OBJECT-ORIENTED METHODOLOGY FOR HOME AUTOMATION APPLICATIONS
BRANZ Study Report SR 52

A H Dechapunya

REFERENCE
Dechapunya, A.H. 1993. Object-Oriented Methodology for Home Automation Applications. Building Research Association of New Zealand, BRANZ Study Report SR 52, Judgeford, New Zealand.

KEYWORDS
Appliances, Automatic Controls, Computer Languages, Controls, Home Automation, Manual Controls, Object-oriented Programming, Remote Controls, Security Systems, Sensors, Smart House, Telecommunications, Thermostats, Timers.

ABSTRACT

The object-oriented paradigm has been examined as a software engineering approach to the development of home automation applications. Object-oriented programming evolves as a consequence of natural progress in programming languages. This type of programming results in reduction in program complexity, modularity, code reusability, and programming efficiency; which in turn brings about cost savings in the development and maintenance of software.

Object-oriented programming mimics human interaction with objects in real life. This makes it easy to model home automation using the object-oriented paradigm. The encapsulation of data and methods within an object makes it easy for interoperability to be implemented. (Interoperability is the ability for different products from different manufacturers to work together.)

Smart House and CAL object specifications for home automation are examined, as is an object-oriented programming language. Finally, the application of object-oriented technology to home automation applications is demonstrated.

CONTENTS

TABLES

1.0 OBJECTIVES

This work sets out to identify the potential of advanced software engineering techniques to provide a structure and platform for an integrated information systems environment, by:

* evaluating object-oriented methodology as a software engineering approach for home automation applications

* producing an example of an object-oriented model for home automation.

2.0 HOME AUTOMATION APPLICATIONS

2.1 Home Automation

Home Automation is a term that denotes the total integration of various activities or services within the home. It links together domestic appliances, lights, telephones etc., in order to give homeowners a means to automate and control home activities. This frees them from day-to-day routine operation.

Home automation provides the following integrated services (Dechapunya, 1992):

* Automation of appliances and lights
* Energy management
* Communication and information
* Entertainment
* Security and safety
* Environmental control.

2.2 Interoperability

Since the mid 1980s, manufacturers, governments, standards organisations, and consortiums all over the world have been working at establishing standards for communication/interconnection between home products. The joint technical committees of the International Standards Organisation and the International Electrotechnical Commission (ISO/IEC) recently established a working group SC25 to establish an international standard for Home Electronic Systems (HES).

The Open System Interconnect (OSI) model is being used as a guiding principle to provide a common set of rules for the development of communications standards. This model was developed by ISO/IEC to serve as a framework for comprehensive and flexible communications for electronic devices. The model consists of seven layers. For each layer, a protocol is defined. The model does not dictate how the layers are to be implemented, it defines **what** is required, not **how** it will be done.

The biggest benefit of the OSI model is that it can make the communications process independent of the message. That is, the message being sent is totally independent of the means by which it is sent.

The seven layers of the OSI model are designed to be largely independent of each other. The layers are:

* **Physical Layer:** Electrical and mechanical parts of communications are defined.

* **Data Layer:** The language of messages is defined.
* **Network Layer:** The process for messages conversion is defined.
* **Transport Layer:** This protocol defines a process by which messages can be transmitted between locations.
* **Session Layer:** The communications process for message translation is defined.
* **Presentation Layer:** The words used to express messages are defined.
* **Application Layer:** The means by which products interact with each other are defined. This layer is responsible for providing an applications language.

The arrival of standards for home electronic systems will result in a new generation of home automation products. One key advantage of these new products will be their ability to interface with each other, allowing for communication between the products of different manufacturers. This function is known as interoperability.

2.3 Requirements for an Application Language

To achieve the goals of interoperation, a framework for comprehensive and flexible application languages is required. Software applications will be written to provide:

* Product-to-product control and communication
* User-to-product control and communication
* Controller-to-product control and communication.

An applications language will allow manufacturers to develop sophisticated home products. Manufacturers will be able to write software applications allowing products from various manufacturers to work together. This will provide an environment so that appliances attached to home communication networks can communicate with each other.

Requirements for an application language for home automation applications include the following capabilities and funtionalities:

* The language must be flexible, and easy to use and implement.

* The language structure must allow source code and data to be easily maintained and modified.

* The language must be able to represent the physical description (data) and the behaviour (functions) of the controls for home products.

* The language must support present and future home automation applications (see Section 2.1).

3.0 OBJECT-ORIENTED METHODOLOGY

3.1 Introduction

3.1.1 Methodologies For Software Development

Methodologies for software development came into being in the 1970s. Two events responsible for this were the arrival of third generation languages and the development of large software projects (military and space programmes).

By the early 1980s, three methodologies were developed (Harmon, 1992):

* Structured Methodology (SM): In this methodology, a program is divided into a number of processes with each process containing input, function and output. This methodology is also known as functional decomposition.
* Entity Relationship (ER): This methodology was developed to deal with database applications. In this methodology, database entities (data structure) are analysed.
* State-Transition Model: This methodology was designed for engineering systems and found to be useful in real-time applications. In this methodology, the focuses are on events and the responses.

By the late 1980s, the SM methodology was enhanced to include the ER and state-transition models. This enhanced SM methodology was accepted and used as a basis for software development for large projects.

Enhanced SM methodology is process-oriented and step-by-step. The SM analysis phase is about what an application will be doing. The SM design phase is about how the application will be implemented.

The automation of software development methodologies has been attempted by many software companies. The end products are known as CASE (Computer-Aided Software Engineering) tools.

Over the past few years, the popularity of object-oriented programming languages has started a new trend in software development methodology. Objects are being increasingly used in all phases of software development:

* Object-oriented analysis (CASE tools)
* Object-oriented design (CASE tools)
* Object programming (C++, Smalltalk, etc.)
* Object storage and sharing (OODBMS e.g., ObjectStore).

3.1.2 The Lifecycle of Software Development

The lifecycle of software development consists of four phases. Each phase has its own goals and outputs. The four phases are (Booch, 1991):

* Analysis: Analysis of the problem.
* Design: Creating a solution based on the analysis.
* Evolution: A series of incremental prototypes that lead to the final implementation.
* Modification: The process of maintaining and enhancing the final prototype to meet users demands.

The analysis phase begins with an attempt to understand the problem. This is normally done by having discussions with the client. The early product of this phase is a statement of the problem describing what the system is trying to achieve. From this description, a model of the problem is constructed and communicated to the client.

Large and complex problems must be broken down into a number of subproblems.

The design phase is the process by which the model(s) of the problem are solved. It begins by studying the model with the aim of developing mechanisms which will provide the foundation for the implementation of that system.

The evolution phase of a system is performed by programmers. The process involves the incremental production of a series of prototypes by coding and testing each prototype separately. These prototypes will evolve into a final system.

The modification phase involves changes to the program ranging from fixing bugs to adding new features to the system.

3.1.3 Conventional and Object-Oriented Programming Languages

There are three fundamental aspects of software programming: data, process and control. Data is an attribute which is operated on by the process (algorithms, operations, rules etc.). Control is the condition from which the process will be performed.

In a conventional programming language (e.g., FORTRAN, C, Basic COBOL, etc.) the task of a programmer is to firstly define data structures, then write a set of procedures to act on the data structures, and lastly link the procedures and the input data together to form an executable program.

The characteristics of the conventional approach are that:

* Fundamental building blocks are built using algorithms.
* The approach is procedure oriented.
* One is modelling a mathematical process.
* The emphasis is on the procedures, not the data structures.
* The program execution is a sequence of procedures acting on data.

Object-oriented programming deviates markedly from conventional programming. In an object-oriented programming language (e.g., Smalltalk, C++ etc.) the task of a programmer is to define objects then write a set of messages for the objects, i.e., messages operate on objects. Computer programs are collections of discrete objects and messages.

The nature of the object-oriented model is that:

* Objects are used as fundamental building blocks of a program.
* Objects are constructed to represent physical models of the real world.
* There is equal emphasis on the data and procedures.
* Program execution leads to a physical model simulating the real world.

Object-oriented programs are often smaller than conventional programs. Wybolt (1990) reported that reimplementing a C program in C++ reduced the code by about 4-5 times. This implies that it requires less time to develop and maintain. Coupling this with the ability to reuse both code and designs will lead to reduction in software development cost; reduction in software maintenance cost; and better response to user requirements.

This is important as the costs of software development and maintenance have not declined at the same rate as hardware costs. Due to reusability, however, software products could become much more cost-effective.

Object-oriented programming has been used for the last 25 years. During the last few years, the combination of affordable hardware and the availability of comprehensive object-oriented tools has resulted in global acceptance of the technology. The brief history of object-oriented programming is shown below (Sigs Publications, 1992):

Year	Event
1967	**Simular-67 was developed**
1972	**Smalltalk-72 was developed**
1980	**Smalltalk-80 was released**
1983	**Object-oriented Pascal was developed**
1984	**Objective-C was created**
1985	**C++ was created**
1989	**Object Management Group was founded**

3.2 Fundamental Concepts

3.2.1 Class and Object

An object has a range of abstractions which describe essential features distinguishing it from other objects. An object is embedded with data (attributes) and methods (operations). An attribute is an inherent characteristic of an object. An attribute normally has a name, type of data, and default value associated with it. A method is a procedure or function which is invoked when the object receives a message under the name of that method. A method performs a number of functions including:

* Manipulating the object's attributes (in response to a command).
* Causing another message to be sent to other objects (in response to a command).
* Outputting data (in response to a query).

Each object has state, behaviour, and identity.

A class is a set of objects which share a common structure and a common behaviour. Classes are related to one another via inheritance relationships (Booch, 1991). An instance of a class is an object. All instances of a class have the same methods. For example, a number of polygon objects (e.g., a Square, a Triangle, a Rectangle) can be abstracted into a single class called Polygon as described below:

```
Class Name:   Polygon
Attributes:   Vertices
              Colour
Operations:   Draw
              Move
```

Table 1 shows another example of classes and objects. Classes of Animal, Mammal and Rabbit are created to describe general characteristics of any animal, any mammal and any rabbit. Object Bunny is used to describe a specific rabbit at any given instance. Note that an instance has real values. These are known as **instance variables.**

The word object has been used to describe both class and instance. Generally, the context of use defines which meaning applies.

In object-oriented applications, an application domain is modelled as one or more objects. The active properties of the physical process are modelled as states of objects. An object has a state associated with it at any point in time. States are altered by internal methods upon receiving messages.

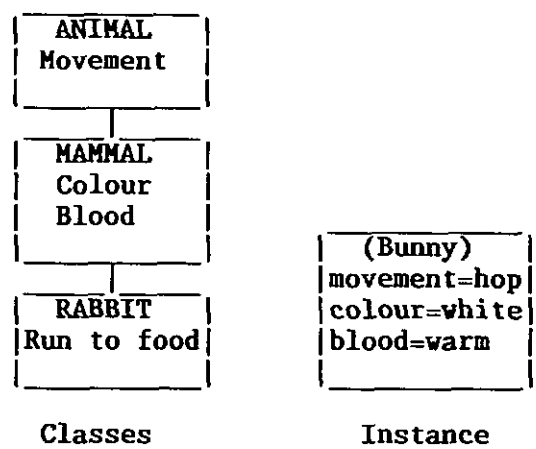


Table 1 Objects Hierarchy

3.2.2 Messages

A message is a means of communication between objects. It is a foundation for object-oriented programming (OOP). It starts an operation by invoking an object's methods. An interesting feature of OOP is the ability of the same message to produce different operations depending the type of the objects it is sent to. For example the message **print** will produce different outputs for the first object which is a matrix printer and for the second object which is a laser printer.

3.2.3 Other Features of The Object Model

Object-oriented programming allows computer programmers to better represent real-world objects in a computer. This capability enables programmers to write programs in a more natural manner. Other major features are discussed below:

Abstraction. An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects (Booch, 1991). Abstractions focus on the external view of objects. In C++, for example, this external view can be implemented by using the keyword 'public'. An abstraction makes it easier to understand an object since it does not show the details of the object.

Encapsulation (Information Hiding). Think of an object as a capsule which contains its data and procedures. The data can only be manipulated by the procedures within the object. The input to an object will be through a program sending messages to it. The outputs will be the changes in its data values.

Encapsulation hides the internal view of an object. In C++, for example, this internal view can be implemented by using the keyword 'private'. Encapsulation means that data and methods are hidden within its class structure. Encapsulation reduces accidental access to data.

Reuse Of Class Libraries. This is one of the most significant benefits of object-oriented programming. Being able to reuse the code will make software development much more productive, allowing programmers to develop, maintain and enhance programs in much less time. Once objects are developed, all a programmer needs to do is to decide what messages will be sent to which object.

The successful reuse of class libraries depends on the analysis, design, and implementation of the classes. An uncontrolled, arbitrary design of classes may result in them not being reusable. An example of non-reusable classes is when they are based on deep and complex hierarchies. Flatter hierarchies will enable economical software modifications.

Polymorphic Messages. 'Polymorphism' is a Greek word meaning 'having many shapes'. In this context it is the ability for the same message to produce different operations depending on the type of object it is sent to. This may prove useful for home automation applications as it enables the same message to perform various operations for different applications.

Inheritance. A class inherits data and methods from its parent classes (superclasses). Table 1 shows the principle of inheritance hierarchy. The hierarchy begins with class Animal which contains data (movement). Class Mammal, a sub class of class Animal, inherits all characteristics from class Animal. Thus, class Mammal contains data (movement, colour and blood). Class Rabbit, which is a sub class of class Mammal, inherits all characteristics from Animal and Mammal. Thus, Rabbit contains data (movement, colour and blood) and methods (e.g., run from loud noise).

Bunny, an instance of class Rabbit, inherits all characteristics from Animal, Mammal and Rabbit. Thus, object Bunny contains data (movement, colour and blood) and method (run from loud noise). However, unlike the classes above it, the attributes of object Bunny have been given values.

3.3 Booch's Methodology

3.3.1 Introduction

According to Booch (1991) "the process of object-oriented analysis (OOA) and object-oriented design (OOD) is neither top-down nor bottom-up, but round-trip gestalt. The process of the round trip gestalt is best described as incremental and iterative." This concept of round trip gestalt is the foundation of analysis and design. The products of this process are: classes and objects, their states, behaviours, and relationships. The activities of analysis and design are shown in Table 2. As shown, although each analysis and design involves the same activities, each views the activities differently.

3.3.2 Identifying Classes and Objects

This activity aims to discover the classes and objects that form the vocabulary of the problem domain. The starting point is looking at what the system is trying to achieve; in other words, the function of the system. Classes and objects which form the domain problem are named and their operations noted. This is sometimes known as domain analysis.

The process involves talking to both users and domain experts to understand the nature of the problem. Once the problem is fully understood, classes and objects can then be named through the process of abstraction.

3.3.3 Identifying the Semantics of Classes and Objects

During this activity, the meaning of the classes and objects will emerge. This is normally done by looking at each object in detail and writing down all that is known about the object. These are classified into object attributes and object functions. Normally, both static and dynamic attributes and functions are noted. The main outputs from this activity are class specifications. A class specification lists the essential aspects of a class in text format.

This takes longer than identifying classes and objects and may involve iterations. Classes and objects may be refined during this activity.

3.3.4 Identifying the Relationships Between Classes and Between Objects

In this activity, one needs to find out how objects and classes relate to each other and how they work together to solve the problem. Two questions to be answered during this activity are:

* How do these objects relate?
* What messages are sent between these objects?

The design phase of this activity may involve prototyping of various models. Again, classes and objects will be refined during this activity.

3.3.5 Implementing Classes and Objects

This activity is for the design phase only. The activity looks at how classes and objects will be represented using an OOP. At this stage, specifications of classes and objects will be sought.

At this point the iterative process is used by going back to the first activity and looping back and forth. That is, the design process is repeated; however, it now focuses on the internal structures (low-level abstractions) of classes and objects.

3.3.6 Booch Notation

Booch notation consists of (Booch, 1991; Vilot, 1990):

* Class diagram
* Category diagram
* State-transition diagram
* Object diagram
* Timing diagram
* Module diagram
* Process diagram.

These diagrams express the logical and physical views as well as the static and dynamic semantics of the system. Table 2 illustrates class, object, state transition, timing diagrams, and relationships between classes.

A class diagram shows classes and their relationships. In the initial stage of OOA, at a high-level of abstractions, class diagrams are drawn to capture the problem domain. At this stage, class specifications are not yet developed. A class diagram shows how classes interact with one another.

Relationships between classes are shown on the class diagram by various lines connecting class icons. A number of relationships can occur between classes. The most important ones are:

* Inheritance (kind-of) relationship: a class inherits data and operations from its parent classes.
* Composition (part-of) relationship: A connection between a pair of classes of different hierarchy.
* Instantiation relationship: A connection between a class and its object.

A state-transition diagram shows the dynamic behaviour associated with classes. The diagrams show the state space of a class, a transition from one state to another.

An object diagram shows an instance of a class and object messages. The purpose of an object diagram is to show the dynamics of the system.

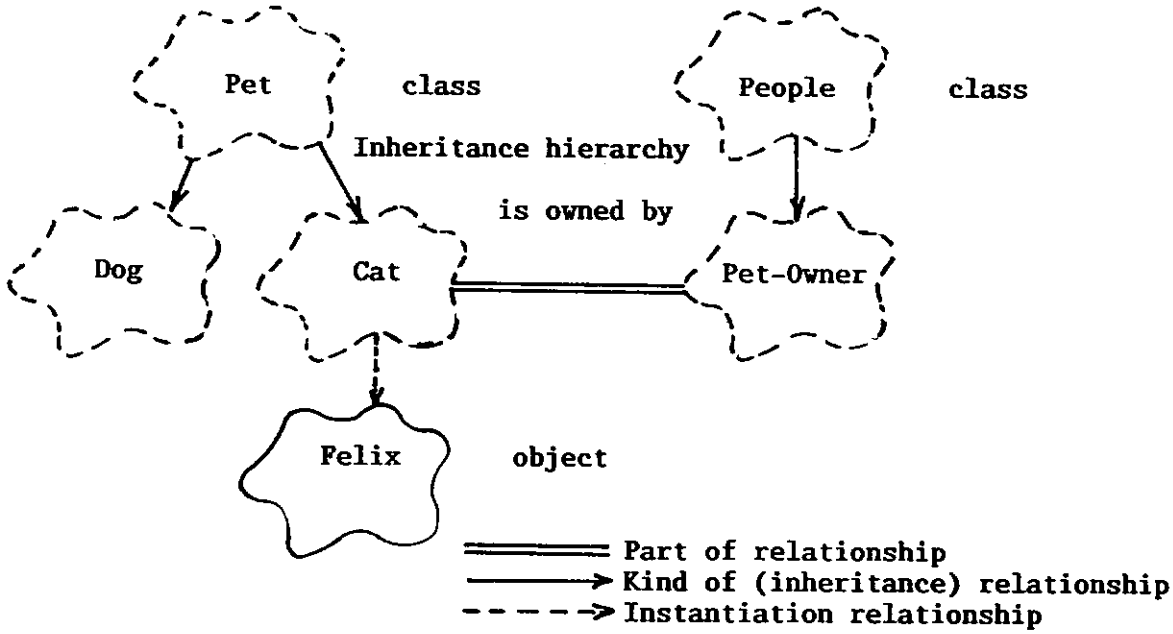
A timing diagram complements an object diagram with information about events and timing of operations of the objects.

ANALYSIS AND DESIGN

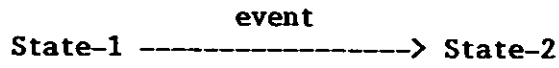
1. Identifying the classes and objects at a given level of abstraction.
2. Identifying the semantics of these classes and objects.
3. Identifying the relationships among these classes and objects.

DESIGN

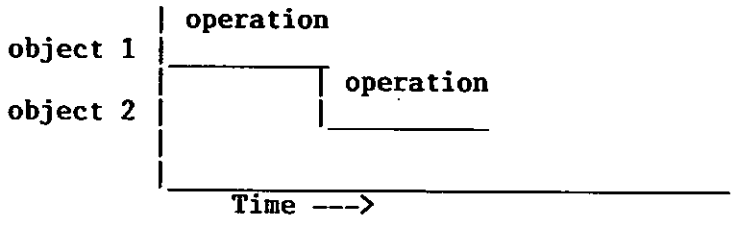
1. Implementing these classes and objects.



CLASS AND OBJECT DIAGRAM



STATE TRANSITION DIAGRAM



TIMING DIAGRAM

Table 2 BOOCH'S METHODOLOGY (Booch, 1991)

3.4 Rumbaugh's Methodology

Rumbaugh's Object Modelling Technique (OMT) (Rumbaugh et al., 1991) is probably the most comprehensive methodology proposed. Essential to the OMT concept is the meaning of "models". A model is defined as an abstraction (essence) of something which is created to understand a problem before implementing a solution. Often, models are built to provide visualisation as a means of communicating with customers.

The OMT methodology uses three views of a system (object model, dynamic model and function model) to model an application domain. The methodology involves four stages of development (analysis, system design, object design and implementation). Thus, the OMT methodology has two dimensions: view and stage of development.

The object model is the most important of the OMT views. The object model provides a means of decomposing a problem into classes of objects. The object model provides the following outputs:

* Object classes
* Attributes and operations of object classes
* Object diagrams
* Relationships between classes.

An object diagram is a graphic notation for representing instances, classes and their relationships. Object class diagrams are used to describe the model of the problem. Object instance diagrams are used to provide examples of the objects in actions.

A graphic notation of an object diagram is shown in Table 3. A class is represented by a rectangle containing three regions: class name, class attributes and class operations. Class and instance notations are shown in Table 3. BRANZ is an instance of class Company and AHD is an instance of class Staff.

An association is a relationship between classes. A link is a relationship between instances. Thus, a link can be seen as an instance of an association. An association can be one-to-one, one-to-many, or many-to-many. An example of a one-to-many association and a link is shown Table 3.

ANALYSIS:

1. Problem Statement
2. Object Modelling
   - Identify objects and classes
   - Prepare data dictionary
   - Identify associations
   - Identify attributes and links
   - Organise and simplify classes via inheritance
   - Verify that access paths exist for queries
   - Iterate and refine the model
3. Dynamic Modelling
   - Identify events between objects
   - Build a state diagram
   - Match events between objects to verify consistency
4. Functional Modelling
   - Identify input and output values
   - Build data flow diagrams showing functional dependency
   - Describe functions
   - Identify constraints

DESIGN:

1. Organise system into subsystems
2. Identify concurrency inherent in the problem
3. Allocate subsystems to processors and tasks
4. Choose approach for management of data stores
5. Handle access to global resources
6. Choose the implementation of control in software
7. Handle boundary conditions

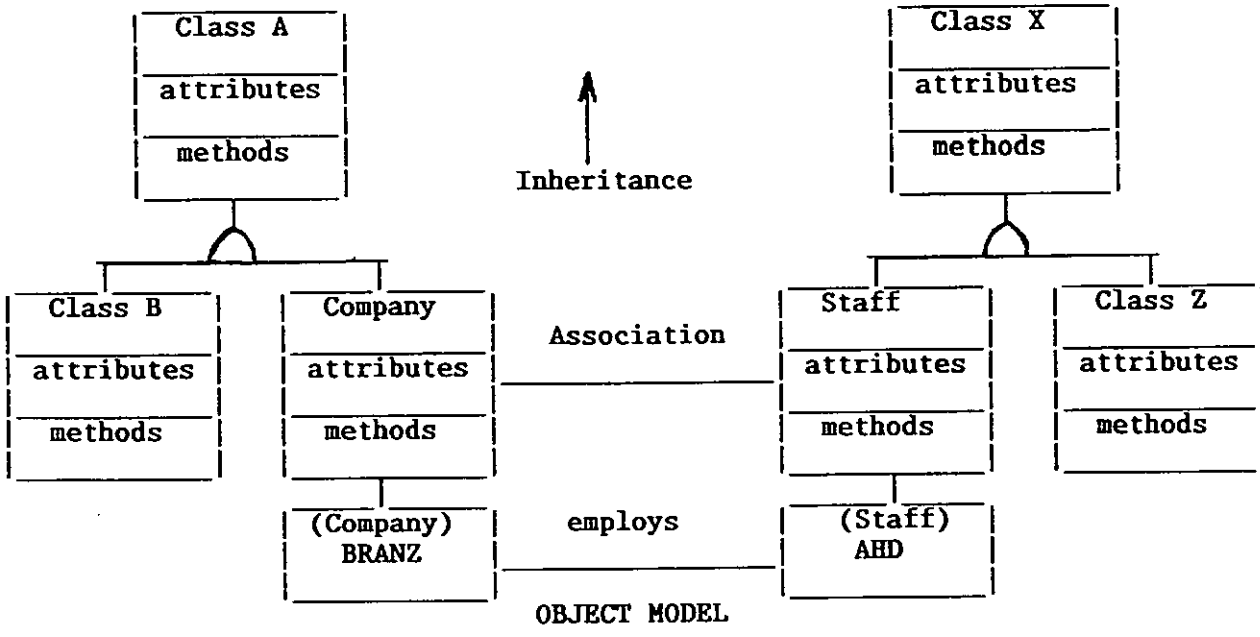


Table 3 RUMBAUGH' S METHODOLOGY (Rumbaugh et al., 1991)

3.5 Summary

3.5.1 Basis of Object-Oriented Methodology

Object-oriented methodology (OOM) applies the concept of looking at the world in terms of objects, to the life-cycle of the development of a computer program. OOM is independent of language. An OOM contains both static information (classes, instances, their relationships) and dynamic information (instances, messages), as shown in Table 4.

Object-oriented analysis (OOA) is the process of identifying classes and objects that form the problem domain. Object-oriented design (OOD) is the process of creating classes and objects and implementing them using an object-oriented programming language. The outputs of OOM include specification, documentation and models of classes and objects and their states, behaviours, and relationships which are used by programmers to implement a solution.

3.5.2 Comparisons of Booch and Rumbaugh Methodologies

The methodologies are similar at the concept level in that they agree on the meaning of OOA and OOD.

The activities of Booch's methodology are the same for both OOA and OOD. However, each views the activities differently. OOA is more concerned with high-level abstractions, whereas OOD is more about low-level abstractions. The activities are not top-down nor bottom-up, but round-trip gestalt; that is, incremental and iterative development.

The OOA and OOD activities of Rumbaugh's methodologies are quite different. OOA is concerned with modelling of the problem, whereas OOD stresses the implementation. The activities can be regarded as a top-down approach.

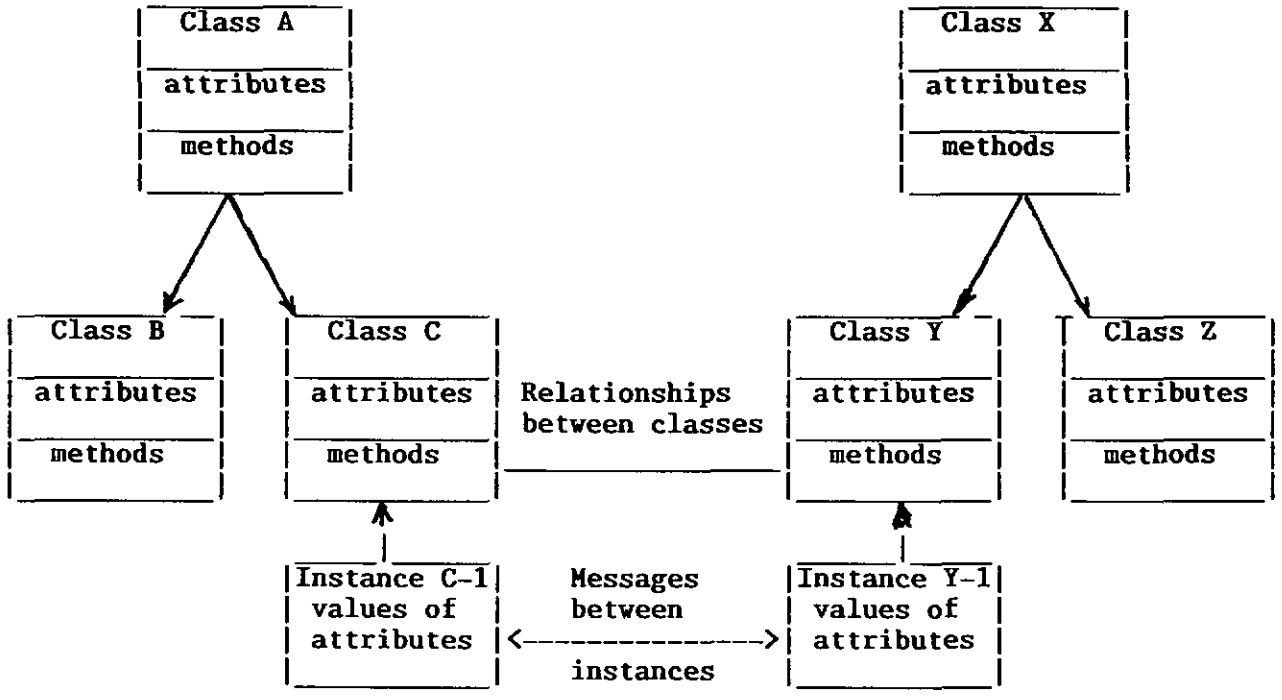


Table 4 Basis of An Object-Oriented Methodology

4.0 APPLICATIONS LANGUAGES FOR HOME AUTOMATION

This section reviews the use of object-oriented technology by two leading home automation systems, Smart House and CEBus, in their implementation of their applications languages.

The application language of Esprit Home System (Home Systems Specification, 1991) is similar to that of the CEBus. Other systems such as D2B and the Japanese HBS (Parks Associates, 1990) are using command tables in their implementation of the applications languages.

It should be noted that these protocols are still evolving and that the two protocols reviewed here should provide a good coverage for the use of object-oriented technology in home automation applications.

ISO/IEC/JTC1 is considering applications languages from a number of home automation systems, including: Batibus (France), CEBus (USA), D2B (The Netherlands), ESPRIT HS (Europe), EIBus (Germany), HBS (Japan), and Smart House (USA). It is expected that evaluation will not be completed until late 1994.

4.1 Smart House Applications Language

4.1.1 Background

Smart House is a new system for the distribution and control of residential energy and communications that enables centralised control of: security, safety, energy management, entertainment, lighting control, and communications (Dechapunya, 1992).

The Smart House applications language plays a key role in the implementation of integrated applications in a Smart House. The applications language allows manufacturers to develop home automation applications by writing software applications which will allow different products from various manufacturers to work together. The language is used for end-to-end communications between appliances.

The applications language used in the Smart House system is object-oriented (Smart House Applications Language Guide, 1992). The Smart House applications language is embedded in home appliances. These appliances, attached to Smart House networks, communicate by sending messages to one another. Appliances can also interact with the system controller.

The environment of the Smart House applications language contains (Smart House Applications Language Guide, 1992):

* Smart House Object Hierarchy,
* Smart House Object Specifications,
* Smart House Applications Function Tables.

4.1.2 Language Environment

Smart House Object Hierarchy. The Smart House object hierarchy is shown in Table 5. It illustrates the hierarchial structure of Smart House objects used in home automation applications. The inheritance is based on a single hierarchy of classes, where all classes have a common ancestor class named object. Dearle (1990) discusses the advantages and disadvantages of single hierarchy and multiple peer hierarchy.

The class Object is a common ancestor class (root class). It is used to provide common attributes and messages for all other instances of objects to inherit.

Smart House Object Specifications. The attributes and operations of Smart House objects are described in the Smart House Object Specifications. Each object specification contains:

* General section,
* Instance Variables section,
* Methods section.

An example of Smart House object specification for object Control is shown below:

GENERAL	
Class Name:	**Control**
INSTANCE VARIABLES	
controlState	**The state of control. The state can be inactive, waiting, or busy.**
METHODS	
reset	**Resets the state of control to "waiting"**
getControlState	**Replies with the state of the control**

The Instance Variables section lists the instance variables of an object. There are two categories of instance variables: static and dynamic. Static variables are fixed by manufacturers. Dynamic variables are subjected to change due to messages received.

The Methods section lists the operations of an object. The methods perform operations on the instance variables of the object. There are two categories of operations: command and query.

Smart House Applications Function Tables. An application is implemented by modelling appliances as units. Each unit consists of one or more panels. The function of each specific applications panel is listed in the Smart House Applications Function Tables.

Developing Applications. Once the model of an application is finalised, the next step is for a product engineer to code messages (command/query) for passing between the objects to perform the required functions.

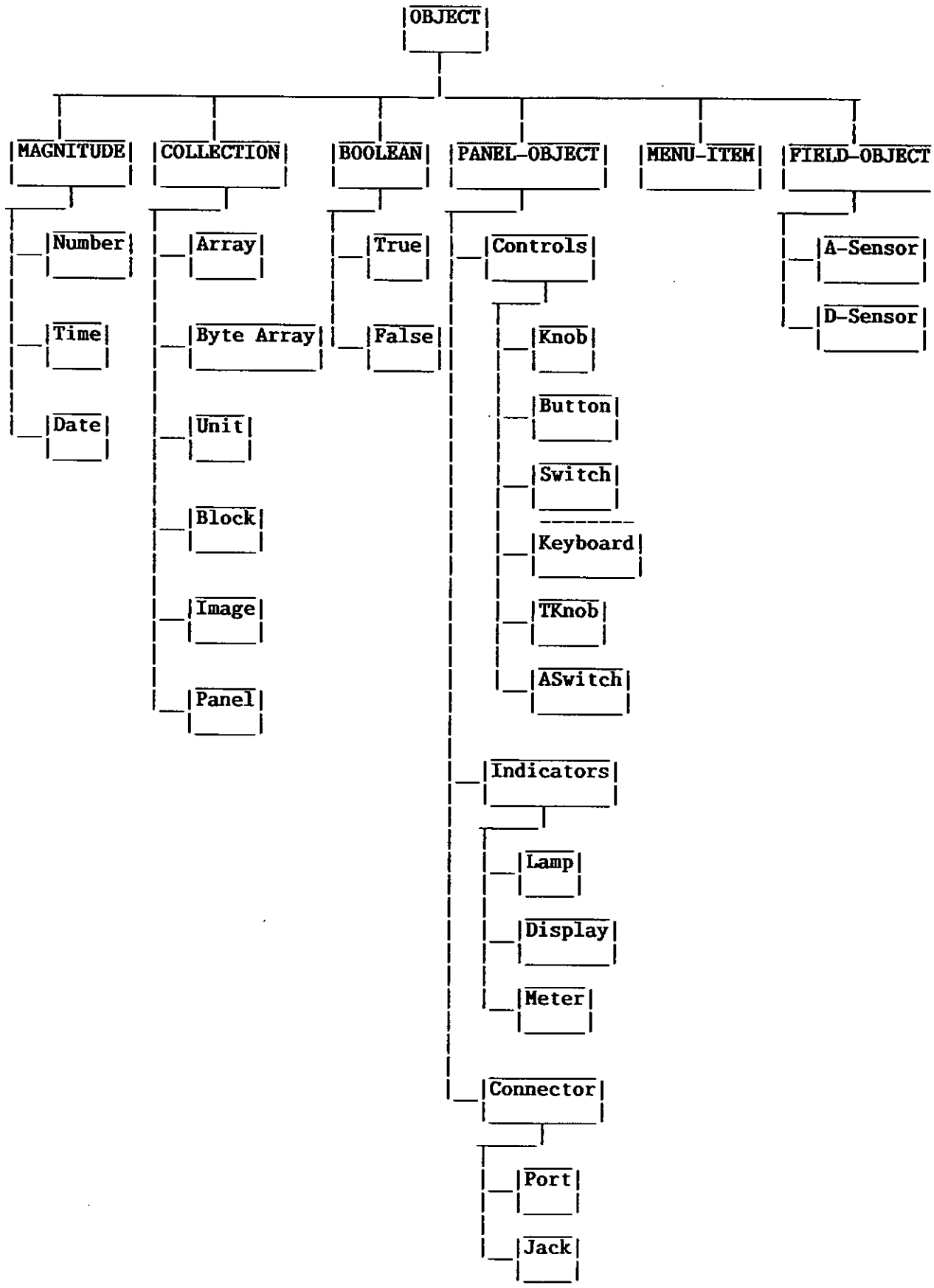


Table 5 Smart House Applications Language Object Tree
(Smart House Applications Language Guide, 1992)

4.2 Common Application Language

4.2.1 Background

CAL (Common Application Language) is an applications language of a CEBus home automation standard (Dechapunya, 1992).

CEBus is a local area network for the home. CEBus allows two-way communication between user and appliance, and between appliance and appliance. The standard is designed to be a comprehensive protocol and language for a number of devices. CEBus products will be able to talk to, and control one another. Products can be classified into system products and user products. System products are those which are not visible to the users. An example of a system product is a router. A router connects various physical media together, to allow a product connected to one medium to communicate to another product on another communications medium. A router collects data on one medium and puts the data onto another medium.

4.2.2 Language Environment

CAL is a language for interproduct communication. It is used by one CEBus device to communicate with another CEBus device. Its architecture is based on the Application layer of the ISO/OSI (Section 2.1) network model. The committee that developed CAL is the biggest CEBus committee. The December 1990 version of CAL has been adopted as an interim standard. The language will require continuous updating as new products and new applications come onto the market.

The design of CAL is based on the object-oriented paradigm. It contains syntax and vocabulary to allow devices to process and communicate commands and status with one another. It is designed to be broad and flexible, allowing products engineers many command options.

CEBus states that its CAL is object-oriented, but features such as inheritance, which is an essential part of an object-oriented language, are not in CAL.

According to the structure of CAL (Electronic Industry Association, 1990), a home device has a single network address and contains one or more contexts. A context is made up of various home automation objects. For example, a TV set is modelled as a device which contains an Audio Process context, and a Video Monitor context. The Audio Process context is made up of various objects such as a Button, or a Switch for controlling the operation of the TV. An object is modelled as collections of one or more variables that can be accessed (read or modified) by a device on the network.

Definitions of contexts, objects, and methods are listed in Table 6. These tables are presently maintained and updated by the Electronic Industry Association (EIA). EIA, however, is in the process of suggesting a new organisation to take over the CEBus home automation standard.

The manipulation of the objects' variables using CAL language is the basis of the CEBus home automation.

The message syntax of CAL is "context object method argument". For example, CAL's message to tell a telephone object to dial 235-7600 would be **"telephone dialer load 235-7600"**.

Each appliance is a node on the CEBus network. Appliances communicate by sending CAL messages to one another. The CAL messages are bundled together to form a packet. Packets travel from one communications medium to another medium using links created by CEBus bridges.

CAL CONTEXT

Audio Process	Time Service Elements	Environmental System
Audio Source	Appliance Control System	Environmental Sensors
Audio Record	Communication System	Air Conditioning System
Video Monitor	Answering Machine	Heating System
Video Source	Intercom	Information System
Video Record	Telephone	Security System
Tuning System		Lighting System

CAL OBJECTS OF TELEPHONE CONTEXT

Switch hook	Receiver (earpiece)
Transmitter (mouthpiece)	Speaker phone
Dialer	Ringer volume control
Keypad	

CAL METHODS

False	Store	Return	Build
True	Swap	Exit	Define
Test	And	Repeat	Do_While
Compare	Or	Next	Dump
Add	Xor	Previous	For
Subtract	Not	Position	If_Then
Load	Branch	X-Translate	Switch
Load_Number	Jump	Y-Translate	While_do
Load_Character	Call	Z-Translate	

USE OF OBJECT DIALER OF TELEPHONE CONTEXT

Method Name	Argument	Action
False		Disable dialing
True		Enable dialing
Load_character	<Phone Number>	Dial <Phone Number>

Table 6 Structure of Common Application Language
(Electronic Industry Association, 1990)

The physical paths for carrying control and communication messages are: Powerline (PLBus); Twisted Pair (TPBus); Coaxial cable (CXBus); Infrared (IRBus); Radio Frequency (RFBus); Audio/video system (AVBus); Fibre optics cable (FOBus).

CAL is an open protocol, Once the CEBus standard is adopted, manufacturers will be able to build products which incorporate CAL to allow them to understand messages or instructions sent by other CAL compatible products.

4.3 Summary

There are strong similarities between Smart House and CAL as shown below:

	Smart House Application Language	Common Application Language
Appliance	Unit	Device
Collection of objects	Panel	Context
Objects	Button, Switch, Knob etc	Button, Switch, Knob etc
Message Syntax	object function messages	context object method

5.0 CLASS SPECIFICATIONS FOR HOME AUTOMATION

5.1 The Basis

The aim of this chapter is to produce an object-oriented model for a basic home automation unit. Analysis and design of the problem domain is attempted using Booch's methodology. The methodology is not used to its full potential since the aim of this part of the work is only to produce hard copy of class descriptions for modelling home automation. Booch's methodology is chosen because it is gaining popularity (Horstmann, 1993) and it provides a more natural means, incremental and iterative development, of solving a problem. Rambaugh's methodology is designed for solving large and complex tasks such as space programs or battle control.

Smart House object specifications are used to provide a guideline for common objects for the home automation domain.

C++ programming language is used as an early attempt to implement the model.

5.2 Application Domain

The application domain selected is a water heater. The function of a water heater is to manage hot water for a home by implementing the control and automation of various devices. To perform these functions, the water heater must be able to communicate with other home automation units as well as the householders. Essential elements required to achieve this are:

* **Language:** This is commonly known as an applications language.
* **Control Unit:** The control unit contains necessary devices to allow communication to occur. These devices or objects include controllers and indicators.

5.3 Identifying Classes and Objects

In order to discover the classes and objects that form the vocabulary of the water heating domain, we first consider the function of the system. The list of tasks and devices which are used to perform the tasks is shown in Table 7.

Tasks	Devices
General management of hot water supply	Unit
Panel containing control and indicator devices	Panel
Power control (on/off)	Button
Thermostat setting	Knob
Information display	Display
Fault indicator	Lamp

Table 7 Devices for Modelling a Water Heater

Thus, a water heater can be modelled as a unit which contains devices which are used for monitoring and controlling the hot water supply. The Unit, Panel, Button, Knob, Lamp, and Display can be modelled as classes.

5.4 Specifying Attributes and Methods

The class specifications of the water heater domain are listed in Table 8.

Class Name:	WaterHeaterUnit
Function:	Management of a hot water system
Attributes:	- model number - unit function - unit status - unit address
Operations:	- get model number - get unit function - get unit status
Used by:	None
Contains:	WaterHeaterPanel, Display
Inherited by:	None
Derived from:	None

Class Name:	WaterHeaterPanel
Function:	Panel for control and indicator devices
Attributes:	- panel function - panel state - default panel state
Operations:	- get panel function - get panel state - enable panel - disable panel
Used by:	WaterHeaterUnit
Contains:	Button, Knob, Lamp
Inherited by:	None
Derived from:	None

Class Name:	Control
Function:	Group all classes which provide control
Attributes:	- control state
Operations:	- get control state
Used by:	WaterHeaterPanel
Contains:	None
Inherited by:	Button, Knob
Derived from:	None

Class Name:	Indicator
Function:	Group all classes which provide messages and display
Attributes:	- indicator status
Operations:	- get indicator status
Used by:	WaterHeaterPanel
Contains:	None
Inherited by:	Lamp, Display
Derived from:	None

Table 8 Class Specifications (continued on next page)

Class Name:	Button
Function:	Provide on/off control
Attributes:	- default button position - current button position
Operations:	- get default button position - get current button position - release button - press button
Used by:	WaterHeaterPanel
Contains:	None
Inherited by:	None
Derived from:	Control

Class Name:	Knob
Function:	Used for setting thermostat
Attributes:	- default knob setting - current knob setting - minimum value - maximum value
Operations:	- get current knob setting - set current knob setting to - show minimum value - show maximum value
Used by:	WaterHeaterPanel
Contains:	None
Inherited by:	None
Derived from:	Control

Class Name:	Display
Function:	Display information
Attributes:	- message - file name
Operations:	- display message - display contents of a file
Used by:	WaterHeaterUnit
Contains:	None
Inherited by:	None
Derived from:	Indicator

Class Name:	Lamp
Function:	Provide fault indicator
Attributes:	- indicator status
Operations:	- get indicator status
Used by:	WaterHeaterPanel
Contains:	None
Inherited by:	None
Derived from:	Indicator

Table 8 Class Specifications (continued from page 21)

5.5 Identifying Relationships Between Classes

In this activity, we see how classes are related to each other and what messages are used to drive these classes. Two pictures are drawn to illustrate part-of and kind-of relationships between these classes.

5.5.1 Kind-Of Relationships

Section 5.3 lists the classes which are used for controlling and monitoring. Kind-of relationships between various classes are shown in Table 9. Knob and Button classes are a kind-of Control class. Display and Lamp classes are a kind-of Indicator class. This relationship is also known as inheritance.

It can be said that a water heater unit is a type of unit and that a water heater panel is a type of panel. Another way of saying this is that a unit is a generic class and a water heater unit is an application-specific class.

Generic Classes	**Application Specific Classes**
Unit	**Water Heater Unit**
Panel	**Water Heater Panel**

5.5.2 Part-Of Relationships

The essential classes required for the control and automation of a water heater are:

* WaterHeaterUnit
* WaterHeaterPanel
* Control
* Indicator.

These form part-of relationships between various classes used for the water heater application. This is illustrated in Table 10. A WaterHeaterUnit is composed of a WaterHeaterPanel and a Display. The WaterHeaterPanel has a Knob object, a Button object, and a Lamp object. That is, each object is part-of WaterHeaterPanel.

Consider a class Knob. The following can be said about this class:

* Knob is PART-OF a class WaterHeaterPanel
* Knob INHERITS attributes and methods from Class Control.

WaterHeaterUnit and its composition is shown in Table 11.

Table 9 Kind-Of Relationships (Inheritance) Diagram

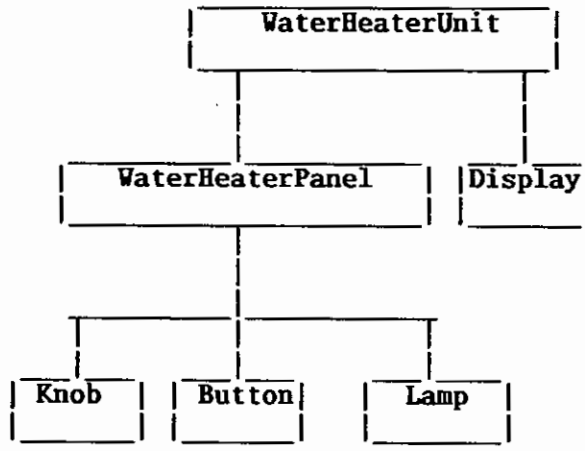


Table 10 Part-Of Relationships (Composition) Diagram

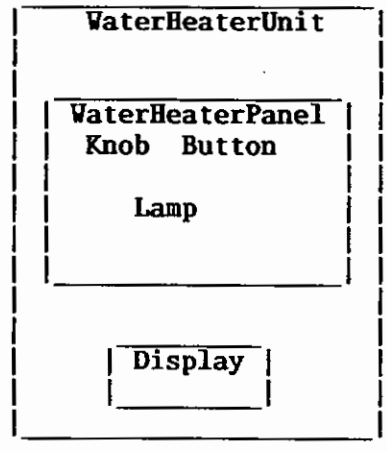


Table 11 WaterHeaterUnit and its Composition

5.6 Representation of Classes and Objects on a Computer

5.6.1 Object-Oriented Programming Language

This Section looks at how classes and objects and their relationships for the water heater domain will be represented on a computer using an object-oriented programming language. The process is iterative, involving the abstraction of classes and objects, and the messages used in the control and automation of the water heater.

An object-oriented programming language, C++, is used in the implementation. C++ (Atkinson and Atkinson, 1992, Borland C++ User's Guide, 1992, Borland C++ Tools and Utilities Guide, 1992, Borland C++ Programmer's Guide, 1992, Dearle, 1990, Vilot, 1990, Voss, 1991, Walker, 1992) was developed by Bjarne Stroustrup at AT and T between 1979 and early 1980.

5.6.2 Representing Classes, Objects and their Relationships

Representing Classes. Table 12 shows a representation of Knob as a class using C++.

```
// KNOB.CPP
// CLASS Knob DEFINITIONS

class Knob : public Control
{
private:
  static int  minValue;
  static int  maxValue;
  static int  range;
  int  stepSize;
  int  defaultKnobSetting;
  int  knobSetting;

public:
       Knob(int knobSetting1);
  void getMinValue();
  void getMaxValue();
  void getStepSize();
  void getCurrentSetting();
  void setStepSizeTo();
  void setCurrentSettingTo();
  void moveUp();
  void moveDown();
};

// FUNCTIONS DEFINITIONS

       etc...
```

Table 12 Representing Class Knob in C++

Class Knob is encapsulated with data and methods. The power of encapsulation of the object-oriented programming method can be shown by rewriting the codes in Table 12 using conventional programming methods. The program is shown below:

```
struct Knob
{
  static int  minValue;
  static int  maxValue;
  static int  range;
  int  stepSize;
  int  defaultKnobSetting;
  int  knobSetting;
};
     ... etc ...
void getMinValue() {...}
void getMaxValue() {...}
void getStepSize() {...}
void getCurrentSetting() {...}
```

The class identifier is replaced by a struct identifier. The content of struct can only be data. This means that the data of struct can be used by any methods anywhere in the program. This can lead to the problem of data integrity. In object-oriented programming, the content of class can be data as well as the methods that manipulate the data. That is, both data and their methods are encapsulated within the specific class structure. This results in clarity, reduction in complexity, modularity, and programming efficiency, which in turn bring about more economical development and maintenance of software lifecycles.

The private data (minValue, maxValue, range, stepSize, defaultKnobSetting, and knobSetting) are accessible only by the methods of class Knob (getMinValue, getMaxValue, getStepSize, getCurrentSetting, setStepSizeTo, setCurrentSettingTo, moveUp, and moveDown). For example, the method setCurrentSettingTo contains:

```
void  Knob::setCurrentSettingTo()
{
  char inchar[3];
  getCurrentSetting();
  Label1:;
  cout<<"\t\t Enter the setting [0-100]:> ";
  gets(inchar);
  if(atoi(inchar) < 0 || atoi(inchar) >= 100)
  {
    cout<<"\t\t The new setting is out of range\n";
    goto Label1;
  }
  knobSetting = atoi(inchar);
}
```

The above C++ representation shows that knobSetting, a private datum (using C++ private specifier) of class Knob, can only be manipulated by Knob's methods.

The interface to class Knob is via the public part of the class. Only method declarations are shown here. These method declarations can be thought as the abstractions of the class. The method definitions are the details of how the class Knob behaves.

Tables 13, 14, and 15 show class representations in C++ of Button, Lamp, and Display, respectively.

```
// BUTTON.CPP
// CLASS Button DEFINTION

class Button : public Control
{
private:
  int buttonPosition;
  int defaultButtonPosition;

public:
       Button(int buttonPosition1);
  void getDefaultPosition();
  void release();
  void press();
  void getButtonPosition();
};

// FUNCTION DEFINITIONS

        etc...
```

Table 13 Representing Class Button in C++

```
// LAMP.CPP
// CLASS Lamp DEFINITIONS

class Lamp : public Indicator
{
public:
  Lamp();
};

// FUNCTION DEFINITIONS

        etc...
```

Table 14 Representing Class Lamp in C++

```
// DISPLAY.CPP
// CLASS Display DEFINITIONS

class Display : public Indicator
{
public:
  Display();
  void messageReport(char fileName[12]);
  void messageDisplay(char message[22]);
};

// FUNCTION DEFINITIONS

        etc...
```

Table 15 Representing Class Display in C++

Representing Objects. An object is an instance of its class. Table 16 shows a few examples of home automation objects. As shown, WHUnitA is an instance of class WaterHeaterUnit. The main difference between WaterHeaterUnit and WHUnitA is that WaterHeaterUnit contains static structure of data and methods. WHUnitA contains dynamic structure with values given.

```
WaterHeaterUnit   WHUnitA("H1", disabled, released, 0);
WaterHeaterPanel  WHPanelA;
Button            ButtonA;
Knob              KnobA;
```

Table 16 Representing Objects in C++

Representing Inheritance Relationships. Table 9 shows kind-of relationships of base class Control with sub classes Knob and Button, and base class Indicator with sub classes Display and Lamp. These relationships between classes can be modelled in object-oriented programming languages by inheritance. The C++ representations of Control and Indicator are shown in Table 17A and Table 17B, respectively. Classes Knob, Button, Display and Lamp inherit from their parent classes by the use of "public" keyword as shown below:

```
class Knob    : public Control { ... };
class Button  : public Control { ... };

class Display : public Indicator   { ... };
class Lamp    : public Indicator   { ... };
```

```
// CONTROL.H
// (C) Copyright 1992 by AHD

#ifndef _CONTROL
#define _CONTROL
#include <iostream.h>

// CLASS Control DEFINITION

class Control
{
private:
  int controlState;

public:
       Control() { controlState = inactive; cout << "Control Initialised"; }
  virtual void reset() { controlState = inactive; }
  virtual void isControlInactive();
  virtual void isControlWaiting();
  virtual void isControlBusy();
  virtual void getControlState();
};

// FUNCTION DEFINITIONS

        ...

#endif
```

Table 17A Representing Inheritance in C++

```
// INDCT.H

#ifndef _INDCT
#define _INDCT

// CLASS Indicator DEFINITION

class Indicator
{
public:
  int indicatorStatus;

  Indicator();
  virtual void getIndicatorStatus();
};

// FUNCTION DEFINITIONS

        ...
#endif
```

Table 17B Representing Inheritance in C++

Representing Composition Relationships. The C++ representation of the WaterHeaterUnit is shown in Table 18. The WaterHeaterUnit is composed of objects WHPanelA and DisplayA. The class WaterHeaterUnit contains: data (unitAddress, modelNumber, unitFunction), objects (WHPanelA, DisplayA) and methods (getFunction, getModelNumber, getUnitStatus).

Table 10 shows the composition of WaterHeaterPanel. It shows that the WaterHeaterPanel contains various objects for controlling and indicating. This composition can be modelled in object-oriented programming languages by building a class which contains objects of other classes.

The C++ representations of the WaterHeaterPanel is shown in Table 19. As shown, the class WaterHeaterPanel contains: private data (panelFunction, defaultPanelState, panelState), private objects (ButtonA, KnobA, and LampA), public methods (reset, getPanelFunction, getPanelState, enablePanel, disablePanel). Because class WaterHeaterPanel contains objects ButtonA, KnobA, and LampA, it has automatic access to all of the methods of these objects.

```
// WATERHEATERUNIT.CPP
// CLASS WaterHeaterUnit DEFINITIONS

class WaterHeaterUnit
{
public:
  Panel    WHPanelA;
  Display  DisplayA;

  char * modelNumber;
  int    unitFunction;
  char * unitAddress;

  WaterHeaterUnit(char * unitAddress1,
    int panelStateA, int buttonPositionA, int knobSettingA);
  void getFunction();
  void getModelNumber();
  void getUnitStatus();
};

// FUNCTION DEFINITIONS

        etc...
```

Table 18 Representing Composition of WaterHeaterUnit in C++

```
// WATERHEATERPANEL.CPP
// (C) Copyright 1992

const int enabled  = 0;
const int disabled = 1;

// CLASS WaterHeaterPanel DEFINITION

class WaterHeaterPanel
{
private:
  int panelFunction;
  int defaultPanelState;
  int panelState;

  Button   ButtonA;
  Knob     KnobA;
  Lamp     LampA;

public:
       WaterHeaterPanel(int panelState1, int buttonPositionA, int knobSettingA);
  void reset() { panelState = defaultPanelState; }
  void getPanelFunction();
  void getPanelState();
  void enablePanel();
  void disablePanel();

//      Accessing ButtonA Object
  void getControlStateButton() { ButtonA.getControlState();}
  void getDefaultPosition() {ButtonA.getDefaultPosition();}
  void release() {ButtonA.release();}
  void press() {ButtonA.press();}
  void getButtonPosition() {ButtonA.getButtonPosition();}

//      Accessing KnobA Object
  void getControlStateKnob() { KnobA.getControlState();}
  void getMinValue() {KnobA.getMinValue();}
  void getMaxValue() {KnobA.getMaxValue();}
  void getStepSize() {KnobA.getStepSize();}
  void getCurrentSetting() {KnobA.getCurrentSetting();}
  void setStepSizeTo() {KnobA.setStepSizeTo();}
  void setCurrentSettingTo() {KnobA.setCurrentSettingTo();}
  void moveUp() {KnobA.moveUp();}
  void moveDown() {KnobA.moveDown();}

//      Accessing LampA Object
  void getIndicatorStatus() {LampA.getIndicatorStatus();}
};

// FUNCTION DEFINITIONS

       etc...
```

Table 19 Representing Composition of WaterHeaterPanel in C++

5.6.3 Representing Messages

In C++, the public functions can be used to access the private data which are encapsulated within the objects. The public functions are said to be the interfaces between a program and the objects. Thus, messages (command or inquiry) to the objects can be represented using these public functions.

The C++ mechanism for sending a message "messageToObject1" to "object1" is "object1.messageToObject1". For example, if it is required to set a thermostat setting for a water heater unit to 100, the steps involved would be:

* an object KnobA is created from a class Knob;
* a message setToPosition(100) is then sent to KnobA;
* the method setToPosition performs an operation on variable currentPosition;
* currentPosition is set to 100.

Table 20 shows examples of sending messages.

```
// MAIN.CPP

main()
{
// Create an object

   WaterheaterUnit WHUnitA("H1", disabled, released, 0);

// Get System Status

   WHUnitA.getFunction();
   WHUnitA.getModelNumber();
   WHUnitA.getUnitStatus();

// Turn On the Water Heater system

   WHUnitA.press();

// Turn Off the Water Heater system

   WHUnitA.release();

// Thermostat Setting

   WHUnitA.setCurrentSettingTo();
}
```

Table 20 Representing Messages in C++

5.7 Summary

Using C++ as an early prototype, a model of a water heater system has been developed using Booch's methodology. It is modelled as a WaterHeaterUnit containing a WaterHeaterPanel and a Display. The WaterheaterPanel contains a Button, a Knob, and a Lamp.

* The iterative process of doing bit-by-bit analysis, design and implementation is a suitable mechanism for modelling home automation objects.

* C++ shows promise as an object-oriented language for implementing a class library of a home automation domain.

* Understanding the concept of an object-oriented approach makes it easier to implement the problem using an object-oriented programming language.

* The power of C++ expressiveness means that, for a relatively small project, object-oriented design can be done using the C++ language.

6.0 CONCLUSIONS

6.1 Object-Oriented Approach

Object-oriented programming is a natural consequence of evolution in programming languages. The results are modularity, reusability, and programming efficiency in developing and maintaining software code.

The object-oriented approach provides a suitable means of representing the attributes and behaviours of the physical controls for home products. A problem is modelled as collections of co-operating objects. Each object is an instance of a class. Classes are related to one another via inheritance and composition.

This makes it a natural process to model home automation using an object-oriented paradigm.

The encapsulation of data and methods within an object enables the concept of interoperability to be easily implemented. An appliance can be modelled as an object with its properties and its interactions with the world encapsulated in the chip embedded within the appliance. Each chip can be implemented using different processors. The implementation of the application language can be tested locally by its developer, since the object is self-contained with its data structures and methods to operate on the data. Thus, an appliance from one manufacturer will be able to work with another appliance from another manufacturer. Each manufacturer does not need to know details of the internal representation of data and methods of each appliance. The only thing required is the interface (messages) for each appliance.

Formal OOA and OOD are designed for large and complex projects such as aerospace and military projects. For smaller projects, parts of OOA and OOD may only be required for software development. This is especially so if an expressive language such as C++ is used. In such cases, you may still need notation to provide roadmaps for the programs.

At present, no existing object-oriented methodologies are accepted as an industry standard. It is expected that these methodologies will undergo further experimentation and development. The starting point is to standardise the design notations (existing notations from various methodologies are complex and quite different).

6.2 Object-Oriented Programming Language

Before any attempt to learn an object-oriented language, the concept of the object-oriented paradigm should be understood first. That is, the programming language should be learned with an object-oriented viewpoint.

6.3 Further Work

This work is a feasibility study of object-oriented technology. It is clear that object-oriented technology provides a natural means for developing software for home automation applications. The work has produced high-level abstractions (names and functions) of classes for a water heater domain.

The output from research in 1993-1994 will include classes to provide a general framework and environment for home automation applications including: automation of appliances and lights, energy management, communication and information, entertainment, convenience, security and safety, and environmental control. Real-world applications for the next 3 years will include prototyping applications for control of appliances and lights, HVAC/energy management, and environmental monitoring and control.

REFERENCES

Atkinson, L., and Atkinson, M. 1992. Using Borland C++ 3 (2nd Edition). Que Corporation, Carmel, Indiana, pp. 1158.

Booch, G. 1991. Object-Oriented Design with Applications. Redwood City, California: Benjamin/Cummings. pp. 580.

Booch, G., and Vilot, M.J. 1992. The structure of C++ programs. C++ Report, October 1992, pp. 20-22.

Borland C++ User's Guide 1992. Borland International, Inc., Scotts Valley, California.

Borland C++ Tools and Utilities Guide 1992. Borland International, Inc., Scotts Valley, California.

Borland C++ Programmer's Guide 1992. Borland International, Inc., Scotts Valley, California.

Dearle, F. 1990. Designing Portable Application Frameworks For C++. The C++ Journal, Summer 1990, pp. 55-59.

Dechapunya, A.H. 1992. Standards For Communications Networks In The Home. Building Research Association of New Zealand, Miscellaneous Report, Judgeford, New Zealand.

Electronic Industries Association 1990. Draft Common Applications Languages (CAL) specification as of 18 December 1990, Washington, D.C.

Harmon, P., 1992. Object-Oriented Methodologies: Part 1, Object-Oriented Strategies, Volume 2, Number 4, California, pp 1-16.

Home Systems Specification, 1991. ESPRIT-HS Consortium, Eindhoven, The Netherlands.

Horstmann, C.S., 1993. Two leading object-oriented design tools. C++ Report, Vol. 5, No. 1, January 1993, pp. 62-67.

Parks Associates, 1990. An Overview of Worldwide Home Automation Standards, Parks Associates, Dallas, Texas.

Rumbaugh, J., Blaha, M., Premerlani, F.E., and Lorenson, W., 1991. Object-Oriented Modelling and Design. Prentice Hall, Englewood Cliffs, NJ, pp. 528.

Sigs Publications, 1992. Happy 25th Anniversary Objects. New York, NY.

Smart House Applications Language Guide, 1992. SH-1401, SMART HOUSE L.P. Upper Marlboro, MD.

Vilot, J.M. 1990. Using Object-Oriented Design and C++. The C++ Journal, Fall 1990, pp. 7-14.

Voss, G. 1991. Object-Oriented Programming: An Introduction. Osbourne Mcgraw-Hill, Berkeley, California.

Walker, G. 1992. Why the choice must be C++. The C++ Journal Vol. 2 No. 1 1992, pp. 52-65.

Wybolt, N. 1990. Experiences with C++ and object-oriented software development. 1990 Usenix C++ Conference, pp. 1-9.

GLOSSARY

Abstraction. The essential characteristics of an object that distinguish it from other objects.

Class. A set of objects which share a common structure and a common behaviour. An instance of a class is an object.

Encapsulation. The putting together of data structure and the methods within its class structure.

Inheritance. A relationship among classes. A class inherits data and methods from its parent class.

Messages. A means of communication between objects. It is a foundation for object-oriented programming by which it starts an operation by invoking an object's methods.

Object. An object is an abstraction which describe essential features of a thing. An object is embedded with data and methods.

Object-oriented programming language. Allow computer programmers to represent real-world objects in a computer.

Object-oriented Analysis. The analysis phase begins with an attempt to understand the problem. The early product of this phase is a statement of the problem describing: what is the system is trying to achieve?; what is the function of the system?.

Object-oriented Design. Creating a solution based on the Object-oriented analysis.

Polymorphic. Ability for the same message to produce different operations depending on the type of objects it is sent to.

BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

BRANZ MISSION

To promote better building through the application of acquired knowledge, technology and expertise.

HEAD OFFICE AND RESEARCH CENTRE

Moonshine Road, Judgeford
Postal Address - Private Bag 50908, Porirua
Telephone - (04) 235-7600, FAX - (04) 235-6070

REGIONAL ADVISORY OFFICES

AUCKLAND
Telephone - (09) 524-7018
FAX - (09) 524-7069
290 Great South Road
PO Box 17-214
Greenlane

WELLINGTON
Telephone - (04) 235-7600
FAX - (04) 235-6070
Moonshine Road, Judgeford

CHRISTCHURCH
Telephone - (03) 663-435
FAX - (03) 668-552
GRE Building
79-83 Hereford Street
PO Box 496