A Survey of Object-Oriented Methods

by Peter Biggs
University of Durham

1. Object Oriented Design Methods

1.1 Introduction

Object orientation is the latest craze to hit software engineering. There are numerous object oriented methods being advocated at the present time. As Rentsch [Rent82] correctly predicted in 1982, "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

Booch [Booc91] feels that: "Because object-oriented design is a relatively young practice, a discipline for effectively applying the elements of the object model has not yet emerged."

This report will take a look at some of the better established object oriented methods currently available, giving an overview of their operation and comparing the notation that they use. It will also mention some of the tools that are available to support the object oriented methods under consideration.

1.1.2 Comparing the methods

In his paper on the Requirements of an Object-Oriented Method, Walker [Walk92] considers the four main criteria that should be considered by an object-oriented method are:

1) **Abstraction**, including rules, guidelines, heuristics for the separation of the problem domain into different abstractions; methods for identifying state, services and interfaces for encapsulating objects; and strategy for abstracting commonality for abstract superclasses.

2) **Notation and representational verisimilitude**, including notation for representing aggregation, inheritance and functional hierarchies, message sending/calling/parameter passing between objects in a seniority hierarchy; representation of autonomous objects and asynchronous message passing; and support for constraint representation and enforcement.

3) **Validation**, including transformation rules between alternative representations of the design; validation between message sends and interface protocols; overcoming implementation language shortcomings; and support for test generation and metrics for test coverage.

4) **Reuse**, including heuristics for identifying pre-existing classes from which to develop new application classes; support for different categories of object class and their reusability characteristics; and language independence for logical or architectural design phase.

This report will consider six main object oriented methods, first giving a general overview of the method itself, considering such particulars as: what levels of abstraction the method covers; how the method is applied; what are the goals of each stage; what deliverables and diagrams will be produced by the method; what verification and validation techniques can be employed; and how the method attempts to support reuse. The next sub-section will look at the notation used in the method. This will be done by use of a simple design to describe how a company functions. An overview of the symbols used is given, followed by the design in the notation for the method being considered. Finally, each method will be analysed to attempt to show some of the good, and not so good, features of the method. A list of tools currently available to support each method will also be given. An appendix is provided which highlights the differences in the use of each method by showing a home heating system in the various notations associated with the methods.

1.2 Object-Oriented Development (OOD)/Booch [Booc91]

1.2.1 The Method

Booch's work on object-oriented design is well matured and respected in the field of object- oriented analysis and design. Object-Oriented design with applications is an ideal text for introducing the concepts of object-orientation. It is divided into three main sections, the first introducing key object-oriented concepts, the second setting out Booch's method for object- oriented design and the third giving five case studies.

In his earlier text Software Engineering with Ada [Booc87], Booch suggests the following steps for analysing a system in preparation for designing a solution in an object-oriented manner:

1. Define the problem.
2. Develop an informal strategy for the software realisation of the real world problem domain.
3. Formalise the strategy.

These steps are based on the earlier work of Abbott [Abbo83]. The problem is defined in a concise, informal textual description, then information on the objects and operations represented in the system can be obtained from this description. Objects are represented by nouns, and operations by verbs. Pressman [Pres87] notes that the first two steps are actually performed during the software requirements analysis stage of development, rather than the design. Booch [Booc86], however, had already noted that object- oriented development, as he described it, is not a complete lifecycle method, but rather, it concentrates on the design and implementation stages.

In his later work [Booc91], Booch suggests the following order of events for the formalisation of the strategy:

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects.

Booch builds upon these principles in expounding his recommended method. He stresses that this is not just a simple sequence of steps to be followed, but rather an iterative and incremental development through refinement of complementary (logical and physical) views of a system. He states that "the process of object-oriented design starts with the discovery of the classes and objects that form the vocabulary of our problem domain; it stops whenever we find that there are no new primitive abstractions and mechanisms or when the classes and objects we have already discovered may be implemented by composing them from existing reusable software components."

Identifying the classes and objects involves finding key abstractions in the problem space and important mechanisms that offer the dynamic behaviour over several such objects. These key abstractions are found by studying the terminology of the problem domain.

Identifying the semantics involves establishing the meanings of the classes and objects identified in the previous stage. The developer should view the objects from the outside, define the object protocol and investigate how each object may be used by other objects.

Identifying relationships extends the previous work to include the relationships between classes and objects and to identify how these interact with each other. Associations such as inheritance, instantiation and uses between the classes are defined, as are the static and dynamic semantics of the mechanisms between the objects. The visibility between classes and objects is also decided upon.

Implementing classes and objects involves delving into the classes and objects and determining how to implement them in the chosen programming language. This is also the step where components are used, and the classes and objects are structured into modules.

In the design method, Booch emphasises the distinction between a 'logical view' of a system, in terms of classes and objects, and a 'physical view' of a system in terms of modules and processes. He also makes a distinction between static and dynamic models of the system. The method he proposes is, however, geared more towards static system descriptions, with less support for dynamic descriptions.

One of the major strength's of Booch's method is the rich notation available. There are diagrammatic notations for producing:

- class diagrams (class structure - static view)
- object diagrams (object structure - static view)
- state transition diagrams (class structure - dynamic view)
- timing diagrams (object structure - dynamic view)
- module diagrams (module architecture)
- process diagrams (process architecture)

The notations for class and object modelling use annotations or variant symbols (e.g. different kinds of arrow) to convey detailed information. Booch suggests that a subset of the notations can be used in the earlier stages of design, with the detail being filled in later. There is also a textual form for each notation, consisting of templates for documenting each major construct. The notations are defined in terms of a large repertoire of symbols, but there seems to be no real definition of syntax or static semantics, only informally in the text accompanying the symbol overview.

1.2.2 Notation

The main diagram used for describing the structure of a system is the class diagram, which shows a static view of the class structure in the object oriented design. The symbols used are as follows:

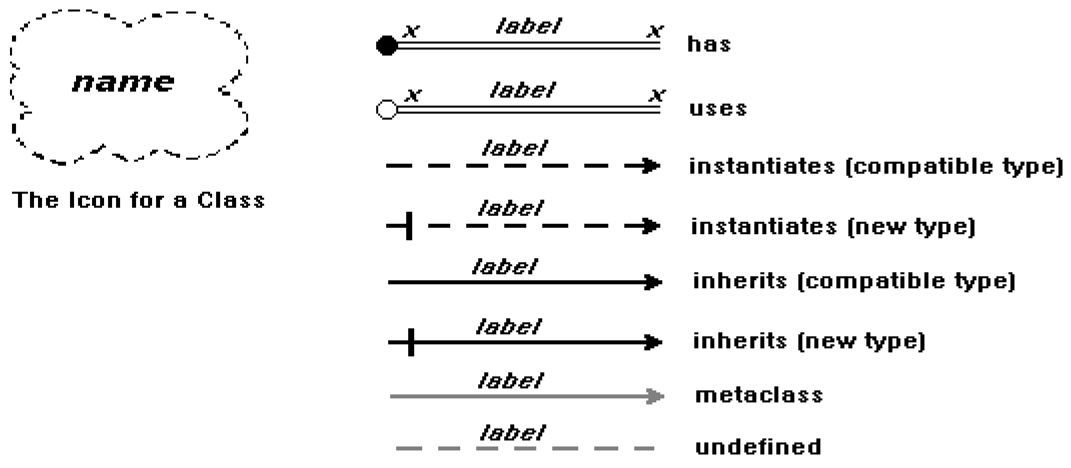
name
The Icon for a Class
x label x has
x label x uses
label instantiates (compatible type)
label instantiates (new type)
label inherits (compatible type)
label inherits (new type)
label metaclass
label undefined

Figure 1.2.1 - A Key to the Booch Class Diagram notation

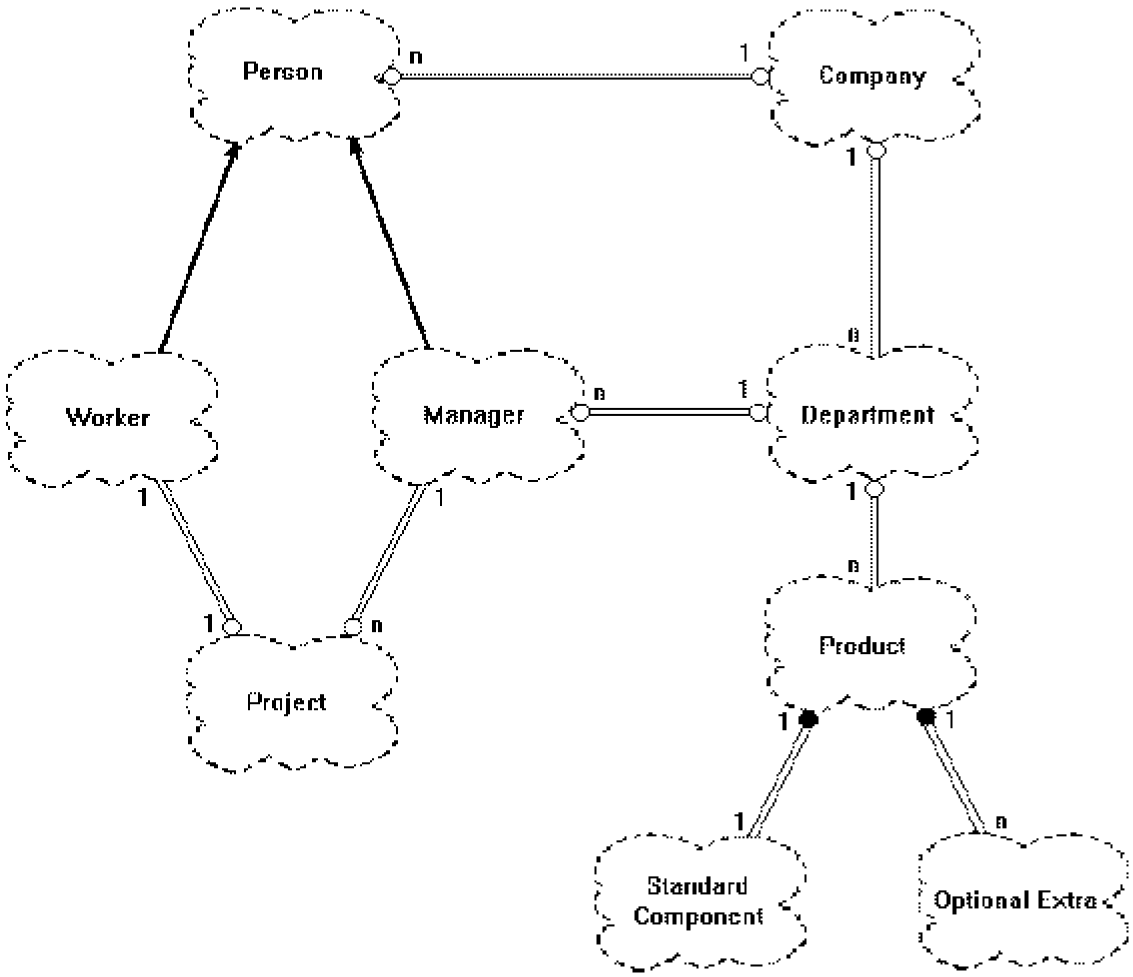

Figure 1.2.2 - A Booch Class Diagram for a Company

1.2.3 Analysis of Method

Booch's approach is essentially pragmatic. The method for object-oriented design is never really developed into a process, but is rather a collection of techniques, formalised ideas and heuristics that can be used when developing object-oriented systems. His book Object-Oriented Design with Applications should certainly be considered a core text in the field of object-oriented design. It contains much good advice in this area, but this is not generally in the form of explicit heuristics. This can be considered as either a strength or weakness of his approach. In considering this method, Ormsby [Orms91] feels that "what the method does not say is quite how the designer should decide whether further decomposition at any stage is possible or desirable. Nor are guidelines provided for judging the quality of a particular decomposition versus another one (though this is very difficult)."

Walker [Walk92] has this to say about Booch's object-oriented approach: "One possible problem is undoubtedly the lack of significant partitioning and layering of diagrams in the Booch method. Booch employs a variety of diagrams for different purposes (e.g. class diagrams, state transition diagrams, object relationship/visibility and message synchronisation diagrams etc.), but within some diagrams, he appears to attempt to cover too much of the system (at least, this appears to be the case in the examples and cases he uses)." He then goes on to say: "Perhaps the criticism of Booch's strategy can be encapsulated in the phrase breadth rather than depth. Booch's diagram notation has sacrificed detail or depth for breadth; this can be useful at the early stages of the design process, but needs to be capable of being reworked to increase the level of detail. In part, this is performed by the textual templates, but they are not structured so as to relate detail in a helpful way to the diagrams; indeed, in some respects, they also lack essential information for the resolution of fine structure within the application."

1.2.4 Support for reuse

Booch [Booc91] considers both design for reuse, and design with reuse. From a process perspective, Booch provides specific guidelines for the construction of reusable components. His method also discusses a project-independent role, that of the object-oriented domain analyst, whose organisation is responsible for the identification, development, marketing, and demonstration of reusable components in conjunction with the projects.

In terms of designing with reuse, Booch is a little more vague. He states that "Actively looking for reusable software components that are relevant to a new system is a very important activity in any development." He goes on to say that "If we can find a component that is relevant to our to our problem, that component becomes a primitive abstraction with which we can compose the system; thus we are left with a smaller problem to solve." However, the details are hidden in the examples given, and no overall recommendations are given. Booch also advocates the use of application generators where applicable.

Tools available

Name of Tool Company Price Platform Rational Rose Rational $749 UNIX MetaEdit MetaCase Consulting $499 Windows ObjectModeler Iconix Software Engineering $1,495 Macintosh, UNIX Paradigm Plus Protosoft $3995 $7770 Windows UNIX S-CASE MultiQuest $249-$995 Windows, OSF/Motif, Open Look, Macintosh System Architect Popkin Software $1395 $1795 Windows OS/2 ObjectMaker Mark V Software Unknown Windows, UNIX, Macintosh Object System/Designer Palladio Software, Inc. Unknown Windows ATRIOM Semaphore Unknown Windows MacAnalyst and MacDesigner Excel Software $995 - $1,595 Macintosh

1.3 HOOD [Hood93]

1.3.1 The Method

Hierarchical Object-Oriented Design (HOOD) has been developed for the European Space Agency as a design/notation method for Ada. Robinson [Robi92a] introduces it in this way: "The top-down hierarchical decomposition approach is not new. After all, this is the method used with data flow diagrams starting from a context diagram which shows all the external interfaces with one central process. This process is then decomposed into other processes with data flows and control flows interacting between them, with consistency checks between levels. In the same way, the purpose of HOOD is to develop the design as a set of objects which together provide the functionality of the program."

The main process in HOOD is called the Basic Design Step. Robinson [Robi92a] describes it admirably: "A Basic Design Step has as its goal the identification of child objects of a given parent object, and of their individual relationships to other existing objects, or the refinement of a terminal object to the level of the code. This process is based on the identification of objects by means of object-oriented design techniques."

In another text [Robi92b], he goes on to further define the Basic Design Step: "A basic design step process is further split into four phases, thus defining a micro life-cycle for a design step." The phases can be summarised as follows:

1. Problem definition.

The context of the object to be designed is stated, with the goal of organising and structuring the data from the requirement analysis phase. This is an opportunity to provide a completeness check on requirements and traceability to design.

1.1 Statement of the problem - the designer states the problem in correct sentences which provides:

- a clear and precise definition of the problem;
- the context of the system to design.

1.2 Analysis and structuring of requirement data - the designer gathers and analyses all the information relevant to the problem, including the environment of the system to be designed.

2. Development of solution strategy.

The outline solution of the problem stated above is described in terms of objects at a high level of abstraction.

3. Formalisation of the strategy.

The objects and their associated operations are defined. A HOOD diagram of the proposed design solution is produced, allowing easy visualisation of the concepts and further formalisation. There are five subphases in the formalisation of the strategy:

3.1 Object identification.
3.2 Operation identification.
3.3 Grouping objects and operations (object operation table).
3.4 Graphical description.
3.5 Justification of design decisions.

4. Formalisation of the solution.

The solution is formalised through:

- formal definition of provided object interfaces
- formal description of object and operation control structures.

1.3.2 Notation

The main diagram used for describing the structure of a system is the HOOD object diagram, which shows a static view of the structure in the hierarchical object oriented design. The symbols used are as follows:

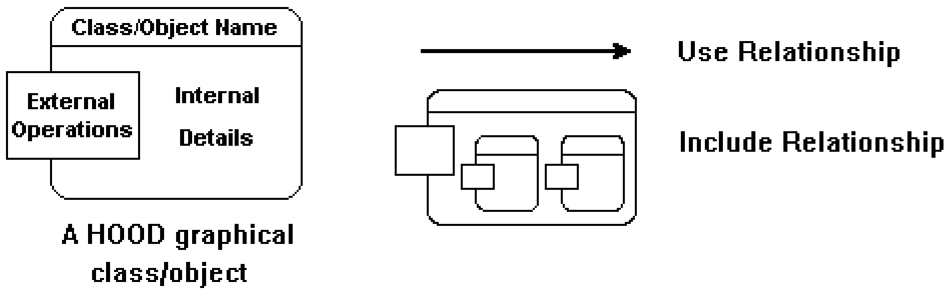

Figure 1.3.1 - A Key to the HOOD Diagram notation

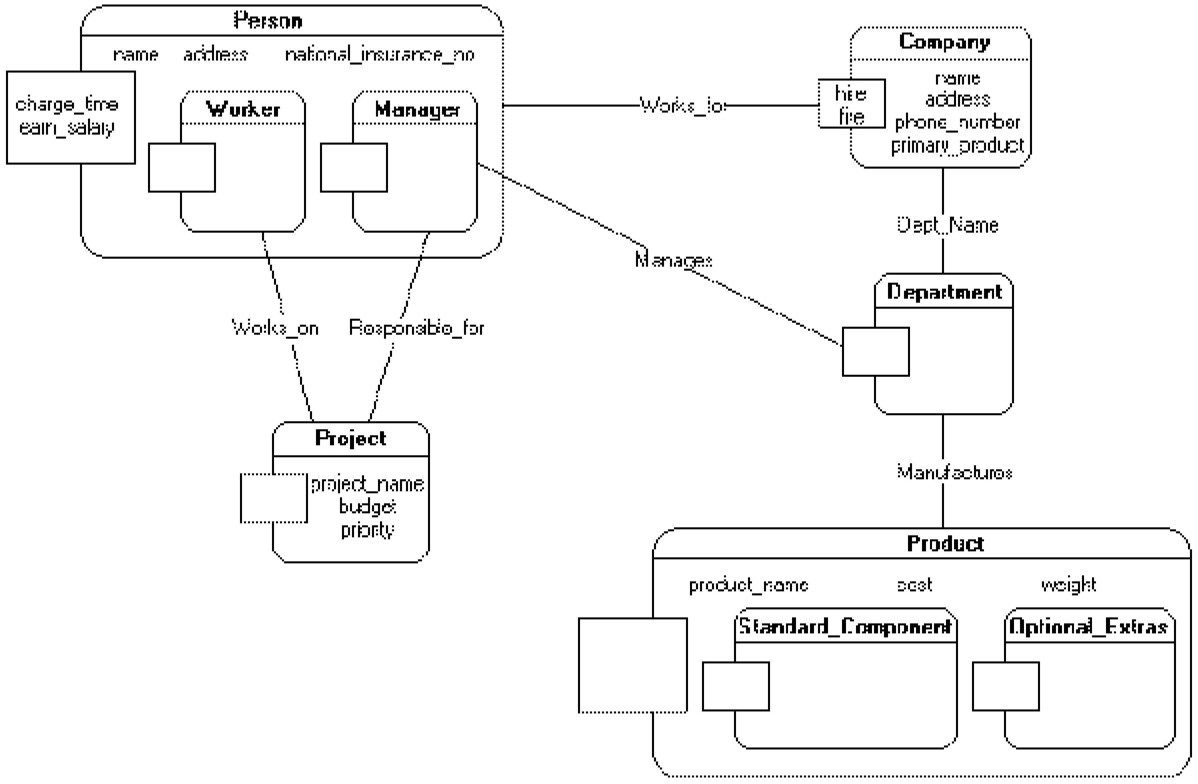

Figure 1.3.2 - A HOOD Diagram for a Company

1.3.3 Analysis of Method

HOOD is heavily geared towards an implementation in Ada. This is obviously ideal for Ada developers, but can be considered to limit its usefulness for any other programming language environment. Jacobson et al. [Jaco92] say: "The method gives the basic design step, but does not give any help in finding the appropriate object structure. Actually HOOD does give strong support for containment structures, but does not for other structures such as use or inheritance structures."

One of the major criticisms of HOOD is its lack of support for some of the more object-oriented techniques available. Support for inheritance is very poor, as evidenced by the lack of any notation to graphically represent it. The method is perhaps more object based than truly object- oriented.

1.3.4 Support for reuse

HOOD has no specific consideration of reuse in its process. Robinson [Robi92c] suggests that the HOOD principles of abstraction, information hiding, locality and modularity support reusability, because of

> " - applicability to the real world and to data entities,
> - an object may be general and therefore be reused in a new configuration".

The only suggestion Robinson gives of designing with reuse is a very vague reference when describing how to produce class objects. "A class object has to be designed separately from the main design as a root object, and either:

> - pre-exists from a previous project
> - from a library of classes
> - has been developed to represent an Ada generic package
> - it is developed specially for the project as it is seen as a general purpose object that can be usefully used in several parts of the design. This is a bottom up aspect of design."

Tools available

Name of Tool Company Price Platform HOOD-SF Virtual Software Factory Ltd. Unknown Unknown Select Select Software Tools Unknown Unknown GraphTalk Rank Xerox Unknown Unknown HOOD Toolset IPSYS Software Unknown UNIX, DOS STOOD Techniques Nouvells d'Informatique Unknown UNIX, RISC, X-windows

1.4 The Object Modelling Technique (OMT) [Rumb91]

1.4.1 The Method

The Object Modelling Technique provides three sets of concepts which provide three different views of the system. There is a method which leads to three models of the system corresponding to these views. The models are initially defined, then refined as the phases of the method progress. The three models are:

The object model, which describes the static structure of the objects in a system and their relationships. The main concepts are:

- class
- attribute
- operation
- inheritance
- association (i.e. relationship)
- aggregation

The dynamic model, which describes the aspects of the system that change over time. This model is used to specify and implement the control aspects of a system. The main concepts are:

- state
- sub/super state

- event
- action
- activity

The functional model, which describes the data value transformations within a system. The main concepts are:

- process
- data store
- data flow
- control flow
- actor (source/sink)

The method is divided into four phases, which are stages of the development process:

1) Analysis - the building of a model of the real world situation, based on a statement of the problem or user requirements. The deliverables of the analysis stage are:

- Problem Statement
- Object Model = Object Model Diagram + data dictionary
- Dynamic Model = State Diagrams + Global Event Flow Diagram
- Functional Model = Data Flow Diagram + constraints

2) System design - the partitioning of the target system into subsystems, based on a combination of knowledge of the problem domain and the proposed architecture of the target system (solution domain). The deliverables of the system design stage are:

- System Design Document: basic system architecture and high-level strategic decisions

3) Object design - construction of a design, based on the analysis model enriched with implementation detail, including the computer domain infrastructure classes. The deliverables of the object design stage are:

- Detailed Object Model
- Detailed Dynamic Model
- Detailed Functional Model

4) Implementation - translation of the design into a particular language or hardware instantiation, with particular emphasis on traceability and retaining flexibility and extensibility.

1.4.2 Notation

The main diagram used for describing the structure of a system is the object model, which gives a model of the class structure specified in the object oriented design. These are the symbols that are used:

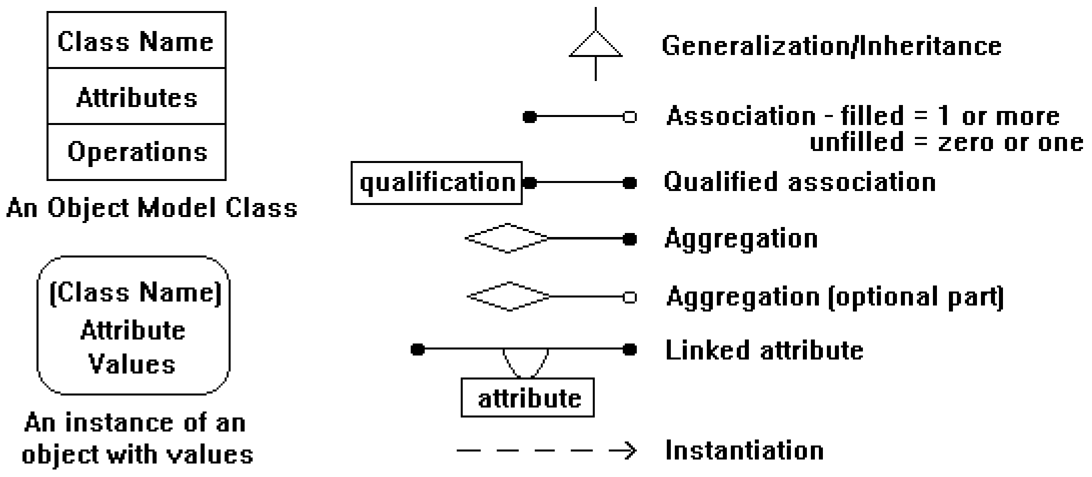

Figure 1.4.1 - A Key to the OMT notation

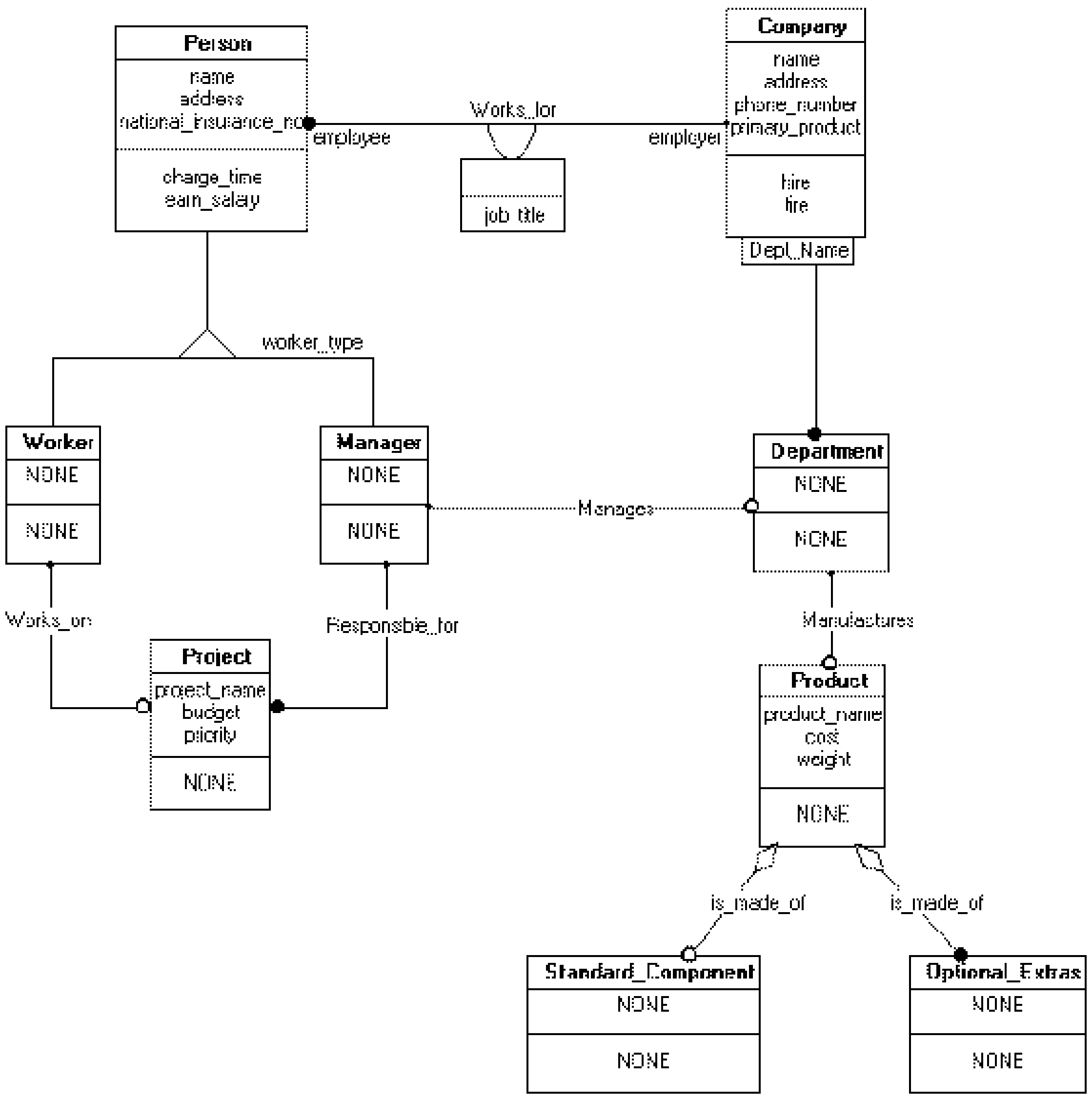

Figure 1.4.2 - An Object Model of a Company

1.4.3 Analysis of Method

The OMT is perhaps one of the most developed of the object-oriented design methods, both in terms of the notation that it uses, and the processes which are recommended for developing an object oriented system. In describing the technique, Rumbaugh et al. [Rumb91] claim that the method attempts to show how to use object-oriented concepts throughout the whole of the software development lifecycle. The inclusion of sections discussing analysis, design, and implementation in both object-oriented and non-object-oriented languages help to validate this claim.

The concept of using three views to represent a system presented in the OMT is potentially very powerful, but can also be considered to be very complex. It is unclear whether the views are to be developed independently of each other, or whether knowledge of one model should be used to influence construction of another. It would seem logical to ensure that the concepts in the different views are properly defined and interrelated, or much confusion could occur at the design stage of the development where the models are to be integrated.

Walker [Walk92] notes two main areas in which the OMT method is lacking. First, "there is no serious attempt to address the systematic reuse of software or design components. Secondly, the management of the software development process, including the establishment of suitable metrics for the measurement of progress and quality, is almost totally neglected." He feels, however, that, in terms of a technical approach to object-oriented software development, "Rumbaugh et al. have clearly established the current state of the art."

It is interesting to note that Rumbaugh is now advocating the inclusion of Jacobson's use cases (see section 1.8.1) as a 'front-end' to the OMT method [Rumb94]. He says, "We feel that use cases fit naturally on the front end of the published OMT process and supplement the existing user-centered features of the method." He goes on to qualify his recommendation of use cases, feeling that "a combination of direct domain analysis and use cases is an effective approach to starting analysis, rather than depending on use cases alone."

According to a recent survey conducted by the Object Management Group, OMT is currently the most popular of the standardised object oriented development methods (in house methods were the most used, with OMT a close second). One of the reasons for its success may be the variety of tool support available for the method, some of which are included in the table below.

1.4.4 Support for reuse

Rumbaugh et al. consider only the reuse of code, with very little practical advice. "Reuse a module from a previous design if possible, but avoid forcing a fit. Reuse is easiest when part of the problem domain matches a previous problem. If the new problem is similar to a previous problem but different, the original design may have to be extended to encompass both problems. Use your judgment about whether this is better than building a new design."

In addition, Rumbaugh et al. discount the utility of defining classes in isolation for general reuse. "Planning for future reuse takes more foresight and represents an investment. It is unlikely that a class in isolation will be used for multiple projects. Programmers are more likely to reuse carefully thought out subsystems, such as abstract data types, graphics packages, and numerical analysis libraries." Their suggestions on 'style rules for reusability' concentrate on methods, and, on the whole, reiterate the standard practices of structured programming, emphasising such styles as encapsulation with minimal coupling and high cohesion.

Tools available

Name of Tool Company Price Platform SelectOMT Select Software Tools 495 Windows OMTool Martin Marietta $995 Unknown StP/OMT Interactive Development Environments $12,000 AIX, DECstation, RS/6000, Sun, HP LOV/OMT Logiscope Unknown UNIX/Motif Excelerator II Intersolv $9,500 WindowsNT MetaEdit MetaCase Consulting $499 Windows ObjectModeler Iconix Software Engineering $1,495 Macintosh, UNIX Object Oriented Designer Taegyun Kim Free UNIX TeamWork/ Objectteam CADRE Technologies $4,000 $8,000 PC UNIX I-CASE OMT Westmount Technology Unknown Windows ObjectMaker Mark V Software Unknown Windows, UNIX, Macintosh ATRIOM Semaphore Unknown Windows MacAnalyst and MacDesigner Excel Software $995 - $1,595 Macintosh

1.5 Responsibility-Driven Design (RDD)/Class-Responsibility-Collaboration (CRC) [Wirf90]

1.5.1 The Method

In Responsibility-Driven Design, a model is developed from the requirements specification by the extraction of nouns and verbs from the specification. This provides a basis for the actual implementation. All of the basic concepts of object-orientation are covered.

In RDD, for each class, different responsibilities are defined which specify the roles of the objects, and their actions. In order to fulfil these responsibilities, classes need to collaborate with each other. Collaborations are defined to show how the objects will interact. The responsibilities are further grouped into contracts which define a set of requests that objects of the class can support. These contracts are further refined into protocols, which show the specific signature of each operation.

Subsystems are introduced to group a number of classes and subsystems of a lower level in order to abstract a certain functionality. Subsystems also have contracts which must be supported by some class in the subsystem. The RDD method consists of two main phases. An overview of these phases is given below:

1) The exploratory phase, which consists of:

Classes
- extract noun phrases from the specification and build a list
- identify candidate classes from the noun phrases
- identify candidates for abstract superclasses
- use categories to look for missing classes
- write a short statement for the purpose of each class

Responsibilities
- find responsibilities
- assign responsibilities to classes
- find additional responsibilities by looking at the relationships between classes

Collaborations
- find and list collaborations by examining responsibilities associated with classes
- identify additional collaborations by looking at relationships between classes
- discard and classes that take part in no collaboration (as client or server)

2) The analysis phase, which consists of:

Hierarchies
- draw inheritance hierarchy graphs
- identify which classes are abstract/concrete
- draw Venn diagrams showing how responsibilities are shared between classes

- refine class hierarchy, checking the allocation of responsibilities
- define how responsibilities are clustered into contracts, and which classes support which contracts

Subsystems
- draw a complete collaborations graph for the system
- identify possible subsystems
- simplify the collaborations between and within subsystems

Protocols
- define protocols for each class, by refining responsibilities into sets of method signatures
- write a specification for each class
- write a specification for each subsystem
- write a specification for each contract

1.5.2 Notation

The main diagram used for describing the structure of a system is the collaboration graph, which represents the subsystems, classes, contracts and collaborations in the system. The symbols used are as follows:

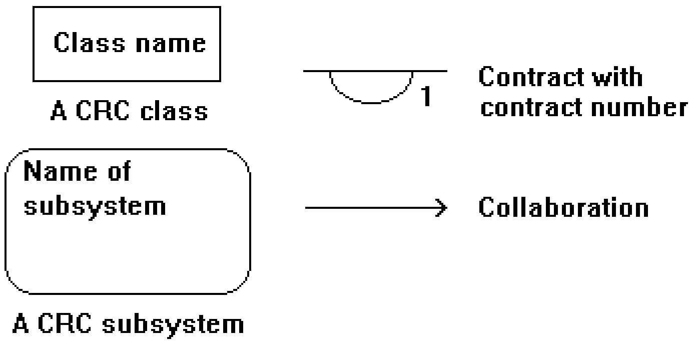

Figure 1.5.1 - A Key to the CRC notation

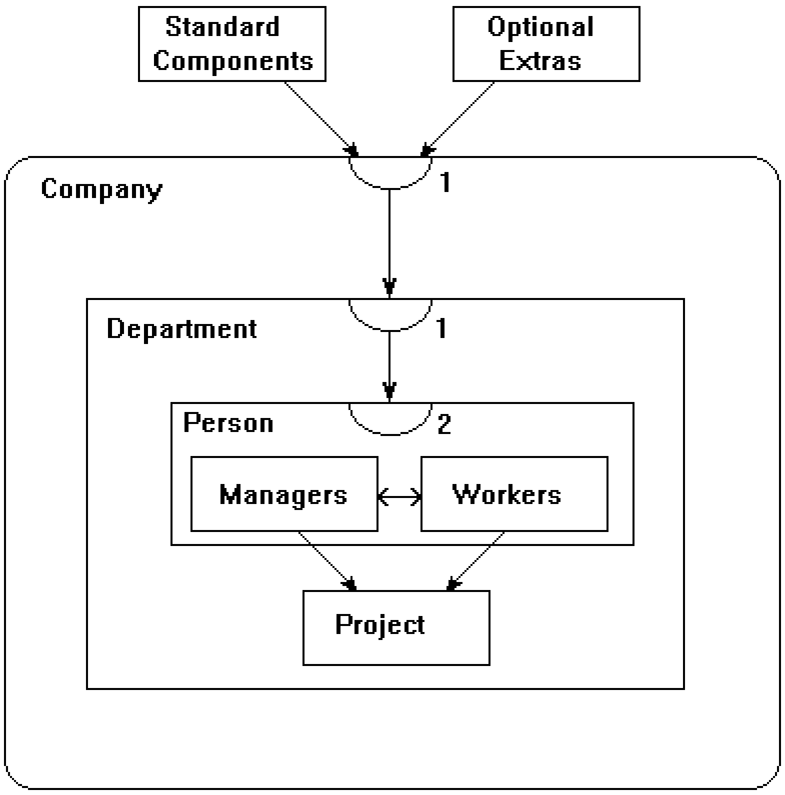

Figure 1.5.2 - A Collaboration Graph for a Company

1.5.3 Analysis of Method

The RDD method tends to use informal techniques and guidelines to develop on appropriate design. It relies heavily on the skill of the designer to find classes and their properties. An important part of the design strategy is the textual analysis of a requirements specification document to identify classes and responsibilities. CRC cards are used to capture class and subsystem specifications. CRC cards are index cards on which the details of a class or subsystem are written. According to Wirfs-Brock et al. [Wirf90], index cards are ideal for the task because they are compact, easy to manipulate and easy to modify or discard. They can also be easily arranged on a desktop, and rearranged to explore a fresh insight, or arranged in such a way to leave empty spaces where it is felt that classes are missing.

This type of simplicity can be considered both a strength and a weakness. It makes the design method much more intellectually and notationally manageable; however, there is a strong dependency on the requirements specification document (or other sources of information) being sufficiently clear, precise and understandable to express a design using the recommended concepts.

RDD only considers the design activity, there is no consideration of domain or problem analysis, or requirements capture. The class concept is central - all other concepts are there to describe classes and their interdependencies. The design activity produces what is really a single model in a single notation. There are no other models, such as models of dynamic behaviour. The notation is also not expressive enough to adequately show the design captured by the process.

The heuristics of this design method are explicit and generally well explained. Gossain [Goss91] feels that Wirfs-Brock et al. provide an extremely useful introduction to the core concepts of object-oriented design, but feels that they miss out on a vision of the 'big picture'. He also feels that their design approach would not scale up well to large systems.

1.5.4 Support for reuse

Wirfs-Brock et al. consider both design for reuse, and design with reuse. When identifying classes and subsystems, the method encourages the use of previously implemented components in the current design. If a previously implemented class or subsystem can fulfil the responsibilities required by the system, it should be reused.

Also, in the identification of candidate classes, Wirfs-Brock et al. direct the designer to "identify candidates for abstract superclasses" which can be made reusable by following their design guidelines.

Tools available

Name of Tool Company Price Platform Excelerator II Intersolv $9,500 WindowsNT HOMSuite Hatteras Software $595 Windows TurboCase StructSoft Unknown Macintosh

1.6 Object-Oriented Analysis (OOA) [Coad90], [Coad91]

1.6.1 The Method

OOA uses basic structuring principles, and joins them with an object-oriented point of view. The method consists of five steps:

1) **Finding Class & Object** - specifies how classes and objects should be found. The first approach is given by starting with the application domain and identifying the classes and objects forming the basis of the entire application and, in the light of this, the systems responsibilities in this domain are analysed.

2) **Identifying Structures** - this is done in two different ways. First, the generalisation-specialisation structure, which captures the hierarchy among the identified classes. Second, the Whole-Part structure, which is used to model how an object is part of another object, and how objects are composed into larger categories.

3) **Defining Subjects** - this is done by partitioning the Class & Object model into larger units. Subjects are groups of Class & Object. The structures identified earlier can be used.

4) **Defining Attributes** - this is done by identifying information and associations for every instance. This involves identifying the attributes needed to characterise each object. The identified attributes are placed at the correct level in the inheritance hierarchy.

5) **Defining Services** - defining the operations of the classes. This is done by identifying the object states and defining services for accessing and altering that state.

The final result of the analysis stage is a model of the problem domain in terms of five layers:

- Subject layer
- Class & Object layer
- Structure layer (i.e. inheritance, relationships)
- Attribute layer
- Service layer

Identifying the elements of each layer constitutes an activity within the method. The order of abstraction is not important, although it is generally easier to move from high levels of abstraction to lower levels.

An object-oriented design model consists of the following components:

- **Problem Domain Component** (PDC) - the results of the object-oriented analysis are put directly into this layer.
- **Human Interaction Component** (HIC) - this involves such activities as: classifying human users, describing task scenarios, designing the command hierarchy, designing the detailed interaction, prototyping the Human Computer Interface, defining HIC classes.
- **Task Management Component** (TMC) - this component consists of identifying tasks (processes), the services they provide, the task priority, whether the process is event or clock driven, and how it communicates (with other processes and the outside world).
- **Data Management Component** (DMC) - this component depends greatly on the storage technology available, and the persistence of data required.

1.6.2 Notation

The main diagram used for describing the structure of a system is the OOA model, which gives a static view of the class structure in the object oriented design. The symbols used are as follows:

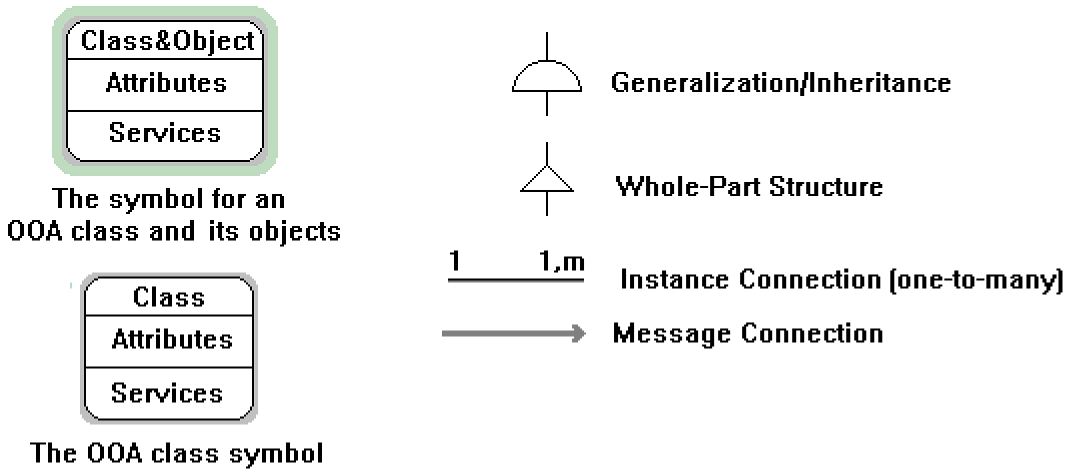

Figure 1.6.1 - A Key to the Coad/Yourdon OOA Model notation

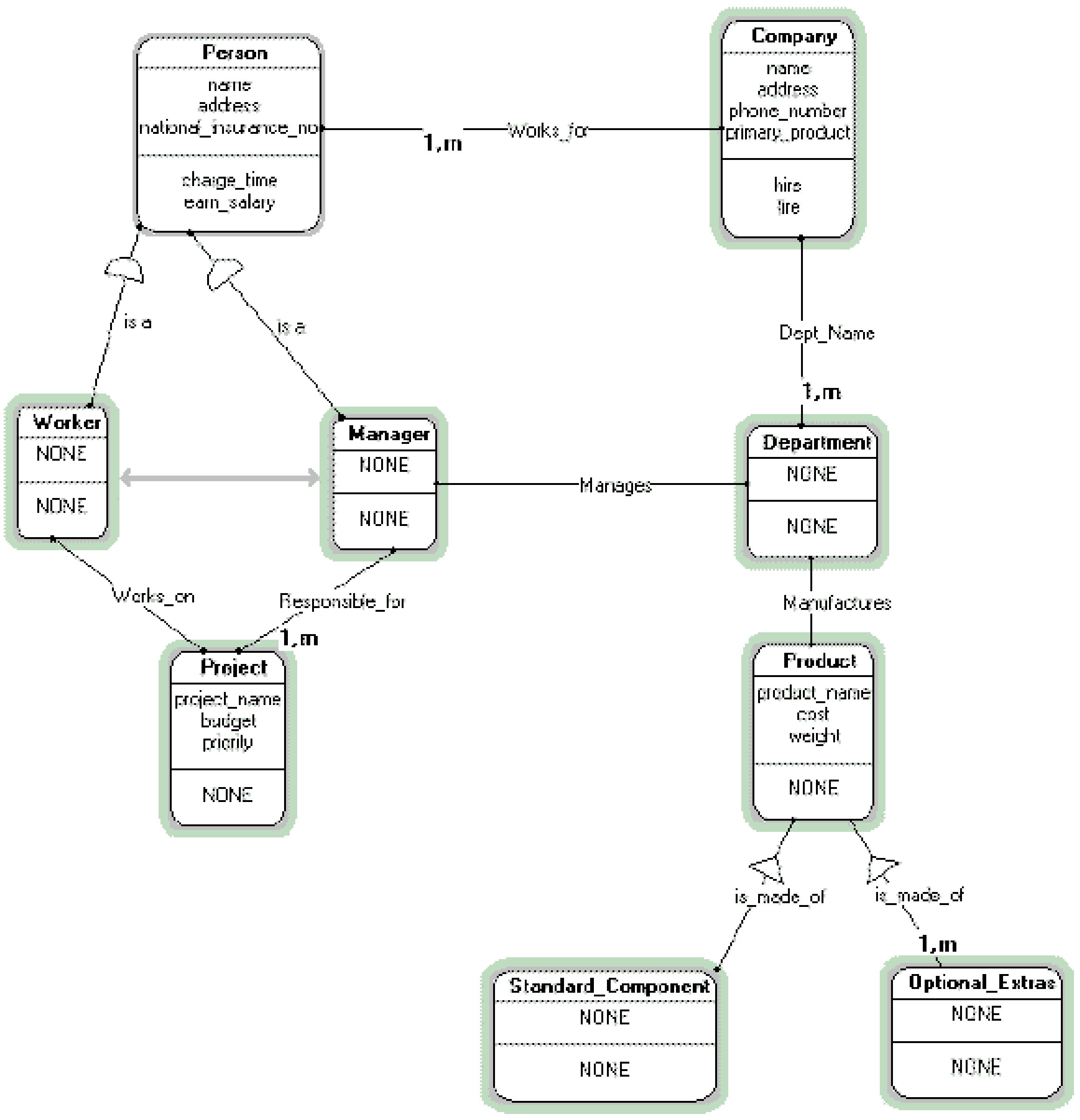

Figure 1.6.2 - A Coad/Yourdon OOA Model of a Company

1.6.3 Analysis of Method

Coad and Yourdon's object-oriented method gives the impression of having been quickly adapted from their other work in time to jump on to the object-oriented bandwagon. Bilow [Bilo91] notes that portions of their exposition of the method are "excessively detailed for management-oriented readers, while other sections are somewhat overly general for designers and implementors."

The four components of the design approach are not presented with enough detail to be put directly into practice. However, their approach to OOA and design is easy to understand, and their method is put across at a level which avoids some of the difficult complexities of formalised object orientation. As such, it gives a good introduction to object orientation. However, there are sufficient flaws in the method to avoid it for large or highly complex systems.

1.6.4 Support for reuse

Coad/Yourdon considers only design with reuse. For example, Coad and Yourdon's discussion of reuse focuses exclusively on the problem domain component, as stated in "More effective analysis requires the use of problem domain constructs, both for present reuse and future reuse." Only one component of their method, that of the Problem Domain, is singled out for potential reuse.

Tools available

Name of Tool Company Price Platform ObjecTool Object International Ltd Unknown Windows, OS/2, HP/Sun UNIX Together/C++ Object International Ltd. 790 Windows MetaEdit MetaCase Consulting $499 Windows ObjectModeler Iconix Software Engineering $1,495 Macintosh, UNIX OOTher Roman M. Zielinski Free Windows Paradigm Plus Protosoft $3,995 $7,770 Windows UNIX Prosa/om Prosa Software Unknown Windows System Architect Popkin Software $1,395 $1,795 Windows OS/2 ObjectMaker Mark V Software Unknown Windows, UNIX, Macintosh ATRIOM Semaphore Unknown Windows MacAnalyst and MacDesigner Excel Software $995 - $1,595 Macintosh

1.7 Object Oriented Structured Analysis (OOSA)/Object Oriented Design Language (OODLE) [Shla92]

1.7.1 The Method

In the method, the software system is initially split up into domains, which can then be broken down into subsystems to be analysed in depth. The method provides comprehensive coverage for the analysis, design and implementation phases of the software development lifecycle. The process can be described in following steps:

1) **Partition the system into domains** - The application domain is defined as the subject matter that is of concern to the enduser of the system. The service domains may include the user interface and other external interfaces of the system. The architectural domain covers the hardware and software which will be used to build and run the desired system. The implementation domains will include the operating system(s) and programming language(s) that will be used to build the system. A Domain Chart is used to depict these domains, and the interrelationships between them (which are referred to as bridges).

2) **Analyse the application domain** - The application domain is analysed using Shlaer/Mellor's integrated set of Object-Oriented Analysis (OOA) models. This produces a set of formal models which are depicted in three related diagrams:

- The Information Model, which defines the conceptual entities or objects in the domain, and the relationships between them.
- The State Models, which describe the behaviour of each object and the interaction between them.
- The Action Data Flow Diagrams, which factor the processing that is required in the State Models.

3) **Verify the analysis through simulation** - The OOA models provide a formal method for verifying the specified behaviour of the system by simulation of the execution of the models. In these simulations, the processes, data and process sequence are used to verify the execution of the models defined. The model execution is simulated by following these steps:

1) Determine the initial state of the data values in the Information Model
2) Initiate the desired behaviour with an event sent to a State Model
3) Execute the processing in the Action Data Flow Diagrams, following the sequence given in the State Models

4) Evaluate the overall outcome against the expected results

4) **Extract the requirements for the service domain** - Each service domain is considered in terms of the requirements that each of the other domains make of it. This allows for the purpose of each domain to be identified for the next step.

5) **Analyse the service domains** - The service domains are analysed using the OOA models, by the same process used in step 2. Due to the client/server nature of the relationships between the domains, server domains must not be analysed until sufficient analysis has been done on their clients to ensure that all the requirements of the server have been identified.

6) **Specify the components of the architectural domain** - The architectural domain is specified using OOA, but can be designed using whichever design method is deemed appropriate. If an object oriented design is required, Shlaer/Mellor's Object Oriented Design LanguagE (OODLE) is recommended. OODLE uses four types of diagram interrelated by a layering scheme to depict the design of an object oriented program, library or environment:

- Inheritance diagrams, which show the inheritance relationships between classes
- Dependency diagrams, showing usage (client/server) and friend relationships between classes
- Class diagrams, which show the external view of a single class
- Class structure charts, showing the structure of the methods and the flow of data and control within a class

7) **Build the architectural components** - There are 2 components in the architectural domain:

mechanisms, which represent the architecture-specific capabilities that must be provided to realise the system. Mechanisms are built by traditional software development methods to produce code in the form of tasks and class libraries.

structures, which represent a prescription for translating the OOA models of the client domains. Structures are built as a combination of code and data that needs to be completed by adding elements from the client domains. When built, structur es are called templates, and are 'populated', or completed using the domain specific information, prior to compilation.

8) **Translate the OOA models of each domain using the architectural components** - The details of the final step depend a great deal on the design chosen for the system, and the architectural components created. However, the general principles for multitasking and multiprocessor systems can be described in two activities:

1) allocate objects to tasks and tasks to processors. The results of this activity are recorded in the Task Communication diagram.
2) create the tasks through a translation of the OOA models.

1.7.2 Notation

There are several diagrams used for describing the structure of a system, namely the inheritance diagram, the dependency diagrams and the class diagram. For convenience, these have been merged to show the object oriented design of a company in figure 1.7.2. The symbols used in the notation are as follows:

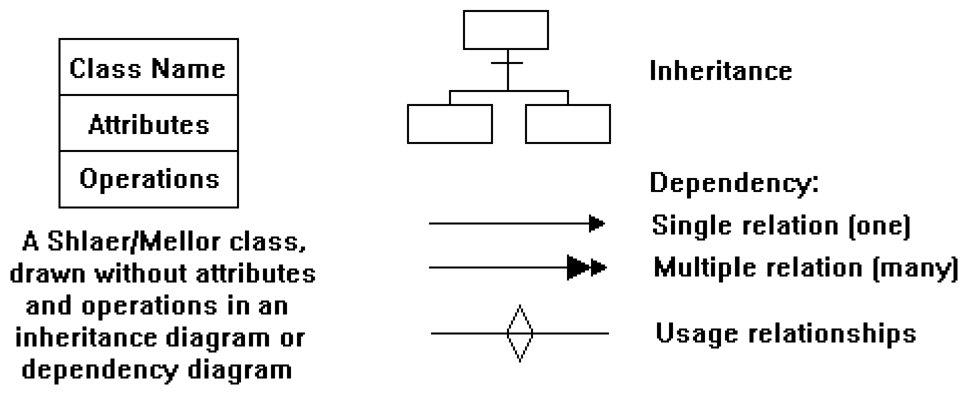

Figure 1.7.1 - A Key to the OODLE Diagram notation

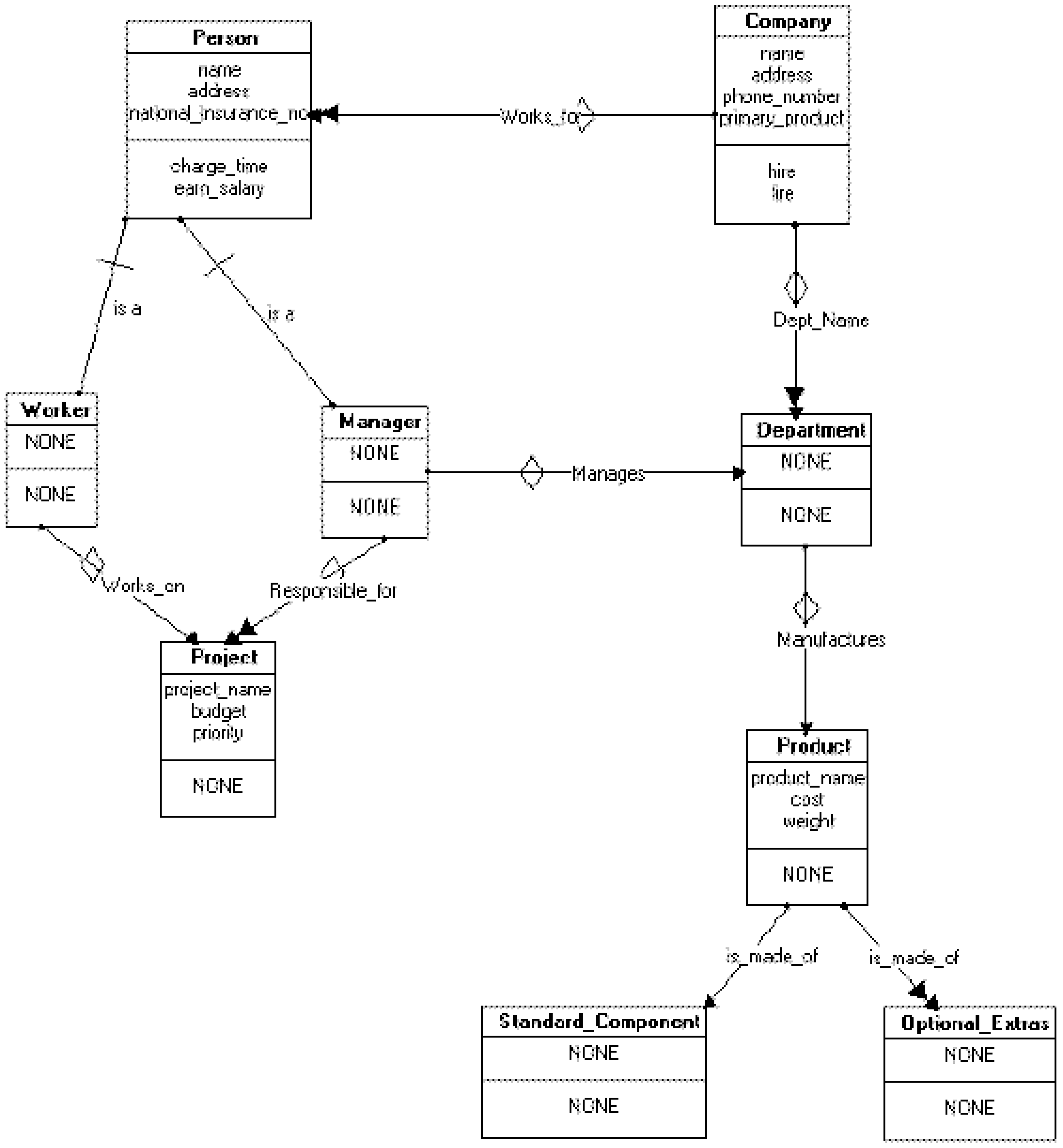

Figure 1.7.2 - A simplified OODLE Diagram for a Company

1.7.3 Analysis of Method

The Shlaer/Mellor method of object oriented analysis and design was one of the earlier attempts at defining a structured method for object orientation. It was originally more object-based than truly object oriented, but has been refined since its initial conception to include many of the key elements of object orientation.

The OOA model simulation is well defined within the method, and provides a means for testing the validity of the system which has been captured in the models. Due to the structured nature of the simulation, much of the process can be automated successfully. The process of populating the structures in the architectural domain can also be partially automated, making the templates built as a result of this method highly reusable.

1.7.4 Support for reuse

Shlaer and Mellor do discuss reuse, but only in terms of reusing processes, and not objects or classes. They suggest that a set of general rules should be applied across all code modules rather than the commonly used 'low-level tinkering' to tune systems. In this way, their recursive design strategy allows the application to be separated from various reusable domains at different levels. It is therefore the interfaces between these domains that need to be carefully considered, allowing reuse to occur within the domains.

Tools available

Name of Tool Company Price Platform Bridgepoint Objective Spectrum $6,000 UNIX TeamWork/ Objectteam CADRE Technologies $4,000 $8,000 PC UNIX Objectbench SES $4,900- $24,300 Macintosh, DOS, UNIX Objective Analyst Objective Spectrum Unknown Unknown Intelligent Analyst Kennedy Carter Unknown Unknown EasyCASE Evergreen CASE Tools $495 Windows, DOS System Architect Popkin Software $1,395 $1,795 Windows OS/2 Toolbuilder IPSYS Software $17,000 Sun Sparc, Apollo, HP, DECstation, RS6000 ATRIOM Semaphore Unknown Windows MacAnalyst and MacDesigner Excel Software $995 - $1,595 Macintosh

1.8 Other Object Oriented Methods

1.8.1 Object Oriented Software Engineering (OOSE) [Jaco92]

Jacobsons OOSE method stresses a pyramid model for the process of developing an object oriented design, in which tools provide support for the activities in three categories:

1. architecture, which signifies the choice of techniques to be utilised;
2. method, which makes explicit the procedures to be followed and
3. process, which allows the method to be scaled up.

One of the key concepts in OOSE, as described by Jacobson [Jaco92] is that of the use case. A use case is defined as "a behaviorally related sequence of transactions" which a user will perform in a dialogue with the system. Thus, each use case is a specific way of using the system, and when a user inputs a stimulus, an instance of a use case executes and starts a transaction belonging to that use case. Based on this view of the system, Jacobson associates the use case model with five other models of the system:

- domain object model - the use case model is expressed in terms of the domain
- analysis model - the use cases are structured by the analysis
- design model - the use case model is realised by the design
- implementation model - this implements the use case model as realised in the design
- testing model - used to test the realisation of the use case model

Although Jacobsons OOSE design method is not particularly strong, his view of the system in terms of use cases is a powerful method for analysing the system in terms of the demands made of the system, and the processing that is required to meet those demands. As noted in section 1.4.3, Rumbaugh is now advocating the inclusion of use cases as a 'front-end' to the Object Modelling Technique.

1.8.2 Object-Oriented Structured Design (OOSD) [Wass90]

OOSD is not so much of a design method as an object-oriented notation. As described by Wasserman et al. [Wass89], the "goal of OOSD is to provide a single architectural design notation that can support every software design."

OOSD is described as being methodology independent. Graham [Grah93] says that it is "strictly speaking, not a method, but a notation to which methodological rules can be added. " As Wasserman et al. [Wass90] put it: "it is not our intent to prescribe a single best method for design...the OOSD notation is neutral...it is expected that designers will develop and use their own design esthetics and design metrics within the framework of OOSD, which simply carries a valid design notation...OOSD supports a wide range of design philosophies."

The main entities in OOSD are:

- **A system**, which consists of a set of units, which are either classes, modules or monitors.
- **A class**, which can inherit from, and use other classes. A class contains public and private units, and can raise exceptions.
- **A module**, which is either a data-object or an operation.
- **A monitor**, which is used to define and use asynchronous processes.

OOSD strengths derive from:

- keeping modularity as the main component of the design method.
- building on a familiar notation.
- attempting to target a wide variety of applications and development languages, while supporting development from a wide range of analysis methods.

Without any method behind the notation, OOSD is useless to those trying to learn the principles of object orientation. It may, however, be of use to those already familiar with a method, who find its notation limiting.

1.8.3 Object Oriented Role Analysis, Synthesis and Structuring (OORASS) [Reen92]

OORASS is an analysis and design method which emphasises the role played by objects in the system. To begin the analysis, areas of concern are identified, which are effectively high level business functions which the system will provide. Each of these areas is modelled using collaborating agents and objects which take various roles in the structure. A role is dependent on the requirements of the system rather than the properties of the object, thus a single object may perform a variety of roles at different stages of the system. Analysis is performed as a top down, but not hierarchical, process until the model is stable and the analysts fully understand the areas of concern. Types of objects are then synthesised based on the roles that they are required to play. Synthesis is the process of creating new objects by inheriting behaviour from simpler objects. Generally useful objects will be those that can play a number of roles within the system, and can therefore be reused throughout the design process. The goals of OORASS are:

1. **Total lifecycle support** - the method should cover the analysis, design, implementation, testing and maintenance of object oriented programs in a seamless fashion
2. **Integrated description** - the method should support an integrated representation of all kinds of informal documentation as well as source code and formal descriptions of the design

3. **Separation of concerns** - the method should allow different concerns to be treated independently, to allow the whole system to be understood in parts
4. **Reuse** - the method should support the notion of reusable ideas, specifications, designs, code and compiled programs
5. **Tool support** - the method will be supported by an integrated support environment for the complete system lifecycle

One of the advantages of OORASS is the ability to construct simple role models representing object interaction, and then combine them in a uniform way to create larger models. This encourages reuse on a larger scale than is advocated in other object oriented methods.

1.8.4 Jackson Structured Design (JSD) [Jack83]

It has been suggested (by Booch [Booc86], among others) that the Jackson Structured Design method may be suitable for use with OOD. The similarities between JSD and object-oriented design have been examined by several authors, including Masiero and Germano [Masi88], who found that, with limitations, "JSD can be used as an object oriented method".

One of the central notions in JSD is that of the Long Running Process (Program) (LRP). An LRP is made up of the time-ordered sequence of actions that an entity performs or suffers. Walker [Walk92] notes that this "view of the 'Long Running Process' with its state information is somewhat analogous to the notion of 'behaviour' (the implementation of an object's responsibilities) in Wirfs-Brock's Responsibility-Driven Approach". The Responsibility-Driven Approach, however, "involves a much more detailed analysis of the nature of the operations performed or suffered by an object with respect to its environment/interacting objects. Suffice to say, the notion of entities/objects being characterised by the set of operations or functions that they perform over their lifetime is clearly very powerful, and is employed in a number of existing design methodologies."

JSD has no concept of inheritance in its method. It is, therefore, more of an object-based method than truly object-oriented. However, there is a remarkable resemblance between the JSD suggestions for identifying entities and actions, as explained by Jackson [Jack83]: "make a comprehensive list of nouns and verbs we encounter in our study of reality: each noun is a possible entity; each verb is a possible action"; and the method for object and operation identification proposed by Booch [Booc91].

2. Object Oriented Design and Reuse

Object-oriented methods have been promoted as inherently supporting reuse. Halladay and Wiebel [Hall93] state that "The most commonly touted benefit of OOP is reuse." Many authors have extolled the advantages of reusability in object-orientation. Atkins and Brown [Atki91] emphasise that reuse is one of the advantages that arise from an object oriented approach, specifically from direct support for abstraction. They suggest that the reuse of classes in a hierarchy, and object libraries are specific examples of reuse that stem from object oriented practices.

Ince [Ince91] says that "Polymorphism allows a developer to build up a library of reusable objects, and contributes greatly towards the ability to develop reusable software." Wiener and Pinson [Wien88] consider one of the main goals of object-oriented software development to be "to shorten the time and lower the cost of development by using reusable software components in the form of baseline classes and by employing incremental problem solving using subclasses." Tsichritzis and Nierstrasz [Tsic89] seem to consider that because of object-oriented programming's heavy emphasis on reuse, that "we can expect extremely large collections of reusable objects to be available to us." They feel that the problems of the future will be associated with managing such large collections of objects, and suggest that perhaps expert systems will be the appropriate tools for helping programmers to find their way through databases of reusable object classes.

It has been seen, however, that among the object-oriented design methods available, there is a lack of explicit provision for reuse [Goss90]. Udell [Udel94] also expresses this opinion: "The traditional OOP vision was, at best, vague on the subject of reuse: Objects would appear as by- products of software development, a market would emerge, and programmers would become producers and consumers of objects." This unstructured, and rather naive, view of reuse can be seen in several object-oriented texts.

Meyer [Meye88] offers considerably more advice on the construction of reusable classes in his text. He suggests that: "A good object-oriented environment will offer a number of predefined classes implementing important abstractions. Designers will naturally look into these to see if there is anything they can use...New applications, if properly done, will also produce more specialized reusable classes. As object-oriented techniques spread, the number and abstraction level of available components grow."

Tello [Tell91] also questions the provision for reuse in object-oriented methods. He states: "some say that the key advantage of OOP is the ability to reuse code for many different programs, but, by itself, this is not significantly different from library functions." Mullin [Mull89] would agree: "As most books available today on OOP say, one of the major benefits of objects is that they are reusable. So are C functions. The difference is that objects, representing both data structures and operations that can be performed on these structures, represent functional packages, requiring no additional work on the part of the programmer to use them."

Raj and Levy [Raj89] note that one of the problems with inheritance in object-oriented systems is that "classes are not automatically reusable". They suggest that for successful reuse, inheritance requires the use of a set of coding rules and a set of design rules. Johnson and Foote [John88] would agree with this second point, as they present a set of 13 rules for designing reusable classes. Udell [Udel94] suggests that "object technology failed to deliver on the promise of reuse", but that componentware; in which components are encapsulated, or combined into a single, separate unit with a well defined interface, in order to make them reusable; is the way forward for reusability.

Winblad et al. [Winb90] note that "Software reuse does not occur by accident, however - even with object-oriented programming languages. System designers must keep the advantages of reusability in mind, planning ahead to reuse what already exists and designing reusability into the new components they create. This requires that programmers adopt new programming behavior, values, and ethics. Borrowing classes created by others must be considered more desirable than implementing a new class. Reviewing existing code to identify opportunities for reuse must have priority over writing new code. Finally, programmers must create simple, reusable classes rather than complex, inscrutable classes. Simplicity is a major tenet of the general philosophy of object orientation."

It is important to note that no one method, technology or technique will solve all the problems associated with reuse. There will always be complications, and these must be expected and planned for. Burd and McDermid [Burd92] note that: "Risks are involved in all software developments, however, often those projects which employ reuse are susceptible to greater risks than those which do not." However, by the use of object-oriented techniques, with their excellent support reuse, these risks, and the difficulties involved in successful reuse can be reduced. This view is confirmed by Burd [Burd93]: "Object oriented design displays the most promise as a re- use methodology...Object orientation on its own isn't sufficient to solve all the problems associated with re-use. This can be achieved only by providing well-defined support that enables re-use to be integrated into a suitable lifecycle model."

It is with this in mind that a CASE tool is proposed which will provide not only support for the use of an object oriented method in software development, but will allow existing code to be reverse engineered to abstract general design decisions, structure the documentation associated with the code in such a way as to make it useful to the developers, and will also provide the techniques and functionality required to support a software reuse program. Although a tool will not solve all the problems that inhibit software developers in a small company from structuring their development in such a way as to gain great benefits for software reuse, it is felt that, with the technology available, it will be easier to put the processes in place that will allow the benefits of software reuse to be maximised.

References

[Abbo83] Abbott, R.; 'Program Design by Informal English Description'; Communications of the ACM; Nov 1983 Vol.26 No.11 P882-894

[Atki91] Atkins, M.C., Brown, A.W.; 'Principles of object-oriented systems'; In: Software Engineer's Reference Book; ed. McDermid, J.A.; Butterworth-Heinemann Ltd.; 1991; P39/3- 39/13

[Bilo91] Bilow, S.; 'Book Review: Object-Oriented Design'; Journal of Object Oriented Programming; Oct 1991; Vol.4 No.6 P73-74

[Booc86] Booch, G.; 'Object-Oriented Development'; IEEE Transactions on Software Engineering; Feb 1986; Vol.12 No.2 P211-221

[Booc87] Booch, G.; 'Software Engineering with Ada (2nd Ed.)'; Benjamin/Cummings, California; 1987

[Booc91] Booch, G.; 'Object Oriented Design with Applications'; Benjamin/Cummings, California; 1991

[Burd92] Burd, E.L., McDermid, J.A.; 'Guiding Reuse with Risk Assessments'; University of York Technical Document YCS 183 (1992); York; 1992

[Burd93] Burd, E.L.; quoted in 'Spiral of Success'; Peltu, M.; Computing; 28 Jan 1993; P24

[Coad90] Coad, P., Yourdon, E.; 'Object-Oriented Analysis'; Yourdon Press, Prentice Hall, New Jersey; 1990

[Coad91] Coad, P., Yourdon, E.; 'Object-Oriented Design'; Yourdon Press, Prentice Hall, New Jersey; 1991

[Cox86] Cox, B.J.; 'Object Oriented Programming - An Evolutionary Approach'; Addison- Wesley, Reading, Mass.; 1986

[Goss90] Gossain, S., Anderson, B.; 'An Iterative-Design Model for Reusable Object-Oriented Software'; ECOOP/OOPSLA'90 Proceedings; Oct 1990; P12-27

[Goss91] Gossain, S.; 'Book Review: Designing Object-Oriented Software'; Journal of Object Oriented Programming; Mar/Apr 1991; Vol.4 No.1 P82-84

[Grah93] Graham, I; 'Object Oriented Methods'; Addison-Wesley; 1993

[Hall93] Halladay, S., Wiebel, M.; "Object Oriented Software Engineering"; Prentice Hall; 1993

[Hood93] Delatte, B., Heitz, M., Muller, J.F. ed.; 'HOOD Reference Manual 3.1'; Masson, Paris; 1993

[Ince91] Ince, D.; 'Object-Oriented Software Engineering with C++'; McGraw-Hill, London; 1991

[Jack83] Jackson, M.A.; 'System Development'; Prentice Hall, New Jersey; 1983

[Jaco92] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.; 'Object-Oriented Software Engineering - A Use Case Driven Approach'; ACM Press, Addison-Wesley, Reading, Mass.; 1992

[John88] Johnson, R.E., Foote, B.; 'Designing Reusable Classes'; Journal of Object-Oriented Programming; Jun/Jul 1988; Vol.1 No.2 P22-30,35

[Jone92] Jones, R.; 'How applicable is the object-oriented approach to the IS environment?'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Masi88] Masiero, P.C., Germano, F.S.R.; JSD as an Object Oriented Design Method; Software Engineering Notes; Jul 1988; Vol.13 No.3 P22-23

[Meye88] Meyer, B.; 'Object-Oriented Software Construction'; Prentice Hall, New Jersey; 1988

[Mull89] Mullin, M.; 'Object Oriented Program Design with examples in C++'; Addison-Wesley, Reading, Mass.; 1989

[Orms91] Ormsby, A.; 'Object-Oriented Design Methods'; In: Object-Oriented Languages, Systems and Applications; ed. Blair, G., Gallagher, J., Hutchison, D., Shepherd, D.; Longman, London; 1991; P203-222

[Pres87] Pressman, R.S.; 'Software Engineering - A Practitioner's Approach (2nd Ed.)'; McGraw- Hill, London; 1987

[Raj89] Raj, R.K., Levy, H.M.; 'A Compositional Model for Software Reuse'; In: ECOOP'89 Proc. of the 1989 European Conference on Object-Oriented Programming; Cook, S. (ed.); Cambridge University Press; 1989; P3-24

[Reen92] Reenskaug, T., Andersen, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., Stenslet, P.; 'OORASS: seamless support for the creation and maintenance of object oriented systems'; Journal of Object Oriented Programming; Oct 1992; Vol.5 No.6 P27-41

[Rent82] Rentsch, T.; 'Object Oriented Programming'; SIGPLAN Notices; Sept 1982; Vol.17 No.12; P51

[Robi92a] Robinson, P.J.; 'Hierarchical Object-Oriented Design'; Prentice Hall, London; 1992

[Robi92b] Robinson, P.J. 'Introduction and overview'; In: Object-Oriented Design; Robinson, P.J. ed.; Chapman & Hall, London; 1992; P1-10

[Robi92c] Robinson, P.J. 'HOOD method tutorial'; In: Object-Oriented Design; Robinson, P.J. ed.; Chapman & Hall, London; 1992; P11-40

[Rumb91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.; 'Object-Oriented Modeling and Design'; Prentice Hall, New Jersey; 1991

[Rumb94] Rumbaugh, J.: 'Getting started: Using use cases to capture requirements''; Journal of Object Oriented Programming; Sept 1994; Vol.7 No.5 P8-12,23

[Shla92] Shlaer, S., Mellor, S.; 'Object Lifecycles: Modeling the World in States'; Prentice Hall; 1992

[Tell91] Tello, E.R., 'Object-Oriented Programming for Windows'; John Wiley and Sons; 1991

[Tsic89] Tsichritzis, D.C., Nierstrasz, O.M.; 'Directions in Object-Oriented Research'; In: Object- Oriented Concepts, Databases, and Applications; Kim, W., Lochovsky, F.H. (ed.); ACM Press, Addison-Wesley, Reading, Mass.; 1989

[Udel94] Udell, J.; 'Componentware'; Byte; May 1994; Vol.19 No.5 P46-56

[Walk92] Walker, I.J.; 'Requirements of an object-oriented design method'; Software Engineering Journal; Mar 1992; Vol.7 No.2 P102-113

[Wass89] Wasserman, A.I., Pircher, P.A., Muller, R.J.; An Object-Oriented Structured Design Method for Code Generation; ACM Software Engineering Notes; Jan 1989; Vol.14 No.1 P32-55

[Wass90] Wasserman, A.I., Pircher, P.A., Muller, R.J.; The Object-Oriented Structured Design Notation for Software Design Representation; IEEE Computer; Mar 1990; Vol.23 No.3 P50-63

[Wien88] Wiener, R.S., Pinson, L.J.; 'An Introduction to Object-Oriented Programming and C++'; Addison-Wesley, Reading, Mass.; 1988

[Winb90] Winblad, A.L., Edwards, S.D., King, D.R.; 'Object-Oriented Software'; Addison- Wesley, Reading, Mass.; 1990

[Wirf90] Wirfs-Brock, R., Wilkerson, B., Wiener, L.; 'Designing Object-Oriented Software'; Prentice Hall, New Jersey; 1990

Appendix A - A Home Heating System Example

This appendix gives an example object-oriented design in the various design notations surveyed earlier in this report. The example used is that of a home heating system, for which a description is given below. The example is taken from Booch [Boo91]. Each design uses identical information, in order to highlight the differences between each of the design notations.

A1. Home Heating System Requirements

A home heating system is required which is to regulate the flow of heat to individual rooms of a home in an attempt to maintain a working temperature (tw) established for each room. The working temperature for each room is calculated by the system as a function of a single desired temperature td (set by the user through a manual input device) and a living pattern, which is set by the user and indicates the times at which the rooms are to be heated.

Each room has a temperature sensor which measures the current temperature (tc) of the room. Each room is equipped with a water valve that controls the flow of hot water into the radiator of the room. The valve can be either fully open or fully closed. If the current temperature is less than the desired temperature and the living pattern indicates that the room is to be heated at this time, the valve is opened to allow hot water to flow into the room.

The timer provides a continuously incrementing count, one increment for every second of elapsed time.

The furnace consists of a boiler, an oil valve, an ignitor, a blower and a water-temperature sensor. The furnace heats the water in the boiler, and the water can then be circulated to one or more rooms of the home.

The furnace is alternately activated and deactivated by the heat-flow regulator as needed to maintain the required temperature for each room. The furnace indicator shows whether the furnace is currently on or off.

If a fault develops in the furnace, the fault indicator is lit. This is associated with the fault reset switch, thus the user can reset the fault indicator by setting the switch to on.

The heat flow regulator interacts with other components of the home heating system to determine the heating needs for each room and to control the flow of heat necessary to satisfy those requirements. The heat flow regulator maintains a working temperature and a living pattern for each room without regard to the running state of the furnace. The heat-flow regulator determines that a given room needs heat whenever the current room temperature is equal to or less than (tw- 2) degrees Farenheit and determines that a room does not need heat whenever the current room temperature is equal to or greater than (tw+2) degrees Farenheit.

This is only if the living pattern shows that the room is currently to be heated. If the furnace is not running, the heat switch and the fault reset switch/indicator are both on, and at least one room needs heat, the heat-flow regulator activates the furnace and then routes heat to the appropriate rooms. If the furnace is running, the heat-flow regulator deactivates the furnace whenever either the heat switch or the fault reset switch/indicator is off and no rooms need heat. The minimum time for furnace restart after prior operation is five minutes.

A2. Booch's OOD

A2.1 Class Diagram

A2.2 Class Templates

```
Name: HeatFlowRegulator
Documentation: text
Cardinality: 1
Hierarchy:
  Superclasses: Object
Public Interface:
          Uses: Furnace
                Home
                OperatorInterface
                Room
    Operations: initialise: furnace: operator:
                needsHeat:
                noLongerNeedsHeat:
                release
                respondToFaultResetSwitch
                respondToFurnaceFault
                respondToFurnaceNotRunning
                respondToFurnaceRunning
                respondToHeatSwitchOff
                respondToHeatSwitchOn
Implementation:
        Fields: roomsNeedingHeat
    Operations: add:
                remove:
                size

Name: OperatorInterface
Documentation: text
Cardinality: 1
Hierarchy:
  Superclasses: Object
Public Interface:
          Uses: HeatFlowRegulator
    Operations: heatStatus
                initialise:
```

```
                release
                reportFault
                reportFurnaceStatus:
Implementation:
          Uses: FaultResetSwitchIndicator
                FurnaceStatusIndicator
                HeatSwitch
        Fields: theFaultResetSwitchIndicator
                theFurnaceStatusIndicator
                theHeatFlowRegulator
                theHeatSwitch
                theView
Concurrency:    active

Name: HeatSwitch
Documentation: text
Cardinality:
Hierarchy:
  Superclasses:
  Metaclass:
Public Interface:
    Operations: initialise:
                switchOn
                switchOff
Implementation:
        Fields: heatSwitchStatus

Name: FaultResetSwitchIndicator
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: ToggleSwitch
  Metaclass:
Public Interface:
    Operations: switchOn
                respondToFault
Implementation:
        Fields: theView
    Operations: switchOff
Name: ToggleSwitch
Documentation: text
Cardinality: 0
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: toggleState
                returnState
Implementation:
        Fields: state

Name: FurnaceStatusIndicator
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: Indicator
  Metaclass:
Public Interface:
    Operations: initialise:
Implementation:
        Fields: furnaceStatus

Name: Indicator
Documentation: text
Cardinality: 0
```

```
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: reportStatus
                setStatus
Implementation:
        Fields: status
                theView

Name: HeatingSystem
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: powerDown
                powerUp
Implementation:
        Fields: theFurnace
                theHeatFlowRegulator
                theHome
                theOperatorInterface
Name: Home
Documentation: text
Cardinality:
Hierarchy:
  Superclasses:
  Metaclass:
Public Interface:
    Operations: closeAllWaterValves
                openWaterValve
                closeWaterValve
                needsHeat
                noLongerNeedsHeat

Name: Timer
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: reportElapsedSeconds
                initialise:
                release
Implementation:
        Fields: elapsedSeconds
                theSemaphore
                timerProcess

Name: ClockCalendar
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: Timer
  Metaclass:
Public Interface:
    Operations: reportCurrentTime
                reportCurrentDate
                initialise:
Implementation:
        Fields: dayOfWeek
                hour
```

```
                 minute
                 second
     Operations: respondToTick

Name: Furnace
Documentation: text
Cardinality: 1
Hierarchy:
  Superclasses: Object
Public Interface:
           Uses: HeatFlowRegulator
     Operations: activate
                 deactivate
                 initialise:
                 release
Implementation:
           Uses: BoilerTemperatureSensor
                 ClockCalendar
                 CombustionFaultSensor
                 Blower
                 OilValve
                 Ignitor
                 OilFaultSensor
         Fields: currentState
                 itsSemaphore
                 theBoilerTemperatureSensor
                 theCombustionFaultSensor
                 theHeatFlowRegulator
                 theBlower
                 theOilValve
                 theIgnitor
                 theOilFaultSensor
                 theView
                 timeDelay
     Operations: postFault
                 postNotRunning
                 update:
Concurrency:     active

Name: Room
Documentation: text
Cardinality: n
Hierarchy:
  Superclasses: Object
Public Interface:
           Uses: HeatFlowRegulator
     Operations: closeWaterValve
                 initialise: location: heatFlowRegulator:
                 name
                 openWaterValve
                 release
Implementation:
           Uses: ClockCalendar
                 CurrentTemperatureSensor
                 DesiredTemperatureSensor
                 LivingPattern
                 WaterValve
         Fields: currentState
                 name
                 theCurrentTemperatureSensor
                 theDesiredTemperatureSensor
                 theHeatFlowRegulator
                 theLivingPattern
                 theSemaphore
                 theView
```

```
                theWaterValve
    Operations: heatingRoom: current: desired:
                setInitialState
                startHeating
                stopHeating
                update:
Concurrency:    active

Name: DesiredTemperatureSensor
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: TemperatureSensor
  Metaclass:
Public Interface:
    Operations: setDesiredTemperature
                initialise:
Implementation:
        Fields: desiredTemperature
                theView

Name: CurrentTemperatureSensor
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: TemperatureSensor
  Metaclass:
Public Interface:
    Operations: initialise:
Implementation:
        Fields: currentTemperature
    Operations: monitorCurrentTemperature

Name: TemperatureSensor
Documentation: text
Cardinality: 0
Hierarchy:
  Superclasses: SimpleSensor
  Metaclass:
Public Interface:
    Operations: reportTemperature
Implementation:
        Fields: temperature

Name: SimpleSensor
Documentation: text
Cardinality: 0
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: respondToExternalChange
Implementation:
        Fields: value
                theSemaphore
    Operations: monitorExternalValue

Name: WaterValve
Documentation: text
Cardinality:
Hierarchy:
  Superclasses: SimpleValve
  Metaclass:
Public Interface:
    Operations: initialise:
```

```
Implementation:
        Fields: waterValveStatus

Name: SimpleValve
Documentation: text
Cardinality: 0
Hierarchy:
  Superclasses: Object
  Metaclass:
Public Interface:
    Operations: openValve
                closeValve
Implementation:
        Fields: valveStatus
```

A3. HOOD

A3.1 HOOD Diagram

A4. Rumbaugh's OMT

A4.1 OMT Object Model

A5. Responsibility-Driven Design/Class-Responsibility-Collaboration

A5.1 Collaboration Graph

A5.2 Contracts

```
1) Heat Flow Regulator
initialise
needsHeat
noLongerNeedsHeat
release
respondToFaultResetSwitch
respondToFurnaceFault
respondToFurnaceNotRunning
respondToFurnaceRunning
respondToHeatSwitchOff
respondToHeatSwitchOn

2) Operator Interface
heatStatus
initialise
release
reportFault
reportFurnaceStatus:

3) Toggle Switch
toggleState
returnState
HeatSwitch: initialise
HeatSwitch: switchOn
HeatSwitch: switchOff
FaultResetSwitchIndicator: switchOn
FaultResetSwitchIndicator: respondToFault
```

```
4) Indicator
reportStatus
setStatus
FurnaceStatusIndicator: initialise

5) Room
closeWaterValve
initialise
name
openWaterValve
release

6) Simple Sensor
respondToExternalChange

7) Temperature Sensor
reportTemperature
CurrentTemperatureSensor: initialise
DesiredTemperatureSensor: initialise
DesiredTemperatureSensor:
setDesiredTemperature

8) Simple Valve
openValve
closeValve
WaterValve: initialise

9) Furnace
activate
deactivate
initialise
release

10) Timer
initialise
reportElapsedSeconds
release
ClockCalendar: initialise
ClockCalendar: reportCurrentTime
ClockCalendar: reportCurrentDate
```

A6. Coad/Yourdon's Object-Oriented Analysis

A6.1 OOA Model

A7. Shaler/Mellor's Object-Oriented Design Language

A7.1 OODLE Diagram