BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

CI/SfB
81 (99.964)
Date August 1994

BRANZ

STUDY REPORT

NO. 56 (1994)

REPORT ON OBJECT-ORIENTED PROGRAMMING FOR HOME AUTOMATION APPLICATIONS

A.H. Dechapunya

This research was supported by funding from the Foundation for Research, Science and Technology's Public Good Science Fund (PGSF) and from the Building Research Levy.

ISSN: 0113-3675

PREFACE

The Building Research Association of New Zealand (BRANZ) produced this report to document its work to date on software applications in home automation, and to suggest ways in which the Home Automation industry in New Zealand might benefit from further advances in this field.

ACKNOWLEDGEMENTS

The work was jointly funded by the Building Research Levy and the Foundation for Research, Science and Technology from the Public Good Science Fund.

The author also wishes to acknowledge the contributions made by Dr Wayne Sharman, Dr John Duncan, and Ian Strawbridge.

Special thanks to Rey Crisostomo for his contribution in C++ programming.

READERSHIP

This report is intended for workers in software engineering, home automation, electronics and manufacturing in New Zealand.

OBJECT-ORIENTED PROGRAMMING FOR HOME AUTOMATION APPLICATIONS

BRANZ Study Report SR56

A H Dechapunya

REFERENCE

Dechapunya, A.H. 1994. Object-Oriented Programming for Home Automation Applications. Building Research Association of New Zealand, BRANZ Study Report SR56, 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

An object-oriented framework for home automation applications is described. The classes of objects developed provide a foundation for applications such as automation of appliances and lights, energy management, entertainment, and security and safety.

The design of the C++ classes employs various object-oriented techniques including encapsulation, inheritance, and polymorphism. The result is a framework for home automation applications which can be extended and maintained efficiently.

PCs are proposed as suitable hardware for providing whole-house controllers.

CONTENTS

TABLES

1.0 INTRODUCTION

1.1 Trends In Home Computers

Computers are becoming more and more of a commodity. PCs are becoming affordable by many New Zealand households (low-end PCs can cost well below $2,000). The concept of utilising these low-end PCs for home automation applications is being investigated. Using a PC as a whole-house controller is a better approach than using a dedicated machine because:

* It can be used for other purposes. That is, users can run home automation tasks at the same time as they are working on their spreadsheets or word processing.
* PCs already possess components which can be used for home automation applications.
* Hardware and software technology is improving continually. This means that a PC-based home automation developer could utilise the technology and pass it on to the users with little cost.
* PCs are more flexible, in terms of adding home automation capabilities, and upgrading hardware and software.

In New Zealand, 20% of households have PCs, normally used for word processing, financial management and entertainment. Clearly, there is a market for using a PC as a platform for integrated home automation. This will add another dimension to the uses of PCs in homes.

The concept of using a PC as distributed home controller is also being investigated overseas. The overseas products (software), however, are designed for technically oriented consumers. The consumers need to apply the system for their applications.

Our system is designed from day one to be application-oriented. The foundation classes developed are application-specific classes.

1.2 Objectives

This work programme sets out to identify, adapt and exploit advanced technologies to simplify and enhance the control of home environments. By providing a structure and platform for modelling, designing and representing physical processes for home applications and services.

Its current objective is to produce foundation classes, which provide a framework for home automation applications (e.g. automation of appliances and lights, energy management, entertainment, security and safety), and implementing these classes on a PC with an object-oriented programming language.

1.3 Report Design

Section 2 outlines the investigation's methodology.

Section 3 describes the main classes for home automation applications and their C++ implementation, and discusses some issues relating to the use of object-oriented programming language.

Section 4 concludes the report and outlines future work.

2.0 METHODOLOGY

2.1 Underlying Methodology

The work strategy is to employ, when possible, standard methodologies using the underlying principles of:

* a uniform user interface for all applications;
* re-engineering rather than reinventing;
* object-oriented technology; and
* integration rather than standalone.

The main technology used is object technology. It is seen as the technology with the greatest promise to achieve the optimum life-cycle of software applications development. It has been examined previously as a software engineering approach to the development of home automation applications (Dechapunya, 1992-1993).

2.2 PC-Based Home Automation

There are two approaches to using a PC as a whole-house controller: standalone, and networking. Table 1 illustrates a PC as a standalone home controller. In this approach, controlled elements are connected directly to the PC. Table 2 illustrates a PC as a networked home controller. In this approach, both PC and controlled elements are connected to the house communications network. The standalone approach is the older concept. The trend is towards network. The network approach is cleaner (does not create a wire mire), integrates into the home infrastructure, and makes use of modern communications technology.

The three basic components required to turn a normal home PC into a PC-based home automation system are: Software; PC hardware interface; and Controlled device hardware interface.

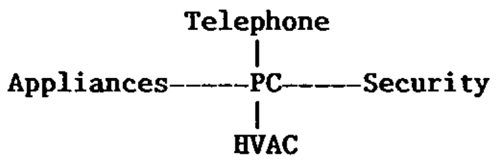

Table 1: Use of a standalone PC for Home Control

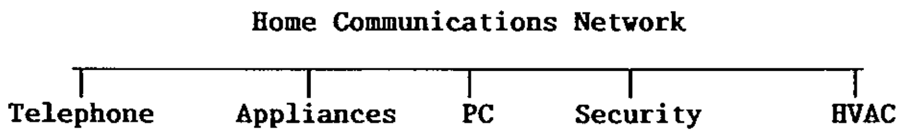

Table 2: Use of a networking PC for Home Control

2.3 Object-Oriented Analysis and Design

An object-oriented analysis and design is a set of guidelines and frameworks, so a software engineer can develop an application model from which object-oriented program can be produced (Branson and Herness, 1993).

The analysis and the design of the problem domain is based on Booch's methodology (Booch, 1991). Booch's methodology has been gaining popularity and it provides a more natural means of solving a problem (Dechapunya, 1993; Horstmann, 1993; Singer, 1993).

2.4 Object-Oriented Programming

C++ is used as an object-oriented (OO) programming language to implement the model. C++ is the language of choice for a number of reasons (Atkinson and Atkinson, 1992; Voss, 1991; Walker, 1992; Wybolt, 1990; Mitchell et al., 1992; Atwood et al., 1993).

C++ is the de facto global industry standard for an object-oriented programming language. Thus, a number of related technologies are available to support the C++ environment including: class libraries, OO databases; and case tools for aiding software developments.

C++ is compatible with C, an industry standard for conventional programming languages. C++ compilers will compile both C and C++ programmes. It is a strongly-typed language. That is, each C++ variable must be defined to be of a particular type. Typing makes coding more efficient and easier to read. Errors in typing will be detected during the program's compilation.

C++ is an expressive programming language, allowing an easy path from designing to programming. In some cases, designing with C++ is as easy as using any specification language.

The downside of C++ is that it is a complex language and not easy to master. Some academics do not favour C++ because it is not a pure object-oriented programming language.

2.5 User Interface

The quality of the user interface is the key factor in gaining acceptance of home automation systems by a more general and non-technical consumer. The user interface must be powerful, easy to use, friendly, hide the complexity of the system, and be based on accepted standards.

User interfaces for home automation applications can be classified as existing technology in the home and modern technology going into homes. Some of the existing technology in the home comprises keyboard, mouse, computer screen, TV screen, remote, telephone, touch pad, and radio devices. The modern technology going into homes comprises voice, sound, and speech recognition.

The user interface plays a crucial role in this work. One of the underlying principles is to provide a uniform user interface for all applications. This means that as a new style of a user interface is designed, it has to be implemented for other applications, and designed to be applied by a range of technologies.

3.0 HOME AUTOMATION MODELS

This chapter reports the investigations main result. It describes foundation classes which provide a general framework for home automation applications and their C++ representations. The chapter begins with an overview of the main classes. Each main class is described in subsequent sections.

Foundation and framework mean that the classes and their implementation presented here will provide the framework from which specific applications can be built.

3.1 Overview

3.1.1 Problem Domain

The application domain chosen for the study is a PC-based Total Home Controller (THC). The network architecture (Table 2) is applied here. Basically, the THC consists of:

* a standard PC running DOS 5.0 or higher;
* a PC interface connecting the PC to the home communications network;
* appliance interfaces connecting appliances to the home communications network; and
* an applications software functioning as the controller and the automator.

The function of the THC is to control and automate the home. The main tasks and devices which are used to perform these tasks are shown in Table 3.

Tasks	Devices
General control and automation	Main Unit
Control of appliances and lights	Appliance Unit
Security and safety	Security/Safety Unit
Energy management	Energy management Unit
Entertainment	Entertainment Unit

Table 3: Devices For Modelling The Total Home Controller

3.1.2 The Main Menu

The main program controls various executions of the objects. The user interface is a menu-based system. The main menu provides users with a top level of operations as shown in Table 4. Each sub-menu allows control over a specific functional area.

```
        TOTAL HOME CONTROLLER
    (c) Home Systems Limited 1993

              MAIN  MENU

A    -    Automatic Mode Menu

1    -    Appliance Operations Menu
2    -    Security Menu
3    -    Energy Management Menu
4    -    Entertainment Menu

Esc -     Exit The System

Enter Your Selection -->
```

Table 4: Main Menu of Total Home Controller

3.1.3 The Main Unit and Applications Units

As shown in Table 3, a total home controller is modelled as a system which contains a number of specific Units which are Unit, AppUnit, SecUnit, EmUnit, and EtUnit. The main Unit is designed to contain all the necessary objects for home automation applications. Other units are classified as views of home automation applications. They are created when required and destroyed when no longer required. The applications units always access the centralised Unit for the processing of their tasks.

The main Unit and applications Units are modelled as classes as shown below:

Task	Class	Object Classification	Section
Control and automation	UNIT	Centralised Object	3.2
Appliances control	APPUNIT	Applications View	3.3
Security and safety	SECUNIT	Applications View	3.4
Energy management	EMUNIT	Applications View	3.5
Entertainment	ETUNIT	Applications View	3.6

The above classes are described in more details in Sections 3.2 to 3.6, respectively. The concept of modelling a Unit is described in Dechapunya (1993). Class Unit, and its composition is shown in Table 5. Table 6 illustrates the class diagram of the total home controller.

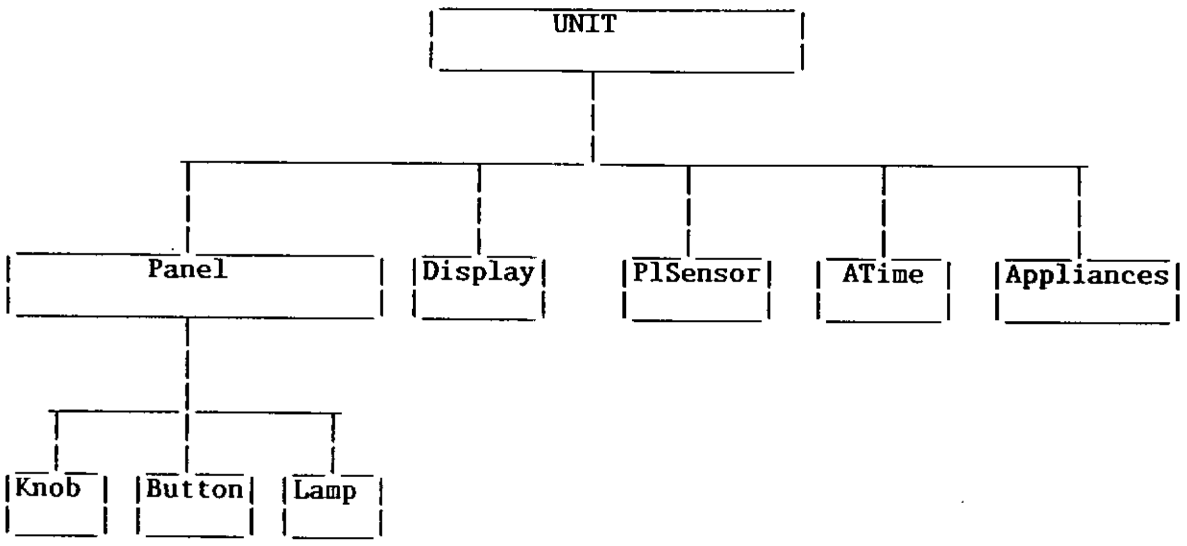

Table 5: Unit and its Composition

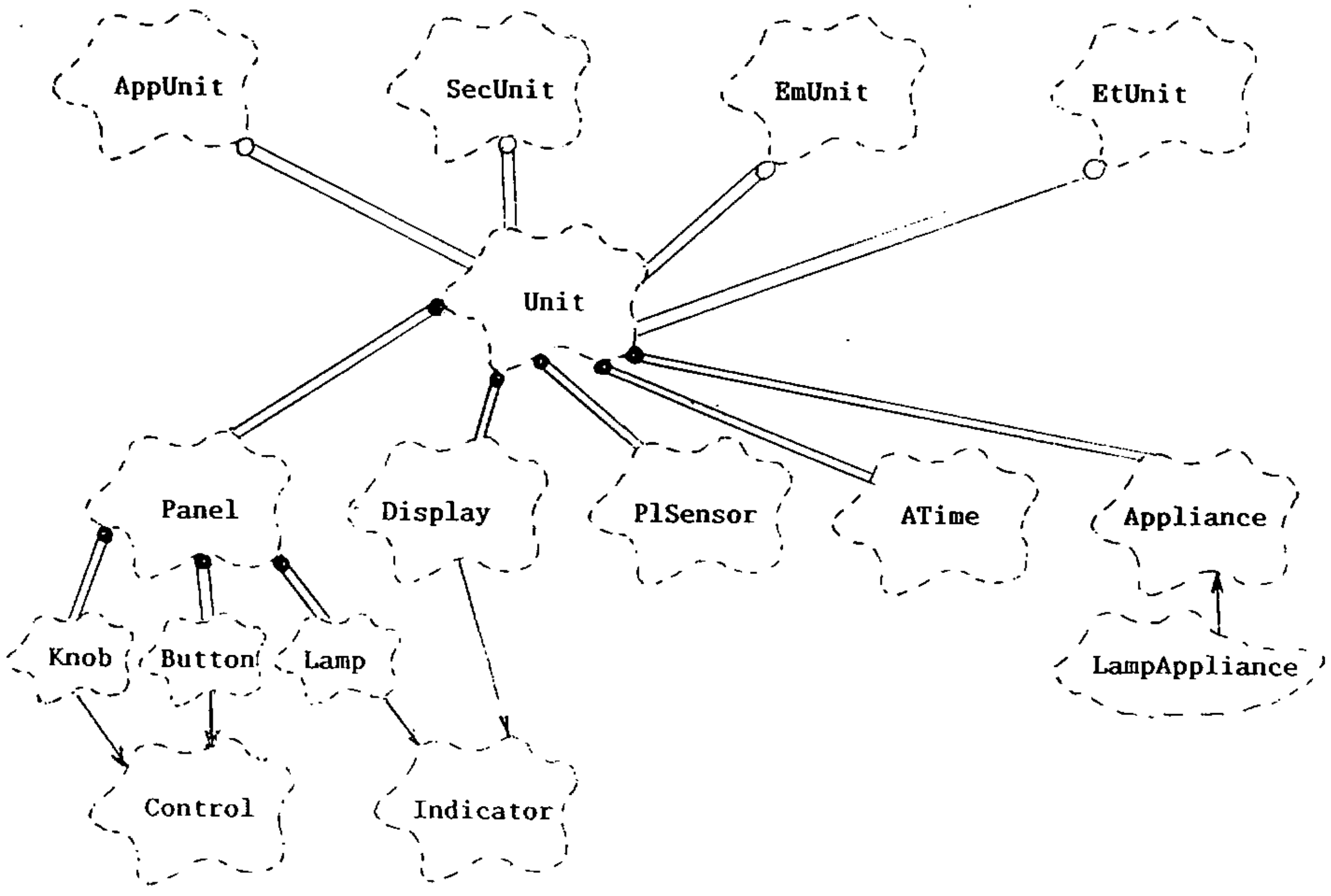

Table 6 : Total Home Controller Class Diagram

3.2 Unit Model

3.2.1 Composition and Design

The physical devices (appliances, lights, etc.) connecting to the home communications network are modelled as a collection of controlled objects.

Class Unit is composed of classes Panel, Display, PlSensor, ATime, and Appliance. Classes Panel and Display have been described in Dechapunya (1993). Unit has one or more appliances. Unit owns the appliances; each appliance, in turn, knows its owner and can access all the devices owned by the Unit.

Class ATime is designed to provide support for data and functions which are associated with times. The data members of class ATime include sunData (contains values of sunrise and sunset), day (indicates it is during the day), weekDay (indicating it is a weekday), winter (for winter) etc. The functions of class ATime include setHousekeepingTime which is used to determine the values of the data members.

Two classes are designed to represent appliances: **Appliance** and **LampAppliance**. Class Appliance has been designed to provide all of the attributes of an appliance. It contains all the data and functions of an appliance. The data members of class Appliance include **name** which is the name of an appliance, **address** which represents a physical location of an appliance, **onTime** which is the time an appliance will be switched on, and **isOn** indicates whether an appliance is on or off. The functions of class Appliance includes **GetName** which is used to obtain the name of an appliance, **GetAddress** is used to get the address of an appliance, **SwitchOn/SwitchOff/Dim/Bright** are used to operate an appliance, **GetType** is used to determine the type of an appliance which can be either normal-appliance or light. Class LampAppliance contains the data and functions of a lamp or incandescant light. Class LampAppliance inherits from class Appliance.

Borland container class library (Borland C++ Programmer's Guide, 1992) is used by class Unit to store an array of appliance pointers as shown below.

```
TIArrayAsVector<Appliance>        appliances;

TIArrayAsVector    =    name of a Borland class template
Appliance          =    name of a class
appliances         =    an array of appliance pointers
```

The function of TIArrayAsVector is to provide a container for an array of objects. The statement "TIArrayAsVector<Appliance>" represents a class of an array of appliance pointers. That is, "appliances" is an instance of a class of an array of appliance pointers. Initially the container is empty. A function "LoadAppliances" is used to place the pointers of each appliance data into the container. Once these pointers are placed in the container, programs can utilise them.

Using container classes means that the number of appliances does not need to be fixed, allowing the user to expand the number of appliances without being concerned about telling the system how many appliances there are.

Since the appliance pointers point to the base class Appliance, and because LampAppliance is derived from the base class Appliance, the pointers also point to LampAppliance. Polymorphism (the ability for the same message to produce different operations depending on the type of object it is sent to) is applied here to ensure that only lamps or lights can be dimmed or brightened. A DIM or BRIGHT message can be sent to any type of appliance, normal or lamp. Once the message is received, the same message will perform different operations for different types of appliances, depending on the type of the appliance. The program representing this concept is shown below:

```
class Appliance
{
public:
   virtual void Dim();
   virtual void Bright();
};
void Appliance::Dim()
{
}
void Appliance::Bright()
{
}

class LampAppliance : public Appliance
{
public:
   virtual void Dim();
   virtual void Bright();
};
void LampAppliance::Dim()
{
   owner->doFunction(GetAddress(), ::Dim);
   isOn = 1;
}
void lampAppliance::Bright()
{
   owner->doFunction(GetAddress(), ::Bright);
   isOn = 1;
}
```

As shown above, the DIM or BRIGHT functions of class Appliance perform nothing. On the contrary, the DIM or BRIGHT functions of class LampAppliance perform dim or brighten.

Object UnitA, the main controlling object, is created at the start of the program and lasts the duration of the program. AppUnit, SecUnit, EmUnit, and EtUnit are the views of the application of Unit. Each of these application units are created and destroyed sequentially. UnitA is created and initialised as shown below. It shows that the UnitA is instantiated with two default messages. H1 is the default appliance, 4 is the default serial port. The port is checked for availability. The Homedata.dat file which contains the data of appliances and lights is then loaded.

```
 Unit UnitA("H1", "4")
 UnitA.checkPortID();
 UnitA.LoadAppliances("c:\\thc\\Homedata.dat");
```

3.2.2 Automatic Operation

In automatic mode, THC runs continuously performing a number of tasks depending on two external forces: time and external events.

Time means minute, hour, day, month, and season. External events can be caused by a number of sources including commands from homeowners, presence of intruders, or requests from other controllers or appliances.

In conventional houses, switches are hardwired into the power circuits and their operations are associated with outlets. In an automated home, switches are addressable and can be controlled from any controller.

Sensors include those for detecting/measuring occupancy, intrusion, temperature, humidity, light, smoke and air quality. Under programmed control, a sensor can provide a trigger to other events. For example, a temperature sensor will cause a fire alarm to ring if it detects abnormally high temperatures.

The design of the user interface for automatic operation is shown in Table 7. The menu provides a general interface for information on the operation of the automatic part of THC.

```
        AUTOMATIC MODE  MENU

    A - Activate Automatic Mode

    1 - General Info
    2 - Info on Security Tasks
    3 - Info on Energy Management
    4 - Info on Housekeeping
    0 - Quit This Menu

    Enter Your Selection -->
```

Table 7: Automatic Operation Menu

3.2.3 Class Specifications

Classes Unit, PlSensor, ATime, and Appliance are described in Tables 8A-8D.

3.2.4 C++ Representation

Tables 9A-9D show an abstract implementation of classes Unit, PlSensor ATime, and Appliance using an object-oriented programming language, C++. Full listings of source codes are not available in this report.

Class Name:	Unit
Function:	General Control and Automation
Attributes:	appliances
	TimeA
	PanelA
	DisplayA
	PlSensorA
	unitAddress
	houseID
	unitID
	securityActivate
	intruder
	home
Operations:	LoadAppliances
	getHouseIDandUnitID
	getUnitStatus
	doFunction
	allUnitsOff
	allLightsOn
	allLightsOff
	startAlarm
	stopAlarm
	automatic
Used by:	None
Contains:	Panel, Display, ATime, Appliance, PlSensor
Inherited by:	None
Derived from:	None

Table 8A: Class Specification of Unit

Class Name:	PlSensor
Function:	Powerline monitoring and sensing
Attributes:	portID
Operations:	checkPortID
Used by:	Unit
Contains:	None
Inherited by:	None
Derived from:	None

Table 8B: Class Specification of PLSensor

Class Name:	ATime
Function:	Provide time in a general sense
Attributes:	sunData
	timeOfDay
	dayName
	day
	evening
	night
	weekDay
	sunRise
	sunSet
	winter
	summer
	wakeUpTime
Operations:	setHousekeepingTime
	showDate
	showTime
Used by:	Unit, AppUnit, SecUnit, EmUnit, EtUnit
Contains:	None
Inherited by:	None
Derived from:	None

Table 8C: Class Specification of ATime

Class Name:	Appliance
Function:	Representing an appliance
Attributes:	name
	address
	onTimeH
	onTimeM
	offTimeH
	offTimeM
	isOn
	owner
Operations:	GetName
	GetAddress
	SetOwner
	SwitchOn
	SwitchOff
	SetOnStatus
	IsOn
	GetOnTime
	hour
	minute
	GetOffTime
	SetOnTime
	SetOffTime
	CheckTime
	GetType
	Dim
	Bright
Used by:	Unit
Contains:	None
Inherited by:	None
Derived from:	None

Table 8D: Class Specification of Appliance

```
// UNIT.H
// CLASS Unit DEFINITIONS

class Unit
{
public:
  TIArrayAsVector<Appliance>  appliances;
  ATime    ATimeA;
  Panel    PanelA;
  Display  DisplayA;
  PlSensor PlSensorA;
public:
  char * unitAddress;
  int    houseID;
  int    unitID;
  Unit(char * unitAddress1, char * portIDA,
    int panelStateA, int buttonPositionA, int knobSettingA);
public:
  int LoadAppliances(const char * filename);
  void getHouseIDandUnitID();

// Appliances Operation
public:
  void getUnitStatus(int applianceSelected);
  int  doFunction(const char * address, int func);
  void allUnitsOff(char homeID);
  void allLightsOn(char homeID);
  void allLightsOff(char homeID);

// Security
public:
  int securityActivate;
  int intruder;
  int home;
  virtual void startAlarm();
  virtual void stopAlarm();

// Automatic Operation
public:
  void automatic();

// Accessing ATimeA Object
public:
  void setHousekeepingTime() {ATimeA.setHousekeepingTime();}
protected:
  void showDate() {ATimeA.showDate();}
  void showTime() {ATimeA.showTime();}

// Accessing PlSensorA Object
public:
  int checkPortID() {return PlSensorA.checkPortID();}
};
...
...
// FUNCTIONS DEFINITION
...
...
```

Table 9A: C++ Representation of Class Unit

```
// PLSENSOR.H
// CLASS PlSensor DEFINITIONS

class PlSensor
{
public:
  char * portID;
  int  checkPortID();
  PlSensor(char * portID1);
};
...
...
// FUNCTIONS DEFINITION
...
...
```

Table 9B: C++ Representation of Class PlSensor

```
// ATIME.H
// CLASS ATIME DEFINITIONS

class ATime
{
private:
  static int  sunData[2][12];
  static char *timeOfDay[3];
  static char *dayName[7];

public:
  int day;
  int evening;
  int night;
  int weekDay;
  int sunRise;
  int sunSet;
  int winter;
  int summer;
  int wakeUpTime;

public:
  time_t      timer;
  struct  tm  *tblock;

public:
  ATime();
  void  setHousekeepingTime();
  void  showDate();
  void  showTime();
};
...
...
// FUNCTIONS DEFINITION
...
...
```

Table 9C: C++ Representation of Class ATime

```
// APPLIANS.H
// CLASS Appliance DEFINITIONS

const int APPLIANCE_NAME_MAXLEN = 25;
const int APPLIANCE_ADDRESS_MAXLEN = 2;

class Unit;

class Appliance {
public:
   Appliance(const char *name, const char *address);
   virtual ~Appliance();
   inline const char *GetName() const;
   inline const char *GetAddress() const;
   inline void SetOwner(Unit *owner);
   void SwitchOn();
   void SwitchOff();
   inline void SetOnStatus(int status);
   inline int IsOn() const;
   void GetOnTime(int *hour, int *minute) const;
   void GetOffTime(int *hour, int *minute) const;
   void SetOnTime(int hour, int minute);
   void SetOffTime(int hour, int minute);
   void CheckTime(int hour, int minute);
   virtual char GetType();
   virtual void Dim();
   virtual void Bright();

protected:
   char name[APPLIANCE_NAME_MAXLEN + 1];
   char address[APPLIANCE_ADDRESS_MAXLEN + 1];
   int onTimeH;
   int onTimeM;
   int offTimeH;
   int offTimeM;
   int isOn;
   Unit *owner;
};

class LampAppliance : public Appliance {
public:
   LampAppliance(const char *name, const char *address);
   virtual ~LampAppliance();
   virtual char GetType();
   virtual void Dim();
   virtual void Bright();
};
...
...
// FUNCTION DEFINITIONS
...
...
```

Table 9D: C++ Representation of Class Appliance

3.3 Appliance Control Model

3.3.1 Composition and Design

The control and operation of appliances and lights are designed as an applications view of a Unit. Class AppUnit is used to implement this task. When created, AppUnit object contains a pointer to an instance of class Unit. This allows AppUnit to utilise the resources of this particular instance of Unit. Object AppUnit is created when needed and destroyed when no longer required.

Object AppUnitA is created dynamically i.e., it is created at run-time. Once the AppUnitA performs its tasks, it can be destroyed, thus saving memory space. The programming method for this dynamic creation AppUnitA object is shown below:

```
AppUnit * AppUnitA;
AppUnitA = new AppUnit(&UnitA);
AppUnitA->applianceMenu();
delete AppUnitA;
```

The dynamic creation method is normally used because it uses memory space more efficiently. This is due to the nature of the program sequence. As shown in the above listing, there is no need for all objects to be allocated at the same time, since each object is required to work independently of any other. Once the object has done its task, it is destroyed. One of the disadvantages of the dynamic creation method is that the object has to be created each time a user selects a particular task.

The design of the user interface for the control of appliances and lights is shown in Table 10. The menu allows users an easy and clear control of appliances and lights.

```
APPLIANCE OPERATIONS MENU

S - Select an Appliance Unit

1 - Turn Off Unit
2 - Turn On Unit
3 - Dim Unit
4 - Brighten Unit
5 - All Lights On
6 - All Lights Off
7 - All Units Off

U - Unit Status
0 - Quit This Menu

Enter Your Selection -->
```

Table 10: Appliance Control Menu

The design of appliances data input is crucial to the program. Basically, the input data cannot be hardcoded in the program since each home will have different requirement. Ideally, the input data should be entered graphically using a standard Windows interface. This will be considered later. At this stage of the development, a text input data file is used, as shown below:

Appliance Type	Appliance Description	Appliance Address
A,	"Study Room Heater",	"K1",
A,	"Study Room Radio",	"K2",
A,	"Study Room Fan",	"K3",
A,	"Coffee Warmer",	"K4",
L,	"Neon Light",	"K5",
L,	"Lounge Table Lamp",	"K6",
L,	"Lounge Light",	"K7",
A,	"Lounge Heater",	"K8",
A,	"Washing Machine",	"H1",
L,	"Outdoor Light",	"A1",
L,	"Telephone Light",	"P10",
L,	"Security Light",	"P15",

As shown above, an appliance can be classified into a normal appliance, A, or a light, L. A normal appliance can not be dimmed or brightened. The input data will be expanded to include more attributes in future work.

3.3.2 Class Specifications

Class AppUnit is described in Table 11.

3.3.3 C++ Representation

Table 12 shows an abstract implementation of class AppUnit using C++. Full listings of source codes are not available in this report.

The relationship between AppUnit and Unit is worth looking at. As shown in Table 12, this relationship is represented by the statements "AppUnit(Unit UnitY);" and "Unit UnitX;". The constructor of AppUnit is represented by the following statement:

```
AppUnit::AppUnit(Unit * UnitY)
: UnitX(UnitY) {}
```

The above statement can be explained as follows: When an instance of AppUnit is created, its constructor will be executed automatically. In this case, it will create a pointer to Unit called UnitY. This pointer, however, will disappear after the constructor is executed since it is a local variable of the constructor. Thus, a formal data member must be defined, UnitX, then, UnitY is assigned to UnitX as shown in the statement "UnitX(UnitY)". Because it is a data member of AppUnit, UnitX will continue to exist throughout the lifetime of the AppUnit instance. This relationship between Unit and AppUnit allows class AppUnit to access the public functions of class Unit as shown in the example below:

```
UnitX->appliances[applianceSelected]->SwitchOn();
UnitX->appliances[applianceSelected]->Dim();
UnitX->allLightsOn('*');
UnitX->allUnitsOff('*');
```

Class Name:	AppUnit
Function:	Control of appliances and lights
Attributes:	UnitX (pointer to Unit)
Operations:	applianceMenu selectAppliance
Used by:	None
Contains:	None
Inherited by:	None
Derived from:	None

Table 11: Class Specification of AppUnit

```
// APPUNIT.H

// CLASS AppUnit DEFINITIONS

class AppUnit
{
public:
  AppUnit(Unit * UnitY);
  Unit * UnitX;

public:
  void applianceMenu();
  int  SelectAppliance();
};
...
...
// FUNCTION DEFINITIONS
...
...
```

Table 12: C++ Representation of Class AppUnit

3.4 Security Model

3.4.1 Introduction

Study has shown that as many as 90% of homeowners who purchased home security systems are not using them (Electronic House, May/June 1989). Basic reasons given were because they were: difficult to use; unreliable; and false alarms occurred.

Home security can be passive, active or a combination of both. A passive system is achieved by making the home look and sound occupied. An active system is achieved by activating certain pre-programmed functions when an intruder is detected.

A typical active system consists of sensors which sense/detect the status of doors/windows, or sense/detect movement of intruders. When an intruder is detected, the sensors will perform alarm functions which include: turn on outdoor lights; sound bell; automatic telephone to security/police; and disarming intruders.

There are a wide range of intruder alarm systems, from basic systems which can be obtained from any retail outlets, to high-end systems, which may require experts to install and maintain.

An effective system is the one which meets the requirements of homeowners, security, police and insurance. These requirements include:

* Simple to Install and Use: A wireless system provides flexibility and portability. The system can be easily re-configured.
* Continuous Operation: Backup power supply is required.
* Reliable Operation: That is, the system must have reliable detectors, reliable communication, reliable system controller, reliable backup power supply, and reliable start/stop.
* Access to the system only by homeowners: The system should be installed and located in a way that prevents intruders accessing it.

3.4.2 Composition and Design

The security and safety functions are designed as an applications view of a Unit. Class SecUnit is used to implement this task. When created, object SecUnitA contains a pointer to object UnitA, allowing SecUnitA to utilise the resources of UnitA. SecUnitA is created when needed and destroyed when no longer required. The programming method for creating SecUnitA is shown below:

```
SecUnit * SecUnitA;
SecUnitA = new SecUnit(&UnitA);
SecUnitA->securityMenu();
delete SecUnitA;
```

The design of a user interface for security and safety is shown in Table 13. The menu allows users to control and manipulate security.

```
                    SECURITY   MENU

             1 - Panic Button: Help Help Help
             2 - Activate Security: Going Out
             3 - Activate Security: Stay Home
             4 - Deactivate Security: I'm Home
             5 - Security Report

             0 - Quit This Menu

             Enter Your Selection -->
```

Table 13: Security Control Menu

3.4.3 Class Specifications

Class SecUnit is described in Table 14.

3.4.4 C++ Representation

Table 15 shows an abstract implementation of class SecUnit using C++. Full listings of source codes are not available in this report.

The present design of class SecUnit, which is an applications view for security and safety services, contains one function only, securityMenu. The data and functions related to security tasks are encapsulated in class Unit. The relationship between Unit and SecUnit is similar to those between Unit and AppUnit discussed in Section 3.3. This relationship allows class SecUnit to access the public functions of class Unit, as shown in the example below:

```
switch(option)
{
  case '1': UnitX->startAlarm();
            UnitX->automatic();
            break;
  case '2': pauseTime();
            UnitX->securityActivate = 1;
            UnitX->home = 0;
            UnitX->automatic();
            break;
  case '3': UnitX->securityActivate = 1;
            UnitX->home = 1;
            UnitX->automatic();
            break;
  case '4': UnitX->securityActivate = 0;
            UnitX->home = 0;
            UnitX->stopAlarm();
            break;
  case '5': UnitX->DisplayA.messageReport("security.log");
}
```

The function automatic currently contains nothing. The detailed study of automatic tasks will be the subject of future work.

Class Name:	SecUnit
Function:	Security and Safety
Attributes:	UnitX (Pointer to Unit)
Operations:	securityMenu
Used by:	None
Contains:	None
Inherited by:	None
Derived from:	None

Table 14: Class Specification of SecUnit

```
// SECUNIT.H
// CLASS SecUnit DEFINITIONS

class SecUnit
{
public:
  SecUnit(Unit * UnitY);
  Unit * UnitX;

public:
  void securityMenu();
};
...
...
// FUNCTION DEFINITIONS
...
...
```

Table 15: C++ Representation of Class SecUnit

3.5 Energy Management Model

3.5.1 Introduction

There are a range of factors affecting household energy consumption including insulation, ventilation, space heating, water heating, space cooling, and the way the house is used.

Of the above factors, only insulation is static. The others can be adjusted dynamically to suit the needs of occupants. The adjustments, if properly done, can lead to the right balance between comfort and saving in energy. Home automation systems can make these adjustments.

3.5.2 Composition and Design

The Energy management function is designed as an applications view of class Unit. Class EmUnit implements this task. When created, Object EmUnitA contains a pointer to object UnitA allowing EmUnitA to utilise the resources of UnitA.

Object EmUnitA is created when needed and destroyed when no longer required. The programming method for creation of EmUnitA is shown below:

```
EmUnit * EmUnitA;
EmUnitA = new EmUnit(&UnitA);
EmUnitA->energyManagementMenu();
delete EmUnitA;
```

The design of a user interface for the energy management operation is shown in Table 16. The menu allows users to manage energy usage.

```
ENERGY  MANAGEMENT  MENU

1 - HVAC: I am home
2 - HVAC: I am going out
3 - Thermostat Setting
4 - Energy Usage Report

0 - Quit This Menu

Enter Your Selection -->
```

Table 16: Energy Management Menu

3.5.3 Class Specifications

Class EmUnit is described in Table 17.

3.5.4 C++ Representation

Table 18 shows an abstract implementation of classes EmUnit using an object-oriented programming language, C++. Full listings of source codes are not available in this report.

The current design of class EmUnit, an applications view for energy management applications, contains three functions: energyManagementMenu, setThermostat, and hvac. The relationship between Unit and EmUnit is similar to that between Unit and AppUnit. This relationship allows class EmUnit to access the public functions and data of class Unit. The implementation of the design of energy management operation, shown in Table 16, using C++ is shown in the example below:

```
switch(option)
{
  case  '1': UnitX->home = 1;
             hvac();
             break;
  case  '2': UnitX->home = 0;
             hvac();
             break;
  case  '3': setThermostat();
             break;
  case  '4': UnitX->DisplayA.messageReport("energy.log");
             break;
  default  : break;
}
```

At this stage of the work, the function hvac() contains nothing. The detailed study of energy management applications is the subject of future work.

Class Name:	EMUnit
Function:	Energy Management
Attributes:	UnitX (Pointer to Unit)
Operations:	energyManagementMenu
	hvac
	setThermostat
Used by:	None
Contains:	None
Inherited by:	None
Derived from:	None

Table 17: Class Specification of EMUnit

```
// EMUNIT.H
// CLASS EmUnit DEFINITIONS

class EmUnit
{
public:
  EmUnit(Unit * UnitY);
  Unit * UnitX;

public:
  void energyManagementMenu();
  void setThermostat();
  void hvac();          //design only
};
...
...
// FUNCTION DEFINITIONS
...
...
```

Table 18: C++ Representation of Class EMUnit

3.6 Entertainment Model

3.6.1 Composition and Design

The Entertainment function is designed as an applications view of a Unit. Class EtUnit is used to implement this task. When created, object EtUnitA contains a pointer to object UnitA. This allows EtUnitA to utilise the resources of UnitA. The programming method for creation of EtUnitA is shown below:

```
EtUnit * EtUnitA;
EtUnitA = new EtUnit(&UnitA);
EtUnitA->entertainmentMenu();
delete EtUnitA;
```

The design of a user interface for the entertainment operation is shown in Table 19. The menu provides users an access to entertainment functions of the total home controller system.

```
ENTERTAINMENT   MENU

1 - Party Time
2 - Party End
3 - Musical Interlude
4 - CD Player

0 - Quit This Menu

Enter Your Selection -->
```

Table 19: Entertainment Menu

At this stage of the design, only two devices are included in the menu, a Sound Card and a Compact Disk Player.

3.6.2 Class Specifications

Class EtUnit is described in Table 20.

3.6.3 C++ Representation

Table 21 shows an abstract implementation of classes EtUnit using an object-oriented programming language, C++. Full listings of source codes are not available in this report.

At the current stage of the work, the functions musicMenu() and cdPlayer() contain nothing. The detailed study of entertainment applications is the subject of future work.

Class Name:	ETUnit
Function:	Entertainment
Attributes:	UnitX (Pointer to Unit)
Operations:	entertainmentMenu
	partyTime
	partyEnd
	musicMenu
	cdPlayer
Used by:	None
Contains:	None
Inherited by:	None
Derived from:	None

Table 20: Class Specification of ETUnit

```
// ETUNIT.H
// CLASS EtUnit DEFINITION

class EtUnit
{
public:
  EtUnit(Unit * UnitY);
  Unit * UnitX;

public:
  void entertainmentMenu();
  void partyTime();
  void partyEnd();
  void musicMenu(); // design only
  void cdPlayer();  // design only
};
...
...
//      FUNCTION DEFINITIONS
...
...
```

Table 21: C++ Representation of Class ETUnit

3.7 Discussion

3.7.1 C++

C++, a hybrid object-oriented programming language, provides a good environment and language support for developing home automation applications. C++ allows home automation objects to be better represented in a computer, in that it allows the total system to decompose into a number of integrated modules. Each module retains its independence, making the system extremely modular. This capability enables programmers to write programs in a more natural manner.

Specifically, the features which prove useful in this study are inheritance, composition, encapsulation (private, protected, public), polymorphism, dynamic binding, and the container class library.

3.7.2 Global Data

Global data is not used as a general design practice since it contradicts the object-oriented principle of encapsulation. The use of global data extensively in a program can potentially lead to difficulty in maintaining the program.

However, there exists the need for global data in home automation applications so that when an object changes some or all of its data, all other objects see the new data. An example is the ability for all objects to check security status, intruder status, and whether the home is occupied.

There are two options to implement this requirement in C++. The first option is to declare the data as true global in the main program as shown below:

```
int securityActivate;
int intruder;
int home;
```

The second option is to declare the data as a static data member as shown below:

```
class Unit
{
protected:
  static int securityActivate;
  static int intruder;
  static int home;
};
int Unit::securityActivate = 0;
int Unit::intruder = 0;
int Unit::home = 1;
```

In the second option, the encapsulation of the data in class Unit ensures that only objects derived from Unit can access the data, and that the static nature of the data means that when an object changes the data, all other objects would see the changes.

Neither method was used in the final implementation. In the final program, the design is based on object UnitA being active throughout, and that the other main objects, applications views, are created as they are required by the users. Thus, by making the above variables (securityActivate, intruder, home) data of the class Unit, these variables will also be active throughout the life of the active program.

3.7.3 Container Class Library

In an early prototype of this study, appliances were modelled as data members as:

```
public:
  char * studyRoomHeaterAddress;
  char * radioAddress;
  char * fanAddress;
  char * coffeeWarmerAddress;
  char * tableLampAddress;
  char * lampAddress;
  char * lightAddress;
  char * neonAddress;
  char * loungeHeaterAddress;
  char * waterHeaterAddress;
  char * washingMachineAddress;
  char * alarmAddress;
```

One of the problems with this method is that homeowners do not have control in the set up of the appliances, since they are hardcoded in the compiled program. Another problem is that the above method does not utilise the advantages of object-oriented programming.

An object-oriented method to get around this problem is to create a class Appliance which encapsulates all attributes and operations of a generic appliance. The base Appliance class can be further classified into normal appliances and lights.

The Borland container class library (Borland C++ Programmer's Guide, 1992) is used to implement the concept. The result is a superior programming routine which allows users to manipulate, edit/delete/modify their appliances independently of the program. The benefits to programmers are enormous. Basically, it provides a complete basis for designing a generic class of appliance which is easy to implement, extend and maintain.

3.7.4 User Interface

The user interface accounts for at least 30% of the programming effort. The present design of the user interface is menu-oriented and implemented in a DOS environment. The basic problem with the menu-based user interface is that users are not in control of the applications. This is because a menu-based system only allows users to access the applications linearly. The interactivity between users and the applications is limited.

It would be better for the applications to be accessed not from top to bottom, but from any way which users choose to elect (non-linear access). Users have their own specific ways of using the applications. They should be able to go to any option, jump to any other option, and get answers any time they need them. This could be done by re-engineering the user interface using the Windows graphical user interface and multimedia methods.

4.0 CONCLUSIONS

4.1 Framework for Home Automation Applications

An object-oriented framework for home automation applications has been developed. The classes of objects developed provides an application framework for automation of appliances and lights, energy management, entertainment, security and safety. In a nutshell, it is a software design for total home control applications using object-oriented technology. The design and the classes have been implemented as a prototype on a PC using object-oriented programming language, C++.

C++ provides an excellent environment and language support for this work. It enables programmers to write programs in a more natural manner. C++ is the de facto global industry standard for object-oriented programming languages. That is, C++ is widely used and accepted throughout the world. The major weaknesses of C++ are its complexity of language and lack of standards for class libraries.

4.2 PC-based Home Controller

The concept of utilising low-end PCs for home automation applications is being investigated. The technology of PCs in both hardware and software is improving continually. This means that PC-based home automation developers can utilise the technology and pass it on to users with little cost.

Using a PC for home automation based applications means that any standard object-oriented programming language can be used as an application language.

4.3 Further Work

The immediate future work is to apply the home automation models developed to housekeeping applications. This will be achieved by extending existing classes as well as creating new classes. The principal application will be in the control and automation of lights and appliances.

Other important work will include the design and development of object-oriented graphical user interfaces and the integration of multimedia technologies.

REFERENCES

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

Atwood, W., Breakstone, A., Britton, D., Burnett, T., Myers, D., and Word, G. 1993. The Gismo Project. C++ Report March-April 1993: 38-43.

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

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

Branson, M., and Herness, E. 1993. The Object-Oriented Development Process. Object Magazine 3(4): 66-70.

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

Dechapunya, A.H. 1993. Object-Oriented Methodology for Home Automation Applications. Building Research Association of New Zealand, BRANZ Study Report SR52, Judgeford, New Zealand.

Horstmann, C.S. 1993. Two leading object-oriented design tools. C++ Report 5(1): 62-67.

Mitchell., E., Becker, P., Dlugosz, J., Finnell-Fruth, C., Free, G., Fruth, R., Herring, B.D., and Schulmeisters, K. 1992. Secrets of the Borland C++ Masters. SAMMS Publishing Carmel, Indiana 46032 USA, pp. 730.

Singer, G. 1993. An Eclectic Approach to Developing an O-O Methodology. Object Magazine 3(4): 36-41.

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

Walker, G. 1992. Why the choice must be C++. The C++ Journal 2(1): 52-65.

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

Copy 2

BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

BRANZ MISSION

To be the leading resource for the development of the building and construction industry.

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
118 Carlton Gore Road, Newmarket
PO Box 99-186, Newmarket

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

CHRISTCHURCH
Telephone - (03) 366-3435
FAX - (03) 366-8552
GRE Building
79-83 Hereford Street
PO Box 496