On the Use of Object Oriented Techniques to Support the Construction of CASE Tools

R. Ian Ferguson  
University of Sunderland, ian.ferguson@sunderland.ac.uk

Andrew Hunter  
University of Sunderland, andrew.hunter@sunderland.ac.uk

Follow this and additional works at: http://aisel.aisnet.org/amcis2000
Abstract

This paper describes the MetaBuilder system: a means of constructing CASE tools by graphically assembling OO components. MetaBuilder belongs to a class of software systems known as metaCASE systems. Certain problems which can occur when attempting to build metaCASE tools are discussed and a solution based upon the use of an OO approach to designing such systems and of persistent object techniques for their implementation is proposed. Details are given of the implementation of the proposed solution as well as examples of its use.

Introduction

A class of software systems known as metaCASE systems are used to help build CASE tools. This paper describes certain problems (discussed in section ) which can occur when attempting to build such metaCASE tools and proposes a solution based upon the use of an OO approach to designing such systems and of persistent object techniques for their implementation.

An initial attempt at building an OO based solution (known as the MetaMOOSE system) allowed the rapid construction of CASE tools from a library of software components (classes). Further development of this approach lead to the production of a diagrammatic notation and associated metaCASE tool (known as MetaBuilder) which allows further CASE tools to be built even more rapidly by graphically composing them from the classes provided by MetaMOOSE. MetaBuilder has been used to implement a variety of CASE tools some of which are presented in section .

Problem

What is MetaCASE

The term MetaCASE (Alderson, 1991) refers to software systems used to build CASE tools. The goal of any MetaCASE system is the rapid, cheap development of CASE tools typically to support customised development methods. A key feature of such systems is that they simplify the task of constructing CASE tools: the CASE tool builder can work at a high level of abstraction using a high level language for describing the entities of the SE process and the operations thereon.

How does MetaCASE work?

Most MetaCASE systems work by providing a generic tool which can then be customised by providing details of the method, notation or process it is to support to produce specific tools. To describe a CASE tool, three models are necessary (summarised in Figure 1.)

i) a data model - which describes the entities and attributes that the CASE tool must be able to manipulate

ii) a notation model - which describes the appearance of those entities and attributes.

iii) function model - which describes the user interface (UI), the actions which can performed upon the entities and the links between the UI and the actions.

Problem

What is MetaCASE

The term MetaCASE (Alderson, 1991) refers to software systems used to build CASE tools. The goal of any MetaCASE system is the rapid, cheap development of CASE tools typically to support customised development methods. A key feature of such systems is that they simplify the task of constructing CASE tools: the CASE tool builder can work at a high level of abstraction using a high level language for describing the entities of the SE process and the operations thereon.

How does MetaCASE work?

Most MetaCASE systems work by providing a generic tool which can then be customised by providing details of the method, notation or process it is to support to produce specific tools. To describe a CASE tool, three models are necessary (summarised in Figure 1.)

i) a data model - which describes the entities and attributes that the CASE tool must be able to manipulate

ii) a notation model - which describes the appearance of those entities and attributes.

iii) function model - which describes the user interface (UI), the actions which can performed upon the entities and the links between the UI and the actions.

Problem

What is MetaCASE

The term MetaCASE (Alderson, 1991) refers to software systems used to build CASE tools. The goal of any MetaCASE system is the rapid, cheap development of CASE tools typically to support customised development methods. A key feature of such systems is that they simplify the task of constructing CASE tools: the CASE tool builder can work at a high level of abstraction using a high level language for describing the entities of the SE process and the operations thereon.

How does MetaCASE work?

Most MetaCASE systems work by providing a generic tool which can then be customised by providing details of the method, notation or process it is to support to produce specific tools. To describe a CASE tool, three models are necessary (summarised in Figure 1.)

i) a data model - which describes the entities and attributes that the CASE tool must be able to manipulate

ii) a notation model - which describes the appearance of those entities and attributes.

iii) function model - which describes the user interface (UI), the actions which can performed upon the entities and the links between the UI and the actions.

Figure 1 - Components of a MetaCASE built CASE tool

MetaCASE research has evolved down several complementary paths, all with the goal of reducing the cost, time and effort required to produce CASE tools. This has been motivated by the need to easily produce CASE tools particularly support for customised methods. Uptake of MetaCASE technology has not been widespread. It is difficult to find definitive reasons for this but the authors believe it is due to a “trade-off” that occurs in all current
approaches between two factors: ease of extensibility and level of abstraction (Isazadeh & Lamb, 1997)(Karrer & Scacchi, 1990). No one MetaCASE system has managed to simultaneously solve these two problems which are discussed in the following sections.

**Ease of extensibility**

The facilities provided by a CASE tool produced using a MetaCASE system depend greatly on what facilities the metatool designers thought the CASE tool users would require. All MetaCASE systems provide toolbuilders with a set of capabilities that they can build into their CASE tools. If some functionality is required by the CASE tool users that was not envisaged by the metatool designers, then the MetaCASE tool will need to be extended to allow it to produce the necessary features in the CASE tools.

One of the shortcomings of current MetaCASE technology is that building such extensions into the metatool is usually difficult - they have not been designed to be easily extensible. This problem manifests itself primarily in the need to use very low level programming techniques to accomplish even the simplest of extensions. Despite published interfaces to many of the OMS systems, interacting with the repository requires profound knowledge of the repository structure/ technique and the use of some very low-level database manipulation functions (Ferguson, 1993).

**Levels of Abstraction**

One of the main problems with MetaCASE systems is that they allow their users only two levels of abstraction when constructing a CASE tool system. At the 'coarse-grained' level, tools, databases, diagrams and notations are described by dedicated high-level languages. At the 'fine grained' level, new operations can be defined upon those databases, diagrams etc. by the use of a lower level language (typically ‘C’). This leads to the situation whereby it is a simple task to construct a simple CASE tool, but an overly complex task to construct a complex CASE tool, particularly one which requires extension in the manner described in the previous section.

What is needed is a means of gradually varying the level of abstraction available to the user to match the complexity of the problem. A system which gracefully reveals its inner workings (like peeling off the layers of an onion) is called for. Scacchi describes the ideal tool as one which provides...

> ...several levels of abstraction appropriate to various users of the metatool with different tool requirements. Simple to create a new tool, more complex for those wishing to do more complex things.(Scacchi & Mi, 1993)

**Solutions**

A MetaCASE system based on OO technology offers a solution to both the problems outlined above. Both the generic and specific parts of a tool can be provided as classes and certain properties of the OO paradigm lend themselves to this solution. Inheritance provides an elegant mechanism for extending the facilities provided by the metaCASE tools and together with encapsulation allows the complexity of the metaCASE system to be hidden from the toolbuilder.

This would allow novice MetaCASE users to easily build simple tools by using the services of the high level classes. As more complex tools are required, the layers of abstraction could be gradually revealed by ascending the inheritance hierarchy of the classes that make up the tool. Only the most complex of tool building tasks would require knowledge and understanding of the low level classes of the MetaCASE system.

The use of this technique means that engineers who wish to construct CASE tools would have a means of gradually learning about the complexities of MetaCASE with highly experienced MetaCASE engineers no longer required for everything but the most trivial tools.

An architecture for an OO MetaCASE tool is therefore proposed with the following features:-

i) The use of classes to describe deliverables from the SE process to enable items such as documents, diagrams, items on diagram, labels, icons etc. to be represented as objects.

ii) Facilities for linking objects via hyperCASE links to allow ease of navigation/integration of tools.

iii) Facilities for defining editors to edit the contents of the repository. These editors would themselves be class based. Facilities for following hyperCASE (Ferguson et al, 2000.) links should be included.

iv) A means of describing graphical representations for items in the repository.

v) A means of describing textual representations for items in the repository.

vi) One language to describe all data, operations and rules in a MetaCASE system. This should be a general purpose OO language augmented with a suitable library of classes to call on.

vii) The built tools should be based on a series of “high-level” classes that make up generic tools. The classes should be provided by the MetaCASE system. Inheritance should be used to specialise and extend the classes for particular tasks.

viii) Tools should be based on a repository implemented as a persistent object database. Access to the repository should be transparent to the tool-
builder in that the toolbuilder can manipulate objects without regard to whether they are held in main memory or in backing store. This would greatly simplify access to data.

Implementing the Solution

History

MetaBuilder has its origins in a series of diagram editors created to support a software engineering method called MOOSE (Method for Object Oriented Software Engineering) (Ferguson et al., 1994). These were engineered in an object-oriented (OO) fashion and as such it soon became obvious that the tools had many of their classes (or components) in common. This set of classes was gradually refined and became known as the MetaMOOSE OO component library. As MetaMOOSE was used in more projects, it became apparent that much of the code needed to combine these generic components into specific tools could be generated automatically from a suitable graphical notation. That notation and its associated tool became known as MetaBuilder.

As MetaMOOSE has been described elsewhere (Ferguson et al., 2000.) the remainder of this section describes the MetaBuilder tool and its associated notation.

MetaBuilder

This section provides a description of the MetaBuilder notation and a simple example of its use.

Whilst the rapid production of CASE tools is possible with MetaMOOSE, it still has several shortcomings. Although much of the code in the system is reused through inheritance, creation of new tools still proceeds in the traditional manner, by editing source code with a text editor. This means that the toolbuilder must have at least a basic knowledge of the target language of MetaMOOSE - Itcl.

The underlying complexity of the MetaCASE system is largely hidden from the toolbuilder (indeed this was one of the main reasons for using an OO approach - so that extraneous detail could be hidden using encapsulation) but some knowledge of the implementation details of MetaMOOSE is still required. Most of the code to create a tool is provided by inheritance (the primary component of MetaMOOSE is the library of classes that must be inherited by a built tool), but much of the code that is written is idiom: i.e. it is repeated with only minor variations in most tools.

The production of CASE tools could thus be faster as the present means is both time consuming in itself and error prone without automated support. The extensive use of idiom within the built CASE tools suggests that there exists considerable opportunity for reuse. One possible solution to the above problem (especially obvious to those who have just built a library of MetaCASE components) is to move from a position where CASE tool building is an exercise in code reuse to one where it is design reuse by:

i) Designing a graphical notation to describe tools assembled from the MetaMOOSE components
ii) Building a tool to edit that notation and generate code from it
iii) Using the MetaMOOSE classes to build that tool.

By embedding knowledge of how to use the MetaMOOSE classes in the tool, the need for the tool builder to know how to use MetaMOOSE or Itcl is removed. All the toolbuilder needs to know is how to build metamodels.

The graphical notation must allow the specification of the three aspects of the model (data, notation and function) and produce the model, view and controller classes to implement the built CASE tools. Since MetaMOOSE is object oriented, the design of a tool based upon it can be expressed in a notation based upon an OO class diagram rather than the ER models used by Toolbuilder/MetaEdit. The diagrammatic notation used could be from any of a number of methods (Booch, Open, Objectory etc.) but since an OO class diagrammer already existed as part of the MOOSE method, it was convenient simply to customise this. The customisation involved was:

i) the addition of a graphical symbol editor to allow specification of a symbol for the notation of each class.
ii) default inheritance of the MetaMOOSE classes.
iii) the addition of a dedicated code generator that is aware of and can generate the idiom used when using MetaMOOSE classes. This is where MetaBuilder embeds the knowledge of how to use all the MetaMOOSE classes.

Items in the target notation are represented by classes ("class" classes and "relationship" classes). Tool construction therefore proceeds by drawing a diagram including classes to represent items on the target notation, adding data members to hold information from the target notation, defining symbols using a built-in drawing package and adding functions to classes to customise event handling/processing/code generation.

The MetaBuilder Metamodelling Notation

The metamodelling technique is best introduced by example. Consider, the diagram shown in Figure 2. This represents a simple type of structured diagram (which we
name box-o-method) consisting solely of labelled nodes joined by edges. In this section, the metamodel representing this kind of diagram is developed, introducing the various modelling concepts in the process.

Before describing the MetaBuilder notation, it is important to distinguish between the metamodelling notation (that used to describe the specific component of a diagrammer) and the target notation (that which is being modelled in this case box-o-method).

The MetaBuilder metamodelling notation is based upon the concepts of object orientation (OO). Any OO modelling notation could have been used as the basis of MetaBuilder, but it was convenient to derive the MetaBuilder tool from an existing object diagrammer built as part of the MOOSE notation.

**Classes** The basic modelling concept in MetaBuilder is therefore the class. The concept is typically introduced (informally) to undergraduates by saying that any “blobs” in the target notation, must be represented by classes in the metamodel.

In the box-O-method example, there is clearly only one type of “blob” - the node - and would be modelled as shown in Figure 3

**Relationships.** The edges in box-o-method are modelled using relationships. Informally, any lines in the target notation become relationships in the metamodel. Stage 2 of the metamodel of box-o-method is shown in

![Figure 3 - Metamodel of box-o-method - Stage 1](image)

**Cardinality Rules.** Restrictions may be placed on the number of edges that may originate or terminate at a node. The restriction is modelled as an integer value (representing the maximum permissible number of edges) placed near the appropriate link. A value of 0 represents an unlimited number of edges. Generalising this, the number of any instances of a given relationship that a class may participate in, can be restricted by the cardinality of the class->relationship and relationship->class links.

Figure 6 shows that an unspecified number of edges may originate and terminate at any node. i.e there is no restriction

![Figure 5 - Metamodel of box-o-method - Stage 3](image)
**Data Members.** The existence of labels on nodes or edges can be modelled using data members. By adding a data member to a class or relationship, a data carrying facility is added to the underlying representation allowing the storage of textual information. Figure 7 shows the “name” data member added to the node class.

**Has_a relationships.** A second example box-o-method diagram (Figure 8) shows that nodes may have “spots” associated with them. This can be designated in the metamodel by the use of the has_a relationship. This implies that a instance of one class maybe directly associated with an instance of another without the mediation of a line or relationship. The metamodel shown in Figure 9 demonstrates this feature.

**Inheritance.** Variants of nodes can be created as shown in Figure 10 where two types of node are present: nodes and coded nodes which have an extra numerical label. Other than this label, coded nodes are identical to ordinary nodes, i.e. they participate in the same relationships and have identical data members.

**Symbols.** The appearance of each class and relationship must be specified to complete the metamodel. The MetaBuilder system includes a simple drawing editor for creating symbols. Figure 12 shows a screen shot of the symbol editor being used to define the box-o-method node.
symbol.

**Textual Annotations.** The symbol editor allows each data member to be associated with a user interface “widget” which allows the contents of the data member to be edited. Three types of widget are currently supported: *textitems* which allow single lines of text to be edited, *textwidgets* which support multiple lines of text and *popup_editors* which allow more sophisticated editing operations to be performed. In addition, simple non-editable labels maybe added to a symbol for non-varying textual annotations.

**Other notation items.** Various other items of notation can appear on a metamodel such as “tools” and “root objects”, but these are “housekeeping” items imposed by the method of implementation. They have no bearing on the metamodeling technique and as such are beyond the scope of this paper.

A screenshot of MetaBuilder being used to prepare the metamodel of box-o-method is shown in Figure 13

**The MetaBuilder Tool**

**Generating a tool.** Once the metamodel is complete, building the target tool is simply a matter of pressing a button. Figure 14 shows the completed box-o-method tool in use.

**Other Facilities of the MetaBuilder System**

**Widgets.** The textitems and text widgets are examples of a general class of items that can be placed on a diagram called widgets. These are items such a buttons, checkboxes and menus familiar from the graphical user interface.

**Functions.** Allow computational behaviour to be added to classes. Triggered by user interaction with widgets or some drawing action such as mouse clicking, symbol movement or dragging and dropping these functions can access the data stored in data members. The functions are expressed in the same computer language used to implement the MetaBuilder system - Itcl(Ousterhout, 1994) (Harrison, 1997).

**HyperCASE Navigation.** The term hyperCASE refers to the ability to easily access related diagrams (or different views of the same diagram) by simple mouse operations on a diagram. MetaBuilder facilitates such navigation by the use of functions.

**Multimedia.** A set of special widgets can be placed on a diagram which allow access to the multimedia facilities of a computer. Video widgets allow short clips of digitised video to be embedded in a diagram. Audio widgets perform the same task for digital sounds.

**Report generation - code generation.** A common requirement having drawn a diagram (at least in the field of computing) is to have a textual report generated automatically from it. This kind of facility is the basis for code generation systems in CASE tools. A default code
generation function is present in all classes in the metamodel. This default function prints out the values of each of the class’s data members and then calls the code generation function of any related classes. This default function can easily be customised to provide the desired format of generated report.

The Advantages of the MetaBuilder Approach

The MetaBuilder approach is characterised by its use of a graphical notation based on OO techniques for metamodelling mapped directly onto an OO implementation of the system. Several practical advantages accrue from using this approach to diagrammer construction that are not all present in other approaches:

Rapid Development. - Complex systems of hyperlinked diagrammers can be built in minutes rather than days. The OO metamodelling approach means that notation specifications seamlessly translate into the implementations. This allows a rapid prototyping approach to be taken to tool construction. If an end user requires changes to a tool or notation, they can be effected rapidly.

Ease of Tool Construction. Experience in teaching the MetaBuilder system suggests that the task of implementing such diagramming tools can be brought within reach of undergraduate computer scientists in a matter of hours. This is due to the fact that MetaBuilder abstracts all of the underlying implementational complexity allowing the toolbuilder to concentrate on the target notation.

Ease of Enhancement. Some advantages of the approach described above pertain to the construction of MetaBuilder itself rather than to the building of diagrammers or to end users. The OO approach to the system allows MetaBuilder to be extended and improved more easily than would otherwise be the case. If some extra facility is required in a tool (e.g. a new type of “widget”) it can be rapidly added by using MetaBuilder to add the new facility to itself, thus making it instantly available for use in all tools built with the MetaBuilder system.

HyperCASE and Multimedia. The HyperCASE facility of MetaBuilder is “Internet aware” in that it allows hyperlinks to diagrams stored on another computer entirely. When combined with the multimedia capabilities, this creates a diagram publishing facility, the possibilities of which have only begun to be explored.

MetaBuilder in use

MetaBuilder has been used extensively within its original intended application domain of CASE tools and to a lesser extent to build diagramming tools in other areas.

This section describes some of the systems built with MetaBuilder.

MOOSE toolset

Although MetaBuilder was originally derived from the MOOSE toolset (see section ) MOOSE has subsequently been rebuilt using MetaBuilder. MOOSE consists of a variety of diagramming and text editing tools which make use of the full range of MetaBuilder’s capabilities. Screenshots of and output from some of the MOOSE tools show the variety of supported notations.

MOOSE class diagrammer. The MOOSE class diagrammer shown in Figure 15 is a typical “node and edge” diagram, supporting a variety of nodes and relationships between them.

MOOSE object diagram. The MOOSE object diagram notation (Figure 16) is composed entirely of classes with has_a relationships between them - no link relationships are present. The diagram is built up by simply dragging one node on to another. This action is sufficient to link the
nodes via the appropriate has_a relationship.

**figure 16 - Object diagrammer**

The VisualMOOSE tool is a user interface design tool. It allows the various screens, forms and dialog boxes of a program’s user interface to be created by dragging and dropping the components from a palette of available controls. The controls that appear on the diagram are not simply icons but are actual working controls. This means that as soon as the control is placed onto the design diagram it is fully functional and will behave as if it were part of a completed program.

**figure 17 - VisualMOOSE**

**Various other CASE tools**

A wide range of diagramming tools has been developed for teaching purposes at the University of Sunderland, these include support for a variety of software engineering methods including Universal Modelling Language (UML), Structured Systems Analysis and Design Methodology (SSADM), Structured Systems Design (SSD) and Soft Systems Methodology (SSM). Of particular utility has been a tool to support Entity-Relationship-Attribute notation which allows database designs to be expressed diagrammatically and subsequently generates code to both create and query the database (Structured Query Language - SQL) and interface it to the World Wide Web (PHP).

**MetaBuilder is Self-defining**

Possibly the most interesting use to which MetaBuilder has been put is re-implementing itself. The MetaBuilder notation is “complete” in the sense that it can describe itself. The current version of MetaBuilder is capable of making amendments and improvements to itself. This greatly speeds up development work!

**Conclusion**

MetaBuilder is an extremely quick means of building diagram editor based CASE tools which can simply be integrated by the use of hyperCASE. Although the creation of advanced tools still requires the toolbuilder to write code, this can be done in a single comfortable environment.

Work on MetaBuilder will continue. It is hoped that “real world” trials of the system can be arranged. The authors wish MetaBuilder to become a vehicle for experimenting with visual languages. To achieve this, the range of tools that can be built must be expanded, perhaps by taking inspiration from current CAD packages for electronic design and adding more complex and powerful graphical widgets.

Development of MetaBuilder should proceed more rapidly now that it is “self-aware” in that it can edit its own definition.

**References**

Alderson, A., “Lecture Notes”. In: European Symposium on Software Development Environments and CASE technology, Königswinter, Germany, June 1991 Lecture Notes in Computer Science 509. Springer-Verlag


Ferguson, R.I., “The Beginners Guide to TBK,” University of Sunderland, School of Computing and Information Systems, Occasional Paper CIS-6-93


Ousterhout, J.K., “Tcl and the Tk Toolkit,” Addison-Wesley, Reading MA, 1994