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.

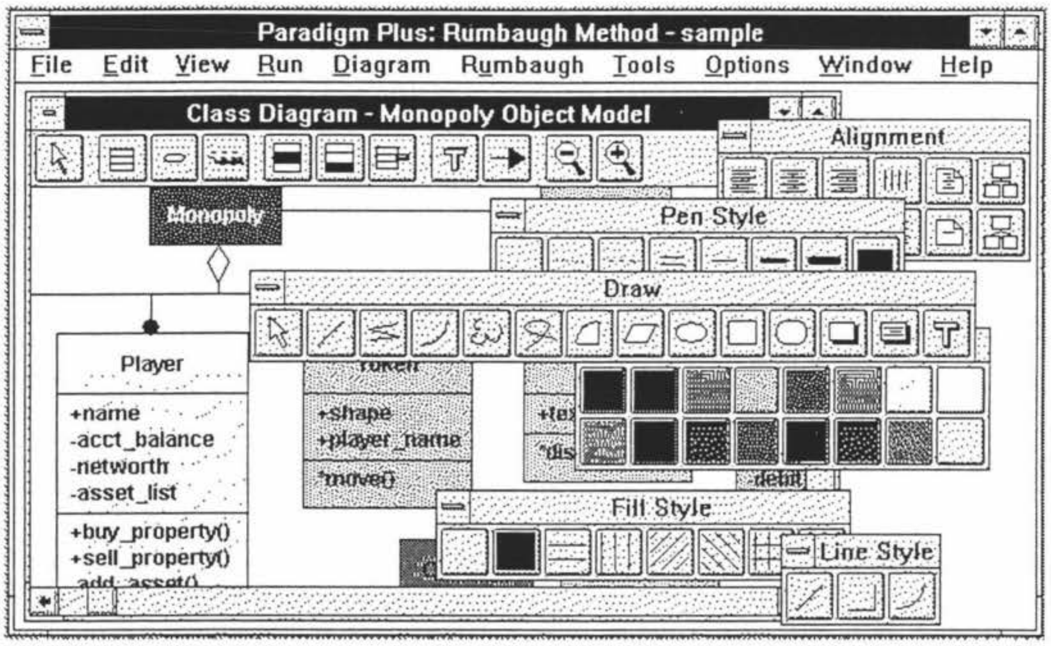

Figure 1-1 – User interface of the Paradigm Plus CASE Tool (Noble, 1996)

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.

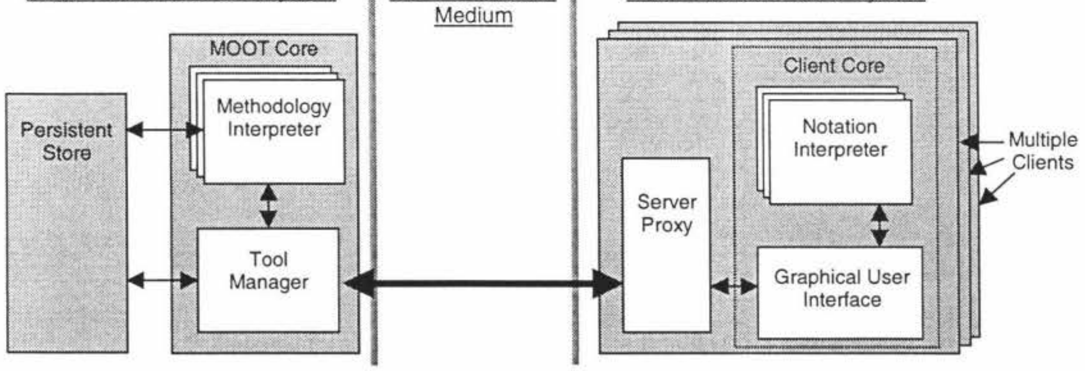

Figure 1-2 – Architecture of the CASE Tool Sub-system of MOOT

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.

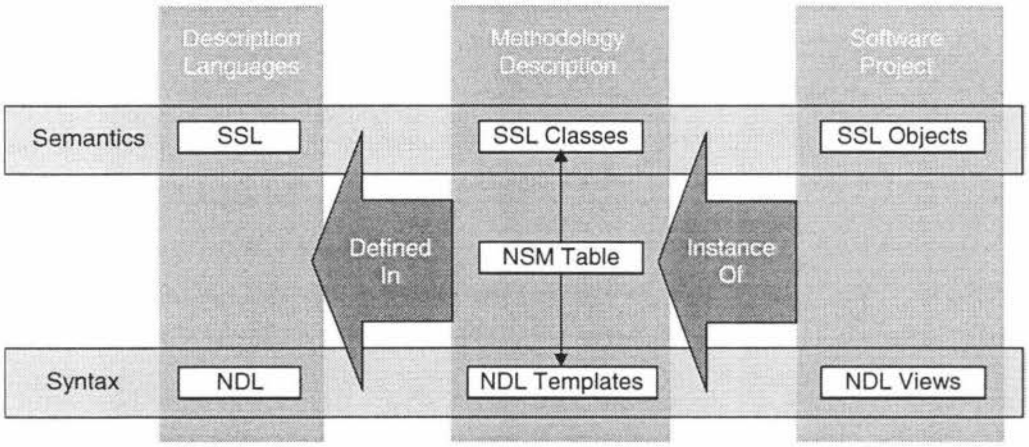

Figure 1-3 – Relationship between software projects, methodology descriptions, and the description languages

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).

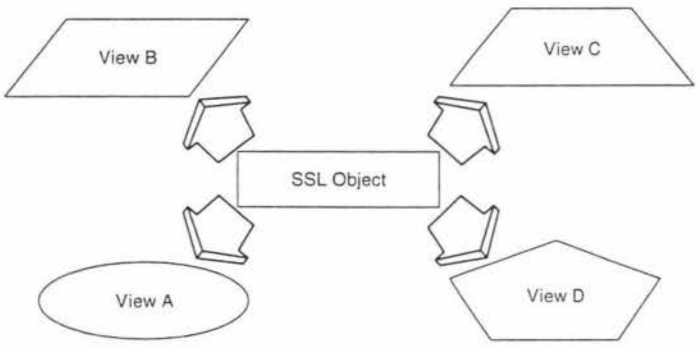

Figure 1-4 – Multiple views of an SSL object

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).

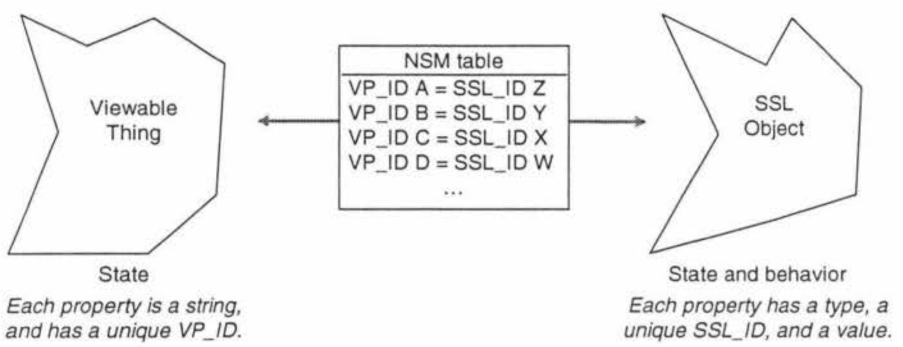

Figure 1-5 – Notation-Semantic Mapping

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.

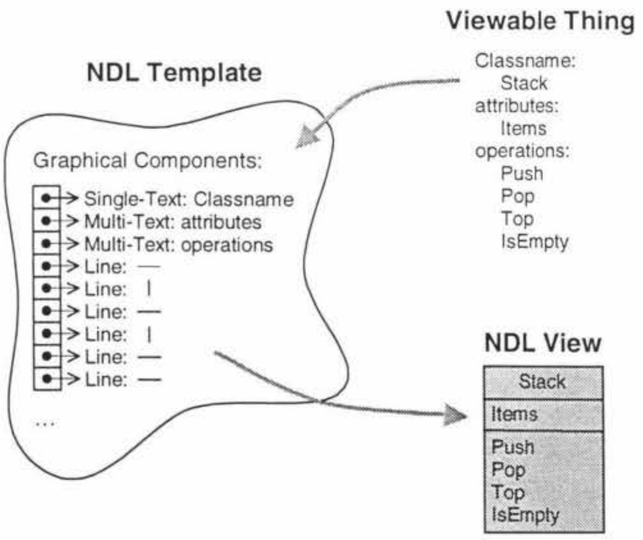

Figure 1-6 – The relationship between Viewable Thing, NDL Template, and NDL View

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.