CHAPTER 5: DEVELOPMENT OF A HYBRID ANALYSIS CASE TOOL

5.1 Introduction
The demand for new and increasingly sophisticated software systems has resulted in greater project development time and the need for more specialized developers. Unfortunately, the first priority of system resources is the maintenance and enhancement of existing systems rather than development of new systems. Coupled with the lack of skilled development staff, this has led to large new software backlogs. In some organizations, software backlogs are as long as five to fifteen years (Stobart, Thompson and Smith 1991). Information Systems managers are under ever-increasing pressure to search for new approaches to shorten the software backlogs. One of the recent technologies that promises to solve some of these problems is Computer-Aided Software Engineering (CASE). This chapter describes CASE, its environment, and its tools. The benefits and risks of using CASE are examined. The reasons for integrating CASE and Object-Oriented technologies are also presented.

5.2 Computer-Aided Software Engineering (CASE)
5.2.1 The Goals of CASE

According to Stobart, Thompson and Smith (1991), the main goals of CASE are to:

- Improve the productivity of the development staff.
- Improve the reliability of the developed software.
- Ensure software systems conforms to requirements specifications.
- Eliminate the wearisome aspects of system development.

5.2.2 CASE Environment

There are three basic layers in the CASE environment (Cheng and Ng 1990):

1. *Innermost layer*. This layer contains a database that stores information related to various projects. It includes technical as well as managerial data such as design specifications, program code, schedule requirements, budget, and personnel allocations.

2. *Middle layer*. This layer contains tools such as compilers, debuggers, operating system utilities, and automatic code generators.

3. *Outer layer*. This layer contains graphics editors and diagrammers.
Figure 5.1 shows this CASE environment.

![Diagram of CASE environment with layers: Graphics editors, Code generators, Compilers, Database, Debuggers, Operating System Utilities]

**Figure 5.1: A CASE environment**

### 5.3 CASE Tools

A CASE tool is a computer program designed for the automation of one software engineering activity. CASE tools can be divided into three main types (Banker and Kauffman 1991; Gillies 1992):

- **Upper/Front-End CASE** supports analysis and design stages of the software development life cycles.
- **Lower/Back-End CASE** supports implementation stages of the software development life cycles.
- **Integrated CASE (ICASE)** supports the whole software development life cycle.
5.4 Benefits of CASE tools

5.4.1 Reducing Development Time and Costs

5.4.1.1 Accelerate the drawing of diagrams
Drawing and redrawing diagrams for large system is labour-intensive using a pencil and paper approach. CASE tools provide automated support for creating and editing diagrams which speed up the drawing process, and therefore reducing the analysis and design time (Stobart, Thompson and Smith 1991; Burkhard and Jenster 1989).

5.4.1.2 Accelerate the generation of documentation
Documentation can be easily generated as information is stored in a data repository. Up-to-date analysis and design specifications can be generated while the development is in progress. This contrasts with the present situation in many information system departments where documentation is produced after the system has been completed. Often this documentation is found to be incomplete and inaccurate (Stobart, Thompson and Smith 1991).

5.4.1.3 Accelerate the creation of systems
The automatic code generator can generate programs instantly. The programming code can be generated and compiled to executable code that can be run immediately.

5.4.1.4 Accelerate the validation of data
Validation of data helps to remove errors at the early stage of development process. Performing validation manually is not only time consuming but also tedious. Automatic validation releases developers to concentrate on essential aspects of the design (Stobart, Thompson and Smith 1991).
5.4.2 Improving Software Quality

5.4.2.1 Consistent diagrams
With all diagram information kept in the data repository, consistency within and between diagrams can be checked automatically (Stobart, Thompson and Smith 1991).

5.4.2.2 Completeness of the development process
Validation can be performed to ensure that the whole development process is completed in accordance with the rules of the methodology being used (Stobart, Thompson and Smith 1991).

5.4.2.3 Standard Conformance
Validation can also be made to ensure that the software conforms to a standard (Stobart, Thompson and Smith 1991). The enforcement of standards and codes of practice helps to improve software quality.

5.4.2.4 Correctness of the programming code
Programming code written manually that is considered to be correct is likely to contain three to eight errors per thousand line (Stobart, Thompson and Smith 1991). The code generator reduces errors in programming code.
5.4.3 Reducing Maintenance Time and Costs
The need for maintenance is largely reduced using CASE tools. CASE tools detect errors and eliminate them during the development process, thus reducing the port-implementation maintenance efforts. If maintenance is still needed, it take less effort to perform with the aid of the accurate documentation generated by the CASE tools (Stobart, Thompson and Smith 1991).

5.5 Risks of CASE tools

5.5.1 Lack of Support for Full Development Life Cycle
Currently, there are no integrated tools available to support the whole software development life cycle (Stobart, Thompson and Smith 1991).

5.5.2 Lack of Integrated Project Support Environment
There is no integrated project support environment. CASE tools that support analysis and design normally contain a data repository or encyclopedia that stores information. When used with multi-user CASE tools, data repositories allow project information to be shared, and therefore assist communication within the organization. Data repositories need to have the capacity to store project information from all project development phases. Current CASE tools cannot provide this integrated project support environment (Stobart, Thompson and Smith 1991).

5.5.3 Lack of Support for Complete System Code Generation
There is no CASE tool to generate complete systems. Current code generators can only generate 'skeleton' code structures that contain branches and loops. Programmers must then add the required programming details to these structures (Stobart, Thompson and Smith 1991).
5.5.4 Lack of Short-term Benefits

CASE tools provide long-term rather than short-term benefits. The strength of CASE tools is to produce quality software that in the long run will reduce the maintenance costs. Many buyers will be disappointed to discover that there is a lack of short-term benefits.

5.6 Integration of CASE and Object-Oriented Technologies

It is generally accepted that for CASE tools to be successful, they need to support a disciplined software development methodology (Stobart, Thompson and Smith 1991; Rowe 1993; Humphrey 1992). The CASE tools that do not adhere to a recognised methodology can help produce systems faster than manual methodology. However, the systems that are produced are likely to be poor in quality.

According to Martin (1993), Taylor and Hecht (1990), and Ward (1990), Object-Oriented methodologies will only be effectively applied to the software development life cycle when CASE tools are available to support them.

5.6.1 Automation of Reusable Software Components

One of the strengths of Object-Oriented methodologies is the reuse of existing software components such as libraries of classes. Reusing existing software components allows programming code to be generated more quickly, less expensively and at a higher level of quality (Stobart, Thompson and Smith 1991). CASE tools can further accelerate the generation of programming code using the class libraries residing in the repository.
5.6.2 Automation of Transition of Phases

Object-Oriented methodologies use the same notations for analysis, design and implementation. This is different from structured methodologies where Data Flow Diagrams used during analysis are transformed into structured charts during design, and structured charts are then transformed into programming code during implementation. The consistent notation of Object-Oriented methodologies simplify the design of the CASE tools. The CASE tools in turn simplify the Object-Oriented development. The use of consistent notations also improves the communication between users, analysts, and programmers and thus reduces the chance of producing incorrect specifications (Martin 1993).

5.6.3 Showing and Hiding Details

CASE tools can easily display and hide details of classes. This facility eases the understanding of classes.

5.6.4 Generate Complete Software Code

CASE tools that support Object-Oriented methodologies can produce a hundred percent programming code because of the uniform notations used throughout the development life cycle (Martin 1993). This means that non-programmers can create programs without knowing programming languages such as C, C++.

5.6.5 Object-Oriented Repository

According to Martin (1993), most of the dominant CASE tools used for non-Object-Oriented development have Object-Oriented repositories. The Object-Oriented nature of the repositories means CASE tools can easily support Object-Oriented methodologies.
5.7 Lack of Object-Oriented CASE tools

Currently, there is a shortage of CASE tools to support Object-Oriented methodologies. This shortage has led several people to adapt structured CASE tools to automate Object-Oriented development. Two of the papers that describe the use of current CASE tools for Object-Oriented analysis and design are those written by Ward (1990) and Taylor and Hecht (1990). Structured CASE tools need to meet several requirements before they can be adapted for Object-Oriented development. A basic requirement is that they must support notations for process and data modeling. Extra modeling guidelines are needed for building Object-Oriented systems. Even though, structured CASE tools can be used for Object-Oriented development, they do not provide support for Object-Oriented notations. Object-Oriented notations provide better representations of Object-Oriented systems.

5.8 The HYCONAN CASE Tool

Coad and Yourdon (1991b) describe their OOA/OOD as a relatively young methodology that will continue to evolve. For this reason, Coad and Yourdon recommend tailoring this methodology to suit the needs of each project. This chapter describes the development of a Hybrid Convergence Analysis CASE tool, the Hybrid Convergence Analysis Editor (HYCONAN). The HYCONAN Graphics Editor is based on the OOPaint program written by Heiny (1992). Firstly, the location of the HYCONAN Graphics Editor in the HCA/HCD CASE system is described. Then, the implementation of each component of the HCA/HCD "Multi-Layer, Multi-Component", model is described separately. The "Concurrent development" principle, based on the "Baseball" model, is applied to the process of developing the HYCONAN Graphics Editor.

5.9 The "Baseball" model of system development

Coad and Nicola (1993) apply the "concurrent development" principle to the development process using the model so-called "Baseball" model, as shown in Figure 5.2, wherein Hybrid Convergence Analysis(HCA), Hybrid Convergence Design(HCD), and programming are concurrently applied during the development process.
5.10 A Hybrid Convergence Analysis/Hybrid Convergence Design CASE System.

The HYCONAN Graphics Editor, a hybrid CASE tool, is a component of a Hybrid Convergence Analysis/Hybrid Convergence Design CASE system to be completed in the future. Figure 5.3 shows this CASE system with the HYCONAN Graphics Editor shaded.
• **Hybrid Convergence Analysis (HCA) Graphics Editor.** This editor is used during the analysis stage to model the system to be developed. The Problem Domain Component is modeled at this stage. The HYCONAN documentation is produced. The information used for producing the HYCONAN documentation is stored in the database.

• **Hybrid Convergence Design (HCD) Graphics Editor.** This editor is used during the design stage. The information produced by the HYCONAN Graphics editor (HCA) is passed to this editor to be used for improving the Problem Domain Component. The Human Interaction, Task Management, and Data Management components are modeled at this stage. The HCD documentation is produced. The information used for producing the HCD documentation is stored in the database.

• **Code Generator.** This code generator is used after the design stage is completed. It is for generating programming codes such as C, C++ or JAVA code. The information can be passed from the HCD Graphics Editor or read from the database.

• **Database.** This database is used for storing information about various projects. It includes the HCA/HCD documentation as well as programming codes.
5.11 Model Development

We create a model to gain a better understanding of the actual entity to be built. If the entity to be built is a software, our model must take a different form. It must be capable of modeling the information that the software transforms, the functions (and subfunctions) that enable the transformation to occur, and the behavior of the system as the transformation is taking place.

During software requirements analysis, we create models of the system to be built. The models focus on what the system must do, not on how it does it. In many cases, the models that we create make use of a graphical notation that depicts information, processing, systems behavior, and other characteristics as distinct and recognizable symbols. Other parts of the model may be purely textual. Descriptive information can be provided using a natural language or a specialized language for describing requirements. The HYCONAN model is a model of both function and behavior.

Functional models – Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem-specific functions. The functional model begins with a single context level model (i.e., the name of the software to be built). Over a series of iterations, more and more functional details are provided, until a thorough delineation of all system functionalities are represented.

Behavioral models – Most software responds to events from the outside world. This stimulus-response characteristic forms the basis of the behavioral model. A computer program always exists in some state - an externally observable mode of behavior that is changed only when some events occurs. A behavioral model creates a representation of the states of the software and the events that cause software to change state.
The HYCONAN model serves a number of important roles:

- The model aids the analyst in understanding the information, function, and behavior of a system, thereby making the requirement analysis task easier and more systematic.
- The model becomes the focal point for review and therefore the key to a determination of completeness, consistency, and accuracy of the specification.
- The model becomes the foundation for design, providing the designer with an essential representation of software that can be translated into an implementation context.

![Diagram](image)

*Figure 5.4: Mapping of the model to a Meta-Case tool in the development of a Case tool.*

### 5.12 The requirements of the Hybrid Convergence Analysis Graphics Editor

The purpose of the HYCONAN Graphics Editor is to produce the HYCONAN documentation. The development of the CASE tool is a two stage process. First, the HYCONAN model is developed and secondly, the model is mapped to a metamodel tool to create the required CASE tool supporting the methodology *(see figure 5.4).* To gain the benefit of using CASE technology, it is crucial to adopt a design method, a tool that fully supports it, and to follow the method as closely as possible. This way the design documents
get a standard appearance and content and are easy for other developers, management, and end users to understand. Also, maintenance gets a lot easier.

Documentation contains the Nine-Concepts HYCONAN model, the Class-&-Object specifications, and other supporting documents. Several symbols are used within each of these components of the HYCONAN documentation. The symbols used within each of these components are described below.

- The Nine-Concepts HYCONAN model. There are nine symbols for the Nine-Concepts HYCONAN model. These symbols include, System, Processes, Class, Instance, Inheritance, Possession, Associations, Composition and State. The symbols are shown in Figure 5.5.

![Symbols used in the HYCONAN model](image)

*Figure 5.5: Symbols used in the HYCONAN model*
In the development of the HYCONAN Graphics editor, METAEDIT Personal metacase tool, a metamodel based information editor that can be extensively modified-customized-to fit with different development methods, their graphical representations, and desired reports was used. A metamodel is a model of your favorite (or any other) design method. By loading it into MetaEdit, you promptly get a CASE tool to support your own method. When creating new method specifications (metamodels) for MetaEdit, the first interest should be in the conceptual content of the method to be modeled-i.e., in identifying the objects and their relationships in the method. When those objects of interest have been identified and named, they are visually represented using the OPRR notation. After that, the representation of the method is specified, and the specifications are translated into a form understood by MetaEdit.

5.13 Method Modeling using MetaEdit

Four domains of Model Information

When specifying methods, two aspects need to be separated: the conceptual content of a method and its representational form. These dimensions are the type-instance dimension and the conceptual dimension.

In the type-instance dimension we distinguish between the types that are included in the metamodel and their instances. The type level determines what is allowed and legal in the instance level. The metamodel with its generic representation definitions belongs to type level. The model the user works with belongs to the instance level.
In the conceptual-representational dimension we observe the difference between concepts and their representations. In MetaEdit, the types and their target level instances belong to the conceptual domain i.e. they do not carry any information about the way they are presented graphically. The graphical representation of a conceptual “thing” is defined by a contextual mapping from conceptual type definitions to their representational definitions.

The representation specification of a methodology should be consistent with the conceptual specification. There should be a straightforward mapping from conceptual to representational specifications. For each object, property, relationship, and role type we should provide an unambiguous graphical definition.
5.14 The Structure of the Meta-Metamodel

The meta-metamodel of MetaEdit is based on the OPRR (Object-Property-Role-Relationship) datamodel. The following definitions define the four basic OPRR concepts:

- **Object** is a thing that exists on its own and is represented by its associated properties. This is always a symbol definition, and its instance is always represented by a symbol (graphical object) that has one or more visible or invisible shape features to which connectors (i.e. lines) can be connected.

- **Property** is describing or qualifying characteristic associated with an object, a relationship, or a role. A property value is presented in a data field. A data field is a slot into which users can enter data. A data field's form and its acceptable values are defined by a data type.

- **Role** is a link between an object and a relationship. It clarifies how an object participates in a relationship. A role is presented by a symbol called a terminal. A terminal is a symbol attached between the end of a connector line and another symbol representing an object instance so that the line ends at the terminal (e.g. an arrowhead).

- **Relationship** is an association between two or more objects, which would not exist on its own (without the associated objects). A relationship instance is always presented by a line that is called a connector. A connector is a straight or joined line between two objects.
An object type always has a symbol definition, and its instance is always represented by a symbol. A symbol is a graphical object that has one or more visible or invisible shape features to which connectors (i.e. lines) can be connected. A symbol may link to textual labels representing the properties of the object instance. Examples of symbols can be seen in figure 5.7. In MetaEdit, each target level object instance is represented by one or more symbols. The graphical behavior of the symbols is defined in the meta level representation definitions.

The property value is presented in a data field. A data field is a slot into which users can enter data. A data field’s form and its acceptable values are defined by a data type. A property may also be shown in the symbol’s label.

A relationship instance is always presented by a line that is called a connector. A connector is a straight or joined line between two objects. For each relationship type, an associated line type is defined. The attributes of a line type are e.g. thickness, color, and solidity.

A role is presented by a symbol called a terminal. A terminal is a symbol attached between the end of a connector line and another symbol representing an object instance so that the line ends at the terminal (e.g. an arrowhead). Terminals differ from object representations in that terminals do not have any “places” to which other connectors can be attached. A terminal is always placed at the connector’s end. A terminal may also contain labels that convey the values of the roles properties. Terminals can also be left undefined, in which case the connector lines attach directly to the object symbols. An example of the terminal used is shown below.
When creating new method specifications (metamodels), the first interest should be in the conceptual content of the method to be modeled, that is, in identifying the objects and their relationships in the method. When those objects of interest have been identified and named, they are visually represented using the OPRR notation shown in figure 5.7.

![OPRR Graphical Symbols](image)

*Figure 5.7: OPRR Graphical Symbols*
5.15 Metamodels of HYCONAN's Diagramming Techniques

The four diagramming techniques of HYCONAN (i.e., the SID, OID, the CRD and the SED) have been specified separately as independent method definitions using the OPRR model. Figures 5.8, 5.9, 5.10, 5.11 show respectively the metamodels of a CRD, SED, OID and SID drawn using the OPRR method of MetaEdit.

![Class Relationship Diagram]

Figure 5.8: An OPRR specification of the Class Relationship Diagram
Figure 5.9: An OPRR specification of the State Evolution Diagram
Figure 5.10: An OPRR specification of the Object Interaction Diagram
Figure 5.11: An OPRR specification of the System Interaction Diagram.
5.16 The Draw Window

All the drawing tasks possible in the selected method are done in MetaEdit’s Draw Window. The Draw Window is shown in figure 5.12 and an example of a window with a SED diagram is shown in figure 5.13. The title bar of the Draw Window contains the name of the picture and the name of the selected method. The drawing tasks supported by the Draw Window are:

- adding objects
- adding relationships
- updating objects and relationships
- deleting objects and relationships

Figure 5.12: The Draw Window in MetaEdit
Figure 5.13 Draw Window with a SED diagram
5.17 Conclusion

It can be easily seen in this chapter that Structured CASE tools can be adapted for Object-Oriented development. However, the lack of support for Object-Oriented notations in these CASE tools limits their usefulness in building Object-Oriented systems. It can also be seen that Object-Oriented methodologies help to build better Object-Oriented CASE tools which themselves support consistent notations of Object-Oriented methodologies. This chapter described the development of a hybrid Structured/Object-Oriented CASE tool called the HYCONAN CASE Tool which has been successfully developed using a MetaCASE tool, namely MetaEdit.

The development environment of MetaEdit is based on the Object-Property-Role-Relationship (OPRR) model, which is a powerful means for developing methods metamodels. It provides generic features such as method definition language, a report definition language and a method help file generator.

This chapter has emphasized on how MetaEdit’s features were used to develop the CASE tool, which encompasses the draw windows of HYCONAN’s diagramming techniques, their reports and their help files.