any of the graphical components, docking areas, or bounding region specified by the group template.

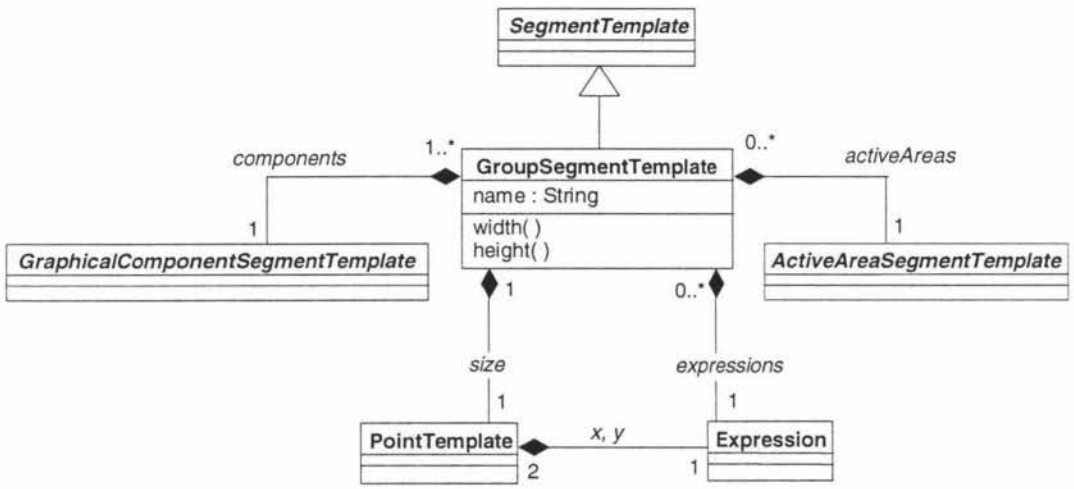



**Figure 3-7 – The Group Segment Template class hierarchy**

Group templates are defined by the following grammar:

```
<Group_Template>      ::= GROUP <Identifier> '{'
                              [ <Equations> ] <Components>
                              [ <Active_Areas> ] <Bounding_Region>
                          '}'
```

The identifier names the group template. Equations and components have been described in sections 3.2 and 3.3 respectively. Active areas and the bounding region are described in sections 3.5 and 3.7 respectively.

### Group Reference Segment Templates

A group reference segment template supports the reuse of group templates by permitting a group template to exist as a component of another group template or notation element. Group templates may be referenced in this manner as many times as required. Group reference segment templates appear in the segment template class hierarchy as a subclass of the Graphical Component Segment Template class, as shown in Figure 3-4.

A group reference component is described by the following grammar:

```
<Group_Reference>    ::= GROUP <Identifier> CONSISTS_OF <Identifier>
                         AT <Point> ';'
```

A group reference specification begins with an identifier that provides a name or alias for the group that is to be referenced. Group-related function expressions (group-width and group-height) use this alias to identify a particular view of a group template and its run-time properties. The alias must be unique within the scope of the notation element that contains the group reference.

Following the alias, a group reference statement identifies the group template that will be used as a component of the notation element currently being defined. A point coordinate specifies the position within the notation element where drawing of the group will begin.

## 3.5. Active Areas

Active area segment templates are adornments to notation elements that specify the regions that a user may select to cause particular actions. In the implementation of the current research prototype, NDL restricts active areas to be rectangular in shape. In future, the prototype can be extended to allow the specification any arbitrary shape for active areas. Each active area segment template defines a rectangular region, specified by the coordinates of its top-left and bottom-right corners. The segment template associates an action with this region that will be executed when the region is selected. Two actions have been identified for implementation in the prototype: text-area updates and group transitions. Active area segment templates have been defined as a class representation included as another branch of the Segment Template hierarchy, as shown in Figure 3-8. The attributes for each leaf class are described in detail later in this section.

Active area segment templates are defined in an NDL specification using the following grammar:

```
<Active_Area>    ::= <Point> TO <Point> <Action> ';'
<Point>          ::= '[' <Expression> ',' <Expression> ']'
<Action>         ::= <Update_Action> | <Transition_Action>
```

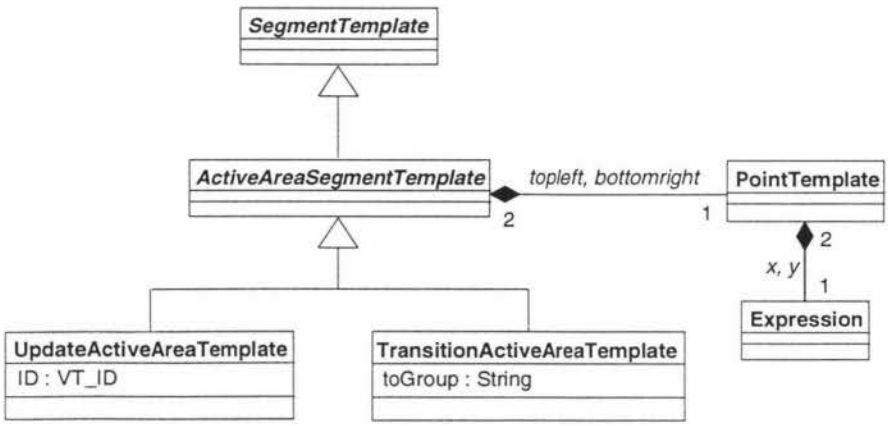



**Figure 3-8 – The Active Area Segment Template hierarchy**

Active areas are grouped in a template definition:

<Active_Areas> ::= ACTIVE_AREAS '{' { <Active_Area> } '}'

Active areas do not produce an image on the display. Usually an active area will be placed on the display to overlay one or more graphical components of a notation element so that the user is made aware of their existence.

**Text Area Updates**

Active areas associated with a text-area-update action are parameterised with an identifier naming the text area that will be updated when the action is executed. The only valid parameters are text areas that have been directly defined as graphical components of the current notation element. This definition excludes text areas that may be defined within any group templates used by the notation element. Performance of a text-area-update action will typically activate a text editor for the text area, allowing the user to change the text as necessary. Text-area-update actions are expressed in NDL by the grammar:

<Update_Action> ::= UPDATE <Identifier>

When specifying a region that will be associated with a text-area-update action, it is a general requirement that the position and dimensions of the region exactly overlay the text area with which the action is associated. This necessity stems from the implementation of text-area-update action event handling in the graphical user interface. The interface uses the position of a mouse click in a text-area-update active area to calculate where text editing should begin in the associated text area. The calculation is based on the relative position of the mouse click to the active area, rather than to the

relative position of the mouse click to the text area. Therefore, if the top-left coordinates of the two areas differ, the text-editing cursor will not be activated in the expected place. Figure 3-9 illustrates the association between a text-area-update active area and its underlying text area, and the steps followed when a text-area-update action is initiated.

Through being a requirement, this particular characteristic of text-area-update active areas lends itself nicely to their automatic generation in a tool that would allow the visual construction of graphical templates.

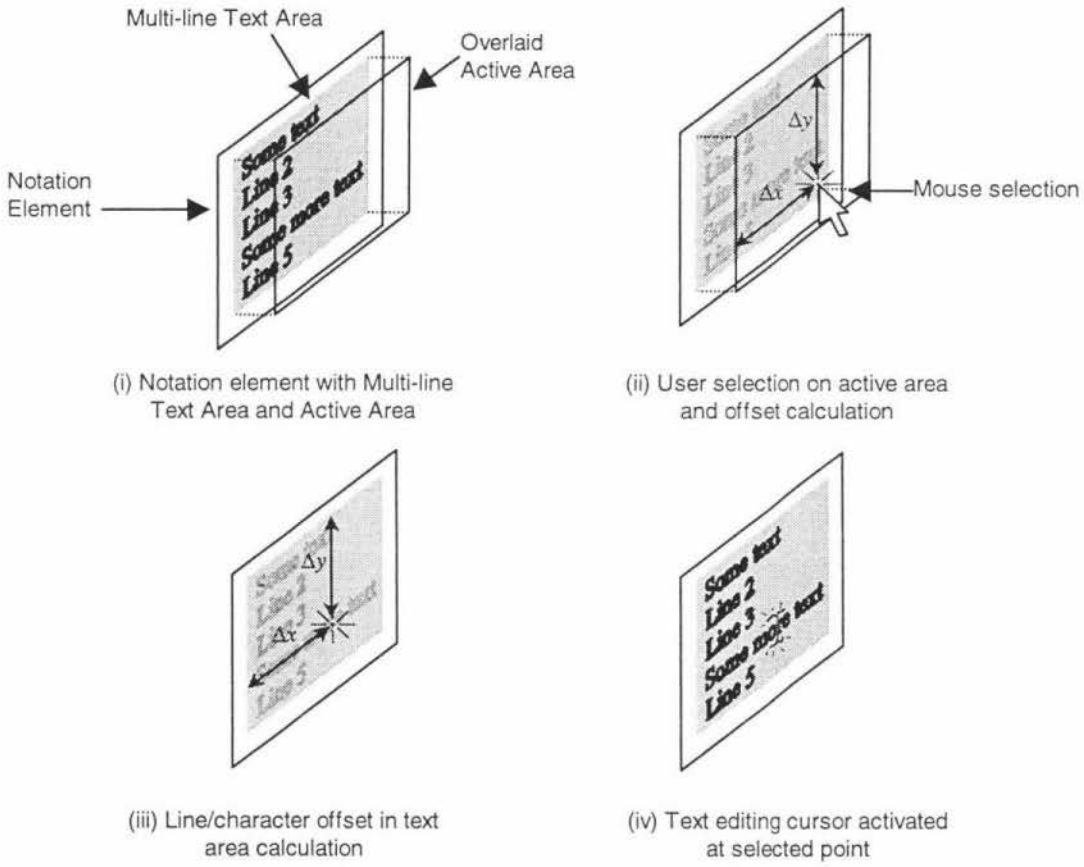



**Figure 3-9 – Association between a "text-area-update" active area and its underlying text area at the user interface level**

### Group Transitions

A group-transition action allows a notation element to alter its visual appearance by instructing the user interface to redraw the element using a different group template. This can be useful in allowing a user to alter the level of visual detail in the information they are provided with. Group-transition actions are most useful when they are applied to a collection of two or more group templates that link each other through transitions.

A user is then free to select the best visual rendering for their needs. Group-transition actions are expressed in NDL by the grammar:

<Transition_Action> ::= TRANSITION TO <Identifier>

Active areas defining group-transition actions are permitted to appear inside group template definitions only. Activating the area instructs the parent template (ie. the template making use of the group as a component) to cease using that particular group template to draw itself with, and instead use the new group template specified as a parameter to the action. Following this, the entire notation element is then redrawn on the display.

Group templates linked through group transition actions may contain text areas identified with the same name. The data contained in the common text area is shared, so that updating the text in one view updates the text in all views (ie. only one copy of the text is stored). This feature allows easy definition of templates that permit multiple levels of detail to be displayed by a single notation element. Figure 3-10 illustrates an example of an object symbol viewed at two levels of detail.

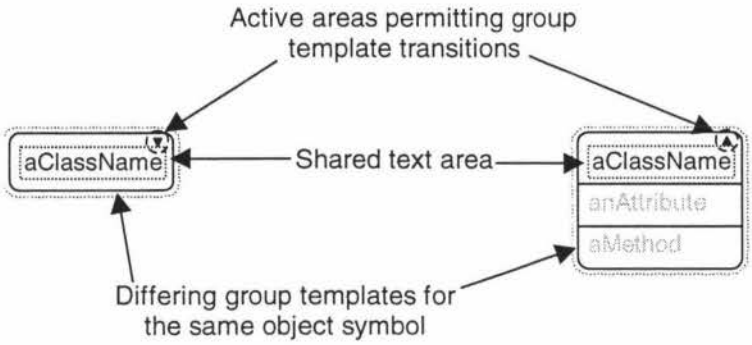



**Figure 3-10 – Common text areas**

Examples of the definition of active area subsection, demonstrating the use of both types of actions that can be associated with notation elements, are as follows. (The examples are based on the active areas shown in the UML class symbol in Figure 3-3(i).)

```
ACTIVE_AREAS {
    [0, 0] TO [Width, 2] TRANSITION TO some_template;
    [W1, 2] TO [+ W1 TEXTWIDTH(classname), H5] UPDATE classname;
```

```
        [2, H2] TO [+ 2 TEXTWIDTH(attributes), H6] UPDATE attributes;
        [2, H4] TO [+ 2 TEXTWIDTH(operations), H7] UPDATE operations;
    }
```

## 3.6. Docking Areas

In most, if not all, diagrams, connections to object symbols are not attached in an ad-hoc manner but instead adhere to particular locations. Certain connections may be permitted to connect only to certain areas on an object symbol, or may not be permitted to connect to a particular type of object symbol at all. For example, in the Coad and Yourdon notation, inheritance relationships are attached to the innermost round rectangle of Class and Class-&-Object symbols. Whole-part relationships and instance connections are attached to the outmost round rectangle (see Figure 2-7, Chapter 2). NDL supports these requirements by allowing the specification of *docking areas* (locations where connections may be attached). These docking areas are used to identify how connections may be associated with object symbols. Individual docking areas are able to limit the types of connections that may be attached, and may limit the quantity of attachments also.

Docking areas are adornments that can be added to object templates and connection symbol templates only[2]. These are the only template types where multiple connections may be made to selective locations on the corresponding template views. Connection terminators always attach to a single connection line at one point. Group templates cannot include docking areas in their definition as group templates may be components of connection terminator templates.

Three types of docking area have been identified, their type-names being linked to the shape that they describe. Point, line, and arc docking areas have been defined to coexist with line and arc graphical components with which a notation element's shape may be described. The three types of docking area are described in the abstract syntax tree by the inheritance hierarchy shown in Figure 3-11. The class Docking Area Segment Template is also a subclass of Segment Template.

[2] Templates are described in Chapter 4.

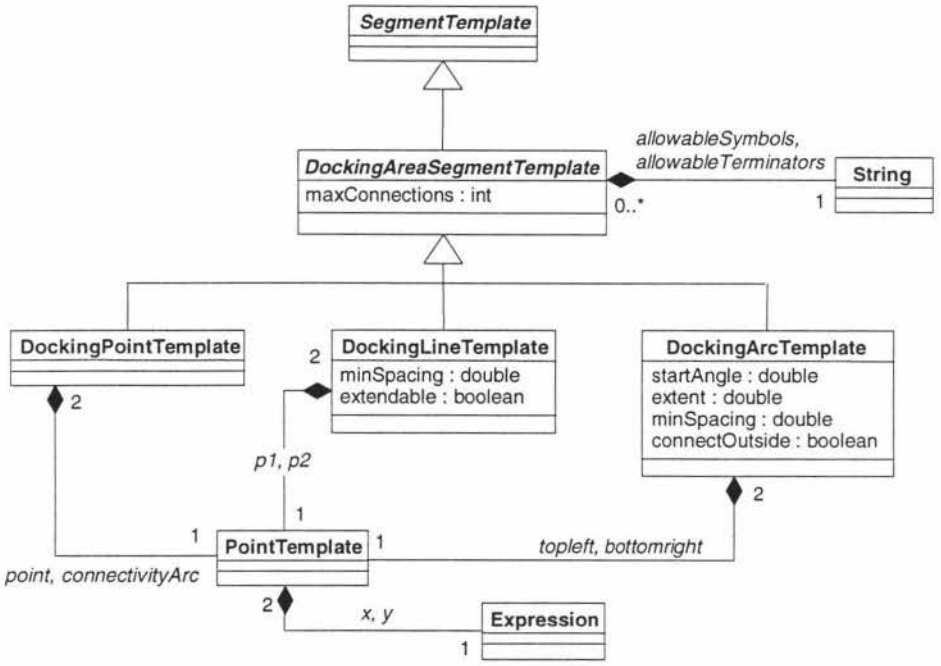



**Figure 3-11 – The Docking Area Segment Template hierarchy**

In line with active area templates, docking areas produce no graphical image. They may or may not be positioned to overlay graphical components of a notation element, depending on the particular requirements. Where a notation element defines more than one docking area that can be attached to a particular type of connection, the decision as to where the connection is finally attached is left as a presentation issue[3]. Docking areas are grouped together in the definition of a notation element using the following grammar:

<Docking_Areas> ::= DOCKING_AREAS '{' { <Docking_Area> } '}'
<Docking_Area> ::= <Point_Docking> | <Line_Docking> | <Arc_Docking>

### Point Dockings

A point docking area represents a single point on a symbol that can allow a connection. Figure 3-12 illustrates an example where a point docking is used.

Generally, it is not allowed for a connection to intersect the symbol to which it is attached. To eliminate possible occurrences of undesirable connections, a point docking area specifies a connection arc. This is an arc, centred on the docking point, through which all connections to that point must intersect. For example, in Figure 3-12 a

[3] The creation of connections in the graphical user interface is discussed in Chapter 6, section 6.5.

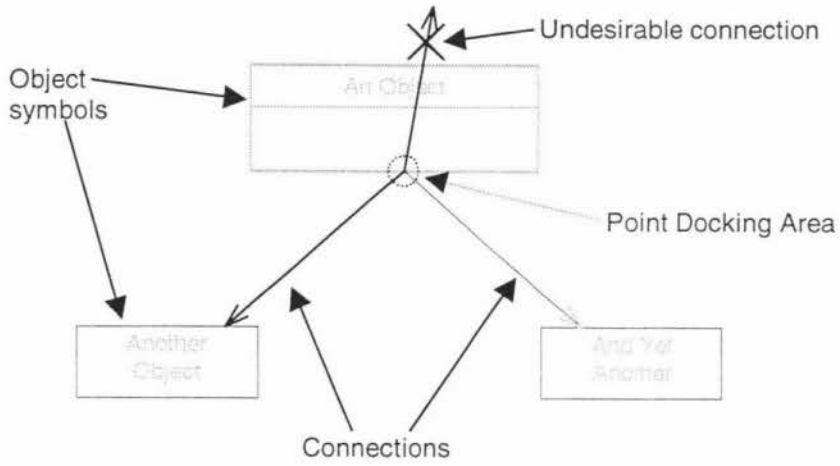



**Figure 3-12 – Point Docking Areas**

suitable connection arc would describe the lower half of a circle around the point. Connection arcs are described in the same manner as the arcs used as graphical components (ie. by start angle and extent).

Point docking areas may also restrict the number of connections that can be attached to the notation element at this point. This is specified by a maximum connection count declared in the definition of the docking area. If no limit is specified, the docking area may have an unlimited number of connections.

Point docking areas may also specify what type of connections may be attached to themselves. This is only relevant to docking areas on object symbols. When required, a list of identifiers that names the allowable connection symbols and connection terminators is included in the definition of the docking point. A connection can only be attached to the docking point if the connection symbol (if one exists in the connection) and connection terminators that describe the pending connection are acceptable to the docking area. If no connection restrictions are defined for a docking point, any type of connection may be attached.

Point docking areas are described in NDL by the following grammar:

```
<Point_Docking>        ::= POINT <Point> START_ANGLE <Direction>
                           EXTENDING <Number>
                           [ WITH <Connection_Count> CONNECTIONS ]
                           [ HAVING <Connection_Types> ]
<Direction>            ::= <Compass_Direction> | <Number>
<Connection_Count>     ::= <Integer> | UNLIMITED
```

```
<Connection_Types>        ::= [ <Terminator_List> ] [ <Connection_Symbol_List> ]
<Terminator_List>         ::= TERMINATORS '(' <Name_List> ')'
<Connection_Symbol_List> ::= SYMBOLS '(' [ NONE [ ',' ] ] <Name_List> ')'
<Name_List>               ::= <String> { ',' <String> }
```

Two examples of point docking areas are described as follows:

```
DOCKING_AREAS {
    ...
    POINT [/ width 2, height] START_ANGLE 270 EXTENDING 0;
    ...
}

DOCKING_AREAS {
    ...
    POINT [0, 0] START_ANGLE 0 EXTENDING 270 WITH 1 CONNECTIONS
        HAVING SYMBOLS  NONE;
    ...
}
```

The first example describes a docking point that could exist in the centre-bottom of a notation element. The identifiers *width* and *height* are assumed to be equated to the width and height of the notation element, respectively. The docking point can accept any number of connections, but from the south direction only.

The second example describes a docking point that is located at the top-left corner of a notation element. It can accept one connection only, and the connection may not contain a connection symbol. Connections can be made to the docking point from any direction between east and south going clockwise (ie. including north and west, but not southeast).

**Line Dockings**

Line docking areas are defined when connections may be made along an entire edge of a notation element. A line docking area inherently restricts the direction from which a connection can be attached to the line. This is indirectly specified by the geometry of the line. A line, specified by a start point, $p_0$, and an end point, $p_1$, is permitted to contain attachments on the right-hand side of the line when traversing from $p_0$ to $p_1$ as shown in Figure 3-13. The docking line in the figure, directed toward the right, can

permit attachments towards the bottom of the diagram (away from the object symbol) but not towards the top (through the object symbol).

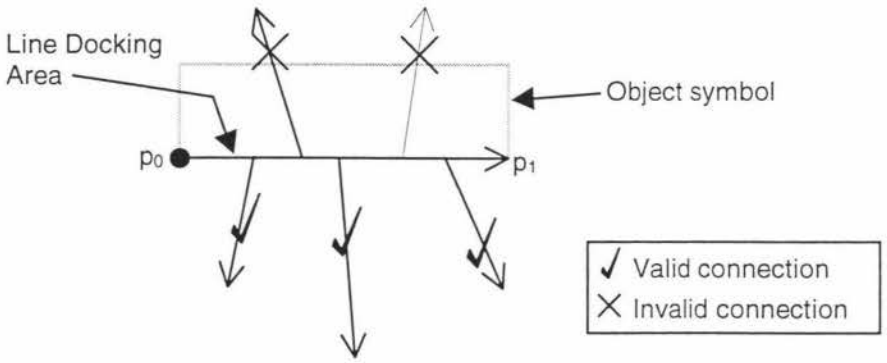



**Figure 3-13 – Line Docking Areas**

A line docking area can be given the property of *extendability*. This means the line can change its length as appropriate to maintain any attachments to it. Generally, this feature is not required for object symbols, but is included to support connection symbols that represent groups of relationships (eg. inheritance and aggregate relationships of Coad and Yourdon).

Figure 3-14 shows two Coad and Yourdon class hierarchies. The first models a single concrete class inheriting from an abstract superclass. The docking area identified by an arrow on the inheritance connection symbol (the half-circle symbol) runs the base of the symbol. It is sufficient to contain the point where the line connected to the concrete class is attached.

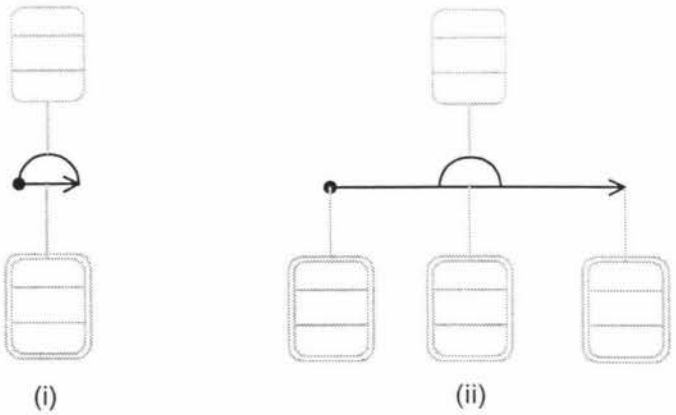



**Figure 3-14 – Coad and Yourdon Line Docking Areas**
**(i) with single connection; (ii) with multiple connections**

In the second case, the connection symbol supports the grouping of three inheritance relationships. However, the base of the original connection symbol is insufficient in space to accommodate all these attachments without an overlapping. To facilitate this, the base line is extended to provide a suitable layout. In NDL, this feature is achieved through defining an extendable line docking area. A docking area defaults to being non-extendable unless an explicit statement is made to the contrary.

Line docking areas, as with point docking areas, allow a restriction on the number of connections that can be attached to the area, and also on the type of connections that can be attached. The details of these are identical to that already described for point docking areas.

A line docking area may also specify a minimum connection distance that must exist between connections attached to the line. This feature has been identified as a presentation issue to avoid overlapping connections. If unspecified, a minimum connection distance of zero is assumed.

Line docking areas are described in NDL by the following grammar:

```
<Line_Docking>      ::= LINE <Point> TO <Point> [IS <Extendable>]
                        [WITH <Connection_Count> CONNECTIONS]
                        [ HAVING <Connection_Types> ]
                        [ MIN_DISTANCE <Number> ] ';'
<Extendable>        ::= EXTENDABLE | UNEXTENDABLE
```

Following are two examples of line docking areas:

```
DOCKING_AREAS {
    ...
    LINE [0, Height] TO [Width, Height] WITH UNLIMITED CONNECTIONS;
    ...
}
DOCKING_AREAS {
    ...
    LINE [Width, 0] TO [0, 0] IS EXTENDABLE HAVING SYMBOLS "Inheritance"
        MIN_DISTANCE 10;
    ...
}
```

The first example describes a line docking area that can accept connections of any type from any southern hemisphere direction, as might exist on the bottom edge of a notation element. The second example describes a line docking that might exist on the top edge of a notation element. It can only accept connections that contain an "Inheritance" connection symbol, the minimum distance between connections is given as ten drawing elements, and the docking line can be extended to accommodate new connections.

**Arc Docking Areas**

Arc docking areas are useful where connections are desired around a curved edge. The shape of the docking area is described in the same manner as a graphical arc component, using a bounding rectangle, a start angle, and an angle of extent. Arc docking areas can restrict the quantity and/or type of connections made to them, and the minimum distance between the connections (which is again zero by default). Arc docking areas also need to specify whether connections should be made to the inside or outside of the curve, as this information cannot be inferred from its basic geometry. Figure 3-15 illustrates contrived examples used to illustrate when connections may be desired on the inside or outside of an arc.

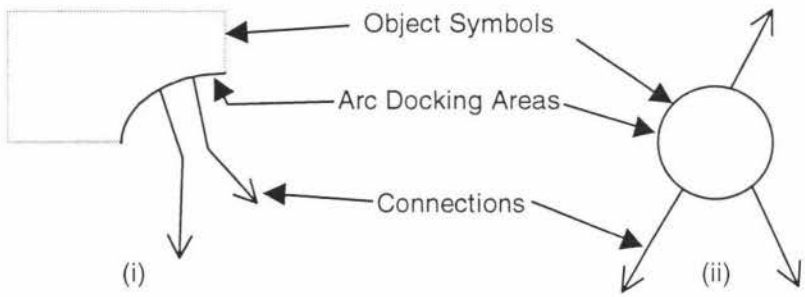



**Figure 3-15 – Arc Docking Areas (i) with connections attached to the inside of an arc; (ii) with connections attached to the outside of an arc**

The grammar below describes the syntax for arc docking areas.

```
<Arc_Docking>        ::= ARC IN BOX <Point> TO <Point> STARTING <Number>
                         EXTENDING <Number>
                         CONNECTED ( INSIDE | OUTSIDE )
                         [ WITH <Connection_Count> CONNECTIONS ]
                         [ HAVING <Connection_Types]
                         [ MIN_DISTANCE <Number> ] ';'
```

Following is an example of an arc docking area. The docking area describes a circle for an object symbol such as that shown in Figure 3-15(ii). Connections are attached to the

outside edge of the circle, and the connections may not contain any connection symbols. Up to five connections can be attached to the docking area:

```
DOCKING_AREAS {
    ...
    ARC IN BOX [0, 0] TO [30, 30] STARTING 0 EXTENDING 360
        CONNECTED OUTSIDE WITH 5 CONNECTIONS HAVING SYMBOLS NONE;
    ...
}
```

## 3.7. Bounding Region

A bounding region is specified for all four template types (groups, object symbols, connection symbols, and connection terminators). The bounding region specifies the smallest region that will fully encompass a view of the template. The current prototype restricts the bounding regions of templates to be rectangular in shape. The bounding region is described by a width and height in the NDL grammar:

```
<Bounding_Region>        ::= BOUNDING_REGION <Point>;
```

In the abstract syntax tree, bounding regions are described by the Point Template class. A Point Template is a container of two expressions. An expression is used for each of the width and height.

Bounding regions are assumed to be relative to the origin ([0,0]) of a template's coordinate system. This assumption is based on the fact that any template can be constructed using only positive ordinates, and is only an issue for manually constructed notation specifications. A visual construction tool that could automatically generate a specification could easily create template descriptions with only positive ordinates.

# Chapter 4

# NOTATION SPECIFICATION AND INTERPRETATION

The individual notation elements that are supported by a particular methodology notation are described in NDL using templates. A template is a composition of a number of the NDL primitives described in Chapter 3, and is a blueprint for the construction of a view. Different template types exists to correspond to the different views (object symbols, connection symbols, and connection terminators) that may be created[1]. Views are constructed by an NDL Interpreter, which executes template specifications.

This chapter describes the scripting of an NDL notation specification, in terms of the templates that it may contain. The composition of each template type is also described. Class hierarchies that contribute to the abstract syntax tree are presented. This chapter also describes the design and operation of the NDL Interpreter and the class hierarchies that are necessary to support its function.

## *4.1. Notation Specifications*

An NDL notation specification is a collection of the template definitions that have been written to describe the notation elements that may appear in a notation. Notation specifications are stored in a plain text format in a notation file. A notation specification begins by identifying the name of the notation, as shown by the following grammar:

<Notation> ::= NOTATION <String> '{' <Template_Collection> '}'

The body of a notation file consists of a collection of the actual template descriptions. The following types of templates may be defined in a notation specification: group templates, object templates, connection symbol templates, and connection terminator

[1] Group templates are excluded here as they are components of other templates and cannot exist in isolation.

templates. These templates may be defined in a notation specification in any quantity and in any order[2]. NDL, however, requires at least one object template and one connection terminator template to be defined in a notation. These are considered the minimum essential for constructing any meaningful diagram. The collection of templates that define the notation syntax is specified according to the following grammar:

```
<Template_Collection>    ::= { ( <Group_Template> | <Object_Template> |
                             <Connection_Symbol_Template> |
                             <Connection_Terminator_Template> ) }
```

The textual notation specifications are parsed by an NDL parser. As a result, an abstract syntax tree is constructed. The root of the abstract syntax tree is represented by the class Notation (Figure 4-1). The Notation class is a container of instances of the classes Object Template, Connection Symbol Template, and Connection Terminator Template. These are the classes that describe notation elements, and represent branches of the abstract syntax tree. Although not actual notation elements, the Notation class is also a container of group templates (represented by the class Group Segment Template). This is to ensure that all group templates are available at any given time, and can be found quickly when necessary. Only one instance of each group and notation element template is ever created, so all references to a particular template reference the same instance of that template. For example, multiple object templates that make use of a common group template all reference the same instance of a Group Segment Template.

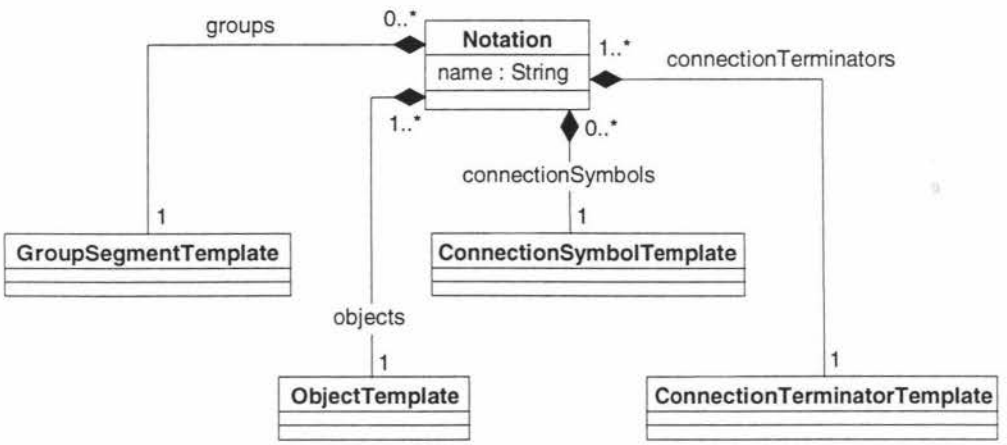



**Figure 4-1 – The composition of the Notation class**

[2] Although templates may be defined in any order, group templates must still be defined before any other template can reference them.

## 4.2. Notation Templates

NDL defines three types of template that correspond to the views that may be created. These are object templates, connection symbol templates, and connection terminator templates. Each of these template types describes a particular type of symbol that may appear in a notation. Templates are composed of segment templates (NDL primitives). Different types of templates may also define special properties that influence template behaviour or increase their functionality.

Each template in a notation is identified by a unique textual name. A template name, taking the form of a string literal, should accurately describe the notation element the template defines. This is necessary as it may be possible for a user to select which notation element to draw only by name.

### 4.2.1. Object Templates

Object templates are used to describe the object symbols used in a notation. Object symbols are visual representations of model concepts such as classes, objects, actors, processes, DFD data stores, etc. An object template specifies the user interactions that can occur with an object view, what connections can be attached to the view, and the positions on the view where these connections may be attached.

The definition of an object template in a notation description consists of a number of compulsory and optional subsections. An object template must at least define one or more graphical components that describe the visible appearance of the view that it represents, and the bounding region of that view. An object template may also optionally define active areas where a user can manipulate the view, and docking areas where connections can be attached to the view. Common expressions that occur in the definition of the various segment templates can be factored out as equations to eliminate redundant calculations.

**Object Template Specification**

Object templates are specified in a notation description using the grammar:

```
<Object_Template>      ::= OBJECT <String> '{'
                             [ <Equations> ] <Components> [ <Active_Areas> ]
                             [ <Docking_Areas> ] [ <Object_Properties> ]
                             <Bounding_Region>
                           '}'
```

The specification begins with the name of the template. The body of the specification composes the equations, primitives, properties, and bounding region that define the symbol.

**Object Properties**

The functionality of object templates can be increased by the declaration of additional properties. In the current implementation of NDL, only one property has been identified for object templates: a default text area. The default text area property is used to direct keystrokes when a user begins to type on an object view without first having selected a particular text area to edit. The default text area automatically becomes the current focus. A default text area would typically be used, for example, to define the text area that holds the name property of an object symbol. This would allow object symbols to be named quickly by simply typing while such a symbol is selected.

The default text area property is utilised by naming the required text area in a subsection of an object template definition devoted to specifying properties. The text area named must be a graphic component of the object template or a graphic component of a group template that is used by the object. If the text area resides in a group template that contains group-transition active areas, a text area of the same name must exist in all groups that can be made visible through such transitions. This ensures that the default text area is always visible to the user.

The "properties" subsection for object templates has the following grammar:

```
<Object_Properties>      ::= PROPERTIES '{' [ <Default_Text_Property> ] '}'
<Default_Text_Property>  ::= DEFAULT_TEXT_AREA <Identifier> ';'
```

The following example declares a text area called *classname* as the default text area for an object symbol:

```
PROPERTIES {
    DEFAULT_TEXT_AREA  classname;
}
```

### 4.2.2. Connection Symbol Templates

Connection symbol templates are used to describe the optional special symbols that may appear in the middle of a connection, typically to denote the type of the relationship the connection represents. A connection symbol template specifies the user interactions that can occur with a connection symbol view, and the particular areas of the view where connections may be attached.

A connection symbol template definition consists of a number of compulsory and optional subjects. A connection symbol template must at least define:

- one or more graphical components that describe the visible appearance of a connection symbol view;
- an entry point for a connection line;
- one or more exit areas for connection lines;
- a bounding region defining the size of a corresponding view.

Additionally, a connection symbol template may declare a list of equations that describe common expressions used in the specification. Certain properties influencing the behaviour of the template may also be declared.

**Connection Symbol Template Specification**

Connection symbol templates are specified using the following grammar:

```
<Connection_Symbol_Template> ::= CONNECTION_SYMBOL <String> '{'
                                 [ <Expressions> ] <Components>
                                 <Connection_Point> <Docking_Areas>
                                 [ <Connection_Symbol_Properties> ]
                                 <Bounding_Region>
                               '}'
```

The specification of a connection symbol template begins with the name of the template. The body of the specification composes the equations, primitives, properties, and bounding region that define the connection symbol. A connection symbol template also defines a connection point.

**Connection Point**

Connection symbols act as visual annotations to alter the meaning of a connection. These symbols are not placed randomly on a line segment representing a connection, but are positioned in a predefined way. To support this, a connection symbol template

defines a connection point. This is a point on the connection symbol where a line segment, stemming from the object symbol initiating the relationship (the parent object symbol[3]), will be attached. In this prototype of NDL, only one connection point is definable. A connection symbol can only have one parent object symbol connection, and it must connect to this point. A connection point is defined using the following grammar:

<Connection_Point > ::= CONNECTION_POINT <Point> ';'

The connection point has a secondary use as the centre of rotation during the construction of a connection symbol template view. Connection symbols appear as annotations to connection lines in a diagram, and are drawn at the same angle as the particular line segment they are connected to (see Figure 4-2). It is obviously impractical to attempt to describe the appearance of a connection symbol at every angle that it may be drawn. Instead, one template is defined that describes the connection symbol as it would appear at some predefined angle, and whenever necessary, the coordinates of the template components, active areas, etc, are rotated about the connection point to align it with a connection line on a diagram. NDL prescribes that connection symbols should be specified in a template as the symbol would appear on a connection line directed from the top of a diagram to the bottom, as shown in Figure 4-2(i).

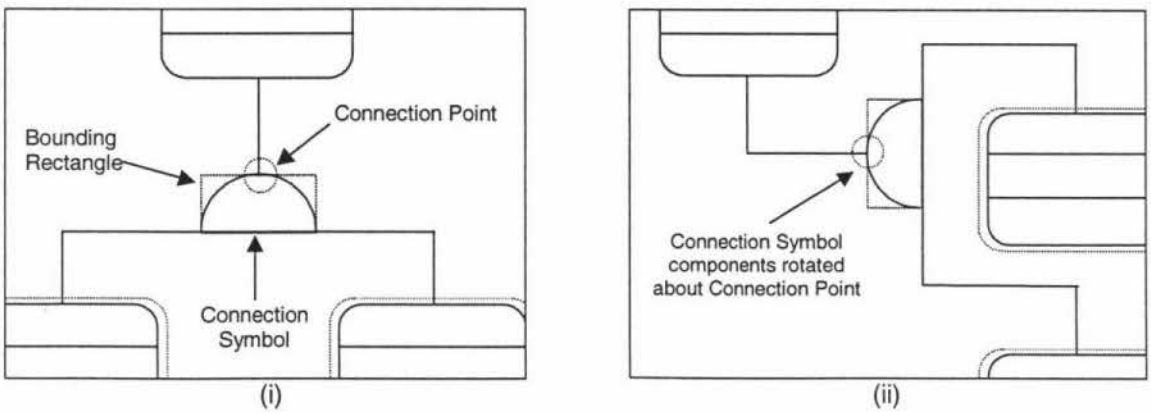



**Figure 4-2 – Connection Symbol Template Rotation: (i) orientation for defining connection symbols; (ii) a connection symbol after it has been rotated 90° anti-clockwise**

[3] The object symbol initiating a relationship is termed the *parent* object symbol. Any object symbols that the relationship is directed towards are termed *child* object symbols.

### Docking Areas

The docking areas of a connection symbol template represent the locations on the connection symbol where connections to child object symbols can take place. A connection symbol template may define as many docking areas as desired. Docking areas are defined in the manner described in Chapter 3, section 3.6.

### Connection Symbol Properties

The behaviour of connection symbol templates can be enhanced by the declaration of additional properties. Connection symbol templates may declare two additional properties: a connection arity and a rotational constraint.

*Connection Arity*

The *arity* of a connection represents the number of object symbols that the relationship is associated with. This does not relate to the presentation of a connection, but specifically to the conceptual relationship(s) expressed by a connection. Figure 4-3 shows an inheritance connection expressed using the Coad and Yourdon notation. The single tree-like structure actually represents three conceptually separate inheritance relationships. Groupings such as these appear in many notations, such as UML (eg. inheritance, aggregation) and Rumbaugh (eg. generalisation).

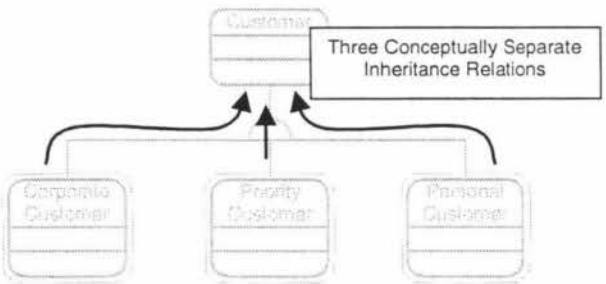



**Figure 4-3 – Grouped inheritance connection in Coad and Yourdon**

NDL defines the arity of a connection symbol to mean *the number of object symbols that a single conceptual relationship is required to be associated with.* It does not consider groupings of these connections into a single visual structure, nor does it constrain whether or not these groupings can take place at all. As examples, inheritance relationships in a class diagram (such as that shown in Figure 4-3) exist between a superclass and a subclass, and so have an arity of two. An m-n-p (ternary) relationship (like the one shown in Figure 4-4) in a relational database connects three tables, and has an arity of three. If unspecified, the default arity of a connection symbol is two.

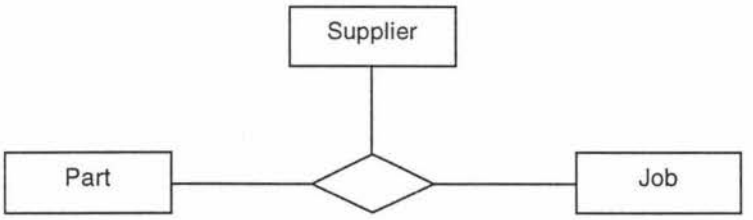



**Figure 4-4 – Ternary relationship**

*Rotational Constraint*

A particular notation may need to restrict the orientation of connection symbols to particular angles. Generally this situation arises when a notation describes connections using only horizontal and vertical lines. NDL supports these requirements by allowing a *rotational constraint* to be placed on connection symbols. The constraint, which in this prototype can be either 'on' or 'off', restricts the rotation of connection symbols to 0°, 90°, 180°, and 270° angles only. Constraining a connection symbol also constrains the connection lines that may be attached to it to horizontal and vertical directions only. If unspecified, a connection symbol by default is rotationally unconstrained.

*Connection Symbol Property Specification*

The "properties" subsection for connection symbol templates has the following grammar:

```
<Connection_Symbol_Properties> ::= PROPERTIES '{' [ <Constraint> ]
                                   [ <Arity> ] '}'

<Constraint>          ::= ( ROTATIONALLY_CONSTRAINED |
                            ROTATIONALLY_UNCONSTRAINED ) ';'
<Arity>               ::= ARITY <Integer> ';'
```

The following example describes properties of the connection symbol shown in Figure 4-4. It states that the connection symbol is rotationally constrained and has an arity of three.

```
PROPERTIES {
    ROTATIONALLY_CONSTRAINED;
    ARITY 3;
}
```

### 4.2.3. Connection Terminator Templates

Connection terminator templates are used to describe the symbols or icons that can appear at the ends of connections. These symbols are typically used to identify a particular type of relationship (eg. inheritance, composition), or to define additional properties of a connection already identified through a connection symbol (eg. cardinality). Various geometric shapes such as small circles, diamonds, or arrowheads are common terminators.

A connection terminator template definition consists of a number of compulsory and optional sub-sections. A connection terminator template must at least define:

- one or more graphical components that describe the visible appearance of a connection terminator view;
- a "head" – a connection point where the terminator is attached to object symbols;
- a "tail" – a connection point where the terminator is attached to a connection line segment;
- a bounding region defining the size of the view.

A connection terminator template may also declare a list of common equations or additional properties that influence the terminator's behaviour.

Connection terminators, like connection symbols, are drawn at the same angle as the connection lines they are attached to. NDL prescribes that connection terminators should be defined the way they would appear if they were attached to the left-hand end of a horizontal line segment, as shown in Figure 4-5. Connection terminator templates also define a rotational constraint property. In this prototype, any connection that is constructed using a connection terminator that is rotationally constrained can only ever be drawn using horizontal and vertical lines.

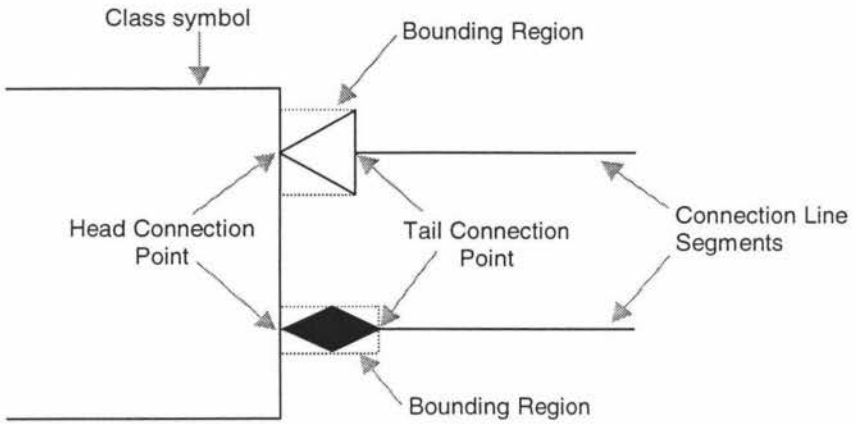



**Figure 4-5 – Features of connection terminators representing the generalisation (top) and composition (bottom) relations in UML**

The grammar for connection terminator templates is as follows.

```
<Connection_Terminator_Template> ::= CONNECTION_TERMINATOR <String> '{'
                                     [ <Expressions> ] <Components> <Head> <Tail>
                                     [ <Connection_Terminator_Properties> ]
                                     <Bounding_Region>
                                   '}'
<Head>                           ::= HEAD <Point> ';'
<Tail>                           ::= TAIL <Point> ';'
<Connection_Terminator_Properties> ::= PROPERTIES '{' [ < Constraint> ] '}'
<Constraint>                     ::= ( ROTATIONALLY_CONSTRAINED |
                                       ROTATIONALLY_UNCONSTRAINED ) ';'
```

### 4.2.4. Template Class Hierarchy

The templates describing notation elements are represented by several classes in the abstract syntax tree. The classes Object Template, Connection Symbol Template, and Connection Terminator template are represented in the inheritance hierarchy shown in Figure 4-6.

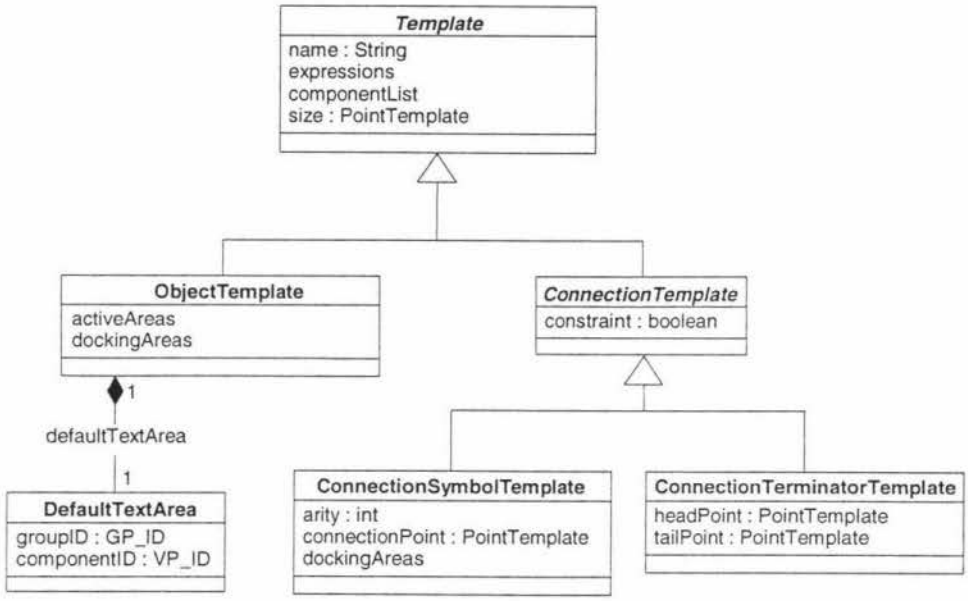



**Figure 4-6 – The Template class hierarchy**

## 4.3. Design of the NDL Interpreter

The NDL Interpreter is used to interpret NDL templates that exists in a notation abstract syntax tree. The result of any such interpretation is the generation of an NDL view. To support the interpretation of templates, the abstract syntax tree was designed using the Interpreter pattern (Gamma *et al*, 1995). The Interpreter pattern defines a representation for a language grammar along with an interpreter that uses that representation to interpret sentences in the language. Each template, segment template, and expression described in the abstract syntax tree is therefore capable of interpreting itself. Templates are executed to generate views. Views are containers of segments, which are generated from the execution of segment templates. Each segment template is composed of a number of expressions, which are evaluated by the interpreter during the construction of segments.

To generate a view, each template requires a set of viewable properties and a context. Each view of a notation element has its own set of viewable properties that provides the values for text areas, the names of any group templates being used to construct the view, and any other properties that are unique to that particular view. The context is an instance of the Visitor pattern (Gamma *et al*, 1995). The use of the Visitor pattern allows the properties of the drawing surface (for example, how long a string of text actually is, in drawing units) to be hidden from the interpreting mechanism.

### 4.3.1. Context

The abstract class Context acts as an interface between a notation specification and the drawing surface. It removes the need for the notation interpreting mechanism to know the details of the underlying graphical system, and allows the implementation of the graphical user interface to be changed, upgraded, or replaced without any interference to the rest of the tool. The class Context defines operations to draw lines, arcs, and text strings on a drawing surface, and to evaluate the height and width (in drawing units) of text strings (Figure 4-7). An implementation of the NDL interpreter in a particular language or for a particular platform includes the implementation of a concrete subclass of Context. The subclass implements the operations defined by Context using whatever facilities are provided by the implementation environment. Porting the NDL Interpreter to a new platform requires only a new subclass of Context to be implemented, rather than the reimplementation of every class that uses context properties (eg. Expressions, Segment Templates, Segments, Views, etc).

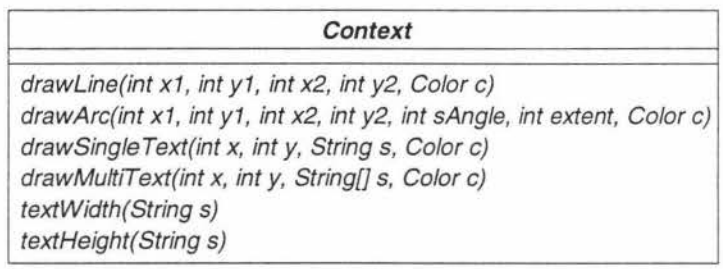



**Figure 4-7 – The Context class**

### 4.3.2. Viewable Properties and Viewable Things

Run-time characteristics of views are described by viewable properties. Viewable properties capture the values of text areas, the state of group references, the extension of line docking areas, and the count of connections that have been attached to individual docking areas. Viewable properties are defined in the class hierarchy shown in Figure 4-8.

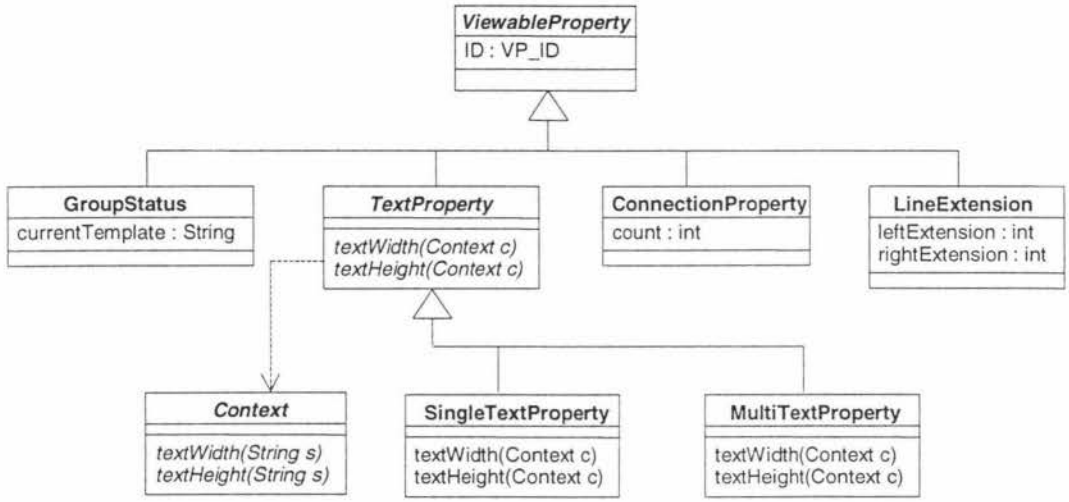



**Figure 4-8 – The Viewable Properties class hierarchy**

The viewable properties of a view are composed into an object of the class Viewable Thing (Figure 4-9). Each view has its own Viewable Thing. A Viewable Thing provides methods to add or retrieve Viewable Properties.

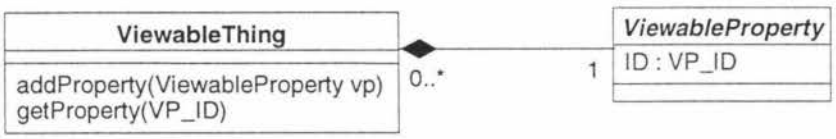



**Figure 4-9 – The Viewable Things class**

A viewable thing acts as a local cache for the property values that it contains. Property values that represent semantic information, such as the value of text areas, are kept under permanent store in SSL objects maintained by the server. In the event that these property values change, the client informs the server at the same time as updating the local cache. Caching these values in the client provides instant access time if views need to be redrawn or reconstructed.

### 4.3.3. Expressions

All segment templates are defined by a number of expressions. A straight line for example is defined by expressions for its two end points. Figure 4-10 shows the Expression class hierarchy. For the purposes of interpretation, the Expression class defines an operation *evaluate* that takes a Context and a Viewable Thing as parameters. Each leaf class implements this operation to evaluate the particular type of expression they represent. The result of any evaluation is a double-precision floating-point numerical value.

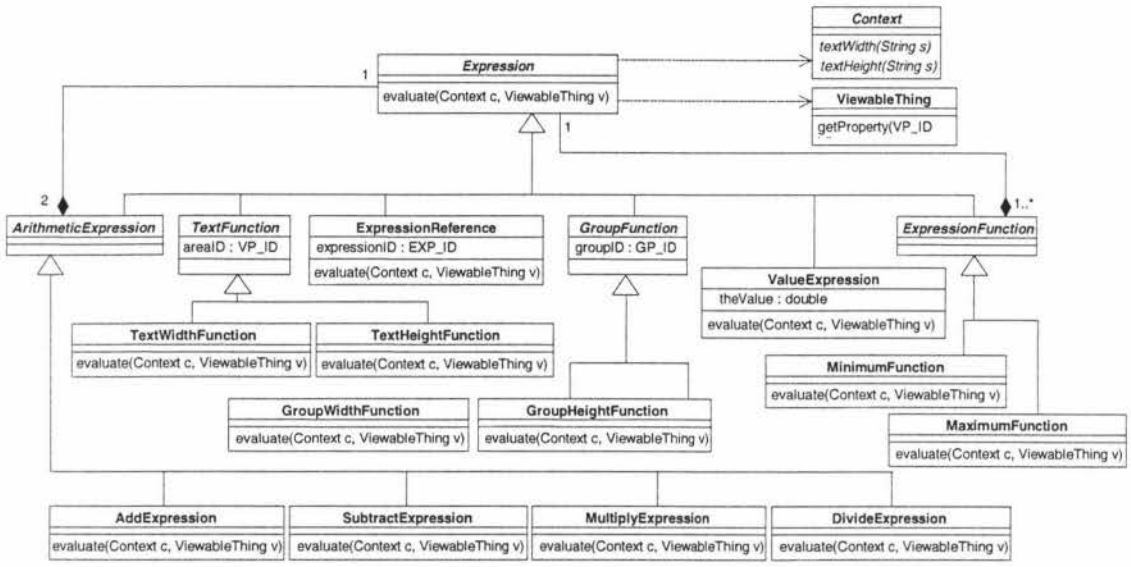



**Figure 4-10 – The Expression class hierarchy**

The *Value Expression* class is the simplest subclass of Expression, and encapsulates just a single numerical constant. Evaluation of an instance of a Value Expression merely returns the value of the constant contained within.

The *Arithmetic Expression* class encapsulates two Expression instances as expression operands. Instances of concrete subclasses of Arithmetic Expression evaluate the two operands and apply a particular operator (such as addition or subtraction) to the evaluation results.

Subclasses of the *Text Function* class are used to calculate either the width or height of a text area. An instance of a concrete subclass of Text Function knows the ID of the text area segment template that the function is to be applied to. The width and height of any text area depends on the context that it is viewed in. This includes the particular font and font size for the text area. The actual responsibility of calculating the size of individual strings in a text area is delegated to the Context object passed as a parameter to the *evaluate* message.

Subclasses of the *Group Function* class are used to calculate either the width or height of a group template. An instance of a concrete subclass of Group Function knows the ID of the group reference segment template that it is to be applied to. The group reference segment template knows the name of the group template that is currently being used by the template. The responsibility of calculating the size of a group is delegated to the particular group template itself. Group templates can calculate their

physical size (in drawing units) by evaluating the expressions defining its bounding region.

Subclasses of the *Expression Function* class are used to apply a particular function to a collection of Expression operands. Concrete subclasses of Expression Function evaluate each Expression that is an operand, then apply a function across the evaluation results.

The *Expression Reference* class allows the reuse of predefined equations to avoid redundancy. When an equation is first evaluated, the result is stored. Subsequent references to the same equation return the previously evaluated result. An equation is only re-evaluated if some viewable property that it depends on alters.

### 4.3.4. Segment Templates

Different parts of the abstract syntax tree class hierarchy for Segment Template were discussed in Chapter 3. A complete class hierarchy is shown in Figure 4-11. The interpreter introduces an operation *build* into the class hierarchy that subclasses of the class Segment Template implement. Segment Template objects respond to the *build* message when given a Viewable Thing and a Context as arguments. The operation causes an instance of a corresponding Segment to be created.

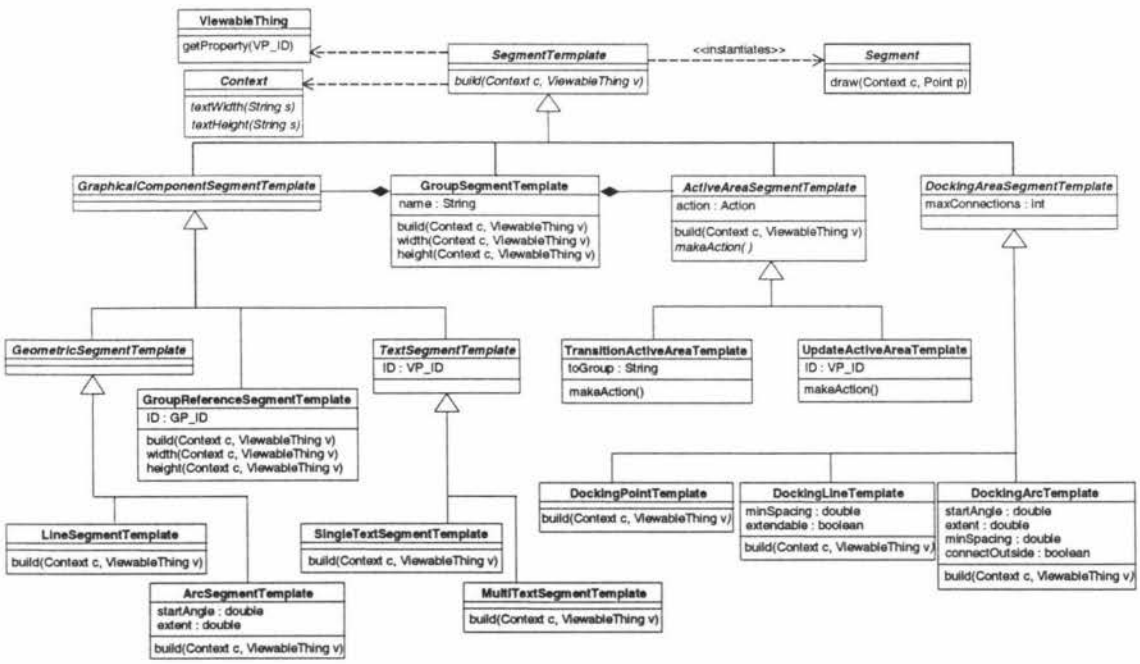



**Figure 4-11 – The Segment Template class hierarchy**

### 4.3.5. Segments

Segments are instantiations of Segment Templates that describe the primitive components in a particular context and with a particular combination of properties. Corresponding Segments exist for all Segment Templates. Segments can draw themselves by responding to a *draw* message when given a Context and a drawing offset as arguments. The drawing offset represents a point within the context drawing surface where the segment should begin drawing. An optional drawing colour may also be specified to override the default drawing colour. Not all segments actually produce a graphical image. For example, active area, docking point, and docking arc segments do not. A docking line segment, however, draws extensions that may have been added to the basic docking line defined. The class hierarchy of Segments is shown in Figure 4-12.

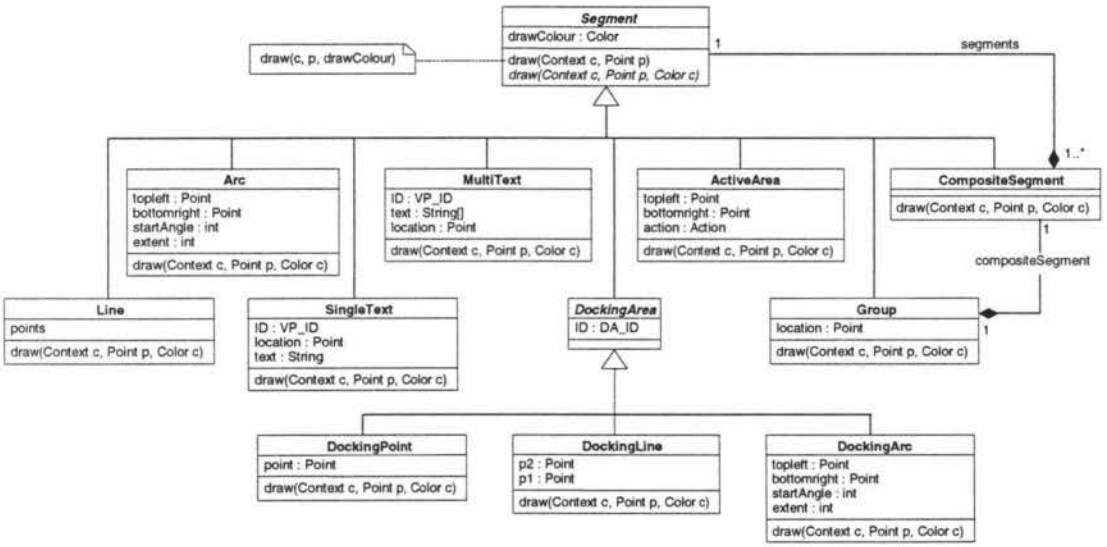



**Figure 4-12 – The Segments class hierarchy**

The leaf classes in the inheritance hierarchy correspond to the different Segment objects that can be created. Each type of Segment is constructed from a corresponding type of Segment Template. Table 4-1 describes the mapping between Segment Templates and Segments.

| Segment Template | Segment |
|---|---|
| Line Segment Template | Line |
| Arc Segment Template | Arc |
| Single Text Segment Template | Single-Text |
| Multi-Text Segment Template | Multi-Text |
| Composite Segment Template | Composite Segment |
| Transition Active Area Template | Active Area |
| Update Active Area Template | Active Area |
| Docking Point Template | Docking Point |
| Docking Line Template | Docking Line |
| Docking Arc Template | Docking Arc |

**Table 4-1 – Mappings between Segment Templates and Segments**

Both subclasses of Active Area Template shown in Table 4-1 produce the same Segment type: an Active Area. To distinguish between the different active area types, an instance of an Active Area Template subclass parameterises the Active Area segment it creates with an Action object. An Action object exists for each type of action supported, as shown in Figure 4-13. Each subclass encapsulates the necessary parameters to support the action.

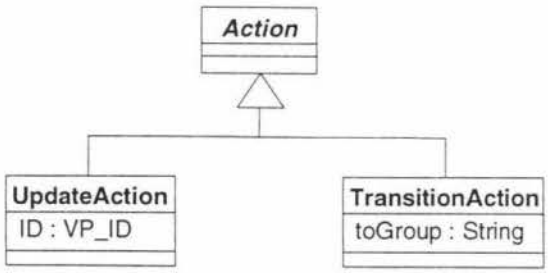



**Figure 4-13 – The Action class hierarchy**

### 4.3.6. Templates

For the purposes of interpretation, the Template class hierarchy presented in Figure 4-6 is extended to include the definition of a method *build*, as shown in Figure 4-14. Templates respond to a *build* message, given a Viewable Thing and a Context, by creating a complementary view. A view is a container of segments that is constructed

by sending a *build* message to each Segment Template contained in the Template. The bounding region of a template is also evaluated to determine the size of the view.

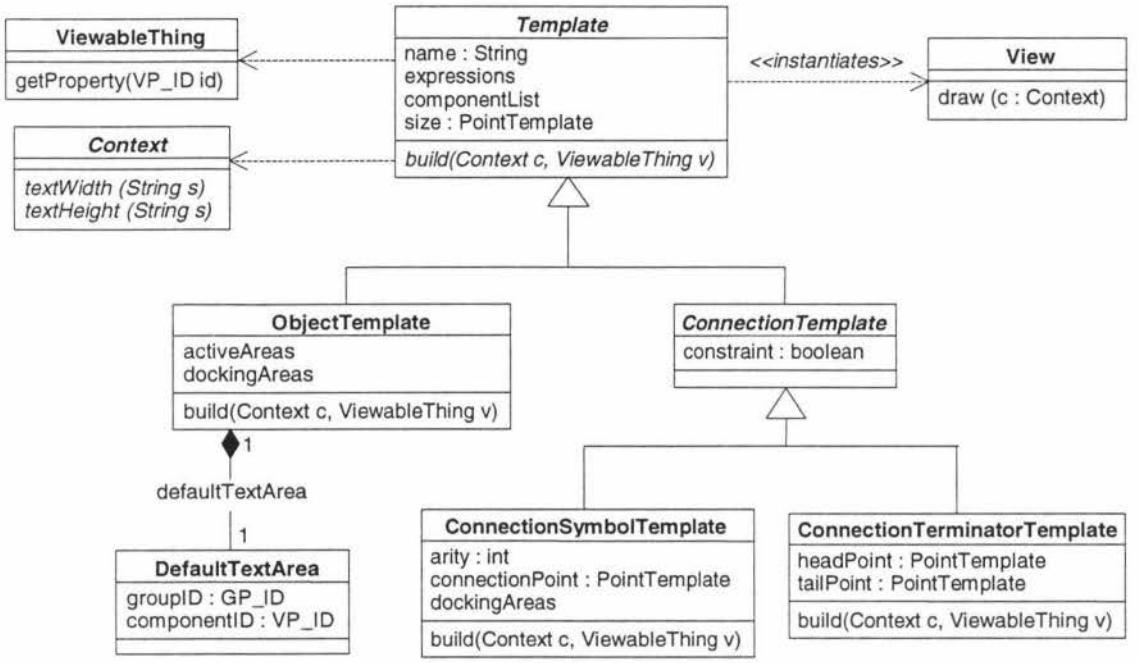



**Figure 4-14 – The Template class hierarchy**

### 4.3.7. Views

Corresponding Views for each of the notation element template types supported in NDL are defined. Every notation element drawn in a diagram has its own View. Different Views are constructed from a single Template object by parameterising the Template's *build* method with a different context or a different set of properties. Views are defined in an inheritance hierarchy as shown in Figure 4-15.

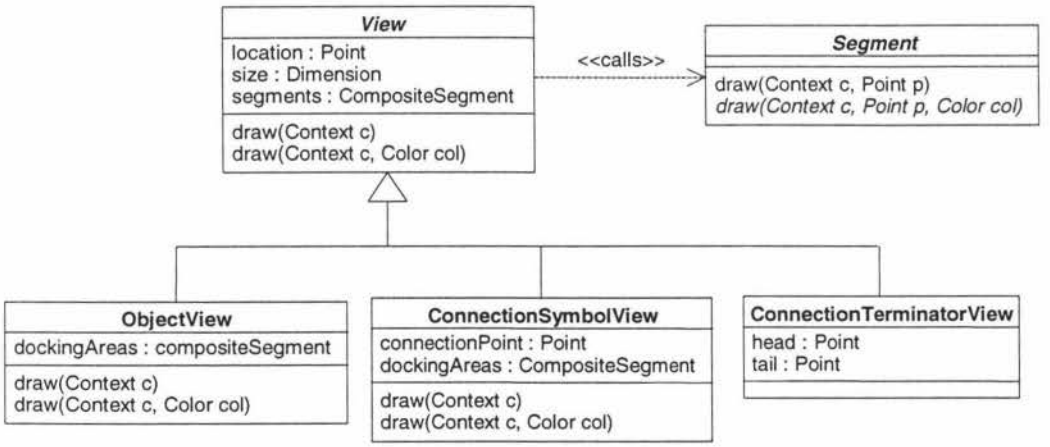



**Figure 4-15 – The View class hierarchy**

The View class implements an operation *draw*. A View responds to a *draw* message when given a Context as an argument by rendering itself onto the drawing surface supplied by the context. An optional drawing colour may also be specified which causes all the segments contained in the view to be rendered in that colour. A view is rendered by sending a *draw* message to each segment contained in the view. Subclasses of View override *draw* where necessary to include additional segments (such as line docking areas) in the rendering.

# Chapter 5

# A CASE Tool Client and the Notation Definition Language

A CASE Tool client user interface has been developed to verify that the approach taken in defining the syntax of a methodology with NDL is efficacious, as well as to determine any deficiencies that still need to be addressed. The client acts as an interface between a user and a notation, providing a means for the user to construct diagrams from the notation elements defined by notation templates. The client concerns itself only with notation syntax, as defined by an NDL specification, and relies on a connection to the CASE Tool server for semantic processing. The client has been designed in a way such that it is as generic as possible, avoiding any methodology or model specific features.

This chapter describes the design of the client, focusing particular attention to its role as a diagram editor. The classes that the client uses to interface to NDL notation templates and template views, and what other classes are required to support full diagram construction are discussed. The chapter also describes how the underlying system model copes with user interactions such as the creation or movement of graph objects.

## 5.1. Overview of Client Interface

The MOOT CASE Tool client user interface (the *client*) is based around a single dialog window residing in a WIMP (Windows, Icons, Menus and Pointer) environment. Most interactions between the user and the client are performed through this main window. The client has been designed using the object-oriented event-driven system model as is standard with windowing environments. An instance of the client remains in an idle state until triggered into performing an action by the occurrence of an event. The types of events that the client may respond to include user-generated actions, such as a mouse click or key press, and events that originate from the remote server.

The client is modelled by the MOOTClient class (see Figure 5-1). This class displays the main interaction window and encapsulates the behaviour necessary to manage the

interaction between the user and the client. The client consists of five core components: the server communications proxy, the project managers, the model editors, the diagram editors, and the graph objects. The first three core components are directly managed by the MOOTClient class, and are described by the classes ServerProxy, ProjectManager and ModelEditor. MOOTClient also instantiates a menu bar that is attached to the main window. The contents of the menu bar may change during the lifecycle of a client instantiation to reflect the current context that the user of the system is working in.

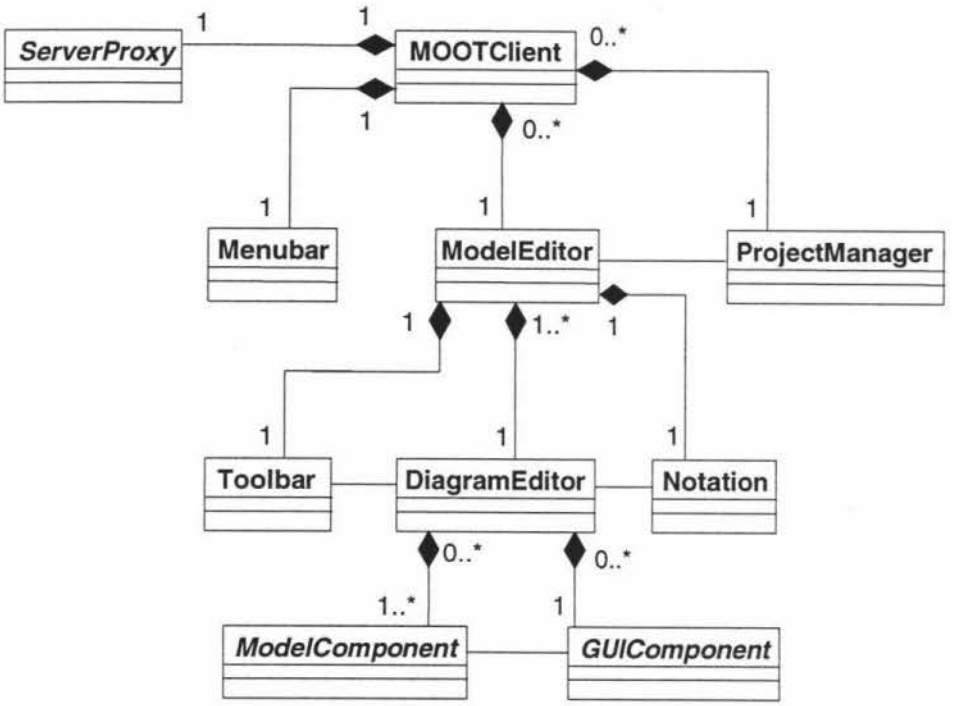



**Figure 5-1 – MOOT CASE Tool Client Model**

The ServerProxy class provides the communications interface between the client and the server. The ServerProxy class is an instance of the Singleton pattern (Gamma *et al*, 1995) as only one instance is created during the lifetime of the client. All user requests that create, modify, or destroy data pertaining to a CASE project require interaction with the server. The proxy assembles these requests into an appropriate form before transmitting to the server, and subsequently returns the server response to the originator of the request. The proxy also handles communications from the server that may not be directly caused by a client request. Such requests include automatic requests to create, modify or delete particular data, generally to maintain consistency between diagrams, models, projects, or other clients connected to the server. The communication between the client and server is further described in Chapter 7.

The ProjectManager class provides a user interface to a CASE project via a dialog window that is separate from the main interaction window. Each project that is created or opened by a user is managed by a new instance of ProjectManager, and hence a new dialog window. The ProjectManager class provides project-level operations that apply to open projects, such as the addition, deletion, or renaming of project models.

The ModelEditor class provides a user interface to a model. An instance of the class presents information to the user via a sub-window of the main window. A model is described using a particular notation, so the ModelEditor class accommodates GUI features that provide access to the elements defined in notations. One way it does this is by creating a toolbar that contains a set of drawing tools based on the notation elements. The toolbar is an instance of the ToolBar class, and is attached to the top of the model editor sub-window. A set of pop-up menus is also created that allow an alternative means of selecting drawing tools. The remainder of the model editor sub-window is occupied by one or more diagram editors. A ModelEditor is a container of these diagram editors, described by the DiagramEditor class.

The DiagramEditor class encapsulates the bulk of the client system by providing support for the construction of diagrams. Each diagram in a model shares the same notation. A DiagramEditor object provides a drawing canvas where a user can add, modify, or delete the object symbols and connections defined in the notation. Interactions with a diagram may cause similar effects in other diagrams of the same model, or in other models. For instance, the creation of an object describing a concept in one model may cause the automatic instantiation of the concept in other models.

A diagram is constructed by the user selecting particular notation elements to draw, and subsequently "painting" on the drawing canvas with these elements. Interactions of this nature cause one or more *graph objects* to be created. A graph object is the smallest visual entity that can be manipulated by the user, and include items such as object symbols, connection symbols, and the individual line segments of a connection. Each graph object is described by a complementary GUIComponent object. GUIComponents can draw, update, and move themselves in response to user interactions. GUIComponents are described in detail in section 5.2.1.

A diagram is not just a collection of graph objects, it is a visual description of a model or part of a model. The graph objects that exist in a diagram must somehow be related back to the more abstract, semantic view of the diagram (ie. a collection of concepts and the relationships between them). While the client is not interested in (and knows

nothing about) the semantics of a diagram, it needs to support the mapping of the diagram to a collection of SSL objects located on the server. To support this mapping, the diagram editor employs a ModelComponent class hierarchy. Objects of ModelComponent classes retain the necessary information that can relate graph objects to semantic entities (ie. concepts and relationships). Each GUIComponent object has an association with a ModelComponent object. Potentially, a semantic entity could have multiple representations, each described by different GUIComponents. Due to the nature of this prototype, each ModelComponent is restricted here to having only one representation. Managing multiple representations is not yet supported (through copy and paste, for example). ModelComponents are described in detail in section 5.2.2.

Figure 5-2 illustrates the client interface in a particular state, and shows how the various interactive classes of the system are presented to the user.

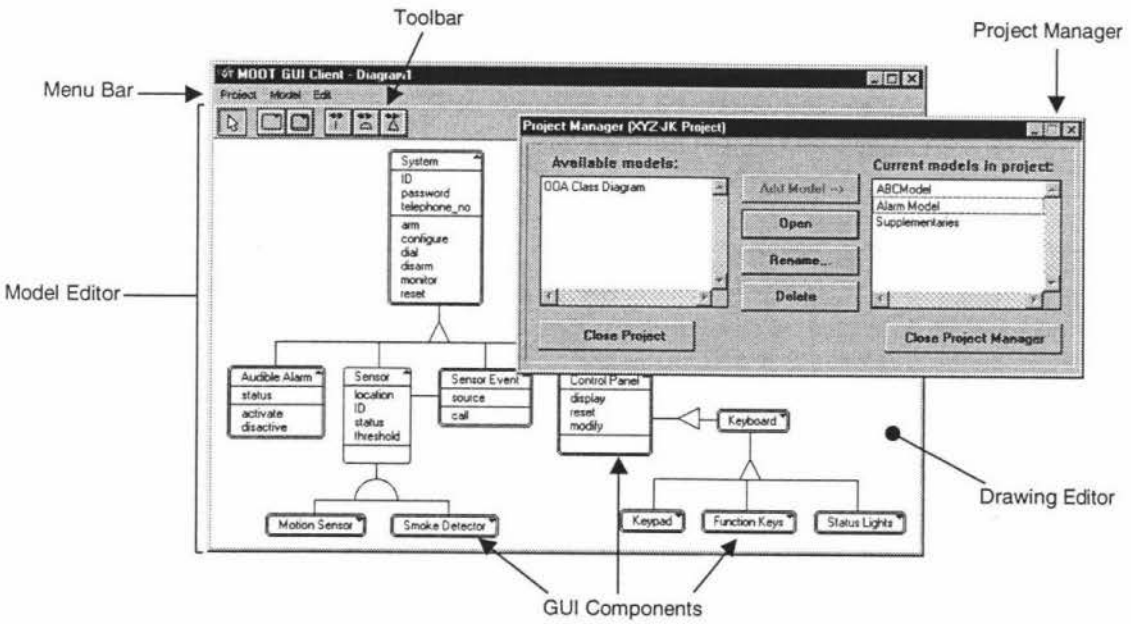



**Figure 5-2 – CASE Tool Client Construction**

The design of the diagram editor is obviously the most interesting aspect of the client, and is the most complicated. The remainder of this chapter focuses on the underlying model of the diagram editor, and describes how an NDL notation specification is used to specify the constructions that can take place in a diagram. Aspects of the graphical user interface design in particular are described in Chapter 6.

## 5.2. NDL and the Diagram Editor

The diagram editor uses an NDL notation specification to provide a basic set of graph objects that can be used for diagram construction. These graph objects constructed from the templates that are defined in the notation specification. Graph objects constructed from notation templates carry a state with them that includes the current values of any viewable properties the template may use to construct a view, and the current view that has been generated from the template. Graph objects supplementing those not defined in a notation (such as the line segments describing connections) are also required. All graph objects in a diagram are not just isolated entities but are generally associated with each other in some way. For example, connections are composed of a number of associated graph objects (connection terminators, line segments, and an optional connection symbol) which are in turn associated to the object symbols that they connect to.

To support "intelligent" graph objects, the diagram editor employs a hierarchy of GUIComponents. Subclasses of GUIComponent are created to encapsulate the specific details of each type of graph object, and also to handle the user interactions which may occur with the graph objects. GUIComponents are described in section 5.2.1. A hierarchy of ModelComponents is also employed to assist the mapping between graph objects and their underlying semantic representation as SSL objects in the CASE Tool server. ModelComponents are described in section 5.2.2.

### 5.2.1. GUIComponents

A diagram is not just a picture, it is a collection of graph objects that describe a system or a part of a system. Each graph object in a diagram has a state that has some semantic meaning (although the client is unable to determine what this semantic meaning is), and allows a particular set of interactions. Interactions may be generic for a particular type of graph object, but may also be defined for a particular type of graph object through the specification of active areas. Graph objects can also be related to each other in some way, although again the client is unable to determine any semantic meaning from these relationships. What the client, or more particularly the diagram editor, must provide is a suitable means for a user to construct diagrams, oblivious to what the diagram actually means, and allow interaction with the individual components that make up those diagrams. The diagram editor makes use of a class of objects called GUIComponents, for this purpose.

The diagram editor makes use of two types of graph object: template and non-template. Template graph objects are based around the notation elements that can be defined in an NDL notation specification: object symbols, connection symbols, and connection terminators. Template graph objects provide an encapsulation of notation template, viewable properties, and template view. Many operations provided by template graph objects make use of one or more of these encapsulations for their functionality. Non-template graph objects describe graph objects for which no notation template is definable in a specification. Only line segments used to connect object symbols together are not defined by NDL templates. The diagram editor provides a class for the description of line segments, and also a class for the composition of multiple line segments into a single entity. All classes describing graph objects are subclasses of an abstract class GUIComponent. The GUIComponent class hierarchy is shown in Figure 5-3. The leaf classes correspond to the possible graph objects that can be created and manipulated by the diagram editor.

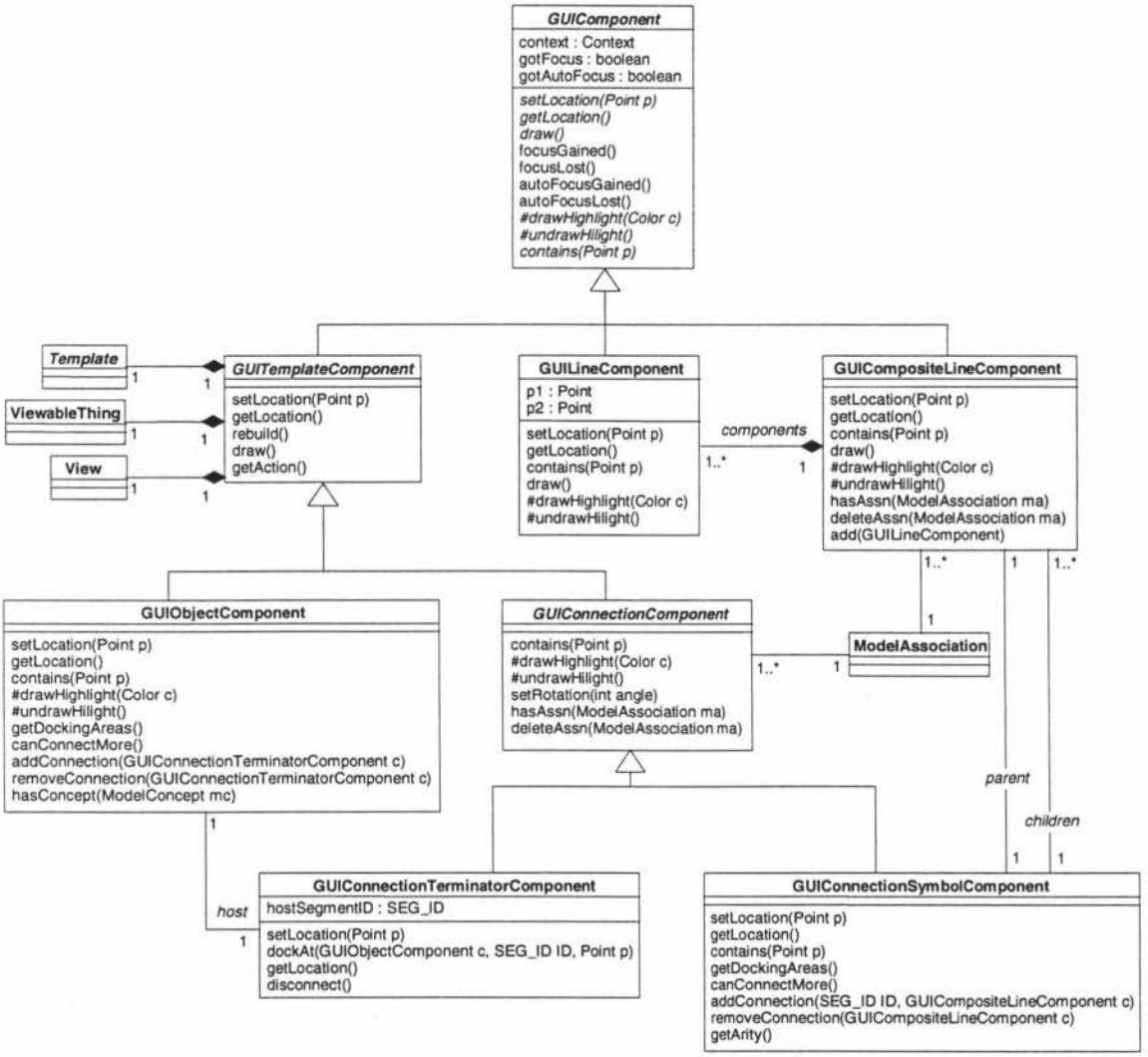



**Figure 5-3 – The GUIComponents class hierarchy**

**The GUIComponent class**

The abstract class GUIComponent defines a set of operations that all subclasses must implement. These operations describe generic operations for graph objects including their positioning, rendering, and highlighting. Also defined is an operation *contains*, which the diagram editor uses to determine what graph object, if any, is present at a particular screen location selected by the user.

GUIComponent objects render themselves by way of a drawing context defined by the Context class. The Context class provides an interface to the graphical sub-system by defining a set of primitive operations for line, arc, and text rendering. GUIComponents use this set of operations to draw the individual graphical primitives that the graph object is usually described by. The implementation of a drawing context eliminates the dependency of graph objects, templates, and views from one particular graphical sub-system model. The Context class was described previously in section 4.3.1 (Chapter 4).

**GUITemplateComponents**

All graph objects drawn from notation templates are represented by a subclass of GUITemplateComponent. This abstract class encapsulates a reference to the Template that describes the symbol, a Viewable Thing that contains a collection of Viewable Properties, and a View that is created from a particular combination of Template, Viewable Properties, and Context. The instantiation of an class that is subclassed from GUITemplateComponent creates a default set of Viewable Properties for the graph object that is based on both the template type and any default values defined in the template. Each time a Viewable Property is added, deleted, or modified, a new View instance is created, resulting in the need for the graph object to be redrawn. GUITemplateComponent implements a method *rebuild* that causes the construction of a new View object. GUITemplateComponent also implements the *draw* method defined by GUIComponent. GUITemplateComponent delegates the actual rendering of the graph object to the current View. Actions associated with active areas of a template are retrieved by the use of the *getAction* method. The use of active areas and actions is described in section 5.3.3.

**GUIObjectComponents**

GUIObjectComponents are used by the diagram editor as an interface to the object symbols that are defined in a notation. The GUIObjectComponent class provides access to the docking areas of an object symbol, and permits connections to be attached or removed from the docking areas. Every GUIObjectComponent is also associated with a ModelConcept (a subclass of ModelComponent). The ModelConcept class maintains

data necessary to map the GUIObjectComponent to an SSL object on the server where its semantics are important. The relationship between GUIObjectComponents and ModelConcepts is described in section 5.3.1.

**The GUIConnectionComponent class and its subclasses**

The GUIConnectionComponent class describes the common properties between connection symbols and connection terminators. These connection components can all be rotated to allow connections to be made at various angles, so the class implements a method *setRotation*. Each connection component is associated with a ModelAssociation (a subclass of ModelComponent) that maintains data necessary to map the various components to a particular semantic relationship that they describe. The relationship between GUIConnectionComponents and ModelAssociations is described in section 5.3.2.

The GUIConnectionSymbolComponent class provides an interface between the diagram editor and connection symbols defined in a notation specification. Methods are provided to access the docking areas of the connection symbol and to add and remove connections to them.

GUIConnectionTerminatorComponents are used by the diagram editor as an interface to connection terminators defined in a notation specification. A method *dockAt* is implemented which causes the terminator to connect to a particular docking area on a GUIObjectComponent. Subsequent disconnection is provided through the *disconnect* method. The *setLocation* method causes the component to move to a new location, however the terminator will restrict its movement to that permitted by the docking area it is attached to.

**GUILineComponents**

The GUILineComponent class provides an implementation of the connecting line segments that join connection symbols and connection terminators together. Each component represents a single line segment, described by two endpoints. The line is rendered by way of the drawing context. To allow reasonable ease of selecting line segments, the class implements the *contains* method so that points up to three drawing units either side of the actual line are considered part of the line.

**GUICompositeLineComponents**

GUICompositeLineComponents are not actually graph objects that are created and manipulated by a user of the diagram editor, but instead serve to group together one or

more connected line segments. In the context of a connection, it is only important that the lines exist (as a GUICompositeLineComponent), rather than how they exists (as a collection of a variable number of GUILineComponents). An instance of GUICompositeLineComponent is associated with one or more ModelAssociation objects that map the composite line to one or more semantic relationships represented in SSL objects on the server. The relationship between GUICompositeLineComponents and ModelAssociations is described in section 5.3.2.

**Composing a diagram**

Figure 5-4 illustrates an example of how the diagram editor uses GUIComponents to construct a simple object diagram. The diagram consists of two objects, the second having an inheritance relationship with the first. An instance association is also defined between the two objects.

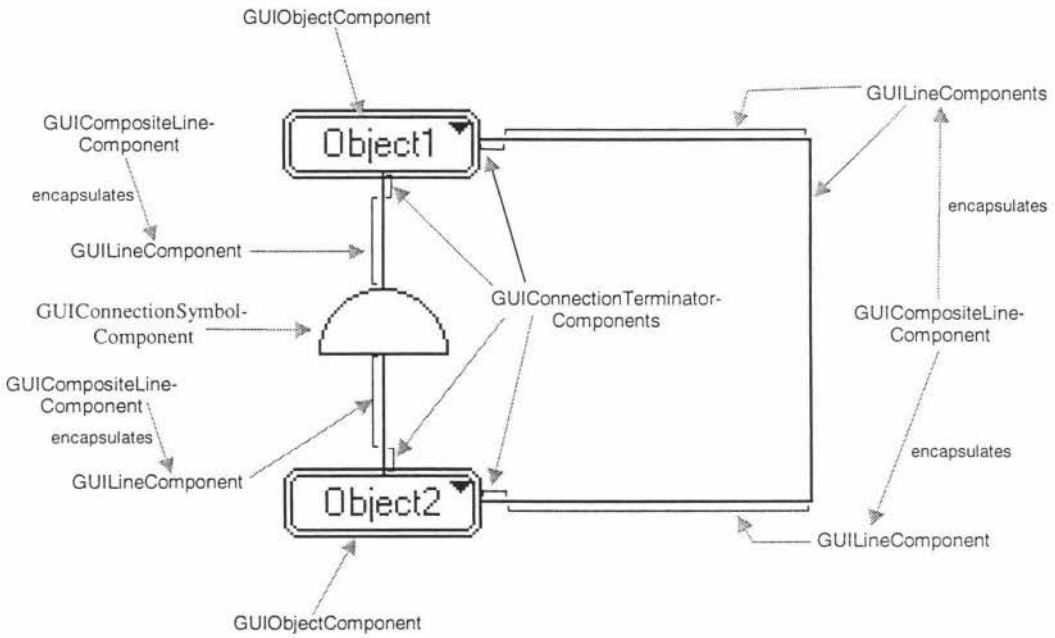



**Figure 5-4 – Composition of a diagram via GUIComponents**

## 5.2.2. ModelComponents

A diagram is described syntactically in terms of a collection of interrelated graph objects, specified by GUIComponents. However, at an abstract or conceptual level it can be described as a collection of concepts and relationships. The CASE Tool server maintains this semantic view of a diagram (and a model) as consisting only of concepts and relationships, and defines SSL objects to describe these. To assist the mapping between graph objects in a diagram and SSL objects describing the underlying semantics, the diagram editor employs a hierarchy of ModelComponents, as shown in

Figure 5-5. Each GUIComponent is related to a ModelComponent that it is conceptually represented by.

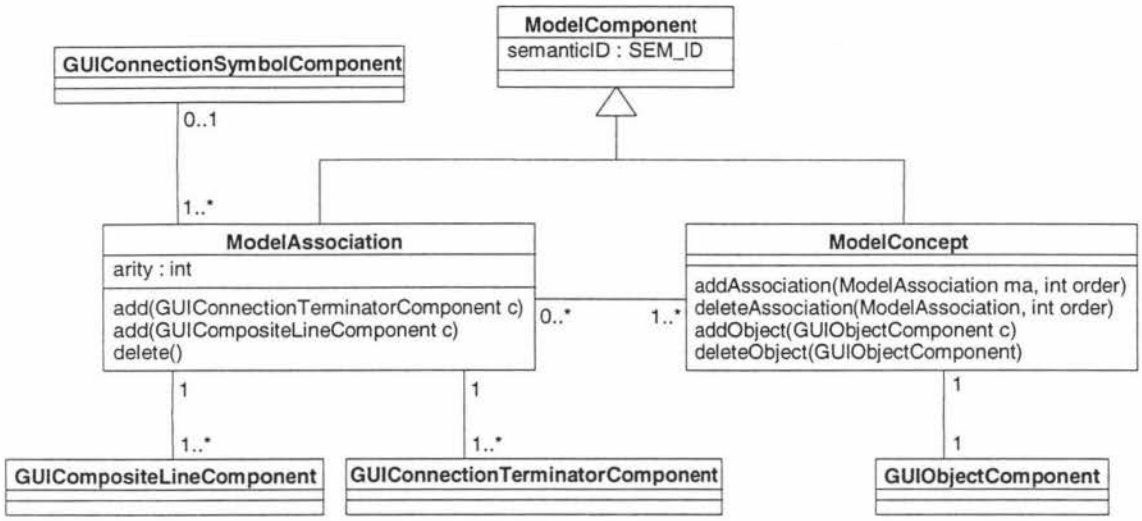



**Figure 5-5 – The ModelComponent class hierarchy**

The ModelComponent class defines an attribute *semanticID*. This is an ID used to identify a particular entity in a model to the server. Each unique entity in a user project has a unique ID. The ID is generated by the server, and is used to reference the entity during client-server communications.

The ModelConcept subclass is used to map GUIObjectComponents to semantic concepts. Multiple GUIObjectComponents that describe different views of a concept may exist in a diagram, model, or project. A single ModelConcept object is used to unify them as just different representations of the same thing. The ModelConcept maintains a reference to each of the GUIObjectComponents. Relationships may also be identified with concepts, and so a ModelConcept object also maintains references to any ModelAssociation that relates to the concept. GUIObjectComponent representations and ModelAssociations are dynamically added and removed from ModelConcepts during the construction or modification of diagrams by the methods implemented by the class.

The ModelAssociation subclass maps connection components and line segments into a single conceptual association. A ModelAssociation object maintains references to all the GUIConnectionTerminatorComponents and GUICompositeLineComponents that make up the association, as well as any GUIConnectionSymbolComponent that may be present in the connection. The maximum number of connection terminators and composite line segments is determined by the arity of the association. The arity of an

association is equal to the arity of the connection symbol where one is present. If no connection symbol is present, a default arity of two is assumed.

As an association always has to exist between concepts, a ModelAssociation object also maintains references to the ModelConcepts that it relates with. References are maintained in a list that is defined in the same order as the connection is drawn by the user. Ordering is important for directed relationships such as inheritance or whole-part.

**Composing a diagram**

Figure 5-6 illustrates an example of how ModelComponents are used in the diagram editor to describe a simple object diagram at a conceptual level. The diagram consists of three object components that are drawn in a grouped inheritance relationship. The concepts and associations that exist in the diagram are noted.

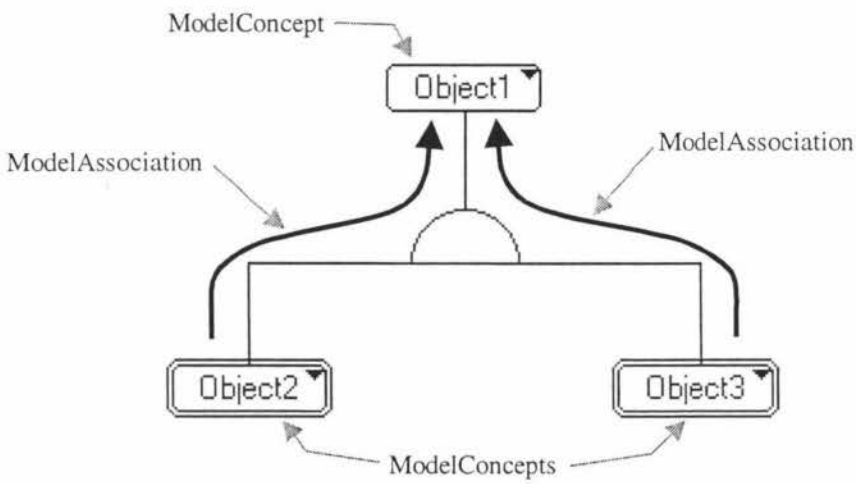



**Figure 5-6 – Composition of a diagram via ModelComponents**

## *5.3. Diagram Construction*

The process of constructing a diagram involves the creation, manipulation, and possible deletion of a number of GUIComponents and ModelComponents. This section describes how the diagram editor handles these construction tasks in terms of the sequence of events that it follows and the object instantiations (or uninstantiations) that it makes.

### 5.3.1. Creating Object Symbols

Object symbols are created in a diagram by the user selecting a drawing tool that describes the object symbol then "painting" on the drawing canvas with that tool. By painting on the canvas in this manner, the user adds objects symbols, and hence semantic concepts, to the diagram. Adding object symbols to a diagram increases the semantic knowledge of the diagram, however the diagram editor knows nothing about what the diagram is actually representing. All the diagram editor knows is a notation, and a limited set of rules that define how the notation can be used. SSL objects instantiated on the server maintain the semantic information about diagrams and models.

The addition of object symbols to a diagram therefore has to be reflected in the server by an SSL object describing the concept being added to the semantic model. To initiate this, the diagram editor sends a request to the server whenever the user wants to add an object symbol, notifying the server what type of object symbol it is. In some instances the server may reject these requests. When this happens, the diagram editor aborts the "add object symbol" operation and presents the user with a reason that is returned from the sever.

If the server can successfully instantiate the concept within the semantic model, it returns an ID to the diagram editor to be used in future reference to that concept. The diagram editor instantiates a ModelConcept object with this ID. The diagram editor also instantiates a GUIObjectComponent object with the notation template of the object symbol that is being added, and associates this GUIObjectComponent object with the ModelConcept. Finally, the diagram editor draws the new graph object on the drawing canvas (by way of the *draw* method defined in GUIComponent), and awaits further input.

In the present implementation of the diagram editor, a ModelConcept is represented by only one GUIObjectComponent. It has been identified that in future it should be possible to copy and paste multiple GUIObjectComponents representing the same model concept into the same diagram, across diagrams, or across models. Each GUIObjectComponent would have access to the same semantic properties but would represent a different view of the concept in a (potentially different) context. A semantic change made to one view could cause other views to change. Across models, GUIObjectComponents could also be rendered with entirely different notation templates. Due to the nature of the current prototype, this has not yet been investigated.

### 5.3.2. Creating Connections

The process of creating a connection is significantly more involved than creating an object symbol, at both the user interface level and the underlying generation of objects by the diagram editor. Objects can limit in some ways the connections that can be created through permitting only certain types of connections to be attached to certain docking areas, however in many (although not all) cases this is just a syntactic presentation issue and nonsensical connections can often still be established. For example, an inheritance connection attached at both ends to the same object symbol is generally easy to create, but has no semantic meaning.

To overcome the semantic issues of connection construction, the diagram editor must communicate with the server to determine if a connection specified by the user is valid in the context of the diagram. After the diagram editor has allowed the user to "paint" the connection they want, the diagram editor assembles the data that describes the connection. This data includes:

- the conceptual object symbols that are being connected, described in terms of the conceptual IDs stored by each of their associated ModelConcepts,
- the template name of any optional connection symbol that is drawn as part of the connection, typically used to identify the type of the relationship, and
- the template names of the connection terminators that attach to each object symbol, as these can also be used to identify the type of relationship, or to qualify it further over a connection symbol.

The ordering of conceptual IDs and terminator names is maintained in the same order as was specified by the user during the painting of the connection, as this information is important for directed relationships.

Once assembled, this data is then transmitted to the server as a request for the establishment of the association. If the request is declined, the server returns an explanation to the diagram editor, which is presented to the user. If, on the other hand, the request is granted, the server returns an ID to the diagram editor to be used for future references to that association. The diagram editor creates an instance of ModelAssociation with that ID. GUIComponents are then created to describe the connection painted by the user. The GUIConnectionSymbolComponent, GUIConnectionTerminatorComponents, and GUICompositeLineComponents that describe the connection are each associated with the ModelAssociation object. The ModelAssociation object also establishes bidirectional references to the ModelConcepts

that the relationship is between. Once this is complete, the diagram editor redraws the connection and awaits further input.

The current prototype of the diagram editor limits a ModelAssociation to having only one graphical representation. A future requirement would support multiple graphical representations to support where an association was copied between diagrams, for example.

### 5.3.3. Interacting with Graph Objects

Graph objects, described by GUIComponents, generally allow some type of interaction to be performed with them. In particular, graph objects described by GUITemplateComponents may contain active areas that define interactions specific to the graph object. The active areas supported in the current implementation of NDL allow actions describing the update of text areas or the transition of group templates within a graph object. These actions are described by Action objects that are contained within the View of the graph object.

When a user selects a GUITemplateComponent graph object, the diagram editor sends the graph object a message to determine the action associated with the specific point selected. If the point does not reside in an active area, there is nothing to do and the diagram editor takes no further action. If the point does reside in an active area, the graph object returns the Action object associated with that active area. The diagram editor examines the Action object to determine what action to invoke. The current actions supported are text-area-updates and group transitions.

Text-area-update actions cause the diagram editor to invoke a text editor, allowing the user to modify the contents of a text area. Once modifications are complete, the diagram editor sends a request to the server, informing it that the user has made a change. This notification is necessary as changes in the value of text areas typically causes the semantic meaning of the graph object to change. Changing the name of a class, for example, is a semantic change. However, the diagram editor cannot distinguish between semantically important and semantically unimportant text areas, as it knows nothing about them other than their ID. The ID of the concept or association (from a ModelComponent object) that contains the text area is included in the request with the updated text area value. If the new value of the text area is deemed valid by the server, the user action is accepted by the diagram editor and the viewable properties

of the graph object are updated. If the update is invalid, the user is presented with an explanation as to why, and the previous value of the text area is restored.

Group transition actions cause only a syntactic change in the diagram by altering the appearance of the graph object concerned. These actions do not add, modify, or delete any semantic information describing the diagram, therefore the diagram editor performs these actions without consultation with the server. The result of the action is the modification of the Group Status viewable property of the graph object to reflect the new group template that is to be used. The graph object is then redrawn with the new template.

Other than actions specified by active areas of GUITemplateComponents graph objects, only one generic interaction is supported by all graph objects. This is the relocation of graph objects to a different position on the drawing canvas. Repositioning graph objects is only a presentation issue, and does not alter the semantics of the diagram[1]. Since the semantics do not change, the diagram editor does not communicate position information to the server.

### 5.3.4. Deleting Object Symbols and Connections

The primary effect of deleting an object symbol or connection is the deletion of the graph objects that represent the entity. However, deleting an entity may cause knock-on effects. For example, deleting an object symbol causes any attached connections to be deleted also. Deleting a line segment that is part of a grouped connection would cause all of the connections in that group to be deleted.

**Deleting Object Symbols**

An object symbol is a particular representation of a model concept. A model concept may have many such representations. Deleting an object symbol does not delete the concept but rather deletes the instance of the representation from a diagram. When deleting an object symbol (a GUIObjectComponent), the diagram editor first removes any connections that are attached to it (as described in Deleting Connections, in this section). The diagram editor then removes the reference to the GUIObjectComponent contained in its associated ModelConcept object. If the ModelConcept contains no

[1] This statement is invalid when considering the interaction diagrams of UML and Booch (for example), as these diagrams do place semantic meaning on the relative position of graph objects. In the current prototype of the MOOT CASE Tool, position information of graph objects is not considered a semantic property, and so the semantic meaning behind these diagrams cannot be fully ascertained. This issue is yet to be addressed.

further references to other GUIObjectComponents, then the ModelConcept is also removed from the diagram. Before such semantic changes, however, the server is notified of the intent to remove the concept. The server may decline the request if appropriate, aborting the deletion process in the diagram editor.

As the current implementation of the diagram editor only ever creates one GUIObjectComponent for each ModelConcept, deletion of the component always results in the deletion of the concept also. In a fully implemented system, where concepts may be shared between anything from a single diagram to multiple projects, deletions may need to be clarified as to their overall effect. By performing a delete, a user may wish to delete:

- just a single representation of a concept from a diagram;
- the actual concept (and all its representations) from a diagram;
- the actual concept (and all its representations) from a model;
- the actual concept (and all its representations) from a project;
- the actual concept (and all its representations) from all the projects system-wide that use it.

Obviously, the effect to the system increases with each possibility, and so does the chance of knock-on effects to other models that use the concept. As this is a yet unexplored area, it is unclear exactly how the server would handle such requests, and whether they would be reversible.

**Deleting Connections**

Connections appearing in a diagram are syntactic representations of associations between concepts. As with concepts, associations may also have more than one such representation, and deleting a representation does not necessarily mean the association no longer exists. A connection is deleted by the user selecting to delete a graph object that is a line segment, a connection symbol, or a connection terminator that is part of the connection. If the graph object is a part of a group of connections, then each of the connections in the group are deleted.

The process of deleting a connection involves the deletion of the GUIComponents that describe it from the diagram. Reference to the GUIComponent is also removed from the ModelAssociation object that describes the conceptual relationship. If any GUIComponents are shared by other connections that are not selected for deletion, then those GUIComponents are left in the diagram and only the association with the ModelAssociation is removed. If, after the deletion, the ModelAssociation remains

with no representations, then it potentially can be deleted from the semantic model (although user confirmation may be required). As with all semantic changes, the server must be consulted if the user desires the ModelAssociation to be deleted.

### 5.3.5. Supporting Undo and Redo

In order to be reasonably useable, the diagram editor needs to provide a facility to undo and redo drawing actions performed by the user. The following questions were posed in determining how the diagram editor would support such a facility:

- How is a diagram described internally, in terms of GUIComponents and ModelComponents, and the relationships between them?
- What effect do create, modify, and destroy user actions have on the internal representation of a diagram?
- How can create, modify, and destroy user actions be reversed?
- How many levels of undo should be supported?

The last question posed here is the only one that deals with useability, and is perhaps is the most important from a user's perspective. In general, the more levels of undo that are supported by a system, the more comfortable a user may feel in exploring the system, knowing that no matter what they do they can always return to an earlier state. An infinite level of undo is desirable, but this is often limited by the memory resources available.

Considering that multiple levels of undo were deemed important, an approach to support this needed to be found. Recording user actions and their effects was considered as a possibility, however it would require a new class of objects to record every conceivable effect of the actions that a user could perform. Actions with a knock-on effect, such as a large delete, could be difficult to record and later reverse. The process of reversing actions is non-trivial in itself, as the reversing of actions with non-trivial effects typically needs to be performed in a particular order. For example, connections can only be recreated after the object symbols they associate with have also been recreated.

A simpler approach was therefore sought. The result was an approach that duplicates the current state of a diagram every time the user performs an action. The state of a diagram is defined by the state of all the GUIComponent and ModelComponent objects of that diagram. Each action performed by a user is carried out on a new copy of the diagram. This requires a two-step process, due to the interrelationships that exist

between individual GUIComponents, between individual ModelComponents, and between GUIComponents and ModelComponents.

The first step in duplicating diagrams involves creating duplicate instances of all the GUIComponents, ModelComponents, and Viewable Things in the diagram. During this step, a reference map is built. The reference map consists of pairs of references that each map an old object instance to its new instance. The second step involves using the reference map to update the references in each new object instance so that they associate with the new instances of other objects, rather than the old instances.

Two new methods were added to the GUIComponent and ModelComponent class hierarchies to support each step respectively. The first of these methods, *clone*, takes a CloneReferenceMap object as a parameter. This method creates a duplicate instance of an object, and adds a mapping between the old and new instances to the CloneReferenceMap object. The method returns the new instance. The second method, *resolveReferences*, takes the constructed CloneReferenceMap as a parameter. This method uses the map to update any references in the object.

To support undo, every time the user performs an action that alters the diagram in some way, the diagram editor saves the current state of the diagram on an "undo" stack before executing the action. A series of user actions causes a series of diagram states to be saved on the stack. To perform an undo, the diagram editor saves the current state on a "redo" stack, then restores the diagram state that is saved on the top of the "undo" stack. A redo action causes the reverse operation to be performed, that is, the current state is saved on the "undo" stack, and the diagram state stored on the top of the "redo" stack is restored to the display. The user is free to move back and forth between any diagram state in this manner. If the user performs a new action while in an "undone" state (ie. the "redo" stack is not empty), the contents of "redo" stack inherently become invalid and are discarded.

There is an efficiency issue that exists in this implementation. Although it is a reliable way to make sure all necessary detail is saved, a lot of unnecessary detail can also be saved. In a small diagram, saving the entire state is not a significant issue. However, in a large and complicated diagram it may well be. There is also a time issue to consider, as duplicating any diagram takes a finite amount of time. In tests, no delay has been noticeable, however the client interface has not yet been exposed to a "live" situation to determine if this approach is feasible in a real implementation.

As a final note, while undo and redo are fully supported by the diagram editor, it is unclear at this stage to what extent actions will be undoable in the server, if at all. Actions involving purely syntactic layout such as component positioning clearly are of no concern to the server and hence supportable, however any creation, deletion, or updating action requires consent from the server. Whether these actions can be undone has not yet been fully investigated by the project team. A delete action can cause the biggest problem, as while "recreate" can undo "delete", recreating a large amount of information after a large-effect delete becomes very sequence specific. Solving this problem, and hence supporting full undo facilities, is an ongoing task.

## 5.4. Maintaining Diagram Integrity

A graph object in a diagram is generally not just isolated entity, but relates to others around it. Individual graph objects of a connection such as line segments or connection terminators are all part of the connection as a whole, and the connection relates to the object symbols that it is attached to. From a purely syntactic perspective, graph objects are "joined" together, and movement of one graph object requires that others that are joined to it to follow. For example, connections attached to an object symbol must remain attached to the object symbol when it is relocated by the user on the drawing canvas. To support this, the diagram editor defines an interface GUIJoinableComponent that is implemented by GUIComponents that can be joined together (Figure 5-7).

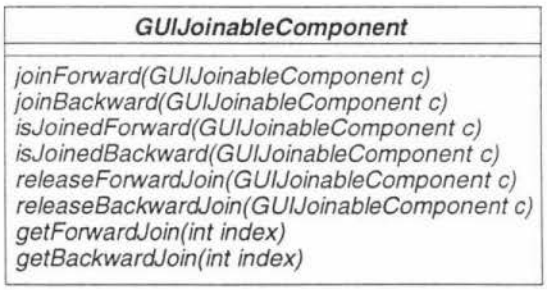



**Figure 5-7 – GUIJoinableComponent interface**

The GUIJoinableComponent interface defines a set of methods that allows physically joined GUIComponents to be logically coupled or decoupled, and allows a particular path of joined components to be followed in both a forward and backward direction. A forward join links to the next GUIComponent in a direction heading away from the source GUIObjectComponent of a relationship. In the example in Figure 5-8, a directed inheritance relationship between two classes is shown. The left-hand group of curved

arrows shows the sequence of GUIComponents that would be obtained by traversing forward though the component chain. The right-hand group of curved arrows shows the complementary sequence that would be obtained by traversing backward.

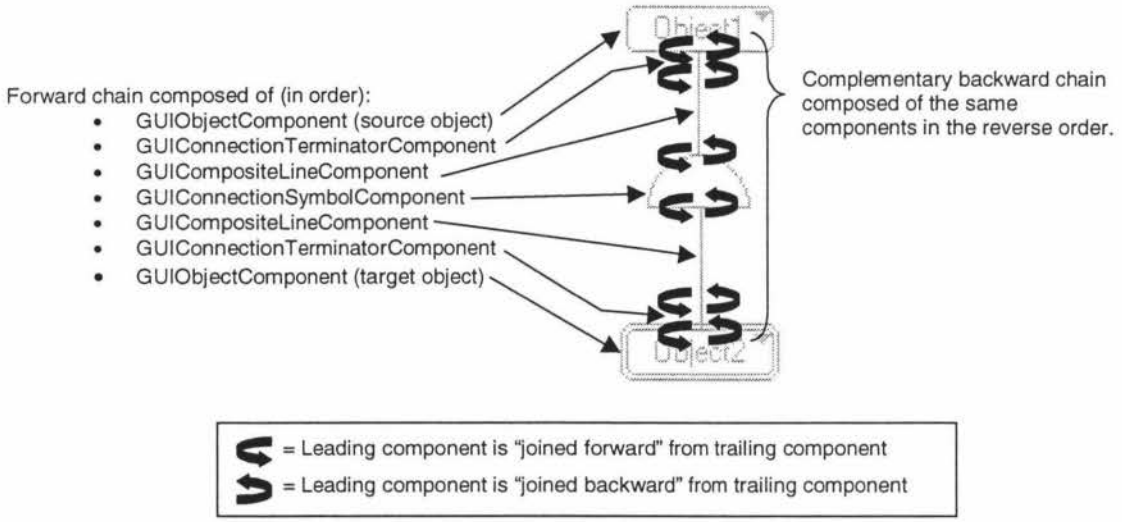



**Figure 5-8 – Forward and backward chaining of GUIComponents**

Within each GUICompositeLineComponent is a similar chain of GUILineComponents. The GUILineComponents are kept separate from the main component chain as the issue of their quantity in a particular representation is less important than the fact that the representation actually exists. A GUICompositeLineComponent acts as a GUILineComponent "manager" in that it controls the creation and deletion of GUILineComponents, ensuring that the endpoints of the composite line always have a continuous line drawn between them.

When any GUIComponent is moved, the system passes messages along any connection chain that the component belongs to, informing neighbouring components of the move. The range of the message passing along the chain depends on the type of component that was moved, and the number of GUIComponents that must be notified to effect the change.

When a GUIObjectComponent is moved, any GUIConnectionTerminator components that are attached to it are informed of the movement and are linearly relocated also. Each terminator that is relocated informs the attached GUICompositeLineComponent that the endpoint of the line is moving. The GUICompositeLineComponent passes that message to the first or last GUILineComponent (depending on the forward or backward direction of the chain messages) of the composition. The GUILineComponent then attempts to relocate itself by repositioning one or both of its endpoints. If both

endpoints are moved, then the next GUILineComponent in the chain is notified and repositioned. The flow of messages continues along the chain of GUILineComponents until no further adjustments are required. In some instances a line may attempt to relocate an endpoint but find it is unable to. For example, the endpoint may be connected to an immovable point docking area. In such cases, the GUILineComponent in question notifies the GUICompositeLineComponent that it is a part of. The GUICompositeLineComponent breaks the line into multiple segments to compensate. Figure 5-9 illustrates an example of how the system responds to a user request for movement in this case. Figure 5-9(i) shows the movement that is desired, while Figure 5-9(ii) shows the sequence of steps the system goes through to produce the result.

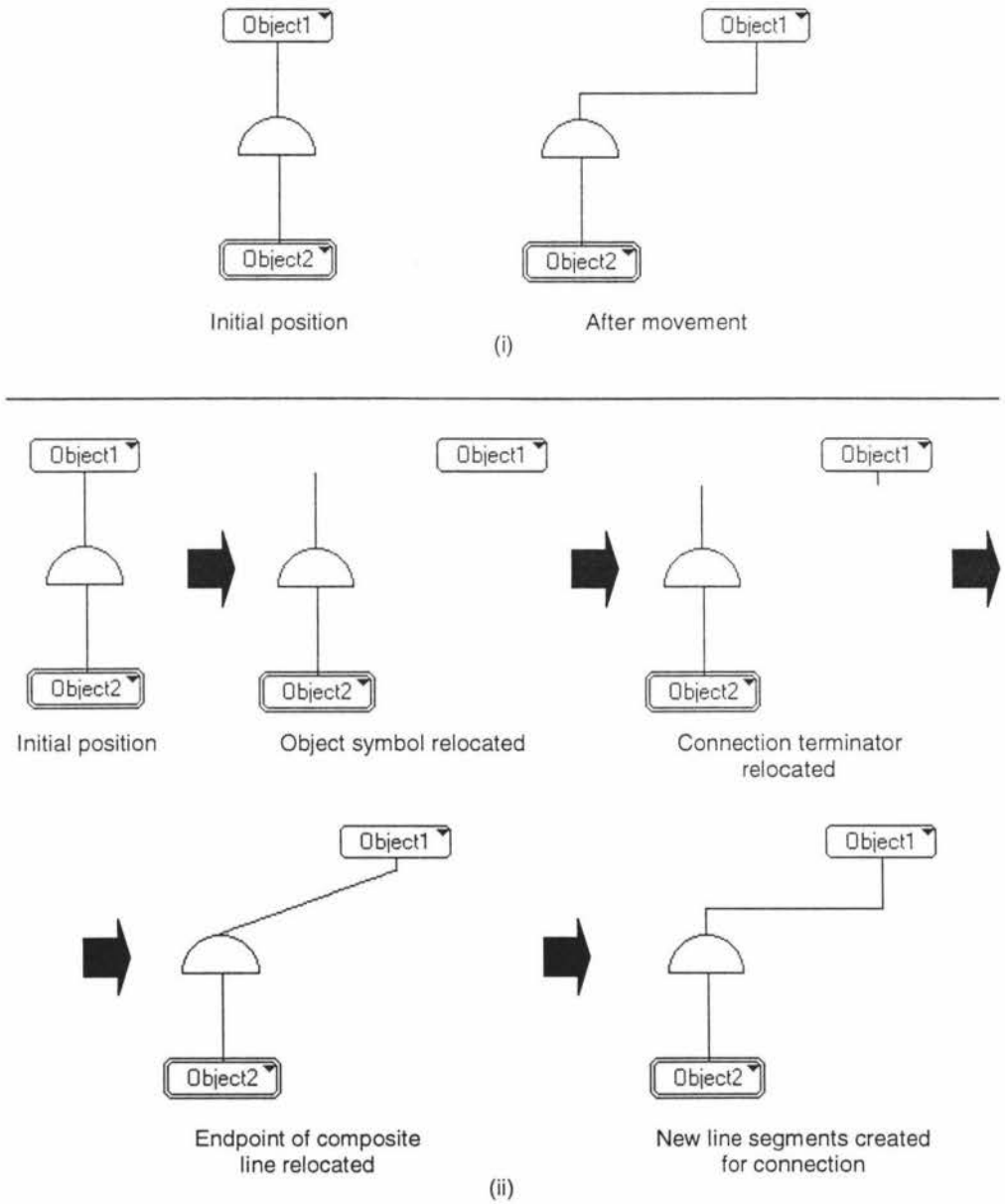



**Figure 5-9 – Movement of object symbols: (i) how the user sees it; (ii) how the movement request is processed as a chain of system events**

The movement of a GUIConnectionTerminator by a user results in a similar sequence of system events. The terminator notifies the GUIObjectComponent it is attached to of its new position and GUIObjectComponent records that information. The terminator also notifies the GUICompositeLineComponent attached to it, which performs the same reconstruction operation as described in the previous paragraph.

GUIConnectionSymbols have GUICompositeLineComponents joined directly to them. When a connection symbol moves, it sends messages directly to each attached composite line. The GUICompositeLineComponent as usual delegates as necessary the reconstruction of the line segments.

GUILineComponents may also be individually selected for movement. In such cases, relocation messages are passed along the chain of line segments, rather than the component chain. If a line segment at the end of the chain needs to pass further messages on, the GUILineComponent notifies its GUICompositeLineComponent "manager". The GUICompositeLineComponent can be attached to either a docking area on a GUIConnectionSymbol, or a GUIConnectionTerminator. If the line is attached to a docking area, an attempt is made to move the connection point to a different part of the docking area. If the line is attached to a GUIConnectionTerminator, an attempt is made to move the terminator to the new position. If in either case the movement is not legal (eg. it would result in the connection becoming disconnected from the docking area), the GUICompositeLineComponent denies the GUILineComponent's request of relocation. Denial causes the whole movement operation to be denied, and no movement takes place.

# Chapter 6

# Design of the Client Graphical User Interface

Modern CASE tools are highly interactive graphics-intensive environments. The software engineer's user interface of a meta-CASE tool such as MOOT is a graphical shell that is parameterised by the notations of the models that are supported by software development methodologies. During the development of the graphical user interface (GUI) for the MOOT CASE Tool client, consideration must be given to all graphical notations that may be possible. No assumptions about the types of notations that a software engineer may use can be made. Developing at such a high level of abstraction causes a loss in design flexibility, as the GUI cannot be tailored to one particular method. It introduces additional challenges in terms of designing effective human-computer interaction mechanisms.

This chapter begins by describing the initial analysis of the software engineer's graphical user interface requirements that was undertaken to determine what should be supported in the initial prototype. The design of the client GUI shell that has been subsequently developed is then described in detail. The main focus of this chapter is aimed at the diagram editor interface component, as this is the most interesting and complex. The other supporting interfaces, such as the project manager dialogs, are touched on briefly.

It should be noted that all the figures appearing in this chapter, unless otherwise stated, are produced from screen captures taken from actual use of the GUI developed.

## 6.1. Analysis of GUI Requirements

The analysis of the GUI requirements is based around the construction and modification of object-oriented (OO) model diagrams, as this is an integral part of any OO methodology[1]. The user interface is based around a sketchpad-like drawing canvas. In

[1] Only OO methodologies have been considered in this prototype as this is the main focus of MOOT. The ultimate aim of MOOT is to support all software engineering methodologies.

this area, the user builds graphical structures that are comprised of a limited number of predefined notation elements. The structures are used to model the system under development, and may represent such things as classes and objects, and the relationships between them.

For each software engineering methodology there are well-defined rules concerning the appearance of each type of notation element, constraints on them, and the ways they can be joined together. These rules are described by a notation specification written in NDL. The goal in designing the GUI is to allow users to place, join and manipulate these elements quickly and easily. The tool must fully support all aspects of the methodology, and must enforce the rules of the methodology (eg. by not allowing illegal connections).

Issues relating to the functional and non-functional requirements for the diagramming component of the GUI have been examined in detail, and are discussed in sections 6.1.1 and 6.1.2. The particular design issues that relate to the implementation of the initial prototype are identified in section 6.1.3. The time available for completion of the thesis was insufficient to examine other components of the user interface, such as the project managers. In comparison to the diagramming editor, however, these other components are uninteresting and follow more orthodox approaches in their design and implementation.

### 6.1.1. Functional Requirements

An analysis of the tasks involved in the construction and modification of model diagrams has been undertaken, and expressed as set of use cases. The analysis produced the following set of broad functional requirements. The user must be able to:

- *Create objects symbols on the drawing canvas.* A notation defines a set of object symbol types. Once an object symbol type has been selected, multiple placements must be possible. It must be possible to enter properties for the current object symbol. It must also be possible to copy object symbols from a repository, and paste them onto the canvas.
- *Make changes to existing object symbols.* Objects symbols must be selectable either singly or in groups. It must be possible to move, copy, paste and delete either individual object symbols or groups of object symbols. Knock-on effects associated with deletion must be detected by the system, and the user prompted for further input if necessary. It must be possible to rename or resize an

individual object symbol, alter its properties, and have its properties displayed on demand.

- *Create connections between object symbols on the drawing canvas.* A notation can define different types of connections. The creation of complex or grouped connections between three or more artefacts must be supported[2].
- *Make changes to existing connections between object symbols.* It must be possible to rename or reshape a connection, to alter its properties (including its cardinality), and to display the properties on demand. It must also be possible to delete connections. Any side effects associated with deletion must be detected by the system, and the user prompted for further input if necessary.
- *Edit diagrams.* It must be possible to re-arrange, annotate, and partition diagrams. A range of general editing functions is also required, including UNDO and REDO.
- *Passively browse diagrams (selectively view and manipulate them at various levels of abstraction).* This includes moving, zooming, and rotating diagrams, in addition to hiding detail.

### 6.1.2. Non-Functional Requirements

It is not enough for the system merely to provide the above functions. The user must be able to carry them out efficiently, flexibly, and easily, with particular reference to:

- the ease of choosing and changing mode;
- the ease of initially placing graph objects on the canvas;
- the ease of selecting graph objects for manipulation;
- the ease of scaling (resizing) graph objects;
- the ease of relocating graph objects, both within a diagram boundary, and across diagrams (the tool must be capable of maintaining diagram integrity);
- the ease of manipulating names, including the assignment of names to graph objects, the scaling of names, the renaming of graph objects, and control over the location of a name on the canvas;
- the ease of viewing diagrams, and selecting viewing options, including the hiding of information, (e.g. message connections);
- the ease of scaling diagrams and portions of diagrams;
- the ease of navigating (paged) diagrams.

---

[2] The term *complex connection* is used to refer to a single relationship between three or more object symbols. The term *grouped connection* is used to refer to multiple relationships that share common connection elements on the drawing canvas, such as the tree-like inheritance structures that can be constructed in the Coad and Yourdon and UML notations.

More generally the system must support:

- the prevention of errors and ease of error recovery, including multi-level UNDO and REDO;
- flexibility of operation;
- consistency of operation across tasks;
- efficiency and spareness of dialogue;
- task conformance, especially for the most common tasks.

The system must also provide:

- an effective help system;
- clear, well-structured and pleasing screens;
- quality feedback at each stage.

Additionally, the system must show clearly at all times: the current mode, the current working position, potential artefacts for selection, the current artefacts being manipulated, and the currently available options.

### 6.1.3. Design Issues

The time available for the development of the user interface was not sufficient to embrace all design issues that were uncovered in sections 6.1.1 and 6.1.2. A smaller set of critical design issues that relate to the look and feel of the diagramming interface was selected for exploration in the initial prototype. These issues are grouped into the following categories: dealing with modality, manipulating objects, manipulating text, and making connections.

**Dealing with Modality**

The interface is required to support the selection of drawing tools (modes) from a palette of graph objects without the user having to shift the mouse cursor from their point of focus on the drawing canvas. The current drawing mode needs to be clearly conveyed to the user. This includes the need for the mouse cursor to carry information on the current drawing mode, and for it to signal potentially illegal operations associated with that mode. For example, a user that is constructing a connection should be made aware if a particular construction is illegal in the current context.

**Manipulating Objects**

The initial placement of graph objects on the drawing canvas, including multiple placements of the same type of graph object, needs to be simple and direct. The

selection and 'picking up' of graph objects for manipulation also needs to be easy and direct. Candidate graph objects that may be selected in the current drawing mode or context should be clearly shown as the mouse cursor is moved over them. Multi-level support for UNDO and REDO operations in connection with diagram construction should be supported. The logical distortion of object symbols, for the purposes of data hiding, needs to be simple and intuitive.

**Manipulating Text**

The initial entry, and subsequent manipulation, of text associated with graph objects needs to be simple and direct. The automatic scaling of objects to accommodate dynamic changes in text height and width should be supported.

**Making Connections**

The construction of complex or grouped connections involving three or more object symbols needs to be simple, intuitive, and direct.

In general, there is a need to make the dialogue efficient and spare, with point-and-click or drag-and-drop methods for most operations. The use of dialog boxes should be kept to a minimum, and only where essential. Keyboard accelerators for common tasks should also be provided, particularly to make use of a user's normally dormant "free hand" (assumed to be the left hand in this prototype).

## 6.2. Design Overview

In order to explore the subset of design issues that were considered for the prototype, an NDL notation specification that describes the Coad and Yourdon OOA model (Coad and Yourdon, 1991a) has been written[3]. This provided a basis for the development and testing of the interface, and allowed reviews and refinements to be added to the final design.

### 6.2.1. Interface Basics

The diagramming interface is based around a sketchpad-like drawing canvas. In this area the user (a software engineer) builds graphical structures comprised of a limited number of predefined notation elements. These notation elements are defined in an NDL notation specification. The notation elements are presented to the user via a

[3] This specification appears in Appendix D.

toolbar that appears along the top edge of the canvas. The toolbar provides a palette of the basic drawing tools for diagram construction. Related drawing tools are grouped together in the toolbar, as this is known to enhance visual coherence resulting in an interface that is easier to comprehend (Constantine, 1996).

In this prototype, the toolbar is constructed from a default selection tool and the object and connection symbols that exist in a notation specification. A connection tool that allows the creation of a connection with no connection symbol is also created by default. The toolbar buttons that represent notation elements are not predefined in the interface but are dynamically drawn at run-time directly from views of the templates in the notation specification. Tools describing connections are identified by a small connection icon in the top-left corner of each button icon. Figure 6-1 illustrates the toolbar that is generated from the Coad and Yourdon notation specification. The selection tool is the currently active tool, as shown by the button appearing pushed in.

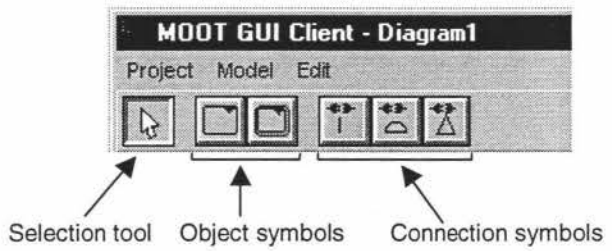



**Figure 6-1 – Toolbar for the Coad and Yourdon OOA methodology**

### 6.2.2. Drawing Tools (Modes)

Before diagram construction can begin, the appropriate drawing tool must first be selected. The GUI provides several different ways to do this:

- The user may select a tool directly from the toolbar by clicking with the mouse. This requires that the mouse cursor be moved away from the drawing canvas and the user's current area of construction.
- The user's free hand (assumed to be the left hand in the prototype) can be used to cycle between the different drawing tools while retaining the mouse cursor at the current point of working. The TAB key on the keyboard can be used to cycle forward between all the different drawing tools. SHIFT-TABBING can be used to cycle backwards. The ESCAPE key can also be used at any time to return to the selection mode.
- By pressing the right mouse button on an empty space on the drawing canvas, a pop-up menu version of the toolbar appears. This menu contains textual descriptions of the toolbar buttons. Figure 6-2 shows the pop-up menu that

relates to the toolbar shown in Figure 6-1. The names for the object symbols and connections appearing on the sub-menus are those given to the templates in the notation specification. The connection tool that is created for the construction of connections with no connections symbol is given the name "Instance".

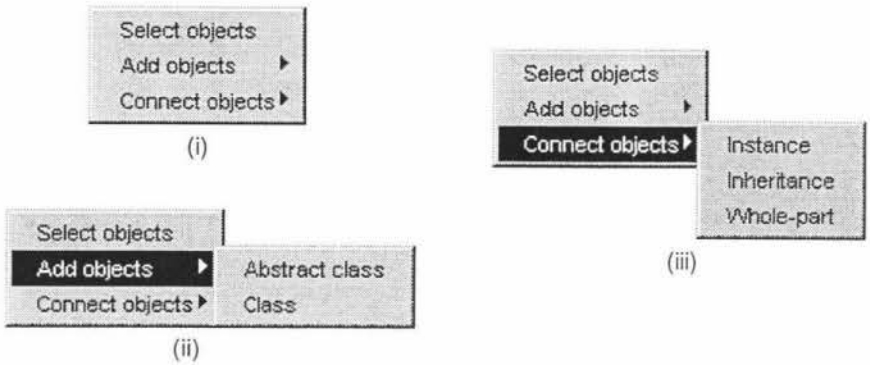



**Figure 6-2 – Pop-up menus for selecting drawing tools: (i) the main menu; (ii) the "Add Objects" sub-menu; (iii) the "Connect Objects" sub-menu**

Information on the current drawing mode is conveyed to the user in several ways. The toolbar always shows the current mode by drawing the appropriate toolbar button in a depressed state. This rule applies independent of the method a drawing mode is selected. Different modes are represented by different mouse cursor shapes, and in the object symbol and connection placement modes the drawing tool is also carried on the mouse cursor in a ghosted form. Figure 6-3 illustrates some examples of how different modes are represented.

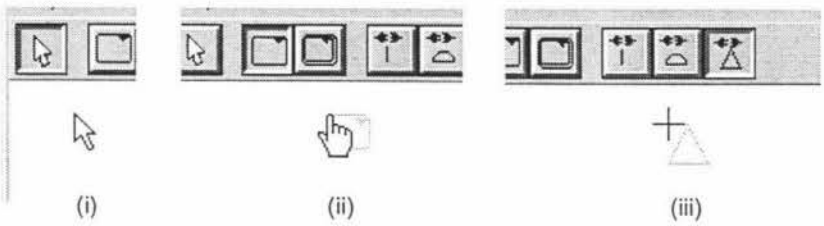



**Figure 6-3 – Conveyance of drawing modes via toolbar and mouse cursor in (i) selection, (ii) object placement, and (iii) connection placement drawing modes.**

## 6.3. Object Symbols

Object symbols are the collection of graph objects that represent individual concepts of a model, including such things as classes, objects, processes, data stores, and so on. Object symbols are grouped together on the tool palette, as shown in Figure 6-1. New object symbols are added to the drawing canvas (and to the model) by selecting the appropriate tool and then clicking on the drawing canvas. This "stamps" the object symbol onto the canvas at the location selected. The ghosted image that follows the mouse cursor provides feedback indicating exactly where placement will occur.

Following placement of an object symbol, the drawing mode is cancelled and the tool reverts to the selection mode. If desired, the drawing mode can be retained by holding down the SHIFT key. This allows multiple object symbols to be placed quickly. Releasing the SHIFT key after one or more placements have been made again causes the selection mode to become active.

### 6.3.1. Auto-highlights and Selection

Graph objects need to be in a selected state prior to the user performing tasks such as relocation, updating, and deletion (the object-action model (Shneiderman, 1998)). Graph objects automatically become selected when they are first placed on the canvas. The GUI facilitates the subsequent selection of candidate graph objects by highlighting the possible selections that can be made as the mouse cursor moves over them[4]. The types of graph object that are highlighted depends on the current drawing mode. Figure 6-4 shows an example of this auto-highlighting, where the mouse cursor is moved across an object symbol.


**Figure 6-4 – Auto-highlighting of a graph object**

All graph objects are selected by clicking with the mouse. Selected objects are highlighted in the same manner as an auto-highlight except in a different colour. The user cannot select objects that are not appropriate to the current drawing mode. In this prototype only one graph object can be selected at any one time, ie. group selections are not supported. Any currently selected graph object can be deselected by either selecting

[4] An option to turn off this feature is included in the implementation of the tool.

another graph object, or clicking on a blank area of the canvas. In the latter case, no new selection is made.

## 6.4. Interacting with Active Areas

Most object symbols defined in a notation specification have one or more active areas associated with them. NDL supports two types of actions for active areas: text-area-updates and group transitions. Handling of both of these has been implemented in the prototype interface.

Active areas of graph objects may be "activated" by the user to cause the action associated with the area to be initiated. In this prototype, the user activates active areas by double-clicking with the mouse while the mouse cursor is within the desired active area. Double-clicking was chosen as the user input sequence to distinguish between activation actions and selection actions (single mouse clicks). The user may also initiate actions by using a pop-up menu. Pop-up menus are opened by right-clicking with the mouse. Pop-up menus are context sensitive in that different menus may appear depending on where the user right-clicks. Figure 6-5 shows examples of the some of the different pop-up menus that may appear.

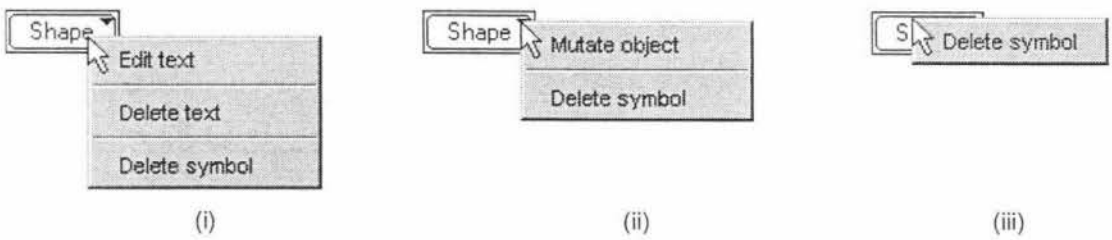



**Figure 6-5 – Context-sensitive pop-up menus appearing on: (i) text-area-update active areas; (ii) group transition active areas; (iii) non-active areas of an object symbol**

### 6.4.1. Text Area Updates

An active area describing a text-area-update action initiates a request to edit the text contained in the text area when it is activated. The area can be activated either through double-clicking on the area, by a pop-up menu as described above, or through the specification of a default text area in the template definition of the graph object. If a graph object's template definition specifies a default text area, pressing any character key on the keyboard while that graph object is selected has the effect of activating a text-area-update action for that text area.

Selecting an active area describing a text-area-update action initiates a request to edit the text contained in the text area. In the case of a text area activated via default text area keyboard input, the action request instead becomes "replace text". Several different types of text editors have been implemented and examined during the phase of GUI development to determine the most efficient and visibly pleasing method of input.

The first text editor employs the use of a separate dialog box to acquisition a line of text. The dialog box contains a text-edit field, an "Okay" button, a "Cancel" button, and some text describing what the dialog was for. In practical use, this proves to be rather intrusive and distracting, and is quite slow and cumbersome to use when multiple text entries need to be made in succession.

The second text editor scales down the dialog box to a small window that floats over the drawing canvas at the site of the active area selected. The window contains enough space on one line only for a text-edit field and two small buttons for "Yes" and "No". While this is a lot quicker in use than the previous dialog box, it is still awkward to use on multiple-line text areas. Each line of a multiple-line entry must be made individually as distinctly separate actions.

The third version of the text editor eliminates any additional dialog completely. This editor overlays itself directly onto the text area of graph objects. The only visual cue as to the presence of the editor is a flashing cursor at the point of text insertion. As characters are typed (or deleted) the graph object containing the text area is dynamically updated and redrawn.

Implementation of this last text editor was by far the most difficult of the three. The first two made use of a text-edit field that is provided as a standard component in Java (the implementation language). The third could not, as the standard Java component did not provide the necessary features to be able to be used in the manner required. This meant that the on-screen editor had to be designed and implemented from scratch, handling cursor display and movement, and text insertion and deletion in its own way. The present implementation of the text editor supports the following features:

- Basic cursor movement. The LEFT and RIGHT keys move the cursor left and right, respectively. The UP and DOWN keys move the cursor between lines in multiple-line text areas. The HOME and END keys move the cursor to the beginning and end of the current line, respectively.
- Advanced cursor movement. With the CONTROL key held down, the left and right cursor keys move a word at a time, moving to the previous or next line as

necessary. CONTROL-HOME and CONTROL-END move to the beginning and end of the text area respectively.

- Text insertion. All text is inserted at the cursor. No overwrite mode is implemented. The DELETE key deletes the character at the cursor, while the BACKSPACE key deletes the character preceding the cursor.
- Multiple line construction. CONTROL-ENTER breaks the current line into two lines at the point of the cursor. CONTROL-BACKSPACE joins the current line with the previous line. This latter action is only permitted if the cursor is in the first column of the current line. Both of these actions have an effect only on multiple-line text areas.
- Termination of editing mode. The ENTER or ESCAPE keys terminate the text editing mode. The ESCAPE key can be used to cancel the text update, returning the original value to the text area if the text editor was activated as the result of a key press activating the default text area of the graph object[5]. The text-editing mode can also be cancelled by selecting an area outside the current text area with the mouse.

The selection of text, and facilities to cut, copy and paste text are *not* supported in the current implementation.

Figure 6-6 shows an example of the on-screen text editor. In this example, the operations of a Coad and Yourdon Class-&-Object symbol are being entered. First, the text area is selected for text entry. The operations *draw* and *move* are then entered. A blank line is then inserted between the two existing lines (by a CONTROL-ENTER keystroke), and a third operation, *erase*, is then entered on the blank line.

This text editor is used in the current implementation of the prototype. Assessment of it in use is positive, particularly on multiple-line text areas where the other text entry methods failed to perform adequately. It is remarkably unintrusive, in that it does not present the user with any more information than they need to perform the required task (the "less is more" principle (Nielsen, 1993)).

[5] If the text editor is started through an explicit active area selection, the original value of the text area can be restored via the UNDO method described later. This apparent inconsistency stems from the generally different circumstances (under normal use) where the different methods of activation would be used. The auto-restore-on-ESCAPE feature also provides a quick recovery mechanism when a key is inadvertently pressed while a graph object with a default text area is selected.

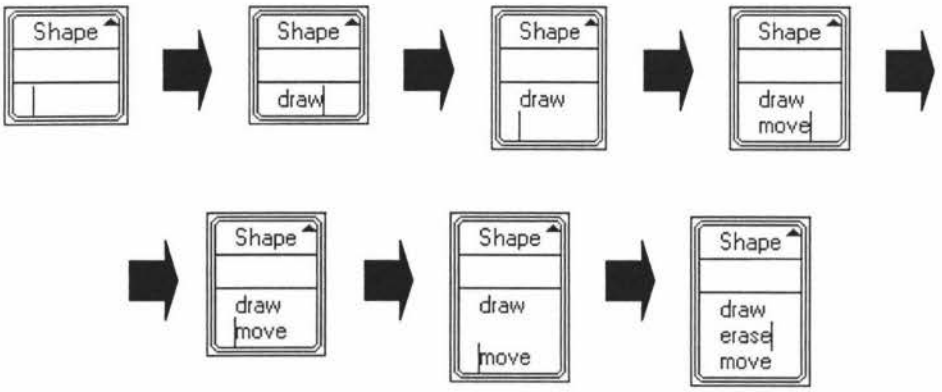



**Figure 6-6 – Text editing via a custom on-screen editor**

Complete lines of text can be deleted from a text area either manually by editing the text area, or through the pop-up menus (Figure 6-7). The pop-up menus for text areas contain an option "Delete Text", which deletes the line of text at the location where the pop-up menu was opened. In the case of multi-line text areas, any lines of text below the line deleted are moved up.

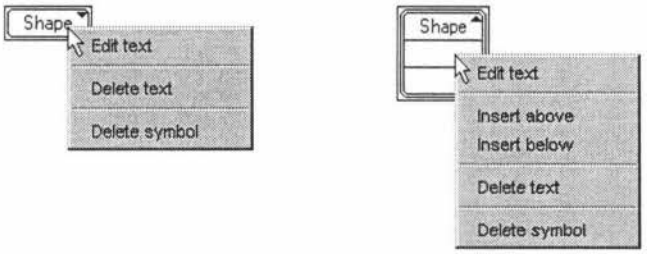



**Figure 6-7 – Pop-up menus on single-line and multi-line text areas**

The pop-up menu for multi-line text areas also contains options for inserting blank lines above and below the selected line. This causes the same action as if CONTROL-ENTER was pressed at either the beginning or end respectively of that selected line. These options were implemented originally for the first two text editors, which required these options to increase (and decrease, with delete) the size of the text area. They are provided still purely as an alternative means of editing for the user.

### 6.4.2. Group Transitions

Group transition active areas are used to alter the visual appearance of a graph object. Selecting the active area causes the GUI to rebuild the template view of the graph object using a different group template. This does not change any information contained in the graph object, but may cause more, less, or just different information to be displayed.

Unlike text areas, which generally contain visible text the user expects to be able to change, active areas associated with group transitions need clear indications of their presence through graphical icons. Without these, a user would be unaware of their existence. Ideally, the icons should portray some clue to the user as to what action their selection would cause. Figure 6-8 illustrates an example where small arrows in the top right-hand corner of a class symbols are used to indicate that the symbol can change in size.

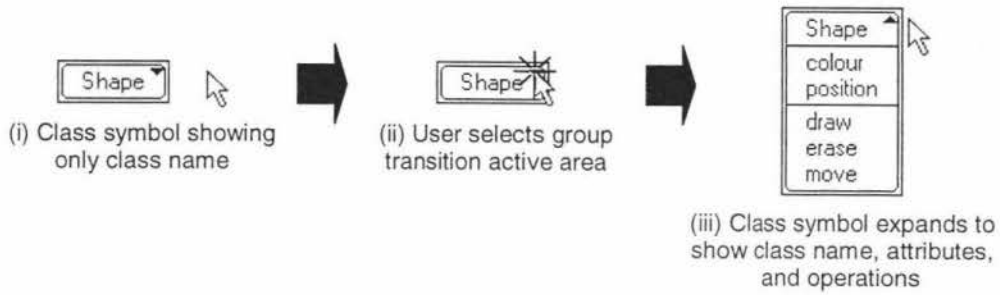



**Figure 6-8 – Group transitions on a class symbol**

## 6.5. Connections

Connections are used to represent relationships that objects may have with themselves or each other. Although a methodology notation can describe relationships in many different ways, construction and manipulation of connections in the user interface follows the same basic principles.

### Creating Connections

The sequence of actions a user goes through to create a connection is aimed at being as simple as possible while still maintaining a useful level of functionality. Much effort has been placed into automating intuitive aspects of their construction, and providing feedback such as connection previews or selection legality.

The connection mode is initiated by selecting a connection tool from the toolbar or pop-up menu (Figure 6-9(i)). To create a connection, the source object symbol is first selected with the mouse (Figure 6-9(ii)). This "hooks on" a connector to the object symbol. Moving the mouse away from the object symbol creates a connecting line between the closest valid docking area on the object symbol and the mouse cursor, bowing to any constraints such as being permitted only horizontal and vertical lines[6] (Figure 6-9(iii), (iv), (v)). If no docking areas on the object symbol allow connections

[6] In this prototype, only horizontal and vertical lines are implemented for simplicity.

in the direction from the object symbol to the mouse cursor, then no connecting line is drawn.

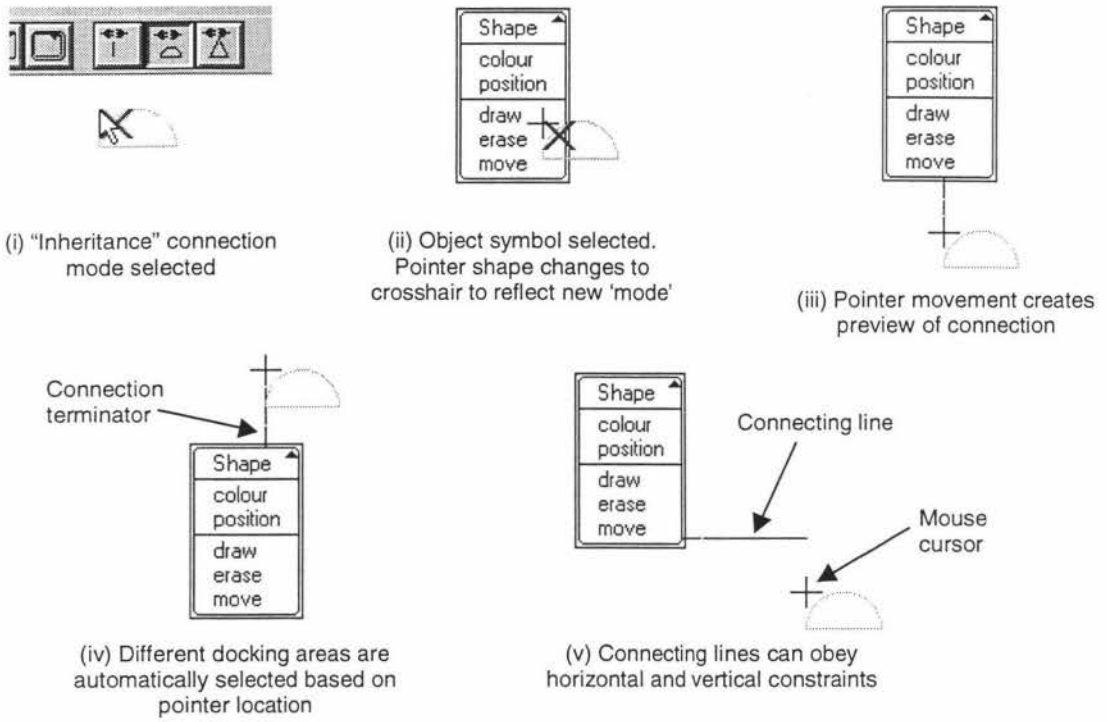



**Figure 6-9 – Creating connections**

Between a connecting line and where it connects with each object symbol is also drawn a connection terminator, as identified in Figure 6-9(iv). Connection terminators are used to identify or qualify particular types of relationships. As there are no toolbar buttons to change the terminator type, a default terminator is chosen for all new connections. In this prototype, the default is the first connection terminator defined in the notation specification. Although not currently implemented, it is expected that a user would be able to change the default terminator during use of the tool.

While creating connections, the user is constantly made aware of illegal selections by the appearance of the mouse cursor. If a particular location is not selectable in the current context, then the mouse cursor is annotated with a large red cross. Figure 6-9 illustrates two examples. In Figure 6-9(i), the mouse cursor location is not selectable as it is not within the bounds of a graph object that can accept a connection. The connection process can only be started from object symbols (and connection symbols, as described later in this section). In Figure 6-9(ii), the mouse cursor location is not selectable as no docking area on the object symbol can create a connecting line towards that direction. Another situation that is illegal is trying to attach the end of a connection

to a graph object that cannot accept a connection (eg. a connection terminator or connecting line segment).

Once a connecting line has been stretched from an object symbol, it can either be directly connected to another object symbol (if there is no connection symbol to place, see below), or anchored to create a "corner". Multiple line segments can be placed by clicking the mouse at the desired locations for each corner, as shown in Figure 6-10. As the prototype implements only horizontal and vertical lines, successive line segments are drawn perpendicular to the last.

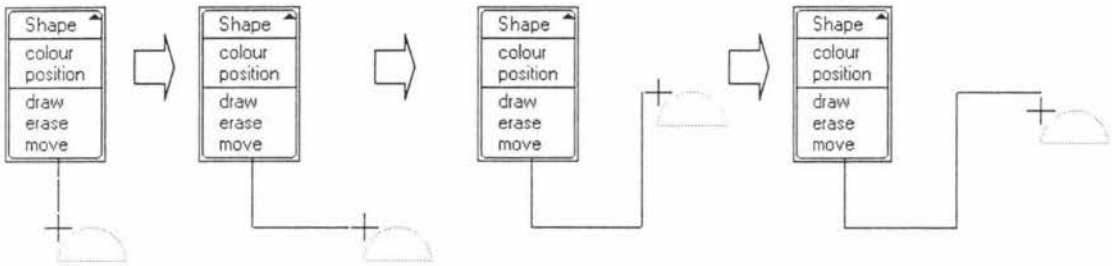



**Figure 6-10 – Creating corners on connecting lines**

If the connection being constructed requires a connection symbol, this must be placed before the tool will permit the connection to be completed. A connection symbol can be placed in the same manner as creating a corner, except by using a double mouse click. This attaches the connection symbol to the end of the last line drawn, correctly rotated as necessary. Figure 6-11 illustrates some examples of a connection symbol drawn at different angles. This figure also illustrates the automatic extension of "extendable" line docking areas. In the first and third examples, the base line of the half-circle connection symbol has been extended to accommodate the line segment stretched from the connection symbol to the mouse cursor. This extension is fully automatic, the length of the extension line being adjusted dynamically to accommodate connections from it.

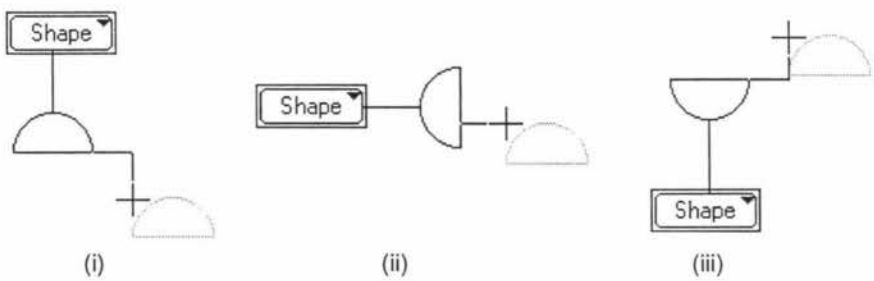



**Figure 6-11 – Connection symbol rotation**

During connection construction, it is possible to undo the placement of connecting line segments and connection symbols by pressing the BACKSPACE key. This removes the the most recent addition to the connection construction. If the whole connection is no longer wanted, the operation can be cancelled completely by pressing the ESCAPE key. Connections can be undone or cancelled in these ways up until the point at which they are completed. After this time, they may be undone as described in section 6.8.

To complete a connection, a connecting line segment must intersect an object symbol on an available docking area. When this situation occurs, the line "hooks on" to the object symbol, and the GUI draws a preview of the connection, shortening the line to fit a connection terminator, as shown in Figure 6-12. If the connection cannot hook onto the object symbol at the specified point, a red cross annotation appears on the mouse cursor and no connection terminator is drawn.

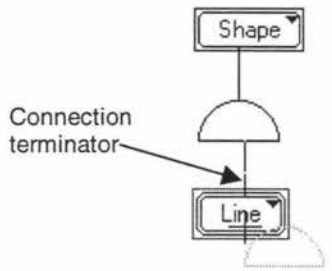



**Figure 6-12 – Preview of a complete connection**

To accept a valid connection, the mouse button is pressed. If the connection represents a binary relationship, the connection process is complete and the connection is added to the diagram. If the connection contains a connection symbol with an arity greater than two, the tool repeatedly asks for connections to be made from the connection symbol until the required number of object symbols have been associated. In this instance, using the BACKSPACE key to undo all intermediate connections is fully supported. Figure 6-13 shows an example of a ternary association, as might be used in a relational database, being created.

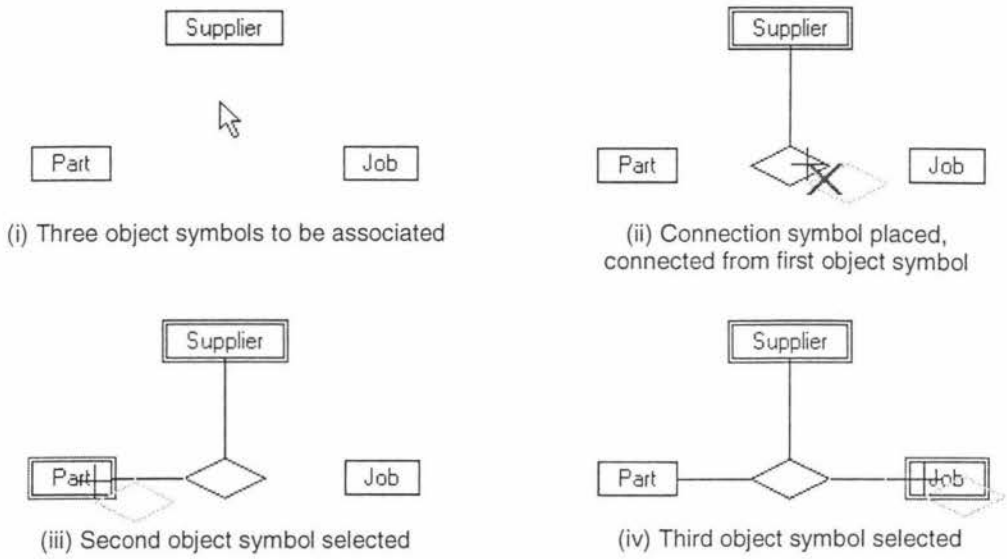



**Figure 6-13 – Creating ternary associations**

After a connection has been successfully completed, the tool returns to the selection mode. It is possible to retain the connection mode by holding down the SHIFT key while completing connections (*cf.* multiple placement of object symbols). If the connection mode was one that did not require a connection symbol, then the state of the interface returns to requiring a source object symbol to connect from. Conversely, if the mode did require a connection symbol, then new connections automatically branch from the connection symbol placed in the first connection, provided that the symbol can still accept more connections (ie. not all docking areas are "full"). Figure 6-14 illustrates an example of creating multiple grouped inheritance connections.

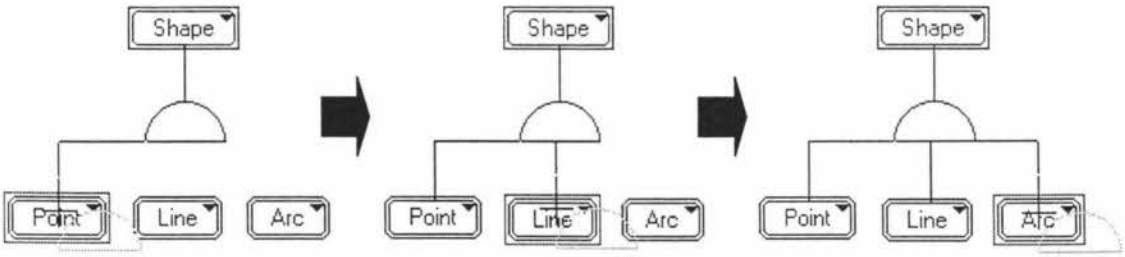



**Figure 6-14 – Creating multiple grouped connections**

Connections can also be added to connection symbols (that can accept further connections) at a later time. The context-sensitive pop-up menu that appears upon a mouse right-click on connection symbols includes an option to add a new connection (Figure 6-15). This option is greyed out (disabled) if no more connections can be made from the connection symbol. Double-clicking on the connection symbol in the selection mode also activates the connection mode. Connections are continued from the

connection symbol in the same manner as was described for creating multiple connections with the SHIFT key in the previous paragraph.

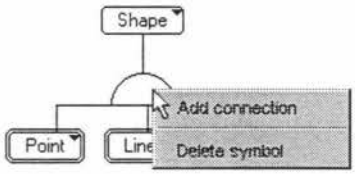



**Figure 6-15 – Context-sensitive pop-up menu for connection symbols**

### Changing Connection Terminators

It is possible for the user to change the connection terminators on the end of connections. Changing a connection terminator can change or add to the meaning of the connection. In line with the design philosophy of the interface, the process of changing connection terminators should be trivial and unobtrusive. Although not implemented, it is proposed that changing connection terminators be accomplished through a dialog box, such as that shown in Figure 6-16. This would be opened either by double-clicking on the connection terminator, or through a context-sensitive pop-up menu. The dialog box would only present the connection terminators that would be valid for the connection type.

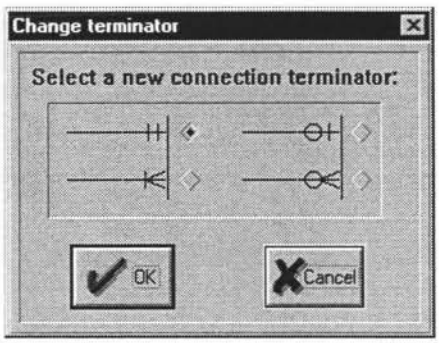



**Figure 6-16 – A mock-up of the "Change Terminator" dialog**

### Text Annotations on Connections

Many notations (eg. Coad and Yourdon, Booch, UML) use textual annotations on connecting lines and connection terminators to represent certain properties of connections such as cardinality. Support for this has yet to be implemented in the prototype as some issues are still to be addressed, two of which are worthy of mention here. The first is a consideration for their initial placement, which can be difficult to determine as connections can occur at any angle and the geometry of graph objects is based on "bounding rectangles". The second issue considers what restrictions would be placed on the movement of textual annotations, such as limiting the distance an

annotation may be placed from the graph object it is associated with. Difficulties may also arise in maintaining placements if connections are moved significantly. It is envisaged that these text areas would be similar in nature (in terms of their editing) to the default text areas found on object symbols.

## 6.6. Moving Graph Objects

All graph objects in a diagram can be repositioned. Graph objects are moved by selecting and dragging with the mouse. If a graph object is connected to others, movement of the initial graph object will affect those that it is connected to. The following paragraphs describe how each type of graph object (object symbol, connection symbol, etc) relates to others associated with it when moved.

Object symbols are regarded as the primary components of a diagram, and so movement of them causes any attached connections to move linearly with them. Connection terminators attached to object symbols maintain their relative positions to docking areas during the movement. Connecting lines joined to the connection terminators also move with the terminators, but may break into multiple line segments to maintain diagram integrity if the movement would cause an invalid layout. Figure 6-17 illustrates an example where the "Point" Class-&-Object symbol is moved up and to the left, and the "Arc" Class-&-Object symbol is moved down and to the right. The line connecting the "Point" Class-&-Object symbol is broken into multiple segments to maintain a correct view of the connection.

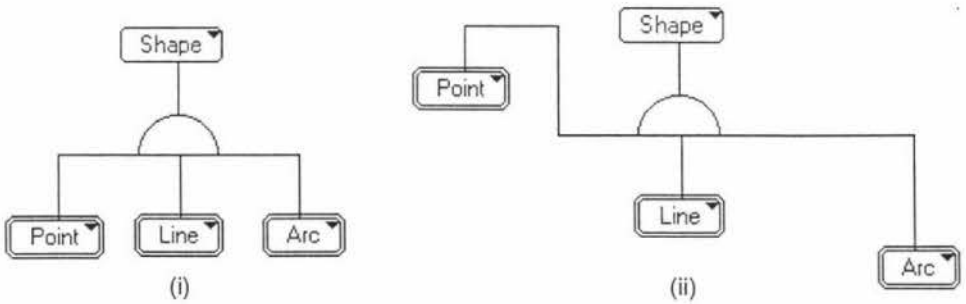



**Figure 6-17 – A model (i) before and (ii) after movement of object symbols**

When connection symbols are moved, any connecting line segments attached to the connection symbol only move with the connection symbol if the line segments would otherwise become disconnected from their associated docking area (eg. point docking areas, or movement past the end of a non-extendable line docking area). Where connecting line segments are moved with connection symbols, again they may break

into multiple segments to maintain diagram integrity, as shown in Figure 6-18(i). If a connection symbol contains an extendable line docking area and movement occurs parallel to that line, the connecting line segments connected to that docking area remain stationary and the line extension readjusts to compensate (Figure 6-18(ii)).

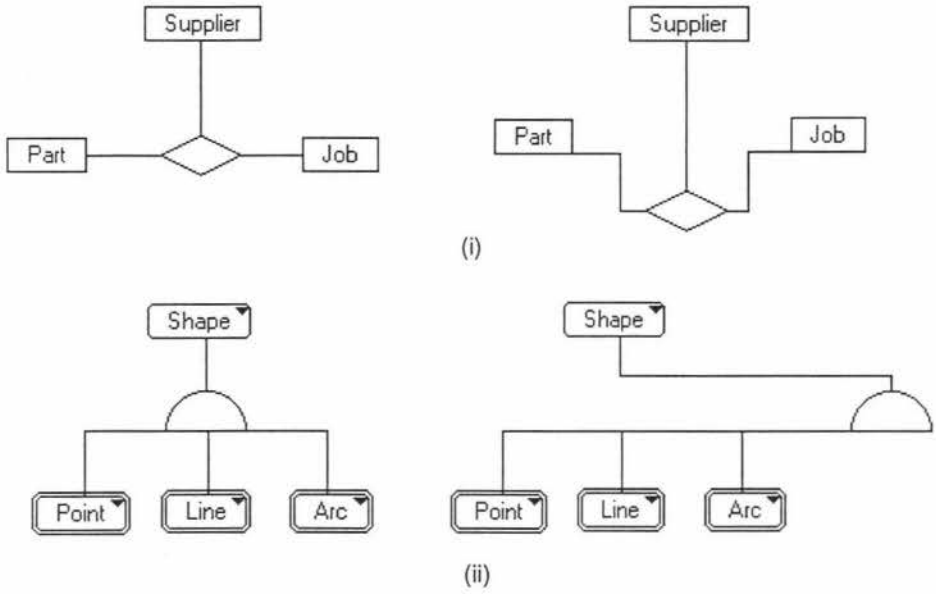



**Figure 6-18 – Movement of connection symbols: (i) movement causing line breakage; (ii) movement causing line extension adjustment**

Connection terminators may only move if they are docked to a line docking area[7]. Movement of connection terminators is restricted to the direction and length of that docking area. The connecting line segment attached to a connection terminator may break if the opposite end of the line is immovable (eg. attached to another object symbol or connection symbol via a point docking).

Connecting lines may be moved in directions perpendicular to their own direction. Their movement may be limited if components other than other line segments (such as connection terminators or object symbols) are attached to one or both of their ends. If so, their movement is restricted to the smallest set of movement capabilities provided by the attachment to these other graph objects. The movement of connecting lines does not normally cause them to break into multiple line segments.

Every effort is made to maintain connection and diagram integrity during the movement of graph objects, however at present there is no layout manager that keeps crossings under control. It is quite easy to rearrange a diagram into a totally confusing layout.

[7] Arc docking areas are currently not supported in the prototype GUI.

Also, while moving graph objects may cause the creation of new connecting line segments, there is presently no implementation to delete excess line segments. Additionally, there are no optimising routines yet implemented to remove very short or zero length connecting line segments where possible.

## 6.7. Deleting Graph Objects

All graph objects in a diagram can be deleted. Delete actions to a graph object existing as part of a connection (such as a connecting line segment) affect the entire connection. Graph objects can be deleted either by using the DELETE key or via the pop-up menus. The pop-up menus for all graph objects have a delete option available.

It is possible for delete operations to cause more than one object symbol or connection to be deleted. If an object symbol selected for deletion contains connections to other objects, then these connections are deleted also (dangling connections are not permitted). A connection symbol that is deleted removes all connections that were associated with it. A connecting line segment that is deleted may remove one or more connections, depending on the context it exists in. Figure 6-19 illustrates a simple model with annotations identifying the number of connections various graph objects are associated with.

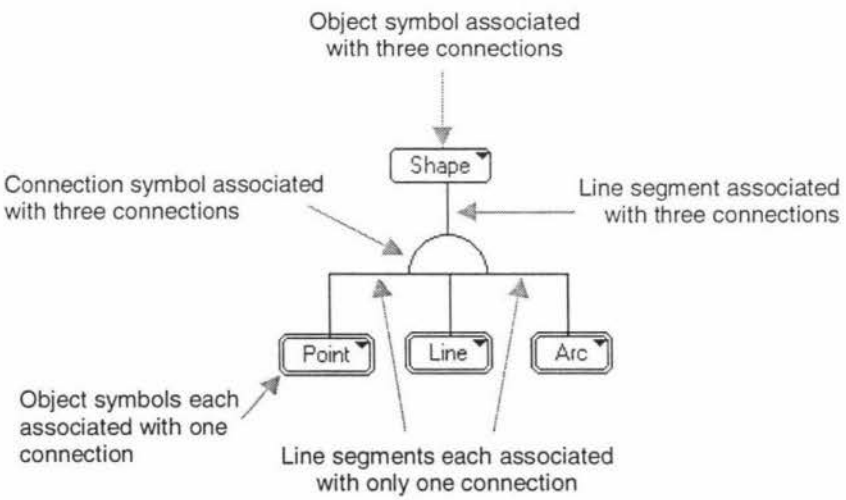



**Figure 6-19 – Graph object associations with connections**

Delete operations that affect only a single object symbol or connection are executed with no further warning. If the deletion may cause knock-on effects, then a warning dialog is presented to the user, informing them what knock-on effects the deletion will have. Figure 6-20 shows an example dialog that would appear after the user selected to

delete the connection symbol in the model. The user is given the option of accepting to delete all the grouped connections, or aborting the action entirely.

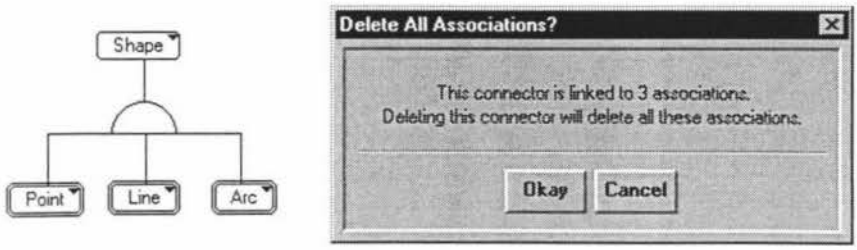



**Figure 6-20 – Example delete warning dialog**

This section so far has dealt only with deleting object symbols and connections at a local level, ie. the deletion affects only the immediate diagram involved. In a large project, it may be desirable to delete an entity from an entire model, or even from an entire project. Such actions would require all models and diagrams containing the entity to be updated, which may cause further knock-on deletions. It is unclear at this stage of the project development exactly how this might be reliably achieved, and if so, whether it would be at all reversible. Such investigations are beyond the scope of this thesis.

## 6.8. Undoing and Redoing Drawing Actions

The tool provides on the main menu bar an Edit menu containing *undo* and *redo* options. These allow drawing actions to be taken back if they proved to produce incorrect or undesirable results, or redone again if necessary. The level of undo is effectively infinite, limited only by memory size. A fully constructed diagram can be repeatedly undone back to the original blank canvas, or repeatedly redone back to the last drawing action made. The names of the menu items are constantly updated to reflect what action selecting them would cause. Some examples are "Undo add symbol", "Undo create association", "Redo move", "Redo delete", and so on.

It should be noted that if actions are undone, any further interaction with the diagram causes any redoable actions to become invalid and subsequently discarded. There is no possibility of recovering these actions when this occurs. This is standard practice with any tool featuring undo facilities.

# Chapter 7

# CLIENT-SERVER COMMUNICATIONS

Many actions that a user can perform with the CASE Tool client have an effect on the semantic meaning of a project, a project model, or an entity that exists in a model. For example, deleting an operation from a class symbol in a UML class diagram modifies the semantics of that symbol. This chapter describes the communications layer between the CASE Tool client sub-system and the CASE Tool server sub-system. The sets of client-initiated and server-initiated requests that have been defined are described. The current methods used for the transportation of notation specifications and diagram state are also described.

## 7.1. Communications Overview

Figure 7-1 illustrates the communications link between the client and server sub-systems of the CASE Tool. Instances of project managers, model editors, diagram editors, and graph objects in the Client Core send requests to and receive requests from the server via the Server Proxy. The Server Proxy transmits and receives requests across some medium that supports communication. The Tool Manager in the CASE Tool server interfaces with this medium to accept and transmit requests, interacting with various Methodology Interpreters for semantic processing.

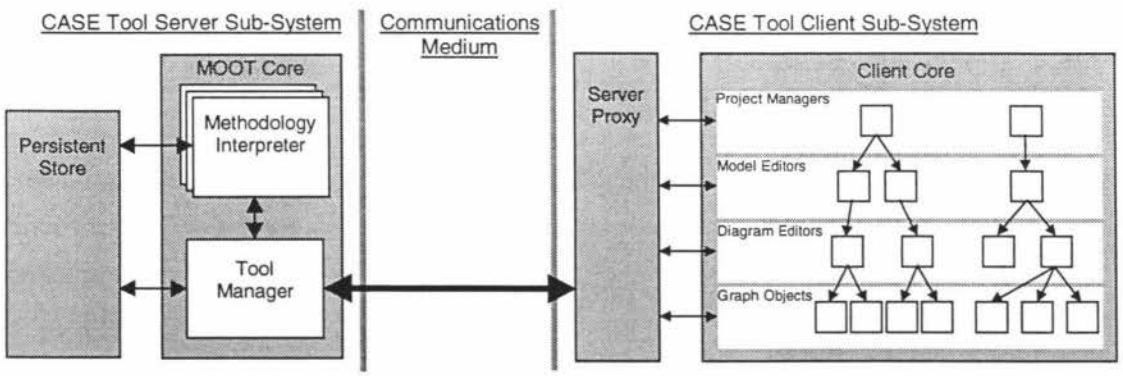



**Figure 7-1 – Client-Server communication sub-systems**

**Server Proxy**

Server communication is supported in the client through an abstract class ServerProxy. This purpose of this class is to eliminate the need for an implementation of the graphical user interface to know the details of the underlying communication layer. The ServerProxy class is an instance of the Proxy pattern (Gamma *et al*, 1995) that defines operations for the transmission of requests from the client to the server, and for the distribution of requests from the server in the client. Subclasses of ServerProxy can implement any method of communication between the client and server (eg. remote method invocation, TCP/IP network message, standalone stub interface, etc) without the rest of the client system requiring reimplementation. The ServerProxy class is also an instance of the Singleton pattern (Gamma *et al*, 1995). The Singleton pattern ensures that only one instance of the ServerProxy class is created during the lifetime of a client instance. Any communication between the client and server passes through the same proxy instance.

**Tool Manager**

Client communication is supported in the CASE Tool server via a Tool Manager. The Tool Manager acts as a communications interface to the MOOT Core, and is responsible for coordinating access to shared resources and monitoring the system's operation. It is responsible for maintaining details specific to each client (such as the projects that are open, the methodologies that they use, etc) and the corresponding Methodology Interpreter for each project. Messages from a client are accepted by the Tool Manager and bound to a message to an SSL object of the appropriate Methodology Interpreter. The Tool Manager generally acts as a slave to clients, responding to requests generated from user actions. The execution of a method in an SSL object however may cause requests destined for one or more clients to be initiated by the Tool Manager. Typically, server-initiated requests are used to keep all client environments (ie. projects, models, and diagrams) in a consistent state.

**Methodology Interpreter**

The semantics of each user project that is open in a client is managed by a corresponding unique instance of a Methodology Interpreter in the CASE Tool server. The client communicates potential semantic changes made by a user to its associated Methodology Interpreter, which in turn creates, modifies, or destroys SSL objects that describe the semantic entities.

Modifications to semantic entities in a diagram may cause side effects to other representations of those entities in other diagrams, models, or projects, or even other

clients. The CASE Tool server is responsible for maintaining consistencies between representations by informing the clients that are viewing any representation when modifications have been made. Consider an example where a user is working with a hierarchy of classes in a UML class diagram, and another user is making use of that class hierarchy in another diagram in another client instance. If the first user deletes an operation from one of the classes, the second user needs to be made aware of that change as it may impact on their work.

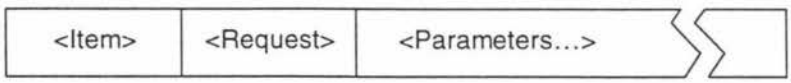
Requests that are initiated by the client are described in section 7.3. Section 7.4 describes the requests that are initiated by the server.

## 7.2. Communication Packets

Three primitive data types are used in the construction of request packets: bytes, integers, and strings[1]. The format and transmission method (eg. big-endian or little-endian) of each of these primitive types must conform to some standard between the client and server. For the purposes of successfully coding and decoding request packets, the following standards were set:

- Bytes are 8-bit unsigned quantities.
- Integers are 32-bit signed quantities. Integers are transmitted and received a byte at a time in the order from high byte to low byte.
- Strings are a null-terminated stream of printable characters. Strings may be of any length. Strings are transmitted and received as a stream of bytes beginning at the start of the string and ending with the null terminating character.

Request packets have the general structure:




The <Item> field identifies the source or destination of the request (eg. model editor, graph object, etc). The <Request> field identifies the particular request (eg. create model, create concept, etc). Any parameters that the request requires are included subsequently in the packet, existing as additional fields.

[1] Structured types such as arrays may be composed from collections of primitive types.

Five items that can send or receive requests have been identified: the client itself, projects, models, diagrams, and graph objects. The possible set of values for the <Request> field of a packet depends on the item that is initiating or receiving the request. Each request also has a different set of parameters. The types of the parameters may vary also. The specifications of the contents and parameters of the request packets described in this chapter are given in Appendix E.

The client-server communication protocol operates in a request-response fashion, ie. all requests generate a response. A positive response may return information from the server where necessary. If a request originating from a client fails, the server provides a textual explanation as a return value. These explanations would typically be presented to the user in some fashion (eg. a dialog box). Requests originating from the server that fail in a client do not include explanations in the response packet. It would be unusual, however, for a client to refuse to acknowledge a request from the server unless a major error had occurred during its execution.

## 7.3. Client-Initiated Communication

The client initiates requests to the server as the result of a user action. All user actions that involve a semantic change to a project, model, diagram, or graph object are communicated to the server. The client-initiated requests that have been defined for inclusion in the initial prototype are documented in Table 7-1. The table groups the requests into various categories that represent the relative levels of communication.

| Client-Level | Project-Level | Model-Level | Diagram-Level | Graph Object-Level |
|---|---|---|---|---|
| Connect | Create Project | Create Model | Create Diagram | Create Concept |
| Disconnect | Open Project | Open Model | Delete Diagram | Create Association |
| Log-in | Rename Project | Rename Model | Rename Diagram | Update Concept |
| Log-out | Close Project | Close Model | | Update Association |
| Get Methodologies | Delete Project | Delete Model | | Delete Concept |
| Get Projects | Save Project | Save Model | | Delete Association |
| | Get All Models | | | |
| | Get Project Models | | | |

**Table 7-1 – Client-initiated requests**

The purposes of the requests shown in the table are described briefly in sections 0 to 7.3.5.

### 7.3.1. Client-Level Requests

Client-level requests are sourced from the client interface. Six client-level requests are currently defined: Connect, Disconnect, Log-in, Log-out, Get Methodologies, and Get Projects.

A *Connect* request is initiated by the client when an instance of the client is first started. This identifies the client to the server, and permits further interaction between the two.

A *Disconnect* request is the last request generated by an instance of a client, and is used to signal that the instance is about to terminate.

A *Log-in* request from the client is used to log a particular user of a client into the server. The server may define a particular environment for each user, in terms of the projects and/or methodologies that are available and the read/write privileges that they are granted. This environment is restored when a user logs-in to the server.

A *Log-out* request closes the user environment in the server and resets the client so it can accept new users.

The *Get Methodologies* request returns an array that contains the names of the software development methodologies that the current user has access to. Each methodology name has an associated ID that is used as a reference to that methodology in the Create Project request.

The *Get Projects* request returns an array that contains the names of the user projects that are available to the user. Each project name has an associated ID that is used as a reference to that project in the Open Project request.

### 7.3.2. Project-Level Requests

Project-level requests are sourced from the project managers of a client. Eight project-level requests are currently defined: Create Project, Open Project, Rename Project, Close Project, Delete Project, Save Project, Get All Models, and Get Project Models.

The *Create Project* request causes the creation of a new user project in the server. The request is parameterised with the ID of a methodology sourced via the server response to the Get Methodologies request. The server responds to a Create Project request by

returning an ID and a default name for the project. It is assumed that a user who creates a new project is automatically given read/write access to the project.

The *Open Project* request causes an existing project to be opened for the user. The request requires the ID of the project and an access mode. The project ID is sourced from the server response to the Get Projects request. The access mode is either "read-only" or "read/write".

The server attempts to provide the user with the access mode requested. Read/write access can only be granted if the user has sufficient access privileges and if the project has not been previously locked by another user opening the project in a read/write mode. If a user requests read/write access and such a request cannot be granted, the project is opened with read-only access instead. When a project is opened with read/write access, the server also locks the project so that it cannot be opened again in read/write mode. A project is unlocked when the user that opened it with read/write access subsequently closes it via the *Close Project* request.

The *Rename Project* request allows a project to be given a new name.

The *Close Project* request closes the project that originates the request. If the project had been opened with read/write access, the server unlocks the project so other users can modify its contents.

The *Delete Project* request allows projects no longer needed to be deleted from the server. The user must have sufficient access rights to delete the project for this request to be successful.

The *Save Project* request saves the current state of the project to permanent storage. Any models or diagrams that have been modified are saved at the same time.

The *Get All Models* request returns an array that contains the names of all the model types that can be added into a project. The ID of the project is supplied as a parameter to the request. Each name is associated with a corresponding ID number that is used to reference the particular type of model in the Create Model request.

The *Get Project Models* request returns an array that contains the names of the models that have been previously added to a project. The ID of the project is supplied as a

parameter to the request. Each name is given a corresponding ID number. These ID numbers are used to reference a particular model in the Open Model request.

### 7.3.3. Model-Level Requests

Model-level requests are sourced from the model editors of a client. Six model-level requests are currently defined: Create Model, Open Model, Rename Model, Close Model, Delete Model, and Save Model.

The *Create Model* request causes the creation of a new model in a project. The request is parameterised with the ID of the project that the model is to be added to, and the ID of the type of model to add (from the *Get All Models* request). The server responds to a Create Model request by returning a new ID for the model, a default descriptive name, and an NDL specification for the model. The notation specification is transmitted to the client as described in sections 7.5 and 7.6. It is assumed a user who creates a new model is given read/write access to the model. New models can only be added to a project that has been opened with read/write access.

The creation of a new model typically causes the automatic creation of one or more diagrams for the model. The automatic creation of model diagrams is a server-initiated request that is described in section 7.4.3.

The *Open Model* request causes an existing model in a project to be opened for the user. The request requires a project ID, a model ID, and an access mode. The model ID is sourced from the server response to the *Get Project Models* request applied to the project. Read/write access is granted where requested if the user has sufficient privileges to make changes to the model, and the model has not been locked by another user. If opened in read/write mode, the server locks the model. When the model is closed by the user (via the *Close Model* request), the read/write lock (if any) is released.

The server responds to an *Open Model* request by returning the descriptive name of the model, the NDL notation the model is constructed with, and the state of any diagrams that exist in the model. The notation is transmitted to the client as described in sections 7.5 and 7.6. The state of diagrams is transmitted to the client as described in section 7.7.

The *Rename Model* request allows a user to give a model a new descriptive name.

The *Close Model* request closes the specified model. If the model had been opened with read/write access, the server unlocks the model so other users can modify its contents.

The *Delete Model* request allows a user to delete existing models from a project. The user must have sufficient access privileges to delete the model for this request to be successful.

The *Save Model* request saves the current state of all the diagrams in the specified model to permanent storage. The transmission of the state of model diagrams is described in section 7.7.

### 7.3.4. Diagram-Level Requests

Diagram-level requests relate to individual diagrams of a model. These requests are generated by a model editor in addition to the requests described in section 7.3.3. Three diagram-level requests are currently defined: Create Diagram, Delete Diagram, and Rename Diagram. These requests, which are all self-explanatory, will only succeed if the particular methodology model in question supports the arbitrary addition, deletion, or renaming of diagrams by a user. For example, a methodology model may spontaneously create or destroy diagrams only as necessary, or may automatically generate the names for individual diagrams.

### 7.3.5. Graph-Object-Level Requests

Graph-object-level requests are sourced from user interactions with the individual graph objects that can exist in a model diagram. Six graph-object-level requests are currently defined: Create Concept, Create Association, Update Concept, Update Association, Delete Concept, and Delete Association.

The *Create Concept* request asks the server to create a new SSL object instance that represents a model concept. The name of the notation template that is used to construct the view of the graph object is used by the server in determining what type of concept is being created. The server returns an ID for the new concept for any future references to that concept.

The *Create Association* request is used to define a semantic association between two or more concepts. The type of association is specified by the name of the (optional)

connection symbol template used in the construction of the connection, and an ordered set of the connection terminators that will be attached to the concepts being associated. An ordered set of the actual concepts being associated is also included with the request. If the association is semantically valid, the server returns an ID for future reference to that association. If not, the association cannot be created and the establishment of the connection in the diagram is refused.

The *Update Concept* request informs the server when the user has modified the value of a viewable property of an object symbol, such as a text area. All the relevant data necessary to locate and update the particular property on the server is included.

The only viewable property update that is defined to be transmitted to the server in the current prototype is the update of text areas. The other types of viewable property are related to the syntactic state of a graph object, and are not maintained by the server. The values of these other types of viewable property are instead saved with a diagram during a Save Model request.

The *Update Association* request is similar to the Update Concept request only it applies to associations. Associations have various components, including connection symbols, connection terminators, and connecting lines that all can have potential textual annotations such as labels or cardinality. The request identifies the particular component that is being updated, and supplies its new value.

The *Delete Concept* request is used to delete a concept from a diagram, model, project, or all projects. A user must have sufficient access privileges in order for the requested delete to be successful[2]. If they do not, the request fails and no delete takes place.

The *Delete Association* request is the same as the *Delete Concept* request only it applies to the associations that can exist between concepts. Again, a user must have sufficient access rights to perform the requested level of deletion.

## 7.4. Server-Initiated Communication

User actions that cause semantic changes to anything in a project result in the client sending a request to the server. In some instances, further notification to that or other

[2] Generally, this is only of concern where the scope of the delete extends past a single model. Within the scope of a model, a user is regarded as having sufficient access rights if the model was opened in a read/write access mode.

clients is necessary to complete the request or to maintain consistencies between multiple views of a concept or association. The server may also be required at times to initiate communication with clients to keep them up to date with the state of the system. The server-initiated requests that have been defined for inclusion in the initial prototype are identified in Table 7-2. No specific server-initiated diagram-level requests are currently defined.

| Client-Level | Project-Level | Model-Level | Graph Object-Level |
|---|---|---|---|
| Add Methodology | Create Model | Create Diagram | Create Concept |
| Add Project | Rename Model | Rename Diagram | Create Association |
| Delete Project | Delete Model | Delete Diagram | Update Concept |
| | | | Update Association |
| | | | Delete Concept |
| | | | Delete Association |

**Table 7-2 – Server-initiated requests**

The purposes of the requests shown in the table are described briefly in sections 7.4.1 to 7.4.4.

### 7.4.1. Client-Level Requests

Client-level requests are destined for the client interface. These requests are generated to keep clients up to date with the overall state of the system. Three client-level requests are currently defined: Add Methodology, Add Project, and Delete Project.

An *Add Methodology* request is generated by the server if a new software development methodology is added to the MOOT system. The request provides clients with the name and ID of the new methodology. The reasoning behind this request is that a CASE Tool client (or a user of a client) may remain connected to the server for an indefinite period of time. Without repeatedly polling the server, clients would otherwise remain unaware of new methodologies that might become available.

The *Add Project* request is initiated by the server as the result of a user of a client creating a new project. The request is sent to all clients other than that which the user who created the project is working. Clients receiving this request add the project to the set that is available for the user to view and/or open.

The *Delete Project* request removes a project from the set of projects that a client knows of. This request is initiated as a result of a user deleting an existing project. These requests are sent to all clients other than that which the user who deleted the project is working.

### 7.4.2. Project-Level Requests

Server-initiated project-level requests are generated by the server when models need to be dynamically and automatically created, renamed, or deleted from a project. For example, the creation of a new project may automatically generate one or more default models. The project-level requests that have been defined are correspondingly Create Model, Rename Model, and Delete Model. The server provides the ID of the new or existing model in the request.

### 7.4.3. Model-Level Requests

Model-level requests are identical in nature to the project-level requests defined in section 7.4.2, however they instead relate to the creation, renaming, and deletion of individual diagrams in a model. The three model-level requests that have been defined are correspondingly Create Diagram, Rename Diagram, and Delete Diagram. The Create Diagram request, for example, is sent to clients immediately after a user creates a new model, since all models must contain at least one diagram[3].

### 7.4.4. Graph Object-Level Requests

The server-initiated graph object-level requests that have been defined are Create Concept, Create Association, Update Concept, Update Association, Delete Concept, and Delete Association. These requests are complementary to the client-initiated requests, providing a means for the server to propagate changes in a diagram to wherever else the particular graph object may be viewed in other diagrams, models, projects, or clients.

[3] Diagrams are not automatically created by the client when a model is created, as a methodology may have cause to create more than one diagram in a new model. It is safer for the client to assume nothing, and to leave diagram creation to the Methodology Interpreter.

## 7.5. Notation Transportation

Notation specifications defined in NDL are stored in a Notation Database that resides in the persistent store of the CASE tool. When a user wants to create a model using a particular notation, the required notation specification must be transferred from the server to the client where the user is working. At some point, the notation specification must be parsed and translated from its textual representation to an abstract syntax tree (AST) representation that can be interpreted. This translation may be administered by either the client or the server. Both of these administration options have been examined in the course of determining what software interface was required between the client and server to support notation retrieval.

*Notation Parser in Client*

In this scenario, shown in Figure 7-2, the server responds to a request that requires a notation specification (eg. a Create Model request) by searching the Notation Database. Upon obtaining the specification, the server sends it directly to the client. The client then proceeds to parse the specification, building the equivalent AST representation.

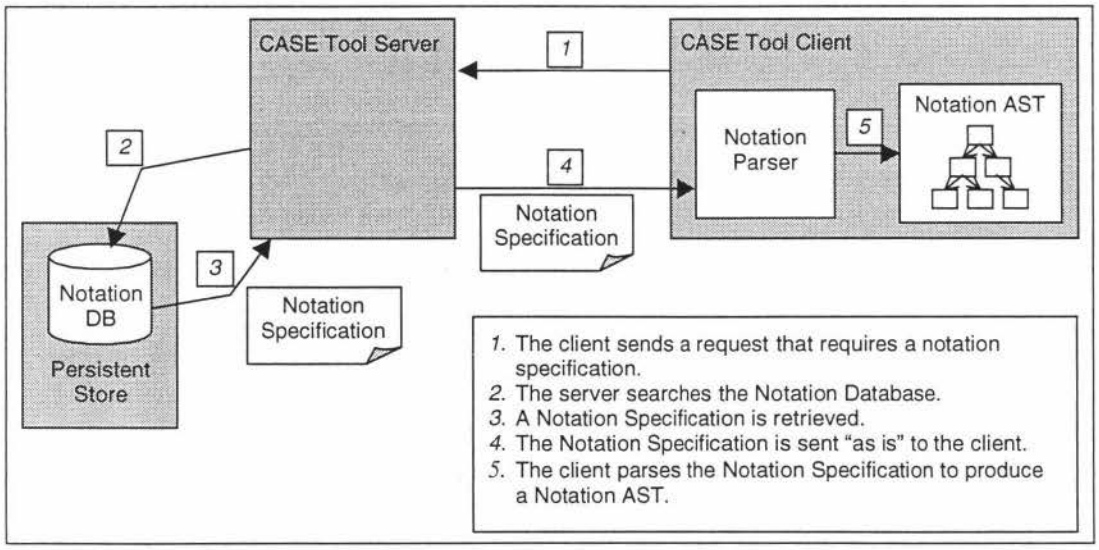



**Figure 7-2 – Notation parsing in the Client**

This method has the advantage that the responsibility of the server is reduced. This may be of benefit if the server is already laboured with requests from other clients. There is a disadvantage however that each client must duplicate the effort required to parse, symbol-match and type-check a notation specification. This process of generating an AST must also be repeated every time a notation is requested. This is clearly redundant effort.

*Notation Parser in Server*

In this alternative scenario, shown in Figure 7-3, a client request that requires a notation results in the server first searching a notation cache (called the Notation AST Store in the figure). This cache stores the AST of serialised notation specifications that have previously been parsed[4]. If the required notation is not present in the AST Store, the server searches the Notation Database for the notation specification. The specification is retrieved and parsed, and the AST generated is serialised and saved into the notation cache for future use. In either case, once the server has obtained a serialised AST, it is transmitted to the client. The client subsequently proceeds to deserialise and restore the AST structure.

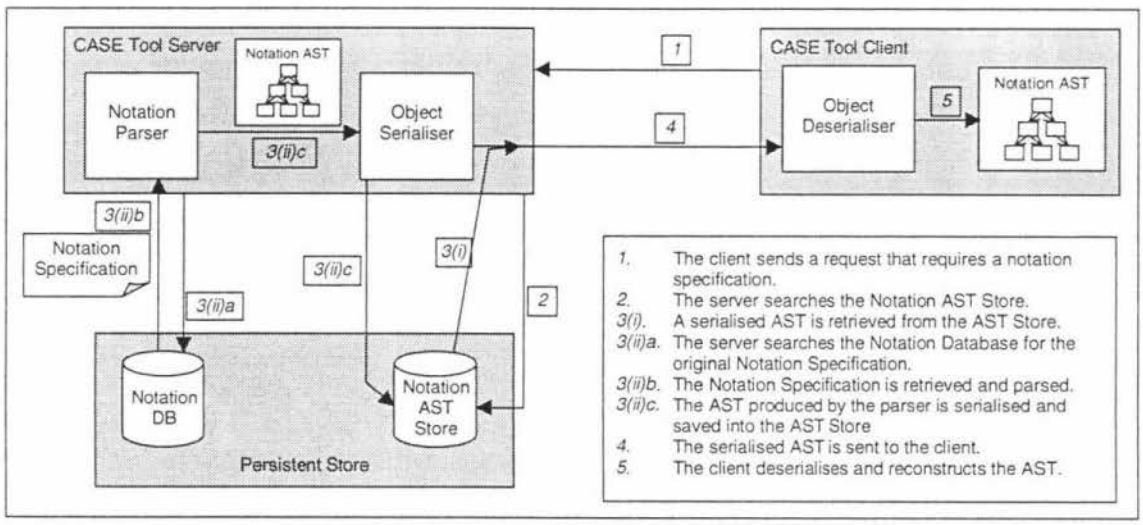



**Figure 7-3 – Notation parsing in the Server**

The advantages of this method include that a notation specification is only ever parsed into an AST once, thus eliminating redundant effort. Clients are also only ever concerned with one notation representation, an AST, a representation that is directly useable. Deserialisation of an object stream is an easier task than parsing a notation specification, as no symbol table, type checking, or consistency matching is required. This method poses the disadvantage however that an extra protocol must be established for the serialisation and deserialisation of notation ASTs.

An extra factor that influenced the direction of the client-server notation interface design was the existence of previous software. Prior to commencement of the thesis research, a notation parser based on the ideas of Clark (Clark, 1994) and Page (Page *et*

[4] An abstract syntax tree exists in memory as a collection of objects. To "serialise" one or more objects is to encode them as a stream of bytes, suitable for I/O. Object deserialisation supports the complementary reconstruction of the objects from the stream.

*al*, 1994) had already been coded. This parser generated AST representations for simple notation specifications (containing object templates only). An AST, described by a collection of objects (vaguely similar to that described in Chapter 4, section 4.3) contained all the same information as the equivalent textual specification. This information could be re-extracted from the AST as the objects supported a text-based serialised stream output of their state. Formalising the syntax for this stream output led to the conclusion that the serialisation would make a suitable input to a client user interface without having to reimplement the original parser in the client. Additionally, it was felt that focusing on the AST would provide a better development approach. The AST, and the NDL Interpreter that was to be built around it, could be designed first, and the grammar for NDL specifications generated around that final design.

After further research and examination, it was concluded that developing the class hierarchies for the NDL Interpreter first was a more profitable venture. Changes to the structure and content of the AST resulted in changes in the original NDL grammar. Since the NDL Interpreter was developed as a component of the client, the serialised AST representation subsequently became an intermediate stage in the communication of notation specifications between the parser in the server and the client user interface.

Following the decision to implement this model, a protocol between the client and server had to be designed to support the serialisation and deserialisation of notation ASTs[5]. The client interface needed to know what to expect on its input, and required sufficient information in that input to reconstruct the original AST. Three elements influenced the decision to work with a textual grammar, much like the NDL grammar but less English-like and more concise and rigid:

- The stream output of ASTs in the original notation parser was intended chiefly for debugging purposes. It had been designed to be human-readable and to be meaningful. The serialisation of object state was therefore already described textually.
- The client and server had the potential to be implemented in different languages and on different platforms. No specific implementation language or platform details were desired in the serialisation.
- Textual descriptions could be easily coded manually for testing and debugging purposes. This later became of particular importance, as the notation parser on the server was not regularly updated to maintain pace with the evolution of NDL.

[5] The class hierarchies that are components of the AST representation of an NDL notation specification were described in Chapter 3 and Chapter 4.

The formalised grammar that was subsequently developed for the serialisation of NDL abstract syntax trees is described in section 7.6.

## 7.6. NDL Serialised Abstract Syntax Tree (AST) Grammar

The client user interface accepts notation specifications in the form of serialised ASTs. The serialisation, structured as a single string of printable characters, is provided via the Server Proxy in the client as the result of a request such as Create Model or Open Model. The client parses the input and reconstructs the structure and state of the original AST defining the notation.

A formalised grammar has been defined to support the serialisation and deserialisation of NDL ASTs. AST state in the serialisation is described using meaningful symbols and numbers. Sections 7.6.1 to 7.6.3 describe the grammar developed using Extended Backus-Nauer form (EBNF). EBNF notation is described in Appendix A. For a complete and concise list of the serialised NDL AST grammar, refer to Appendix C. For an example of a complete NDL notation specification describing the Coad and Yourdon notation, and its representation as a serialised AST, refer to Appendix D.

### 7.6.1. AST Notation Specification

The root of a notation AST is represented by the class Notation. The Notation class encapsulates the name of the notation being defined and four template collections which comprise the groups, object symbols, connection symbols, and connection terminators of the notation. A serialisation of the AST must therefore begin with the serialisation of an instance of the Notation class. The serialisation has the following grammar:

```
<Notation>          ::= '[' NTN: <String>
                            <Groups> <Objects> <Connection_Symbols>
                            <Connection_Terminators>
                        ']'
```

The four template collections comprising the notation are each serialised as individual subsections of the notation serialisation. The syntax of each subsection is described by the grammar:

```
<Groups>                  ::= '[' GRPS: { <Group_Template> } ']'
<Objects>                 ::= '[' OBJTS: { <Object_Template> } ']'
<Connection_Symbols>      ::= '[' CONSYMS: { <Connection_Symbol_Template> } ']'
<Connection_Terminators> ::= '[' CONTERMS:
                                  { <Connection_Terminator_Template> } ']'
```

Templates are composed of various collections of NDL primitives such as graphical components, active areas, and docking areas. The shape and position of individual primitives are described by a number of expressions. Common expressions may be factored out from other expressions in an NDL specification so they are only defined once. These expressions are included in the composition of templates. Based on this, individual templates may therefore require some or all of the following subsections in their serialisation:

- common expressions;
- graphical components;
- active areas;
- docking areas;
- a bounding region.

The grammar for the serialisation of these subsections (the NDL primitives) is described in section 7.6.2. The grammar for the serialisation of actual templates is described in section 7.6.3.

### 7.6.2. Serialisation of Notation Primitives

Each type of notation primitive that NDL supports (graphical components, active areas, docking areas, etc) is grouped into a single composition in a notation template. These compositions are placed into clearly identifiable subsections in a serialisation. The subsections that may appear in a serialisation are correspondingly the specification of: common expressions, graphical components, active areas, docking areas, and the template's bounding region.

#### Common Expressions

Common expressions are written as equations in an NDL specification so that they are equated with some textual identifier. When parsed by an NDL parser, *all* textual identifiers in a notation specification are turned into unique IDs. This eliminates the need for the deserialiser to consider symbol matching or name clashes. The IDs for common expressions are based on their order of definition, ie. the first equation has an

ID equal to zero, the second has an ID equal to one, and so on. These ID numbers are not stored with the expressions (as their ordered position in the set of expressions is indicative of their ID), but are used when referring to the expressions in other parts of the template definition. In the serialisation, only the expressions themselves are output, and they are output in the same order as they were defined. As a set, the expressions are serialised as a subsection using the following grammar:

```
<Expressions>          ::= '[' EXPS: { '[' <Expression> ']' } ']'
```

Zero or more expressions may be defined in the subsection. Expressions are written in the same form in a serialisation as they are written in an NDL specification (ie. Reverse Polish Notation). Expressions are serialised from, and deserialised to, one or more instantiations of the classes in the Expression class hierarchy (Figure 4-10, Chapter 4). The grammar for the serialisation of individual expressions is as follows:

```
<Expression>           ::= '+' <Expression> <Expression> |
                           '-'<Expression> <Expression> |
                           '*' <Expression> <Expression> |
                           '/' <Expression> <Expression> |
                           <Term>
<Term>                 ::= <Number> | <Function> | <Expression_Reference>
<Function>             ::= <Text_Function> | <Group_Function> |
                           <Expression_Function>
<Expression_Reference> ::= '$exp$(' <Integer> ')'

<Text_Function>        ::= <Text_Width_Function> | <Text_Height_Function>
<Group_Function>       ::= <Group_Width_Function> | <Group_Height_Function>
<Expression_Function>  ::= <Max_Function> | <Min_Function>

<Text_Width_Function>  ::= '$txtWidth$(' <Integer> ')'
<Text_Height_Function> ::= '$txtHeight$(' <Integer> ')'
<Group_Width_Function> ::= '$grpWidth$(' <Integer> ')'
<Group_Height_Function>::= '$grpHeight$(' <Integer> ')'
<Max_Function>         ::= '$max$(' <Expression> { ',' <Expression> } ')'
<Min_Function>         ::= '$min$(' <Expression> { ',' <Expression> } ')'
```

Note that function names are enclosed in dollar sign ('$') symbols. This is purely a syntactic convention to distinguish function expressions from other types of identifiers in the serialisation[6].

**Graphical Components**

The graphical components subsection contains the graphical component segment templates that describe the shape of a template. The subsection is denoted by the grammar:

```
<Graphical_Components> ::= '[' COMPS: { <Graphical_Component > } ']'
<Graphical_Component>  ::= <Line> | <Arc> | <Single_Text> | <Multi_Text> |
                           <Group_Reference>
```

Five types of segment template can describe the shape of a template. These are line, arc, single-text, multi-text, and group references. Line segment templates are defined by a series of points, and so are described in the serialisation by the grammar:

```
<Line>    ::= '[' LN:  <Point> '-' <Point> { '-' <Point> } ']'
<Point>   ::= '(' <Expression> ',' <Expression> ')'
```

Arc segment templates are defined by the top-left and bottom-right corners of a bounding rectangle, a start angle for drawing, and the angle that the arc subtends:

```
<Arc>     ::= '[' ARC: <Point> '-' <Point> ',' <Number> ',' <Number> ']'
```

Text segment templates are defined by a unique ID, a location, and possible initial values:

```
<Single_Text>   ::= '[' ST: <Integer> ',' <Point> [ <String> ] ']'
<Multi_Text>    ::= '[' MT: <Integer> ',' <Point> { <String> } ']'
```

A group reference segment template is defined by a unique ID, the name of the initial group template, and a location:

```
<Group_Reference>   ::= '[' GRP: <Integer> <Identifier> AT <Point> ']'
```

[6] Remember that there is no symbol or type checking carried out by the deserialiser.

### Active Areas

Active areas are grouped in a serialisation using the following grammar:

```
<Active_Areas>          ::= '[' AA: { <Active_Area> } ']'
```

The grammar for the serialisation of individual Active Area Segment Templates is as follows:

```
<Active_Area>           ::= '[' AAT: <Point> '-' <Point> <Action> ']'
<Action>                ::= <Update_Action> | <Transition_Action>
<Update_Action>         ::= '[' UACT: <Integer> ']'
<Transition_Action>     ::= '[' TACT: <Identifier> ']'
```

An active area is defined in a serialisation by the specification of an active region. The active region is defined by top-left and bottom-right coordinates. The active region is followed by the associated Action object to denote the particular specialisation of Active Area Segment Template that is being serialised.

### Docking Areas

The docking area subsection in a serialisation is defined by the grammar:

```
<Docking_Areas>         ::= '[' DA: { <Docking_Area> } ']'
<Docking_Area>          ::= <Point_Docking> | <Line_Docking> | <Arc_Docking>
```

Each docking area template has a number of attributes that define the connectability of the area. Point Docking Templates are described by a location, a maximum connection count (in the serialisation, "u" signifies an unlimited count), a connectivity arc, and (where required) a list of connection symbols and connection terminators that are permitted to attach to the docking area:

```
<Point_Docking>         ::= '[' PT: <Point> ',' <Connection_Count> ','
                            <Connectivity_Arc> <Allowable_Connectors> ']'

<Connection_Count>      ::= 'u' | <Integer>
<Connectivity_Arc>      ::= '[' CONARC: '(' <Number> ',' <Number> ')' ']'
<Allowable_Connectors>  ::= '[' [ <Allowable_Terminators> ]
                                [ <Allowable_Symbols> ] ']'
```

```
<Allowable_Terminators>  ::= TERMS: <Name_List>
<Allowable_Symbols>      ::= SYMBOLS: <Name_List>
<Name_List>              ::= '(' { <String> } ')'
```

Line docking areas can be either extendable or not extendable. This is represented by a flag value in the serialisation. The shape of the docking area is defined by two endpoints. Line docking areas also define a maximum connection count, a minimum connection spacing, and a list of symbols that may connect to the docking area:

```
<Line_Docking>           ::= '[' LN: <Flag> ',' <Point> '-' <Point> ','
                             <Connection_Count> ',' <Number>
                             <Allowable_Connectors> ']'
<Flag>                   ::= 0 | 1
```

When an Arc Docking Segment Template object is serialised, the shape is first described. The shape is defined by two points representing the top-left and bottom-right points of the enclosing region, and two numbers representing the start angle and extent of the arc. A flag value indicating the state of the *connectOutside* attribute follows. Defined next is the maximum connection count allowed for the docking area, followed by the minimum connection spacing and a list of symbols that may connect to the arc. The grammar for the serialisation is given here:

```
<Arc_Docking>            ::= '[' ARC: <Point> '-' <Point> ',' <Number> ',' <Number> ','
                             <Flag> ',' <Connection_Count> ',' <Number> ','
                             <Allowable_Connectors> ']'
```

**Bounding Region**

The bounding region of a template describes the template's dimensions. The dimensions of a bounding region are serialised using the grammar for a point, as this is a suitable notation (a point defines two ordinate expressions). Bounding regions are described in the serialisation using the grammar:

```
<Bounding_Region>        ::= '[' BR: <Point> ']'
```

### 7.6.3. Template Serialisation

A notation is comprised of up to four different template types: group templates, object templates, connection symbol templates, and connection terminator templates. The following subsections that may appear in the grammar of template serialisations have been described in detail in 7.6.2. (The EBNF non-terminal symbol that defines them in the serialisation grammar is noted in parentheses.)

- common expressions (<Expressions>)
- graphical components (<Graphical_Components>)
- active areas (<Active_Areas>)
- docking areas (<Docking_Areas>)
- bounding region (<Bounding_Region>)

**Group Templates**

Group templates are identified by a string denoting the name of the group. Group templates are composed of lists of common expressions, graphical components, and active areas (refer Figure 3-7, Chapter 3). A group template also defines a bounding region. The serialisation of group templates has the grammar:

```
<Group_Template>    ::= '[' GST: <Identifier> <Expressions>
                        <Graphical_Components> <Active_Areas>
                        <Bounding_Region> ']'
```

**Object Templates**

The class Object Template is a specialisation of the class Template (refer Figure 4-6, Chapter 4). The serialisation of an Object Template causes the attributes of the Template superclass to be serialised as a separate subsection. Based on previous conventions, it appears in the stream that an Object Template encapsulates a Template. This particular design was chosen as it eliminated repeated operations in the notation deserialiser and allowed reuse of code fragments. The Template class has a large number of attributes, and the concrete subclasses of Template all need to restore these attributes when reconstructed. Object Templates are recreated from the stream by first deserialising and instantiating a Template object. The Template object is then converted to an Object Template, and the remaining attributes deserialised. Note that the serialisation of Templates can only exist within the serialisation of its concrete subclasses.

The grammar for the serialisation of the Template and Object Template classes is as follows:

```
<Template>              ::= '[' TMPL: <String> <Expressions>
                            <Graphical_Components> <Bounding_Region> ']'
<Object_Template>       ::= '[' OBJTMPL:  <Template> <Active_Areas>
                            <Docking_Areas> <Default_Text_Property> ']'
<Default_Text_Property> ::= '[' DEFTXT: '(' <Integer> ',' <Integer> ')' ']'
```

The optional default text area property of object templates is defined by two integers representing a group ID and a component ID. The group ID identifies the group that contains the text area with the ID that matches the component ID.

**Connection Symbol Templates**

The class Connection Symbol Template is a subclass of Connection Template, which itself is a subclass of Template (refer Figure 4-6, Chapter 4). Similarly to Object Templates, a Connection Symbol Template encapsulates and extends the definition of a Connection Template in the serialisation. A Connection Template encapsulates and extends the definition of a Template. This is better seen in the grammar:

```
<Connection_Template>   ::= '[' CONTMPL: <Template> <Flag> ']'
<Connection_Symbol_Template> ::= '[' CONSYMTMPL: <Connection_Template>
                            <Point> <Docking_Areas> <Arity> ']'
<Arity>                 ::= '[' ARITY: <Integer> ']'
```

A Connection Template extends a Template with the definition of a rotation constraint (a <Flag>). A Connection Symbol Template extends a Connection Template with a connection point, a list of docking areas, and an arity.

**Connection Terminator Templates**

Connection Terminator Templates are also a subclass of Connection Template, and so encapsulate the definition of a Connection Template in a serialisation. A Connection Terminator Template extends a Connection Template by specifying the head and tail connection points, as shown by the following grammar:

```
<Connection_Terminator_Template> ::= '[' CONTERMTMPL:
                                <Connection_Template> <Point> ',' <Point> ']'
```

## 7.7. Diagram Transportation

The client interface must be able to save and later reload diagrams when a user chooses to save or open a model in their project. The state of a diagram is maintained in the diagram editor by a collection of ModelComponent objects and a collection of GUIComponent objects. GUIComponent objects that are built from a notation template description (ie. GUITemplateComponents) each contain a reference to the template they are built from, an associated set of viewable properties (stored in an object of the ViewableThing class), and a current view that is rendered on to the computer display.

Not all of this information needs to be saved in order to be able to successfully reload a diagram. Notation templates do not need to be saved with GUITemplateComponents, as the templates already exist in a notation specification. Notation specifications are reloaded with diagrams when a model is reopened. The GUITemplateComponents, however, do need to save the name of the template they use, so that the reference can be reinstantiated upon reloading. Views of GUITemplateComponents do not need to be saved with a diagram, as these can be recreated from the notation template and viewable properties contained in a GUITemplateComponent at any time. Objects of the following classes, and attributes of these classes, *must* however be saved:

**A.** ModelComponent class hierarchy:

- The semantic ID of components;
- The GUIComponent objects associated with a component;
- The cross-references made between concepts and associations.

**B.** GUIComponent class:

- For graph objects constructed via templates: the name of a notation template, and the set of viewable properties (stored in a ViewableThing);
- For other graph objects: the details necessary (eg. line endpoints) to redraw the graph object;
- The ModelComponent objects associated with each graph object;
- The cross-references made between joined graph objects;
- Any other necessary details, such as which connections are attached to which docking areas.

**C.** ViewableThing class:

- All viewable properties.

Objects of these classes can only be saved and loaded by a serialisation mechanism. It has been identified that a formalised grammar, similar to that described in section 7.6 for notation ASTs, needs to be defined for the serialisation of objects recording diagram state. Due to time constraints, however, no such grammar has yet been defined.

As an interim measure, the client interface supports diagram saving and loading by using the object serialisation features native to the language of implementation (which is Java (Sun, 1998)). This is unsatisfactory in the long term for three reasons: consistency, portability, and efficiency. The current method of serialising objects for the purposes of saving and loading diagrams is inconsistent with the other method of serialisation used in the client, which is based around a formalised textual grammar. Using features of a particular implementation language restricts portability, as a serialisation could only ever be used by a client implemented in the same language, with the same ModelComponent, GUIComponent, and ViewableThing class hierarchies. If these class hierarchies (including the implementation of individual classes) are modified, then previous serialisations cannot be reloaded due to class version mismatches[7]. The size of object serialisations using the native features is also exorbitant, and the time taken to save or load the small-scale diagrams that can be created in the current client prototype is in the order of tens of seconds. This is clearly inefficient, and extremely unsatisfactory when extrapolated to the larger-scale diagrams that will be supported in the future.

[7] This was unfortunately discovered through experience while developing the client.

# Chapter 8

# IMPLEMENTATION OF THE MOOT CASE TOOL CLIENT

A prototype CASE Tool client based on the design of the NDL Interpreter and graphical user interface described in this thesis has been implemented to verify if the approach taken is efficacious. The implementation of the prototype in Java (Sun, 1998) constitutes a significant component of the research effort achieved by the author. This chapter describes the implementation phase of the research.

## 8.1. Implementation Language

The CASE Tool client has been implemented using version 1.1 of the Java Development Kit (Sun, 1997). Java is most often associated with the World Wide Web and the applets (small application programs) that can be attached to Web pages. However, positive reports on the use of the Java language for more complicated "real-world" applications have started to appear (Cooper and Werstein, 1998; Grundy *et al*, 1998). The design of the Java language has also been considered as reflecting good software engineering practice (Schach, 1997). Java was chosen as the implementation language for the CASE Tool client for reasons including object-orientation, network support, platform independence, robustness and reliability, and memory management.

**Object-Orientation**

Java is an object-oriented programming language similar in structure and syntax to C++ (Stroustrup, 1997). The object-oriented nature of Java moulds well into the underlying object model of the MOOT system, and is ideal for the event-driven nature of window-based graphical user interfaces. User-action events, or events sourced from the CASE Tool server, correspond to messages to objects capable of handling these events.

**Network Support**

One of the primary design considerations for Java was to include extensive network support. Java is intended for use in networked/distributed environments (Sun, 1996). Implementing client/server applications in Java is simple, and interaction with remote

systems can be achieved in the same manner as interacting with a local file system. The CASE Tool client by nature relies on a network connection to a CASE Tool server, so the network support of Java was important.

**Platform Independence**

Platform independence was an important issue as it increases the target audience for the system. Java source code is compiled into a byte-code binary representation. This byte-code representation is not platform specific, and can be executed on any platform that implements a Java Interpreter (which currently includes the Sun Solaris, PC, and Macintosh platforms). Implementing the CASE Tool client in Java allows any users of these systems access to the MOOT CASE Tool, supported by a graphical user interface that is consistent in look and feel.

**Robustness and Reliability**

The design of the Java language results in a robust and reliable implementation environment. Java is a strongly typed language (more so than C++) that allows extensive compile-time and link-time type checking to be performed to guard against potentially illegal declarations. The Java language removes the notion of pointers present in many other languages, resulting in a robust environment that is "safe" to implement in. Memory cannot become corrupted, as unauthorised access to memory can never occur. A significant advantage of the removal of pointers is that Java implements "true" arrays that do not use pointer arithmetic (as in C++). This permits array indexing to be checked at run-time to ensure that the possibility of overwriting memory and corrupting data is eliminated. The extensive support for exceptions (errors, or other exceptional circumstances, that are treated as distinct from the normal flow of program control) in Java further reduces the possibility that a program will fail without warning or explanation.

**Memory Management**

The memory management features of Java are significantly improved over similar languages such as C++. The removal of pointers eliminates potential hazards when dealing with memory issues. Java also provides an automatic garbage collector. This eliminates the need for the programmer to think about memory management, and allows them to instead concentrate more on the problem itself.

While not crucial to the implementation of the prototype CASE Tool client, performance issues with a Java implementation were also considered during the selection of an implementation language. Java is an interpreted language, and is

therefore naturally slower in execution than native code compilers such as C++. Just-In-Time (JIT) compilers that translate Java byte-code to native machine code can aid this speed deficiency somewhat, however these compilers are platform dependent. The automatic garbage collector can also cause a performance hit when invoked by the Java Interpreter (Sun, 1996). Performance issues were not considered crucial to the implementation of the client as by its nature the user interface spends a significant amount of time waiting for a user event to occur. The tasks that the client has to perform are generally responses to these user events, and individually are not time-consuming.

## 8.2. Implementation Details

Due to time constraints imposed on the research, the prototype CASE Tool client only implements a subset of the total design requirements identified in this thesis. The present level of implementation achieved, however, accounts for a considerable time component of the research, and approximately 17,500 lines of Java code have been written[1]. The following aspects of the CASE Tool client have been implemented in the current prototype:

- An NDL abstract syntax tree deserialiser (~6% of code). The deserialiser uses the grammar defined in Chapter 7 to reconstruct the serialised abstract syntax tree representations of notation descriptions.
- The class hierarchies defined in Chapter 3 and Chapter 4 that represent NDL and the NDL Interpreter (~21% of code). All responsibilities have been implemented and the NDL Interpreter is fully operational.
- The classes and class hierarchies defined in Chapter 5 that model the CASE Tool client, with the following restrictions:
  - The ProjectManager class is defined, however it does not produce an interactive dialog window for project management. The class implements only basic operations for creating, renaming, and deleting models.
  - The ModelEditor class has been implemented, however it supports only one diagram for each model.
  - The DiagramEditor class has been implemented with the graphical user interface design described in Chapter 6. The editor currently supports only horizontal and vertical lines for connections. The minimum distance specification between connections to line docking areas is

[1] This figure includes white space and syntactic sugar in its calculation. Ignoring these, approximately 8500 lines of "useful" code remains.

ignored during connection construction or movement. Arc docking areas are currently not supported by the diagram editor. Textual annotations to connections (for cardinality, for example) are currently not supported in the editor either[2]. The implementation, testing, and debugging of the DiagramEditor class and its responsibilities was the most time-consuming of the implementation tasks. The DiagramEditor class itself is responsible for approximately 14% of the total code produced.

- The GUIComponent and ModelComponent class hierarchies have been fully implemented (~15% of code).

- Very little of the client communications interface described in Chapter 7. The ServerProxy class has been defined with only the following operations: Create Model, Open Model, Save Model, Create Concept, and Create Association. A subclass of ServerProxy, StandaloneClient, has been implemented to support these operations. The Create Model, Open Model, and Save Model operations simply present dialog boxes that interface to the local file system. The Create Concept and Create Association operations return sequential ID numbers for the generation of ModelConcept and ModelAssociation objects in the diagram editor.

The client operates principally around the diagram editor interface. Upon start-up, the client automatically creates an instance of the ProjectManager class. This project manager is the only one created during the lifetime of the client and is tied to no particular methodology. Models that are created or opened are added to this project. Models may be saved, but not projects. Saving a model saves the state of the diagram that belongs to the model. Each instance of the ModelEditor class supports only one diagram, and each diagram supports only one representation of each concept or association that is created. The size of each diagram is limited to the size of the window that is used to display the client interface, as no scrollbars for navigation around a larger drawing canvas have been implemented. Due to the lack of server communication, any diagram that can be constructed with a notation specification may be drawn, regardless of the semantic meaning it may or may not otherwise portray.

The overall presentation of the client is sparse in that very few dialog boxes have been implemented for the display of information. These aspects were not considered to be of high priority during the implementation of the prototype. The presentation methods that

[2] They could be supported easily by including a text area (and associated text-update active area) as part of a connection symbol or connection terminator template definition. However, these text areas would not be able to be repositioned if the text they contained happened to overwrite other graph objects in the diagram. The issue of annotation placement and movement has yet to be resolved in this implementation.

have been implemented in the prototype (such as the toolbar of drawing tools) use many graphical components based on those described in the Graphic Java Toolkit (GJT) (Geary, 1997). The GJT was written for Java 1.0, and the many of the components used in the client graphical user interface have been re-implemented using the Java 1.1 event model for the prototype

## 8.3. Results of the Implementation

The results of the initial CASE Tool client prototype have allowed the production of small-scale diagrams. Figure 8-1 illustrates an example model that was created using the Coad and Yourdon notation specification listed in Appendix D. The notation supports the Class and Class-&-Object object symbols, and the Instance, Gen-Spec, and Whole-Part connections of Coad and Yourdon, as shown in the toolbar in the figure. Coad and Yourdon Subjects and Message Connections are not supported. Subjects are composite notation symbols, which are considered beyond the scope of this thesis. Message Connections are drawn as thick arrowed lines, and may be supported once the specification of optional line styles is incorporated into NDL[3].

The model (Figure 8-1) represents a simple alarm system that is composed of an audible alarm, various types of sensor, sensor events, and a control panel. The control panel has a keyboard that contains a keypad, function keys, and status lights.

[3] This feature was removed from previous implementations of NDL as noted in the review in Chapter 2.

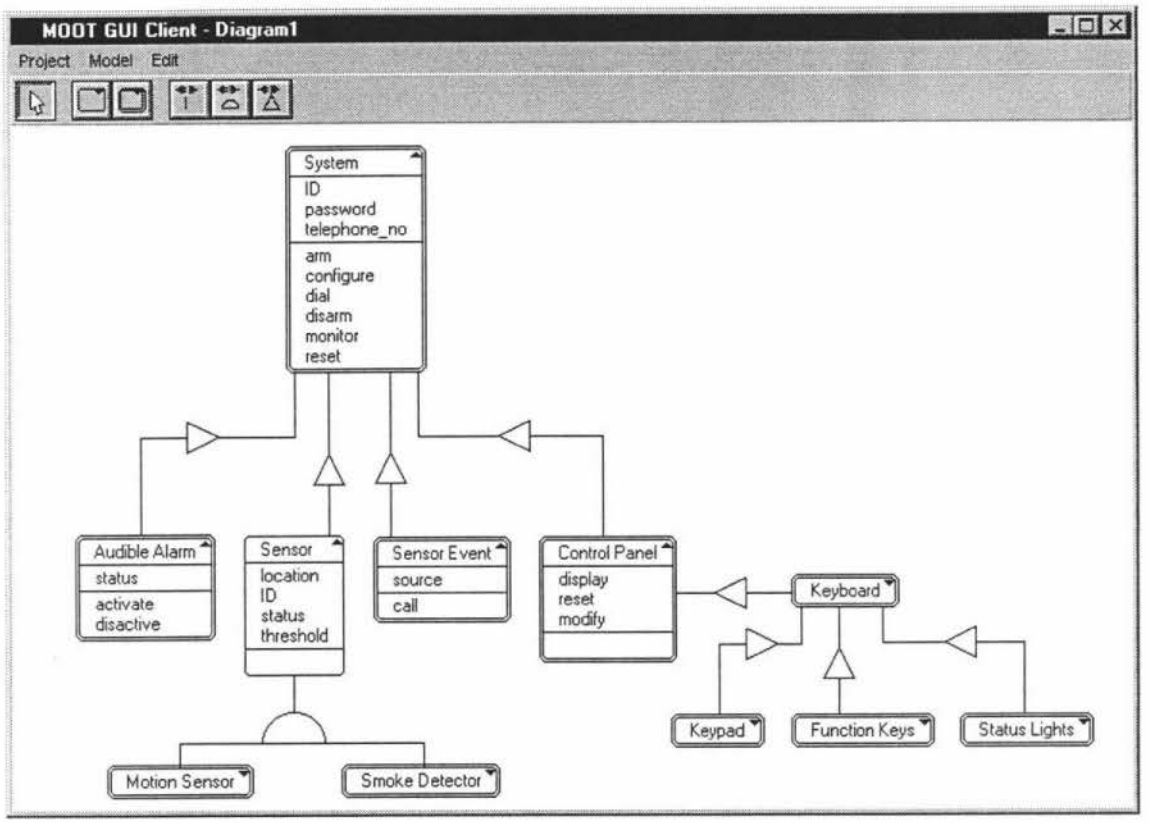



**Figure 8-1 – Sample Coad and Yourdon class diagram**

# Chapter 9

# CONCLUSION

The overall aim of the research presented in this thesis was the development of a prototype MOOT CASE Tool client that supported the use of arbitrary methodology notations for the construction of small-scale diagrams. A notation definition language, NDL, was to be used to specify the various notation elements that exist in a notation. The graphical user interface of the client needed to be simple, non-intrusive, and efficient, yet flexible, useable, and helpful. The communications protocol between the client and the MOOT CASE Tool server that applies the semantic definitions of a methodology to a project had to be considered also.

## 9.1. Review of the Notation Definition Language

The current version of NDL that has been developed in this thesis has proven, in use, to meet the initial design requirements. Various methodology notations have been examined, and a description in NDL of the Coad and Yourdon notation has been composed. The implementation of the language has allowed deficiencies to be found, enabling further proposals to be made to direct its evolution.

A notable problem that still exists with NDL is in the specification of connections. The current implementation of connections (using connection symbols and connection terminators) alleviates some of the existing problems that were inherent in previous research. The creation and placement of grouped connections is now trivial when the grouping is based around a connection symbol. The selection of individual graph objects of a connection for manipulation is also now possible due to each graph object being described by separate GUIComponent objects.

The current implementation however appears adequate only for notations where the type of a connection is specified by a connection symbol (eg. Coad and Yourdon). In many notations connection symbols do not exist, and the type of a connection is specified by connection terminators. The current implementation of NDL and the CASE Tool client graphical user interface make it difficult (although not impossible) to

create such connections. If a notation contained no connection symbols, only a single type of connection would be present in the toolbar of the model editor (a simple "association" type connection)[1]. To create a particular type of connection, an "association" would first have to be made, and then the appropriate connection terminators would have to be changed to reflect the type of connection required. This process actually changes the *type* of the connection, rather than *properties* of the connection, and would be very time consuming when numerous connections were required to be made.

To alleviate this problem, a connection template definition needs to be reintroduced into NDL. A connection template would not take away from the benefits currently provided by connection symbols and connection terminators, but rather would enhance their use. It is proposed that each type of connection that a notation supports would be defined specifically by a connection template. A connection template would define a connection in terms of:

- the arity (binary, ternary, etc) of the connection;
- a connection symbol (if any);
- the (possibly different) connection terminators that by default appear on the ends of connecting lines;
- other connection terminators that may be substituted for the defaults;
- line styles for the connecting lines (eg. thick lines (Coad and Yourdon Message Connection) or double lines (Booch (1991) Uses Relationship);
- how connection grouping may be supported.

The result would also improve the specification of docking areas that restrict the type of connections that can be attached to them. Restrictions could be described in terms of complete connection types, rather than the individual components of a connection.

To support connection templates in the toolbar of a model editor, each connection template may also need to define a small iconic representation of the connection it represents. This could be constructed easily through a collection of graphical components.

Another interesting question that has arisen through the development of NDL and the CASE Tool client is whether an NDL specification in its original representation would

[1] Currently, the model editor creates toolbar buttons for connections from the connection symbol templates described in a notation specification. A basic "association" type connection that contains no connection symbol is created by default on the toolbar.

ever be used. Currently the client uses only the serialised abstract syntax tree representation. An abstract syntax tree representation would typically also be used in a visual construction tool that supported the creation, modification, and testing of methodology notation descriptions. Assuming a complete MOOT system would include such a tool, it is feasible to state that an abstract syntax tree representation need be the only representation of a NDL notation specification necessary. A visual construction tool could output serialised notation abstract syntax trees for storage in the persistent store of MOOT. A serialised abstract syntax tree would then subsequently be passed to a client when it required a notation (as occurs presently). This process would remove an unnecessary representation, and unnecessary translation steps between the representations.

### 9.1.1. Future Work

Many additional improvements and extensions could be made to NDL to increase its flexibility and the level of notation description. An obvious extension is to reintroduce support for the specification of colours, line styles, fills, patterns, text fonts and styles, and other such features that can increase the visual appearance of notation elements.

Another significant improvement that could be made is to enhance the functionality of text areas. Two extensions are proposed here. These are the support for text wrapping and conditional expressions.

**Text Wrapping**

Text wrapping would be useful were lines of text became too long to be practically presentable in a notation element. For example, consider the process bubble shown in Figure 9-1. It would be very impractical to enlarge the bubble to the diameter that would be required to contain the text on a single line.


**Figure 9-1 – Text wrapping in a process bubble**

Text wrapping may also be useful in notation elements that contain multiple text areas. If the text contained in one text area was particularly wide relative to the other text areas, then a large amount of white space may be unnecessarily consumed by the

notation element. For example, consider the two UML class symbols shown in Figure 9-2. The class symbol on the left occupies a large amount of screen real estate, much of which is empty space. In an environment where space is limited (ie. a computer screen), the second symbol would be a preferred display representation, as the amount of screen space occupied is much less.

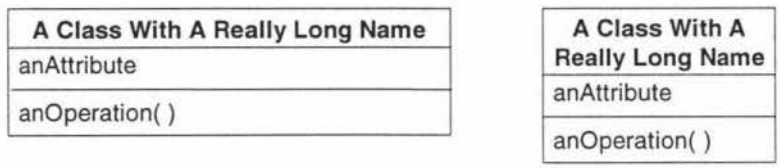



**Figure 9-2 – Text wrapping in a UML class symbol**

The functionality of text wrapping cannot be fulfilled by the current multi-line text areas as these text areas treat each line of text as an individual item. Manipulating this information would be prove tedious and there would be no simple way of identifying what was supposed to be a multi-line text area or a single wrapped line of text. A better alternative would be to define text wrapping as a property of a single-line text area that could be "enabled" or "disabled".

### Conditional Expressions

UML defines presentation techniques that allow text areas of notation elements to be expressed at various levels of detail. For example, two different ways that the attributes of a class symbol may be expressed are shown in Figure 9-3. The selective suppression of specific parts of a viewable property string is currently not supported in NDL. This may require some form of conditional expression syntax to be introduced into NDL. It would also mean that a method of allowing a user to change the level of expression would be required, along with a method for the system to remember the current expression level. This problem is similar to the logical distortion of entire symbols, however the problem is made more difficult as it applies to a basic notation primitive.

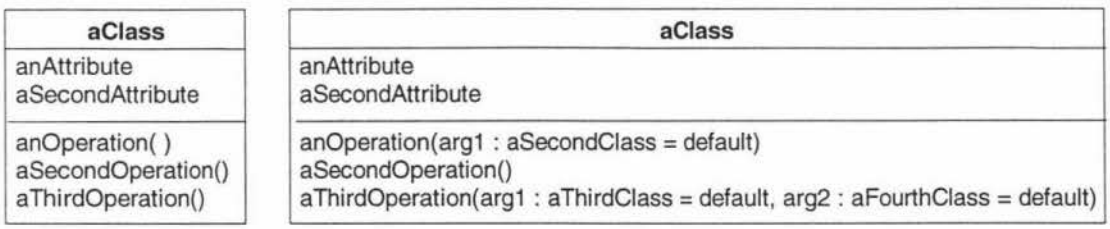



**Figure 9-3 – Text areas in UML class symbols showing varying levels of detail**

A further improvement that could be made to NDL is the support for purely text-based notations. The current version of NDL cannot handle these effectively, and would only be able to do so through describing notation elements by text areas alone.

## 9.2. Review of the CASE Tool Client Interface

The CASE Tool client prototype that has been developed in this thesis adequately meets most of the initial design requirements. Unfortunately due to time constraints, the implementation could not be completed. Text annotations to connections are not currently supported, and no implementation on the communications interface has been trialed. Despite this, the prototype has shown that the current approach to NDL is efficacious, and that the syntax and semantics of a methodology description can successfully be separated in definition.

Still to be evaluated is the proposed communications protocol between the CASE Tool client and server sub-systems. Once the communications interface has been implemented, a detailed examination of the flow of communications during a "typical" working session needs to be performed to determine if any problems exist. One particular concern is the delay which may be incurred whenever a user performs an action that causes a semantic change to a diagram. It is hoped that this delay will not be significant enough to be distracting, however this cannot be ascertained until a practical situation is tested.

### 9.2.1. Future Work

Many extensions need to be made to the client before it would satisfy the full requirements of a MOOT CASE tool. Some of these extensions include support for:

- notations where relative position information of object symbols carries semantic meaning;
- hyperlinks between diagrams (for example, the expansion of a process in a data flow diagram);
- more comprehensive distortion display techniques, including physical distortion.

In order to support reuse by importation of existing models or classes, the client also needs to provide a means to determine whether individual graph objects may be modified or not. For example, a class that was imported into a model from a previously defined library should not be able to be modified in the context of the new model.

An interface description language (IDL) may also need to be defined for the client graphical user interface. Such a language would allow the appearance of the interface to be customised for different software development methodologies. For example, different status windows could be defined and positioned alongside the main drawing editor. The drawing toolbar may also be customised further.

An IDL would also support the specification of interactive dialog boxes that may be opened as a result of user actions. For example, if a user selected an operation of a class symbol, a dialog box could be opened that allowed the specification of all the properties of that operation. In many circumstances such as this, it is impractical for every property of a text item to be displayed in the one text area. It would be difficult to display and navigate such information.

While an IDL would support customisation of the interface, the overall look and feel of the CASE Tool client would remain the same across all methodologies.

## 9.3. Conclusion

The progress that has been made in the development of NDL has shown that it is a viable method for describing the syntax of software engineering methodologies. Through additional work, NDL is set to become a powerful and flexible language that will be able to support all past, present, and future methodology notations. The development of the CASE Tool client, through its generic, methodology independent design, has provided a pilot with which future ideas may be explored. These generic features, coupled with the expressive power of NDL, promote a positive outlook to the MOOT CASE Tool project.

# GLOSSARY

**Active area** (NDL) – A region on a **notation element** that may invoke an action when selected by a user.

**CASE** – An acronym for Computer Aided Software Engineering.

**CASE tool** – A software tool that supports software engineers in part or all of the software development lifecycle.

**Client** – The part of a system that a user interacts with to perform a task. Clients themselves interact with a **server** for essential services.

**Concept** – A "thing" that exists in a particular problem domain, such as a class, object, or data store.

**Connection** – A visual representation of an association between two **concepts**.

**Connection symbol (NDL)** – A graphical symbol that exists in the middle of a **connection** to distinguish the type of association that the **connection** represents.

**Connection terminator (NDL)** – A graphical symbol that exists at the point where **connections** attach to **object symbols**. Connection terminators server to identify a particular type of association, or to enhance the meaning of an association.

**Context** (GUI) – A particular environment that **diagrams** may be viewed in.

**Diagram** – A visual representation of a **model** or part of a **model**.

**Docking area (NDL)** – The position on an **object symbol** or **connection symbol** where **connections** may be attached.

**Graph object (GUI)** – The smallest graphical entity existing in a **diagram** that can be manipulated by a user.

**Grouped connection (GUI)** – A collection of two or more **connections** in a single visual structure.

**Methodology** – A prescription of a process for one or more phases of the software development lifecycle. Different methodologies use their own particular set of **models** to describe a software artefact.

**Model** – A description of a particular aspect of a problem domain, such as the concepts that exist in it or the interaction between these concepts. A model is described in terms of one or more **diagrams**.

**MOOT** – An acronym for Massey's Object Oriented Tool. MOOT is a research project aimed at development a CASE tool that can support all methodologies in an interface that is consistent in look and feel.

**NDL** – An acronym for **Notation Definition Language**.

**Notation** – The visual syntax of a methodology.

**Notation Definition Language** – A language that supports the specification of arbitrary **methodology notations**.

**Notation element** – An **object symbol** or **connection** that may exist in a notation.

**Object symbol** – A visual representation of a **concept** that exists in a problem domain.

**Project** – A description of a software artefact. A project contains a number of **models** that have been created by the application of a particular **methodology**.

**Segment (NDL)** – A primitive component of a **view**, such as a line or a text string.

**Semantic Specification Language** – A language that supports the specification of the semantics of a **methodology**.

**Server** – The host of a system that provides common essential services to **clients**. This architecture allows the common services to be shared across a large number of simultaneous users of the system.

**SSL** – An acronym for **Semantic Specification Language**.

**Template (NDL)** – A definition of a **notation element**. Templates are composed of **template segments**.

**Template segment (NDL)** – A definition of a primitive component of a template, such as a line, text area, docking area, or active area.

**View** – A representation of an instance of a **notation element**. **Object symbols**, **connections symbols**, and **connection terminators** are views of their respective templates.

**Viewable property** – A property of a **view** that can be seen and possibly manipulated be a user.

**Viewable thing** – A collection of **viewable properties**. An NDL **view** is constructed by the application of a viewable thing to a **template**.

# REFERENCES


AHO, A.V., SETHI, R., and ULLMAN, J.D. (1986): *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, Massachusetts.

APPERLEY, M. and CHESTER, M. (1995): *Tree Browsing*, Working Paper 96/13, Department of Computer Science, The University of Waikato, Hamilton, New Zealand.

BOOCH, G. (1991): *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Co, Inc., Redwood City, California.

BOOCH, G. (1994): *Object Oriented Analysis and Design with Applications*, 2nd edition, The Benjamin/Cummings Publishing Co, Inc., Redwood City, California.

BROUGH, M. (1992): *Methods for CASE: A Generic Framework*, Advanced Information Systems Engineering: 4th International Conference CAiSE'92, Springer-Verlag, Berlin.

CHURCHER, N. and CERECKE, C. (1996): *GroupCRC: Exploring CSCW Support for Software Engineering*, Proceedings of OzCHI'96, Hamilton, New Zealand, pp 62-68.

CLARK, P. (1994): *Methodology Independent CASE Tool - A Prototype*. Massey University Masterate Thesis, Department of Computer Science, Massey University, Palmerston North, New Zealand.

COAD, P., YOURDON, E. (1991a): *Object-Oriented Analysis*, 2nd edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

COAD, P., YOURDON, E. (1991b): *Object-Oriented Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

D. COLEMAN, D., AENOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F. and JEREMAES, P. (1993): *Object-Oriented Development: The Fusion Method*, Prentice Hall, Inc., Englewood Cliffs, New Jersey.

COLLINS, D. (1995): *Designing Object-Orientated User Interfaces*, The Benjamin/Cummings Publishing Co, Inc., Redwood City, California.

CONSTANTINE, L. (1996): *Visual Coherence and Usability: A Cohesion Metric for Assessing the Quality of Dialogue and Screen Designs*, Proceedings of OzCHI'96, Hamilton, New Zealand, pp 115-121.

COOPER, C., and WERSTEIN, P. (1998): *The Use of Java to Develop a Microprocessor Emulator*, Software Engineering: Education and Practice 1998, Dunedin, January 1998.

DASARI, S., MEHANDJISKA, D. and PAGE, D. (1995): *Construction of a Generic Knowledge Base for a Methodology Independent CASE Tool*, Addendum to the ANNES'95 Proceedings, The Second NZ International Two-Stream Conference on Artificial Neural Networks and Expert Systems, Dunedin, November, 1995, pp 466-473.

FOWLER, M. and SCOTT, K. (1997): *UML Distilled – Applying the Standard Object Modelling Language*, Addison-Wesley Publishing Co., Reading, Massachusetts.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. (1995): *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, Massachusetts.

GEARY, D. and McCLELLAN, A. (1997): *Graphic Java – Mastering the AWT*, SunSoft Press, Prentice Hall, New Jersey.

GRAHAM, I. (1994): *Migrating to Object Technology*, Addison-Wesley, Workingham, UK.

GRUNDY, J., HOSKING, J., FENWICK, S. and MUGRIDGE, W. (1995): *Visual Object-Oriented Programming*, Chapter 11, edited by M. Burnett, A. Goldberg and T. Lewis, Manning/Prentice-Hall, New Jersey.

GRUNDY, J., HOSKING, J., and MUGRIDGE, R. (1998): *Experiences in Using Java on a Software Tool Integration Project*, Software Engineering: Education and Practice 1998, Dunedin, January 1998.

HAM, J. (1994): *Template Generator for a Methodology Independent Object-Oriented CASE Tool.* Massey University Honours Report, Department of Computer Science, Massey University, Palmerston North, New Zealand.

HENDERSON-SELLERS, B. and EDWARDS, J. (1994): *The Working Object*, Book Two of Object Oriented Knowledge, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

HENDERSON-SELLERS B. and GRAHAM, I. (1996): *OPEN: Toward Method Convergence?*, IEEE Computer, April, 1996.

IIVARI, J. (1996): *Why Are CASE Tools Not Used?*, Communications of the ACM, Vol. 39, No. 10, October 1996, pp. 94-103.

JACOBSON, I., CHRISTENSON, M., JONSSON, P., and ÖVERGAARD, G. (1994): *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley, Wokingham, England.

LANG, B. (1991): *CASE Support for the Software Process: Advances and Problems*, ESEC'91, edited by A. Lamsweerde and A. Fugetta, Springer-Verlag, Berlin.

LEUNG, Y. K., and APPERLEY, M. D. (1993): *A Taxonomy of Distortion-Oriented Display Techniques for Graphical Data Presentation*, Proceedings of HCI International'93, Orlando, Florida.

MARTIN, J. and ODELL, J. (1995): *Object-Oriented Methods*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

MARTTIIN, P. (1994): *Towards Flexible Process Support with a CASE Shell*, Advanced Information Systems Engineering, 6th International Conference, CaiSE '94, edited by G. Wijers, S. Brinkkemper, and T. Wasserman, Springer-Verlag, Berlin.

MEHANDJISKA, D., PAGE, D., CLARK, P. (1994): *An Intelligent Object-Oriented CASE Tool*, Proceedings of the 1st International Conference on Object-Oriented Information Systems, Springer-Verlag, London, December, 1994.

MEHANDJISKA, D., APPERLEY, M., PHILLIPS, C.H.E., PAGE, D. and CLARK, P. (1995): *A Methodology Independent Object-Oriented CASE Tool*, Proceedings NZCS'95.

MEHANDJISKA, D., PAGE, D. and CHOI, M.D. (1996a): *Meta-modelling and Methodology Support in Object-Oriented CASE Tools*, in Proceedings OOIS'96, Springer-Verlag, London, 1996.

MEHANDJISKA, D., PAGE, D. and DASARI, S. (1996b): *Generic Knowledge Base for a Methodology Independent Object-Oriented CASE Tool*, IASTED International Conference of Artificial Intelligence, Expert Systems and Neural Networks, Honolulu, Hawaii, USA, August 1996.

MEHANDJISKA, D., PAGE, D., GRIFFIN, D. and USHERWOOD, L. (1997): *Methodology Knowledge Representation and Interpretation for a Methodology Independent OO CASE Tool*, IASTED International Conference on Software Engineering, November, 1997.

NIELSEN, J. (1993): *Usability Engineering*, Academic Press Ltd, London.

NILSSON, E. (1990): *CASE Tools and Software Factories in Advanced Information Systems Engineering*, CAiSE'90, edited by G. Goos and J. Hartmanis, Springer-Verlag, Berlin.

NOBLE, J. (1996): *A Diagram Editor with a Minimal Interface*, in Proceedings of OzCHI'96, Hamilton, New Zealand, pp 163-165.

PAGE, D., CLARK, P. ad MEHANDJISKA-STAVREVA, D. (1994): *An Abstract Definition of Graphical Notations for Object Orientated Information Systems*, in Proceedings OOIS'94, Springer-Verlag, London.

PAGE, D., GRIFFIN, D., USHERWOOD, L. and MEHANDJISKA, D. (1997): *Implementation of a Semantic Specification Language Interpreter for a Methodology Independent OO CASE Tool*, IASTED International Conference on Software Engineering, November, 1997.

PAGE, D., MEHANDJISKA, D., PHILLIPS, C.H.E. (1998): *Methodology Independent OO CASE Tool: Supporting Methodology Engineering*, Software Engineering: Education and Practice 1998, Dunedin, January 1998.

PAGE, D. (to appear): *Methodology Representation for a Methodology Independent Object Orientated Case Tool.* Massey University Doctorate Thesis, Department of Computer Science, Massey University, Palmerston North, New Zealand.

PAPAHRISTOS, S. and GRAY, W. (1991): *Federated CASE Environment in Advanced Information Systems Engineering*, CAiSE'91, edited by G. Goos and J. Hartmanis, Springer-Verlag, Berlin.

PLAISANT, C., CARR, D. and SHNEIDERMAN, B. (1995): *Image-Browser Taxonomy and Guidelines for Designers"*, IEEE Software, March 1995, pp 21-32.

RATIONAL SOFTWARE CORPORATION (1997): *UML Notation Guide, version 1.0*, http://www.rational.com

RATIONAL SOFTWARE CORPORATION (1998): *Rational Rose*, http://www.rational.com

ROSSI, M., GUSTAFSSON, M., SMOLANDER, K., JOHANSSON, L. and LYYTINEN, K. (1992): *Meta-Modelling Editors as a Front End Tool for a CASE Shell*, CAiSE'92, edited by P. Loucopoulos, Springer-Verlag, Berlin.

RUMBAUGH, J., (1991): *Object Oriented Modelling and Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

RUMBAUGH, J. (1995a): *OMT: The Functional Model*, Journal of Object-Oriented Programming, Vol. 8, No. 1, 1995.

RUMBAUGH, J. (1995b): *OMT: The Object Model*, Journal of Object-Oriented Programming, Vol. 7, No. 8, 1995.

SCHACH, S.R. (1997): *Software Engineering with Java*, Richard D. Irwin (Times Mirror Higher Education Group, Inc.), Chicago.

SHLAER, S. and MELLOR, S.J. (1991): *Object Lifecycles: Modelling the World in States*, Yourdon Press/Prentice Hall.

SHNEIDERMAN, B. (1998): *Designing the User Interface*, Addison-Wesley Publishing Co., Reading, Massachusetts.

SMITH, R. and ANDERSON, P. (1996): *Relating Distortion to Performance in Distortion Orierted Displays*, Proceedings of OzCHI'96, Hamilton, New Zealand, pp 6-11.

SMOLANDER, K, LYYTINEN, K., TAHVANAINEN, V.-P. and MARTTIIN, P. (1991): *MetaEdit: A Flexible Graphical Environment for Methodology Modelling*, in Advanced Information Systems Engineering, in Proceedings CAiSE'91 edited by G. Goos and J. Hartmanis, Springer-Verlag, Berlin, 1991.

SORENSON, P. (1988a): *First Generation CASE Tools: All Form but Little Substance*, Research Report, Department of Computational Science, University of Saskatchewan, Saskatchewan, Canada.

SORENSON, P. (1988b): *On The Metaview System for Many Specification Environments*, IEEE Software, Vol 5, No. 2, March, 1988.

STROUSTRUP, B. (1997): *The C++ Programming Language*, 3rd edition, Addison-Wesley Publishing Co., Reading, Massachusetts.

SUMNER, M. (1992): *The Impact of Computer-Assisted Software Engineering on Systems Development*, IFIP Transactions - The Impact of Computer Supported Technologies on Information Systems Development, edited by K.E. Kendall, K. Lyytinen, and J.I. DeGross, Elsevier Science Publishers, Amsterdam.

SUN Microsystems (1996): *The Java Language: An Overview*, http://java.sun.com/docs/overviews/ java/java-overview-1.html

SUN Microsystems (1997): *The JDK 1.1.1 Documentation*, http://java.sun.com/ products/jdk/1.1/docs

SUN Microsystems (1998): *Java Home Page*, http://java.sun.com

WIRFS-BROCK, R. (1990): *Designing Object Oriented Software*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey.

# Appendix A

# EBNF Definition

**Standard BNF**

A grammar is used to describe what sorts of phrases exist in a language, and how these phrases may be built up. Backus-Nauer Form (BNF) (Aho *et al*, 1986) is a common notation that can be used to described these phrases as a set of production rules. For example, a grammar for Java would contain the phrase types <Statement> and <Condition>, and would have a rule to state that:

a <Statement> can be "if ( <Condition> ) <Statement>;"

In BNF, this is written:

<Statement> ::= if ( <Condition> ) <Statement>;

The symbol "::=" is read as "can be". The phrase types defined in the grammar, such as <Statement> and <Condition> are meta-symbols of the language. Meta-symbols are used when talking about the language, rather than being actual symbols of the language. In BNF, meta-symbols are always enclosed in angle brackets, and are referred to as non-terminal symbols. Symbols that do not appear in angle brackets are called terminal symbols. Terminal symbols are symbols such as characters, numbers, or strings that actually appear in sentences of the language when an instantiation of the grammar is made. Terminal symbols do not represent phrases, and cannot be constructed out of smaller phrases. In the example above, the symbol "if" is a terminal symbol. Terminal symbols can never appear on the left-hand side of a production rule.

The BNF notation supports a shorthand notation for expressing production rules that have the same left-hand side. For example, consider the following grammar that describes a set of colours:

```
<Colour>        ::= blue
<Colour>        ::= red
<Colour>        ::= green
<Colour>        ::= yellow
```

These production rules can be written as a single rule as follows:

```
<Colour>        ::= blue | red | green | yellow
```

The vertical bar symbol, '|', represents an "exclusive-or" relationship, meaning that one (and only one) selection may be made to satisfy the production rule. In a particular context, a <Colour> therefore can be either blue or red or green or yellow.

**EBNF**

Extended Backus-Nauer Form (EBNF) is an extension to the standard BNF notation that adds support for optional or repeating groups of symbols in a production rule. Optional groups of symbols are delineated in a production rule by square brackets ('[' and ']'). Optional symbols may appear zero or once in instantiations of that production rule. For example, consider the following grammar:

```
<Draw_Command>  ::= draw [ <Colour> ] <Figure>
<Colour>        ::= blue | red | green | yellow
<Figure>        ::= line | arc
```

A <Draw_Command> phrase consists of the terminal symbol "draw," optionally followed by a drawing colour (blue, red, green, or yellow), and concluded with a figure to draw (a line or arc).

Repeating groups of symbols are delineated by braces ('{' and '}'). Repeating groups may appear zero or more times. The following example illustrates this through the definition of a number, which can consist a digit optionally followed by any number of further digits:

```
<Number>        ::= <Digit> { <Digit> }
<Digit>         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

EBNF also supports parentheses ('(' and ')') to group alternatives. This is typically used in optional or repeating groups to limit the scope of other modifiers (such as '|').

To illustrate this, the previous grammar could be extended to incorporate optional sign information for numbers:

```
<Number>        ::= [ ( '+' | '-' ) ] <Digit> { <Digit> }
<Digit>         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This says that a <Number> can optionally be preceded by a sign, and the sign can either be a positive or negative sign. Note that the single character signs are enclosed in single quotes. This distinguishes them as actual symbols of the language, and improves clarity where confusion may arise with the meta-characters | [ ] { } < and >. It is often a good idea to place all single character, non-alphanumeric characters in quotes, as in the example above, for the reason of clarity. If a meta-character symbol is required to appear as part of the grammar itself, then it needs to be enclosed in single quotes to point this out, as all symbols enclosed in single quotes are interpreted as terminal symbols. For example, the <Statement> production rule that was described earlier would be better written as follows, with the parentheses and semi-colon enclosed in quotes:

```
<Statement>     ::= if '(' <Condition> ')' <Statement> ';'
```

# Appendix B

## Notation Definition Language Grammar

This appendix lists the complete EBNF grammar for the Notation Definition Language (NDL) that was described in Chapter 3 and Chapter 4. It is grouped into subsections for easier navigation.

**Notation:**

```
<Notation>              ::= NOTATION <String> '{' <Template_Collection> '}'

<Template_Collection>   ::= { ( <Group_Template> | <Object_Template> |
                              <Connection_Symbol_Template> |
                              <Connection_Terminator_Template> ) }
```

**Templates:**

```
<Group_Template>        ::= GROUP <Identifier> '{'
                              [ <Equations> ] <Components>
                              [ <Active_Areas> ] <Bounding_Region>
                            '}'

<Object_Template>       ::= OBJECT <String> '{'
                              [ <Equations> ] <Components> [ <Active_Areas> ]
                              [ <Docking_Areas> ] [ <Object_Properties> ]
                              <Bounding_Region>
                            '}'
<Object_Properties>     ::= PROPERTIES '{' [ <Default_Text_Property> ] '}'
<Default_Text_Property> ::= DEFAULT_TEXT_AREA <Identifier> ';'
```

```
<Connection_Symbol_Template> ::= CONNECTION_SYMBOL <String> '{'
                                 [ <Expressions> ] <Components>
                                 <Connection_Point> <Docking_Areas>
                                 [ <Connection_Symbol_Properties> ]
                                 <Bounding_Region>
                               '}'
<Connection_Point >          ::= CONNECTION_POINT <Point> ';'
<Connection_Symbol_Properties> ::= PROPERTIES '{' [ <Constraint> ] [ <Arity> ] '}'
<Constraint>                 ::= ( ROTATIONALLY_CONSTRAINED |
                                   ROTATIONALLY_UNCONSTRAINED ) ';'
<Arity>                      ::= ARITY <Integer> ';'

<Connection_Terminator_Template> ::= CONNECTION_TERMINATOR <String> '{'
                                 [ <Expressions> ] <Components> <Head> <Tail>
                                 [ <Connection_Terminator_Properties> ]
                                 <Bounding_Region>
                               '}'
<Head>                       ::= HEAD <Point> ';'
<Tail>                       ::= TAIL <Point> ';'
<Connection_Terminator_Properties> ::= PROPERTIES '{' [ < Constraint> ] '}'
```

**Equations:**

```
<Equations>                  ::= EQUATIONS '{' { <Assignment> } '}'
<Assignment>                 ::= <Identifier> '=' <Expression> ';'
<Expression>                 ::= '+' <Expression> <Expression> |
                                 '-' <Expression> <Expression> |
                                 '*' <Expression> <Expression> |
                                 '/' <Expression> <Expression> |
                                 <Term>
<Term>                       ::= <Number> | <Identifier> | <Function>

<Function>                   ::= <Text_Function> | <Group_Function> |
                                 <Expression_Function>
<Text_Function>              ::= <Text_Width_Function> | <Text_Height_Function>
<Group_Function>             ::= <Group_Width_Function> | <Group_Height_Function>
<Expression_Function>        ::= <Max_Function> | <Min_Function>

<Text_Width_Function>        ::= TEXTWIDTH '(' <Identifier> ')'
```

<Text_Height_Function> ::= TEXTHEIGHT '(' <Identifier> ')'
<Group_Height_Function>::= GROUPHEIGHT '(' <Identifier> ')'
<Group_Width_Function> ::= GROUPWIDTH '(' <Identifier> ')'
<Max_Function> ::= MAX '(' <Expression> { ',' <Expression> } ')'
<Min_Function> ::= MIN '(' <Expression> { ',' <Expression> } ')'

**Graphical Components:**

<Graphic_Components> ::= COMPONENTS '{' { <Component_Template> } '}'
<Component_Template> ::= <Line_Template> | <Arc_Template> |
<Single_Text_Template> | <Multi_Text_Template> |
<Group_Reference>

<Line_Template> ::= LINE <Point> TO <Point> { TO <Point> } ';'

<Arc_Template> ::= ARC IN BOX <Point> TO <Point>
STARTING <Direction> EXTENDING <Number> ';'
<Direction> ::= <Compass_Direction> | <Number>
<Compass_Direction> ::= NORTH | SOUTH | EAST | WEST | NORTHEAST |
NORTHWEST | SOUTHEAST | SOUTHWEST

<Single_Text_Template> ::= SINGLETEXT <Identifier> <Point> [ <String> ] ';'
<Multi_Text_Template> ::= MULTITEXT <Identifier> <Point>
[ <String> { ',' <String> } ] ';'

<Group_Reference> ::= GROUP <Identifier> CONSISTS_OF <Identifier>
AT <Point> ';'

**Active Areas:**

<Active_Areas> ::= ACTIVE_AREAS '{' { <Active_Area> } '}'
<Active_Area> ::= <Point> TO <Point> <Action> ';'

<Action> ::= <Update_Action> | <Transition_Action>
<Update_Action> ::= UPDATE <Identifier>
<Transition_Action> ::= TRANSITION TO <Identifier>

**Docking Areas:**

```
<Docking_Areas>          ::= DOCKING_AREAS '{' { <Docking_Area> } '}'
<Docking_Area>           ::= <Point_Docking> | <Line_Docking> | <Arc_Docking>

<Point_Docking>          ::= POINT <Point> START_ANGLE <Direction>
                             EXTENDING <Number>
                             [ WITH <Connection_Count> CONNECTIONS ]
                             [ HAVING <Connection_Types> ]
<Direction>              ::= <Compass_Direction> | <Number>
<Connection_Count>       ::= <Integer> | UNLIMITED
<Connection_Types>       ::= [ <Terminator_List> ] [ <Connection_Symbol_List> ]
<Terminator_List>        ::= TERMINATORS '(' <Name_List> ')'
<Connection_Symbol_List> ::= SYMBOLS '(' [ NONE [ ',' ] ] <Name_List> ')'
<Name_List>              ::= <String> { ',' <String> }

<Line_Docking>           ::= LINE <Point> TO <Point> [IS <Extendable>]
                             [WITH <Connection_Count> CONNECTIONS]
                             [ HAVING <Connection_Types> ]
                             [ MIN_DISTANCE <Number> ] ';'
<Extendable>             ::= EXTENDABLE | UNEXTENDABLE

<Arc_Docking>            ::= ARC IN BOX <Point> TO <Point> STARTING <Number>
                             EXTENDING <Number>
                             CONNECTED ( INSIDE | OUTSIDE )
                             [ WITH <Connection_Count> CONNECTIONS ]
                             [ HAVING <Connection_Types]
                             [ MIN_DISTANCE <Number> ] ';'
```

**Bounding Region:**

```
<Bounding_Region>        ::= BOUNDING_REGION <Point>;
```

**General:**

```
<Point>                  ::= '[' <Expression> ',' <Expression> ']'

<Identifier>             ::= <Letter> { <Alpha_Numeric> }
<String>                 ::= '"' { ( <String_Character> | ' ' ) } '"'
<String_Character>       ::= <Alpha_Numeric> | <Special_Character>
<Alpha_Numeric>          ::= <Letter> | <Digit> |
```

<Letter> ::= <Lowercase_Letter> | <Uppercase_Letter> | '_'

<Lowercase_Letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<Uppercase_Letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<Special_Character> ::= ` | ~ | ! | @ | # | $ | % | ^ | & | * | '(' | ')' | - | = | \\ | + | '|' | '[' | ']' | '{' | '}' | ; | : | '"' | , | . | / | '<' | '>' | ? | \"

<Number> ::= <Real> | <Integer>

<Real> ::= <Integer> '.' <Digit> { <Digit> }

<Integer> ::= [ '-' ] <Digit> { <Digit> }

<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# Appendix C

## NOTATION ABSTRACT SYNTAX TREE SERIALISATION GRAMMAR

This appendix lists the complete EBNF grammar for the serialisation of the abstract syntax tree representation of a notation specification, as described in Chapter 7. It is grouped into subsections for easier navigation.

**Notation:**

```
<Notation>                ::= '[' NTN: <String>
                                <Groups> <Objects> <Connection_Symbols>
                                <Connection_Terminators>
                              ']'

<Groups>                  ::= '[' GRPS: { <Group_Template> } ']'
<Objects>                 ::= '[' OBJTS: { <Object_Template> } ']'
<Connection_Symbols>      ::= '[' CONSYMS: { <Connection_Symbol_Template> } ']'
<Connection_Terminators> ::= '[' CONTERMS: { <Connection_Terminator_Template> }
                              ']'
```

**Templates:**

```
<Group_Template>          ::= '[' GST: <Identifier>
                                <Expressions> <Graphical_Components>
                                <Active_Areas> <Bounding_Region>
                              ']'

<Template>                ::= '[' TMPL: <String>
                                <Expressions> <Graphical_Components>
                                <Bounding_Region>
                              ']'
```

```
<Object_Template>          ::= '[' OBJTMPL: <Template>
                                  <Active_Areas> <Docking_Areas>
                                  <Default_Text_Property>
                               ']'
<Default_Text_Property>    ::= '[' DEFTXT: '(' <Integer> ',' <Integer> ')' ']'

<Connection_Template>      ::= '[' CONTMPL: <Template>
                                  <Flag>
                               ']'
<Flag>                     ::= 0 | 1

<Connection_Symbol_Template> ::=
                               '[' CONSYMTMPL: <Connection_Template>
                                  <Point> <Docking_Areas> <Arity>
                               ']'
<Arity>                    ::= '[' ARITY: <Integer> ']'

<Connection_Terminator_Template> ::=
                            '[' CONTERMTMPL: <Connection_Template>
                               <Point> ',' <Point>
                            ']'
```

**Expressions:**

```
<Expressions>              ::= '[' EXPS: { '[' <Expression> ']' } ']'

<Expression>               ::= '+' <Expression> <Expression> |
                               '-'<Expression> <Expression> |
                               '*' <Expression> <Expression> |
                               '/' <Expression> <Expression> |
                               <Term>
<Term>                     ::= <Number> | <Function> | <Expression_Reference>
<Function>                 ::= <Text_Function> | <Group_Function> |
                               <Expression_Function>
<Expression_Reference>     ::= '$exp$(' <Integer> ')'

<Text_Function>            ::= <Text_Width_Function> | <Text_Height_Function>
<Group_Function>           ::= <Group_Width_Function> | <Group_Height_Function>
<Expression_Function>      ::= <Max_Function> | <Min_Function>
```

```
<Text_Width_Function>    ::= '$txtWidth$(' <Integer> ')'
<Text_Height_Function>   ::= '$txtHeight$(' <Integer> ')'
<Group_Width_Function>   ::= '$grpWidth$(' <Integer> ')'
<Group_Height_Function>::= '$grpHeight$(' <Integer> ')'
<Max_Function>           ::= '$max$(' <Expression> { ',' <Expression> } ')'
<Min_Function>           ::= '$min$(' <Expression> { ',' <Expression> } ')'
```

**Graphical Components:**

```
<Graphical_Components> ::= '[' COMPS: { <Graphical_Component > } ']'
<Graphical_Component>  ::= <Line> | <Arc> | <Single_Text> | <Multi_Text> |
                           <Group_Reference>

<Line>                 ::= '[' LN: <Point> '-' <Point> { '-' <Point> } ']'
<Arc>                  ::= '[' ARC: <Point> '-' <Point> ',' <Number> ',' <Number> ']'
<Single_Text>          ::= '[' ST: <Integer> ',' <Point> [ <String> ] ']'
<Multi_Text>           ::= '[' MT: <Integer> ',' <Point> { <String> } ']'
<Group_Reference>      ::= '[' GRP: <Integer> <Identifier> AT <Point> ']'
```

**Active Areas:**

```
<Active_Areas>         ::= '[' AA: { <Active_Area> } ']'
<Active_Area>          ::= '[' AAT: <Point> '-' <Point> <Action> ']'

<Action>               ::= <Update_Action> | <Transition_Action>
<Update_Action>        ::= '[' UACT: <Integer> ']'
<Transition_Action>    ::= '[' TACT: <Identifier> ']'
```

**Docking Areas:**

```
<Docking_Areas>        ::= '[' DA: { <Docking_Area> } ']'
<Docking_Area>         ::= <Point_Docking> | <Line_Docking> | <Arc_Docking>

<Point_Docking>        ::= '[' PT:
                               <Point> ',' <Connection_Count> ','
                               <Connectivity_Arc> <Allowable_Connectors>
                           ']'
<Connection_Count>     ::= 'u' | <Integer>
<Connectivity_Arc>     ::= '[' CONARC: '(' <Number> ',' <Number> ')' ']'
<Allowable_Connectors> ::= '[' [ <Allowable_Terminators> ]
                               [ <Allowable_Symbols> ] ']'
```

<Allowable_Terminators> ::= TERMS: <Name_List>
<Allowable_Symbols> ::= SYMBOLS: <Name_List>
<Name_List> ::= '(' { <String> } ')'

<Line_Docking> ::= '[' LN:
<Flag> ',' <Point> '-' <Point> ',' <Connection_Count>
',' <Number> <Allowable_Connectors>
']'

<Arc_Docking> ::= '[' ARC:
<Point> '-' <Point> ',' <Number> ',' <Number> ','
<Flag> ',' <Connection_Count> ',' <Number> ','
<Allowable_Connectors>
']'

**Bounding Region:**

<Bounding_Region> ::= '[' BR: <Point> ']'

**General:**

<Point> ::= '(' <Expression> ',' <Expression> ')'

<Identifier> ::= '_' { <Alpha_Numeric> }
<String> ::= '"' { ( <String_Character> | ' ' ) } '"'
<String_Character> ::= <Alpha_Numeric> | <Special_Character>
<Alpha_Numeric> ::= <Letter> | <Digit>
<Letter> ::= <Lowercase_Letter> | <Uppercase_Letter> | '_'
<Lowercase_Letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<Uppercase_Letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<Special_Character> ::= ` | ~ | ! | @ | # | $ | % | ^ | & | * | '(' | ')' | - | = | \\ | + | '|' | '[' | ']' | '{' | '}' | ; | : | '"' | , | . | / | '<' | '>' | ? | \"
<Number> ::= <Real> | <Integer>
<Real> ::= <Integer> '.' <Digit> { <Digit> }
<Integer> ::= [ '-' ] <Digit> { <Digit> }
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# Appendix D

## SAMPLE NOTATION SPECIFICATIONS

This appendix lists the notation specifications that were written for the Coad and Yourdon notation described in use in this thesis. An NDL specification, and the equivalent serialised abstract syntax tree representation, is given. The specifications are instantiations of the EBNF grammars described in Appendix B and Appendix C, respectively.

### NDL Specification

```
NOTATION "Coad&Yourdon" {

   GROUP upArrow {
      COMPONENTS {
         LINE [3,0] TO [3,0];
         LINE [2,1] TO [4,1];
         LINE [1,2] TO [5,2];
         LINE [0,3] TO [6,3];
      }
      BOUNDING_REGION [6, 3];
   }

   GROUP dnArrow {
      COMPONENTS {
         LINE [0,0] TO [6,0];
         LINE [1,1] TO [5,1];
         LINE [2,2] TO [4,2];
         LINE [3,3] TO [3,3];
      }
      BOUNDING_REGION [6, 3];
   }

   GROUP classgroup {
      EQUATIONS {
         lineAY = + 4 TEXTHEIGHT(classname);
         tbBY = + 2 lineAY;
         lineBY = + + 4 lineAY TEXTHEIGHT(attributes);
         tbCY = + 2 lineBY;
         height = + + 4 lineBY TEXTHEIGHT(operations);
         width = + 4 MAX(TEXTWIDTH(classname), TEXTWIDTH(attributes),
                         TEXTWIDTH(operations));
         upArrowX = - - width GROUPWIDTH(upArrow) 1;
         aawidth = - width 4;
      }
      COMPONENTS {
         TEXT classname [2, 2];
         LISTTEXT attributes [2, tbBY];
```

```
        LISTTEXT operations [2, tbCY];
        LINE [3, 0] TO [- width 3, 0];
        LINE [width, 3] TO [width, - height 3];
        LINE [- width 3, height] TO [3, height];
        LINE [0, - height 3] TO [0, 3];
        ARC IN BOX [0, 0] TO [6, 6] STARTING 90 EXTENDING 90;
        ARC IN BOX [- width 6, 0] TO [width, 6] STARTING 0
            EXTENDING 90;
        ARC IN BOX [0, - height 6] TO [6, height] STARTING 180
            EXTENDING 90;
        ARC IN BOX [- width 6, - height 6] TO [width, height]
            STARTING 270 EXTENDING 90;
        LINE [0, lineAY] TO [width, lineAY];
        LINE [0, lineBY] TO [width, lineBY];
        GROUP arrow CONSISTS_OF upArrow AT [upArrowX,2];
    }
    ACTIVE_AREAS {
        [upArrowX, 2] TO
            [+ upArrowX GROUPWIDTH(upArrow), + 2 GROUPHEIGHT(upArrow)]
            TRANSITION TO classnamegroup;
        [2, 2] TO
            [+ 2 aawidth, + 2 TEXTHEIGHT(classname)] UPDATE classname;
        [2, tbBY] TO [+ 2 aawidth, + tbBY TEXTHEIGHT(attributes)]
            UPDATE attributes;
        [2, tbCY] TO [+ 2 aawidth, + tbCY TEXTHEIGHT(operations)]
            UPDATE operations;
    }
    BOUNDING_REGION [width, height];
}

GROUP classnamegroup {
    EQUATIONS {
        height = + 4 TEXTHEIGHT(classname);
        width = + 4 TEXTWIDTH(classname);
        dnArrowX = - - width GROUPWIDTH(dnArrow) 1;
    }
    COMPONENTS {
        TEXT classname [2,2];
        LINE [3, 0] TO [- width 3, 0];
        LINE [width, 3] TO [width, - height 3];
        LINE [- width 3, height] TO [3, height];
        LINE [0, - height 3] TO [0, 3];
        ARC IN BOX [0, 0] TO [6, 6] STARTING 90 EXTENDING 90;
        ARC IN BOX [- width 6, 0] TO [width, 6] STARTING 0
            EXTENDING 90;
        ARC IN BOX [0, - height 6] TO [6, height] STARTING 180
            EXTENDING 90;
        ARC IN BOX [- width 6, - height 6] TO [width, height]
            STARTING 270 EXTENDING 90;
        GROUP arrow CONSISTS_OF dnArrow AT [dnArrowX, 2];
    }
    ACTIVE_AREAS {
        [dnArrowX, 2] TO
            [+ dnArrowX GROUPWIDTH(dnArrow), + 2 HEIGHT(dnArrow)]
            TRANSITION TO classgroup;
        [2,2] TO
            [+ 2 TEXTWIDTH(classname), + 2 TEXTHEIGHT(classname)]
            UPDATE classname;
    }
    BOUNDING_REGION [width, height];
}
```

```
SYMBOL "Abstract class" {
   COMPONENTS {
      GROUP symbolgroup CONSISTS_OF classnamegroup AT [0,0];
   }
   DOCKING_AREAS {
      LINE [0,3] TO [0, - GROUPHEIGHT(symbolgroup) 3]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS;
      LINE [3, GROUPHEIGHT(symbolgroup)] TO
         [- GROUPWIDTH(symbolgroup) 3, GROUPHEIGHT(symbolgroup)]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS;
      LINE [GROUPWIDTH(symbolgroup), - GROUPHEIGHT(symbolgroup) 3]
         TO [GROUPWIDTH(symbolgroup), 3]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS;
      LINE [- GROUPWIDTH(symbolgroup) 3, 0] TO [3, 0]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS;
   }
   PROPERTIES {
      DEFAULT_TEXT_AREA classname;
   }
   BOUNDING_REGION [GROUPWIDTH(symbolgroup),
                    GROUPHEIGHT(symbolgroup)];
}

SYMBOL "Class" {
   EQUATIONS {
      classheight = + 4 GROUPHEIGHT(classnamegroup);
      classwidth = + 4 GROUPWIDTH(classnamegroup);
      lineY = - classheight 5;
      lineX = - classwidth 5;
   }
   COMPONENTS {
      GROUP symbolgroup CONSISTS OF classnamegroup AT [2, 2];
      LINE [5, 0] TO [lineX, 0];
      LINE [classwidth, 5] TO [classwidth, lineY];
      LINE [lineX, classheight] TO [5, classheight];
      LINE [0, lineY] TO [0, 5];
      ARC IN BOX [0, 0] TO [10, 10] STARTING 90 EXTENDING 90;
      ARC IN BOX [- classwidth 10, 0] TO [classwidth, 10]
          STARTING 0 EXTENDING 90;
      ARC IN BOX [0, - classheight 10] TO [10, classheight]
          STARTING 180 EXTENDING 90;
      ARC IN BOX [- classwidth 10, - classheight 10] TO
          [classwidth, classheight] STARTING 270 EXTENDING 90;
   }
   DOCKING_AREAS {
      LINE [0, 5] TO [0, lineY] IS NON_EXTENDABLE WITH UNLIMITED
         CONNECTIONS HAVING SYMBOLS NONE, "Whole-part";
      LINE [5, classheight] TO [lineX, classheight]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS
         HAVING SYMBOLS NONE, "Whole-part";
      LINE [classwidth, lineY] TO [classwidth, 5] IS NON_EXTENDABLE
         WITH UNLIMITED CONNECTIONS
         HAVING SYMBOLS NONE, "Whole-part";
      LINE [lineX, 0] TO [5,0] IS NON_EXTENDABLE WITH UNLIMITED
         CONNECTIONS HAVING SYMBOLS NONE, "Whole-part";
      LINE [2, 5] TO [2, lineY] IS NON_EXTENDABLE WITH UNLIMITED
         CONNECTIONS HAVING SYMBOLS "Inheritance";
      LINE [5, - classheight 2] TO [lineX, - classheight 2]
         IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS
         HAVING SYMBOLS "Inheritance";
```

```
            LINE [- classwidth 2, lineY] TO [- classwidth 2, 5]
               IS NON_EXTENDABLE WITH UNLIMITED CONNECTIONS
               HAVING SYMBOLS "Inheritance";
            LINE [lineX, 2] TO [5,2] IS NON_EXTENDABLE WITH UNLIMITED
               CONNECTIONS HAVING SYMBOLS "Inheritance";
         }
         PROPERTIES {
            DEFAULT_TEXT_AREA classname;
         }
         BOUNDING_REGION [classwidth,classheight];
      }

      CONNECTION_SYMBOL "Inheritance" {
         COMPONENTS {
            LINE [0, 20] TO [40, 20];
            ARC IN BOX [0, 0] TO [40, 40] STARTING 0 EXTENDING 180;
         }
         CONNECTION_POINT [20,0];
         DOCKING_AREAS {
            LINE [0, 20] TO [40, 20] IS EXTENDABLE
               WITH UNLIMITED CONNECTIONS;
         }
         PROPERTIES {
            ROTATIONALLY_CONSTRAINED;
            ARITY 2;
         }
         BOUNDING_REGION [40, 20];
      }

      CONNECTION_SYMBOL "Whole-part" {
         COMPONENTS {
            LINE [0, 20] TO [20, 20] TO [10, 0] TO [0, 20];
         }
         CONNECTION_POINT [10, 0]
         DOCKING_AREAS {
            LINE [0, 20] TO [20, 20] IS EXTENDABLE
               WITH UNLIMITED CONNECTIONS;
         }
         PROPERTIES {
            ROTATIONALLY_CONSTRAINED;
            ARITY 2;
         }
         BOUNDING_REGION [20, 20];
      }

      CONNECTION_TERMINATOR default {
         COMPONENTS {
            LINE [0, 3] TO [12, 3];
         }
         HEAD [0, 3];
         TAIL [12, 3];
         PROPERTIES {
            ROTATIONALLY_CONSTRAINED;
         }
         BOUNDING_REGION [12, 6];
      }
   }
```

## AST Notation

```
[NTN: "Coad&Yourdon"

[GRPS:
   [GST: _upArrow
      [EXPS:]
      [COMPS:
         [LN: (3,0)-(3,0)]
         [LN: (2,1)-(4,1)]
         [LN: (1,2)-(5,2)]
         [LN: (0,3)-(6,3)]
      ]
      [AA:]
      [BR: (6, 3)]
   ]

   [GST: _dnArrow
      [EXPS:]
      [COMPS:
         [LN: (0,0)-(6,0)]
         [LN: (1,1)-(5,1)]
         [LN: (2,2)-(4,2)]
         [LN: (3,3)-(3,3)]
      ]
      [AA:]
      [BR: (6, 3)]
   ]

   [GST: _classgroup
      [EXPS:
         [+ 4 $txtHeight$(9)]                    # 0: lineAY
         [+ 2 $exp$(0)]                          # 1: tbBY
         [+ + 4 $exp$(0) $txtHeight$(12)] # 2: lineBY
         [+ 2 $exp$(2)]                          # 3: tbCY
         [+ + 4 $exp$(2) $txtHeight$(15)] # 4: height
         [+ 4 $max$($txtWidth$(9), $txtWidth$(12), $txtWidth$(15))]
                                                 # 5: width
         [- - $exp$(5) $grpWidth$(30) 1]  # 6: upArrowX
         [- $exp$(5) 4]                          # 7: aawidth
      ]

      [COMPS:
         [ST: 9, (2, 2)]
         [MT: 12, (2, $exp$(1))]
         [MT: 15, (2, $exp$(3))]
         [LN: (3, 0)-(- $exp$(5) 3, 0)]
         [LN: ($exp$(5), 3)-($exp$(5), - $exp$(4) 3)]
         [LN: (- $exp$(5) 3, $exp$(4))-(3, $exp$(4))]
         [LN: (0, - $exp$(4) 3)-(0, 3)]
         [ARC: (0,0)-(6, 6), 90, 90]
         [ARC: (- $exp$(5) 6, 0)-($exp$(5), 6), 0, 90]
         [ARC: (0, - $exp$(4) 6)-(6, $exp$(4)), 180, 90]
         [ARC: (- $exp$(5) 6, - $exp$(4) 6)-($exp$(5), $exp$(4)),
               270, 90]
         [LN:  (0, $exp$(0))-($exp$(5), $exp$(0))]
         [LN:  (0, $exp$(2))-($exp$(5), $exp$(2))]
         [GRP: 30 _upArrow AT ($exp$(6), 2)]
      ]
      [AA:
```

```
         [AAT: ($exp$(6), 2)-
               (+ $exp$(6) $grpWidth$(30), + 2 $grpHeight$(30))
               [TACT: _classnamegroup]]
         [AAT: (2, 2)-(+ 2 $exp$(7), + 2 $txtHeight$(9))
               [UACT: 9]]
         [AAT: (2, $exp$(1))-
               (+ 2 $exp$(7), + $exp$(1) $txtHeight$(12)) [UACT: 12]]
         [AAT: (2, $exp$(3))-
               (+ 2 $exp$(7), + $exp$(3) $txtHeight$(15)) [UACT: 15]]
      ]
      [BR: ($exp$(5), $exp$(4))]
   ]

   [GST: _classnamegroup
      [EXPS:
         [+ 4 $txtHeight$(9)]              # 0: height
         [+ 4 $txtWidth$(9)]               # 1: width
         [- - $exp$(1) $grpWidth$(31) 1] # 2: dnArrowX

      ]
      [COMPS:
         [ST: 9, (2, 2)]
         [LN: (3, 0)-(- $exp$(1) 3, 0)]
         [LN: ($exp$(1), 3)-($exp$(1), - $exp$(0) 3)]
         [LN: (- $exp$(1) 3, $exp$(0))-(3, $exp$(0))]
         [LN: (0, - $exp$(0) 3)-(0, 3)]
         [ARC: (0, 0)-(6, 6), 90, 90]
         [ARC: (- $exp$(1) 6, 0)-($exp$(1), 6), 0, 90]
         [ARC: (0, - $exp$(0) 6)-(6, $exp$(0)), 180, 90]
         [ARC: (- $exp$(1) 6, - $exp$(0) 6)-($exp$(1), $exp$(0)),
               270, 90]
         [GRP: 31 _dnArrow AT ($exp$(2), 2)]
      ]
      [AA:
         [AAT: ($exp$(2), 2)-
               (+ $exp$(2) $grpWidth$(31), + 2 $grpHeight$(31))
               [TACT: _classgroup]]
         [AAT: (2, 2)-(+ 2 $txtWidth$(9), + 2 $txtHeight$(9))
               [UACT: 9]]
      ]
      [BR: ($exp$(1), $exp$(0))]
   ]
]

[OBJTS:
   [OBJTMPL:
      [TMPL: "Abstract class"
         [EXPS:]
         [COMPS:
            [GRP: 21 _classnamegroup AT (0,0)]
         ]
         [BR: ($grpWidth$(21), $grpHeight$(21))]
      ]
      [AA:
      ]
      [DA:
         [LN: 0, (0, 3) - (0, - $grpHeight$(21) 3), u, 10]
         [LN: 0, (3, $grpHeight$(21))-
              (- $grpWidth$(21) 3, $grpHeight$(21)), u, 10]
         [LN: 0, ($grpWidth$(21), - $grpHeight$(21) 3)-
              ($grpWidth$(21), 3), u, 10]
```

```
            [LN: 0, (- $grpWidth$(21) 3, 0) - (3, 0), u, 10]
         ]
         [DEFTXT: (21,9)]
      ]

      [OBJTMPL:
         [TMPL: "Class"
            [EXPS:
               [+ 4 $grpHeight$(22)] # 0: classheight
               [+ 4 $grpWidth$(22)]  # 1: classwidth
               [- $exp$(0) 5]        # 2: dockingY
               [- $exp$(1) 5]        # 3: dockingX
            ]
            [COMPS:
               [GRP: 22 _classnamegroup AT (2, 2)]
               [LN: (5, 0)-($exp$(3), 0)]
               [LN: ($exp$(1), 5)-($exp$(1), $exp$(2))]
               [LN: ($exp$(3), $exp$(0))-(5, $exp$(0))]
               [LN: (0, $exp$(2))-(0, 5)]
               [ARC: (0, 0)-(10, 10), 90, 90]
               [ARC: (- $exp$(1) 10, 0)-($exp$(1), 10), 0, 90]
               [ARC: (0, - $exp$(0) 10)-(10, $exp$(0)), 180, 90]
               [ARC: (- $exp$(1) 10, - $exp$(0) 10)-($exp$(1), $exp$(0)),
                     270, 90]
            ]
            [BR: ($exp$(1), $exp$(0))]
         ]
         [AA:
         ]
         [DA:
            [LN: 0, (0, 5)-(0, $exp$(2)), u, 10
                 [SYMBOLS: ("$none$", "Whole-part")]]
            [LN: 0, (5, $exp$(0))-($exp$(3), $exp$(0)), u, 10
                 [SYMBOLS: ("$none$", "Whole-part")]]
            [LN: 0, ($exp$(1), $exp$(2))-($exp$(1), 5), u, 10
                 [SYMBOLS: ("$none$", "Whole-part")]]
            [LN: 0, ($exp$(3), 0)-(5, 0), u, 10
                 [SYMBOLS: ("$none$", "Whole-part")]]
            [LN: 0, (2, 5)-(2, $exp$(2)), u, 10
                 [SYMBOLS: ("Inheritance")]]
            [LN: 0, (5, - $exp$(0) 2)-($exp$(3), - $exp$(0) 2), u, 10
                 [SYMBOLS: ("Inheritance")]]
            [LN: 0, (- $exp$(1) 2, $exp$(2))-(- $exp$(1) 2, 5), u, 10
                 [SYMBOLS: ("Inheritance")]]
            [LN: 0, ($exp$(3), 2)-(5, 2), u, 10
                 [SYMBOLS: ("Inheritance")]]
         ]
         [DEFTXT: (22,9)]
      ]
]

[CONSYMS:
   [CONSYMTMPL:
      [CONTMPL:
         [TMPL: "Inheritance"
            [EXPS:]
            [COMPS:
               [LN: (0, 20)-(40, 20)]
               [ARC: (0, 0)-(40, 40), 0, 180]
            ]
            [BR: (40, 20)]
```

```
         ]
         1                # rotation constrainment
      ]
      (20, 0)             # connection point
      [DA:
         [LN: 1, (0, 20)-(40, 20), u, 10]
      ]
      ARITY: 2
   ]

   [CONSYMTMPL:
      [CONTMPL:
         [TMPL: "Whole-part"
            [EXPS:]
            [COMPS:
               [LN: (0, 20)-(20, 20)-(10, 0)-(0, 20)]
            ]
            [BR: (20, 20)]
         ]
         1
      ]
      (10, 0)
      [DA:
         [LN: 1, (0, 20)-(20, 20), u, 10]
      ]
   ]
]

[CONTERMS:
   [CONTERMTMPL:
      [CONTMPL:
         [TMPL: "default"
            [EXPS: ]
            [COMPS:
               [LN: (0, 3)-(12, 3)]
            ]
            [BR: (12, 6)]
         ]
         1    # rotation constrainment - 0=false, 1=true
      ]
      (0, 3)  # head
      (12, 3) # tail
   ]
]
]
```

# Appendix E

## SPECIFICATION OF PACKET CONTENTS FOR CLIENT-SERVER COMMUNICATION

This appendix defines the structure and content of the communication packets that were described in Chapter 7.

**Data Types**

The primitive data types used in the construction of request packets are bytes, integers, and strings. These are denoted by the types <BYTE>, <INT>, and <STRING> respectively. Bytes are 8-bit unsigned quantities. Integers are 32-bit signed quantities. Strings are a null-terminated sequence of printable characters.

**Packet Structure**

Request packets are described in the following text in the following way:

*Request:* <Item> | <Request> | <Parameter1> | <Parameter2> | ...

The <Item> field identifies the source or destination of the request (eg. model editor, graph object, etc). The <Request> field identifies the particular request (eg. create model, create concept, etc). Any parameters that the request requires are included subsequently in the packet, written as additional fields.

The <Item> and <Request> fields of the packet are both of the type <BYTE>. Five items have been identified for initiating or receiving requests: the client, projects, models, diagrams, and graph objects. Each of these have been defined a byte field value for representation in the <Item> field of a request as shown in Table E-1.

| Item | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Client | ITEMclient | $00 |
| Project | ITEMproj | $01 |
| Model | ITEMmdl | $02 |
| Diagram | ITEMdgm | $03 |
| Graph Object | ITEMgobj | $04 |

**Table E-1 - Notation and field values for request items**

The possible values for the <Request> field of a packet depends on the source or destination of the request. Each request also has a different set of parameters, and the types of these parameters may vary. All packets for transmission (including the response packets described below) are prefixed by a sequence number (an <INT>), and are terminated by a null character (a <BYTE>).

The communication protocol operates in a request-response fashion. That is, all requests generate a response. A response packet begins with the <Item> and <Request> field values of the request it is responding to, however, the high bit (bit 7) of the <Item> field is used to flag the success or failure of the request. If the request was successful, this bit is set to one, else it is set to zero.

A successful request may include some return values for the client. If so, this is denoted in the text by describing a response packet using the notation:

*Response:* <Parameter1> | < Parameter2> | ...

If a request from a client fails, the server provides an explanation (a <STRING>) as a return value. These explanations would typically be shown to the user via a dialog box or some other means. Requests from the server that fail in a client do not include explanations in the response packet.

In the following sections, response packets are only shown for requests that succeed and the return values to the originator of the request. Successful responses that return no extra information other than the success of the request are not shown for brevity. Packets for unsuccessful responses are not shown either, again for brevity, as they all have the same format and do not need further explanation.

## E.1. Client-Initiated Communication

### E.1.1. Client-Level Requests

Client-level requests are sourced from the client interface. Table E-2 documents the values for the <Request> field of the request packet for client-level requests.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Connect | CLIENTconnect | $00 |
| Disconnect | CLIENTdisconnect | $01 |
| Log-in | CLIENTlogin | $02 |
| Log-out | CLIENTlogout | $03 |
| Get Methodologies | CLIENTgetmeth | $04 |
| Get Projects | CLIENTgetproj | $05 |

**Table E-2 – Client-level requests from the client interface**

**Connect Request**

*Request:* ITEMclient | CLIENTconnect

**Disconnect Request**

*Request:* ITEMclient | CLIENTdisconnect

**Log-in Request**

*Request:* ITEMclient | CLIENTlogin | UserName | UserPassword

*Parameters:* UserName: <STRING>
UserPassword: <STRING>

**Log-out Request**

*Request:* ITEMclient | CLIENTlogout

**Get Methodologies**

*Request:* ITEMclient | CLIENTgetmeth

*Response:* MethCount | MethID | MethName | ...

*Parameters:* MethCount: <INT>
MethID: <INT>
MethName: <STRING>

This request returns an array of size MethCount. Each element of the array contains an ID and a name of a methodology.

**Get Projects**

*Request:* ITEMclient | CLIENTgetproj

*Response:* ProjectCount | ProjectID | ProjectName | ...

*Parameters:* ProjectCount: <INT>

ProjectID: <INT>

ProjectName: <STRING>

This request returns an array of size ProjectCount. Each element of the array contains an ID and a name of a project.

### E.1.2. Project-Level Requests

Project-level requests are sourced from the project editors in a client. Table E-3 documents the values for the <Request> field of the request packet of project-level requests.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Project | PROJcreate | $00 |
| Open Project | PROJopen | $01 |
| Rename Project | PROJrename | $02 |
| Close Project | PROJclose | $03 |
| Delete Project | PROJdelete | $04 |
| Save Project | PROJsave | $05 |
| Get All Models | PROJgetallmdls | $06 |
| Get Project Models | PROJgetprojmdls | $07 |

**Table E-3 – Project-level requests from the client**

**Create Project**

*Request:* ITEMproj | PROJcreate | MethID

*Parameters:* MethID: <INT>

*Response:* ProjectID | ProjectName

*Parameters:* ProjectID: <INT>

ProjectName: <STRING>

**Open Project**

*Request:* ITEMproj | PROJopen | ProjectID | AccessMode

*Parameters:* ProjectID: <INT>

AccessMode: <BYTE>

*Response:* AccessMode

*Parameters:* AccessMode: <BYTE>

The request requires a project ID and an access mode. The valid field values for the access mode are given in Table E-4.

| Access Mode | Field value (hexadecimal) |
|---|---|
| Read Only | $00 |
| Read/Write | $01 |

**Table E-4 – Modes for accessing existing data**

**Rename Project**

*Request:* ITEMproj | PROJrename | ProjectID | NewName

*Parameters:* ProjectID: <INT>

NewName: <STRING>

**Close Project**

*Request:* ITEMproj | PROJclose | ProjectID

*Parameters:* ProjectID: <INT>

**Delete Project**

*Request:* ITEMproj | PROJdelete | ProjectID

*Parameters:* ProjectID: <INT>

**Save Model**

*Request:* ITEMproj | PROJsave | ProjectID

*Parameters:* ProjectID: <INT>

**Get All Models**

*Request:* ITEMproj | PROJgetallmdls | ProjectID

*Parameters:* ProjectID: <INT>

*Response:* ModelCount | ModelID | ModelName | ...

*Parameters:* ModelCount: <INT>

ModelID: <INT>

ModelName: <STRING>

This request returns an array of size ModelCount. Each element of the array contains an ID and a name of a model.

**Get Project Models**

*Request:* ITEMproj | PROJgetmodels | ProjectID

*Parameters:* ProjectID: <INT>

*Response:* ModelCount | ModelID | ModelName | ...

*Parameters:* ModelCount: <INT>

ModelID: <INT>

ModelName: <STRING>

This request returns an array of size ModelCount. Each element of the array contains an ID and a name of a model that has been previously added to the project.

### E.1.3. Model-Level Requests

Model-level requests are sourced from the model editors in a client. Table E-5 documents the values for the <Request> field of the request packet of model-level requests

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Model | MDLcreate | $00 |
| Open Model | MDLopen | $01 |
| Rename Model | MDLrename | $02 |
| Close Model | MDLclose | $03 |
| Delete Model | MDLdelete | $04 |
| Save Model | MDLsave | $05 |

**Table E-5 – Model-level requests from the client**

**Create Model**

*Request:* ITEMmdl | MDLcreate | ProjectID | ModelID

*Parameters:* ProjectID: <INT>

ModelID: <INT>

*Response:* ModelID | ModelName | Notation

*Parameters:* ModelID: <INT>

ModelName: <STRING>

Notation: <STRING>

**Open Model**

*Request:* ITEMmdl | MDLopen | ProjectID | ModelID | AccessMode

*Parameters:* ProjectID: <INT>

ModelID: <INT>

AccessMode: <BYTE>

*Response:* ModelName | Notation | AccessMode | Diagrams

*Parameters:* ModelName: <STRING>

Notation: <STRING>

AccessMode: <BYTE>

Diagrams: <STRING>

**Rename Model**

*Request:* ITEMmdl | MDLrename | ProjectID | ModelID | NewName

*Parameters:* ProjectID: <INT>

ModelID: <INT>

NewName: <STRING>

**Close Model**

*Request:* ITEMmdl | MDLclose | ProjectID | ModelID

*Parameters:* ProjectID: <INT>

ModelID: <INT>

**Delete Model**

*Request:* ITEMmdl | MDLdelete | ProjectID | ModelID

*Parameters:* ProjectID: <INT>

ModelID: <INT>

**Save Model**

*Request:* ITEMmdl | MDLsave | ProjectID | ModelID | Diagrams

*Parameters:* ProjectID: <INT>

ModelID: <INT>

Diagrams: <STRING>

### E.1.4. Diagram-Level Requests

Diagram-level requests are also sourced from the model editors in a client. Table E-6 documents the values for the <Request> field of the request packet of diagram-level requests.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Diagram | DGMcreate | $00 |
| Rename Diagram | DGMrename | $01 |
| Delete Diagram | DGMdelete | $02 |

**Table E-6 – Diagram-level requests from the client**

**Create Diagram**

*Request:* ITEMdgm | DGMcreate | ProjectID | ModelID

*Parameters:* ProjectID: <INT>
ModelID: <INT>

*Response:* DiagramID | DiagramName

*Parameters:* DiagramID: <INT>
DiagramName: <STRING>

**Rename Diagram**

*Request:* ITEMdgm | DGMrename | ProjectID | ModelID | DiagramID | NewName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
NewName: <STRING>

**Delete Diagram**

*Request:* ITEMdgm | DGMdelete | ProjectID | ModelID | DiagramID

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>

### E.1.5. Graph-Object-Level Requests

Graph-object-level requests are sourced from user interactions with the individual graphic objects in a model diagram. Table E-7 documents the values for the <Request> field of the request packet of graph-object-level requests.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Concept | GOBJcreatecon | $00 |
| Create Association | GOBJcreateassn | $01 |
| Update Concept | GOBJupdatecon | $02 |
| Update Association | GOBJupdateassn | $03 |
| Delete Concept | GOBJdeletecon | $04 |
| Delete Association | GOBJdeleteassn | $05 |

**Table E-7 – Graph-object-level requests from the client**

### Create Concept

*Request:* ITEMgobj | GOBJcreatecon | ProjectID | ModelID | DiagramID | TemplateName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
TemplateName: <STRING>

*Response:* ConceptID

*Parameters:* ConceptID: <INT>

### Create Association

*Request:* ITEMgobj | GOBJcreateassn | ProjectID | ModelID | DiagramID | ConnectionSymbolName | ConceptID | ConnectionTerminatorName | ConceptID | ConnectionTerminatorName | ...

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConnectionSymbolName: <STRING>
ConceptID: <INT>
ConnectionTerminatorName: <STRING>

*Response:* AssociationID

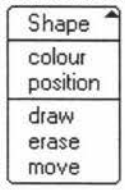
*Parameters:* AssociationID: <INT>

### Update Concept

*Request:* ITEMgobj | GOBJupdatecon | ProjectID | ModelID | DiagramID | ConceptID | ViewablePropertyID | NewValue

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConceptID: <INT>
ViewablePropertyID: <INT>
NewValue: <STRING>

The only viewable property update that is transmitted to the server in the present implementation of the client is the update of text areas. For single-line text areas, the NewValue parameter is equated simply to the new value of the text area. For multi-line text areas, each line is enclosed in double quotes and separated by commas for transmission. If a line contains a double quote character, it is prefixed by the escape character '\'. As an example, consider this Coad & Yourdon class symbol:




The class contains three operations: draw, erase, and move. If these were entered anew for the text area, the NewValue string parameter in an Update Concept request to the server would be as follows:

"draw", "erase", "move"

**Update Association**

*Request:* ITEMgobj | GOBJupdateassn | ProjectID | ModelID | DiagramID |
AssociationID | AssociationComponent | ViewablePropertyID | NewValue

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
AssociationID: <INT>
AssociationComponent: <BYTE>
ViewablePropertyID: <INT>
NewValue: <STRING>

This request is similar to the Update Concept request only it applies to associations. Associations have various components, including connection symbols, connection terminators, and connecting lines that all can have potential textual annotations such as labels or multiplicity. The particular component that is being updated is indicated by the AssociationComponent parameter, which can take the values specified in Table E-8.

| With connection symbol | | Without connection symbol | |
|---|---|---|---|
| Component | Field value (hexadecimal) | Component | Field value (hexadecimal) |
| Connection symbol | $00 | Connection line | $00 |
| Connection line | $01..$nn | Terminator | $01..$02 |
| Terminator | $nn+1..$nn+nn | | |

**Table E-8 – Field values for AssociationComponent parameter to Update Association Request: nn = total number of connecting lines (ie. GUIComposite-LineComponents) representing the association. This is equal to the number of connection terminators also.**

The request supplies the ID of the viewable property within the particular component that is being updated, and the string value that was entered by the user.

**Delete Concept**

*Request:* ITEMgobj | GOBJdeletecon | ProjectID | ModelID | DiagramID | ConceptID | DeleteLevel

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConceptID: <INT>
DeleteLevel: <BYTE>

The DeleteLevel parameter is used to denote the severity of the delete. The valid values for the DeleteLevel field are shown in Table E-9.

| Delete level | Field value (hexadecimal) |
|---|---|
| Diagram only | $00 |
| Model | $01 |
| Project | $02 |
| All projects | $03 |

**Table E-9 – Field values for specifying the severity of a Delete Concept request**

**Delete Association**

*Request:* ITEMgobj | GOBJdeleteassn | ProjectID | ModelID | DiagramID | AssociationID | DeleteLevel

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
AssociationID: <INT>
DeleteLevel: <BYTE>

## E.2. Server-Initiated Communication

### E.2.1. Client-Level Requests

The <Request> field values for the request packets of client-level requests are given in Table E-10.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Add Methodology | CLIENTaddmeth | $00 |
| Add Project | CLIENTaddproj | $01 |
| Delete Project | CLIENTdeleteproj | $02 |

**Table E-10 – Client-level requests from the server**

**Add Methodology**

*Request:* ITEMclient | CLIENTaddmeth | MethID | MethName
*Parameters:* MethID: <INT>
MethName: <STRING>

**Add Project**

*Request:* ITEMclient | CLIENTaddproj | ProjectID | ProjectName
*Parameters:* ProjectID: <INT>
ProjectName: <STRING>

**Delete Project**

*Request:* ITEMclient | CLIENTdeleteproj | ProjectID
*Parameters:* ProjectID: <INT>

### E.2.2. Project-Level Requests

The <Request> field values for the request packets for project-level requests are given in Table E-11.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Model | MDLcreate | $00 |
| Rename Model | MDLrename | $01 |
| Delete Model | MDLdelete | $02 |

**Table E-11 – Project-level requests from the server**

**Create Model**

*Request:* ITEMproj | MDLcreate | ProjectID | ModelID | ModelName | Notation

*Parameters:* ProjectID: <INT>
ModelID: <INT>
ModelName: <STRING>
Notation: <STRING>

**Rename Model**

*Request:* ITEMproj | MDLrename | ProjectID | ModelID | NewName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
ModelName: <STRING>

**Delete Model**

*Request:* ITEMproj | MDLrename | ProjectID | ModelID

*Parameters:* ProjectID: <INT>
ModelID: <INT>

### E.2.3. Model-Level Requests

The <Request> field values for the request packets for model-level requests are given in Table E-12.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Diagram | DGMcreate | $00 |
| Rename Diagram | DGMrename | $01 |
| Delete Diagram | DGMdelete | $02 |

**Table E-12 – Model-level requests from the server**

**Create Diagram**

*Request:* ITEMmdl | DGMcreate | ProjectID | ModelID | DiagramID | DiagramName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
DiagramName: <STRING>

**Rename Diagram**

*Request:* ITEMmdl | DGMrename | ProjectID | ModelID | DiagramID | NewName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ModelName: <STRING>

**Delete Diagram**

*Request:* ITEMmdl | DGMdelete | ProjectID | ModelID | DiagramID

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>

### E.2.4. Graph-Object-Level Requests

The <Request> field values for the request packets for model-level requests are given in Table E-13.

| Request | Denoted by | Field value (hexadecimal) |
|---|---|---|
| Create Concept | GOBJcreatecon | $00 |
| Create Association | GOBJcreateassn | $01 |
| Update Concept | GOBJupdatecon | $02 |
| Update Association | GOBJupdateassn | $03 |
| Delete Concept | GOBJdeletecon | $04 |
| Delete Association | GOBJdeleteassn | $05 |

**Table E-13 – Graph-object-level requests from the server**

### Create Concept

*Request:* ITEMgobj | GOBJcreatecon | ProjectID | ModelID | DiagramID | ConceptID | TemplateName

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConceptID: <INT>
TemplateName: <STRING>

### Create Association

*Request:* ITEMgobj | GOBJcreateassn | ProjectID | ModelID | DiagramID | AssociationID | ConnectionSymbolName | ConceptID | ConnectionTerminatorName | ConceptID | ConnectionTerminatorName | ...

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
AssociationID: <INT>
ConnectionSymbolName: <STRING>
ConceptID: <INT>
ConnectionTerminatorName: <STRING>

### Update Concept

*Request:* ITEMgobj | GOBJupdatecon | ProjectID | ModelID | DiagramID | ConceptID | ViewablePropertyID | NewValue

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConceptID: <INT>
ViewablePropertyID: <INT>
NewValue: <STRING>

### Update Association

*Request:* ITEMgobj | GOBJupdateassn | ProjectID | ModelID | DiagramID | AssociationID | AssociationComponent | ViewablePropertyID | NewValue

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
AssociationID: <INT>
AssociationComponent: <BYTE>
ViewablePropertyID: <INT>
NewValue: <STRING>

The values for the AssociationComponent parameter were specified in Table E-8.

**Delete Concept**

*Request:* ITEMgobj | GOBJdeletecon | ProjectID | ModelID | DiagramID | ConceptID

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
ConceptID: <INT>

**Delete Association**

*Request:* ITEMgobj | GOBJdeleteassn | ProjectID | ModelID | DiagramID | AssociationID

*Parameters:* ProjectID: <INT>
ModelID: <INT>
DiagramID: <INT>
AssociationID: <INT>