INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

An Object-Oriented Application of Framework Control Systems

Vincent A. G. Martins

A Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

February 1995

Abstract

An Object-Oriented Application of Framework Control Systems

Vincent A. G. Martins

We must face the fact that today, systems are becoming fairly large and more complicated than ever before. As a result of this, the programming environment is also changing. Every programmer knows the feeling of anxiety and frustration when the time comes for a system modification, maintenance, quality, efficiency, etc.

Several experts in the field of programming believe that the popular concept of object-oriented programming is a very useful and effective paradigm in the software industry. It advocates a partly bottom-up style of programming that offers tremendous benefits that fall in line with the goals of software engineering methodology.

Object-oriented programming is widely used in a variety of applications. One important area where this form of programming is continuously growing is in the field of control systems. In this report, the concepts of object-oriented programming are discussed. The Model-View-Controller (MVC) framework is explained and applied to the simulation of control systems using object-oriented techniques in the C++ language. The various benefits and pitfalls of using the object-oriented programming paradigm are also discussed.

Acknowledgments

I wish to thank my supervisor Dr. Peter Grogono for the advice he has given me throughout the course of my project.

I am especially grateful to my parents and family for their encouragement, patience, confidence and support. I acknowledge the effort of Claude Martins for drawing the figures used in this project.

Table of Contents

List of Figures

1. Introduction

The art of programming has changed dramatically over the years. Many programmers are now concerned more with the maintenance of old programs than writing completely new ones beginning from elementary steps. The psychological perspective of the modern programmer is that discarding a program is unnecessary if it can be altered and reused for a new application. The concept of recycling is even applied to this field where programming methodology is progressing rapidly in the hopes of developing a framework to aid the programmer to develop reliable software for applications.

Object-oriented methodology was developed after structured programming and has become increasingly popular in the software industry. As software becomes more complex and difficult to manage, another concept in the process of developing software has been introduced which is different from conventional programming. This concept is called the object-oriented programming paradigm.

The term "object-oriented" is such a popular concept that it tends to be overused, and is misleading particularly in the field of software development where different design methods, environments and languages tend to be classified in this methodology. By defining the term and describing the basic principles of object-oriented programming we will try to clarify some misconceptions.

1.1. Scope

In this report we discuss the features of the object-oriented programming paradigm. In particular, the Model-View-Controller (MVC) framework is described and applied to the simulation of two control system applications.

Chapter two defines the meaning of the object-oriented paradigm, what led to the motivation for its discovery and popularity. The requirements for a language to be considered as an object-oriented language are also explained.

Chapters three and four describe the main features of an object-oriented language such as objects, classes, inheritance, polymorphism and binding along with the advantages of using this form of programming methodology.

Chapter five presents the Model-View-Controller (MVC) architecture which is a good tool for the simulation of control systems.

Chapter six illustrates the MVC framework by simulating two applications of control systems-- a cruise control system and a home heating system. Common features of control systems are also identified.

Chapter seven discusses the results, problems encountered and solutions found in developing the framework for control systems. Future trends in programming paradigms are also introduced.

2. Object-Oriented Programming Paradigm

What is the object-oriented programming paradigm?

The object-oriented programming paradigm is a software development process that aims to provide an organizational framework for the development of large, complex systems. It is a tool designed to aid the programmer in the development of an application which is beneficial to a variety of users in the long run. It uses a "data-oriented" approach (as opposed to the functional approach) for software development where the data is encapsulated into objects and manipulated via operations or methods, thus strongly linking data and behavior together. Although several authors in this area have defined this methodology, one that offers a clear conceptual definition is given by Booch [1991];

> "Object-oriented programming is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

He points out that there are three parts to this definition:

(i) object-oriented programming uses objects instead of algorithms as the building blocks,

(ii) each object is an instance of some class, and

(iii) there is an inheritance relationship among the classes.

Depending upon the characteristics displayed by a programming language, it can be placed within one or more of the following sets. [Wegner 1990]

An *object-based* programming language is one where by its syntax and semantics support the creation and functionality of objects without their management (e.g. Ada, Actors). A *class-based* programming language is one that has the characteristic of an object-based language in addition to supporting the creation of classes and management of objects but offering no support for class management (e.g. Clu). An *object-oriented* programming language (e.g. C++, Simula, Eiffel, Smalltalk) supports the creation and functionality of objects, object management by classes, and management of classes through two types of inheritance--single inheritance (a subclass is derived from one superclass) or multiple inheritance (a subclass is derived from more than one superclass). Thus an object-based language is a subset of a class-based language which in turn is a subset of an object-oriented language.

2.1. Motivation for Object-Oriented Programming

The growing size and complexity of programs in software engineering led to the motivation for object-oriented programming as different tools are needed to tackle the various engineering problems. These problems include:

- High cost of software development and maintenance in delivering low quality systems.
- The lack of good tools to create, use, access and interpret data according to user specification.
- Development needed of sophisticated, self-documented, user-friendly interfaces and extendible programming environments that can be easily customized according to the users' requirements.
- The need for advanced tools for rapid prototyping, including the development of libraries adapted for a wide range of applications.
- Improvements required both in the areas of heterogeneous knowledge representation and programming environments that can provide several programming paradigms to deal with problems according to their nature. [Masini 1991]

It is conceivable that the concept of object-oriented programming will provide some of the methodologies that will enable a programmer to develop better programs that will in turn, hopefully, lead to reliable systems.

2.2. Popularity of Object-Oriented Programming

Why is object-oriented programming popular?

Object-oriented programming has adopted a set of principles that, if used, help the programmer initially to tackle complex tasks. The basic principles of modularity (splitting the problem domain into smaller manageable components that can be tackled separately), abstraction (separation of concerns among different components), and information hiding (concealing irrelevant detail at a particular level of abstraction), have contributed to the popularity of the object-oriented programming paradigm.

As the name implies, the programming structure for an application is achieved through using "objects". Real world scenarios can be viewed as collections of objects that have certain characteristics and operations which can be performed on those objects. For example, automobiles are objects that consist of objects such as *engine*, *accelerator*, *brake*, *gear* etc. and perform operations such as *start car*, *depress accelerator*, *depress brake*, *shift gear* etc.

The object-oriented approach views the program as a "model" of the system being studied consisting of a set of interacting "objects", thus closely resembling real world situations. If the software product is produced according to object-oriented principles,

the expectation is that it can be easily modified, extended and maintained, thus promoting software reusability and interoperability of large complex systems.

As today's software is geared towards reusability, maintainability and extendibility, the object-oriented programming paradigm offers some advantages over traditional programming, making it a popular tool for developing complex systems in the software industry.

2.3. Requirements for Object-Oriented Programming Languages

In order for a programming language to be classified as an object-oriented language, certain requirements must be fulfilled. Some of these are the support for the creation of objects and abstract data types, strong typing, encapsulation such that information pertaining to specification and implementation are hidden appropriately, inheritance, genericity, concurrency, some form of message passing between modules, binding and the support for polymorphism characteristics (see Fig. 2.1). [Nielsen 1992]

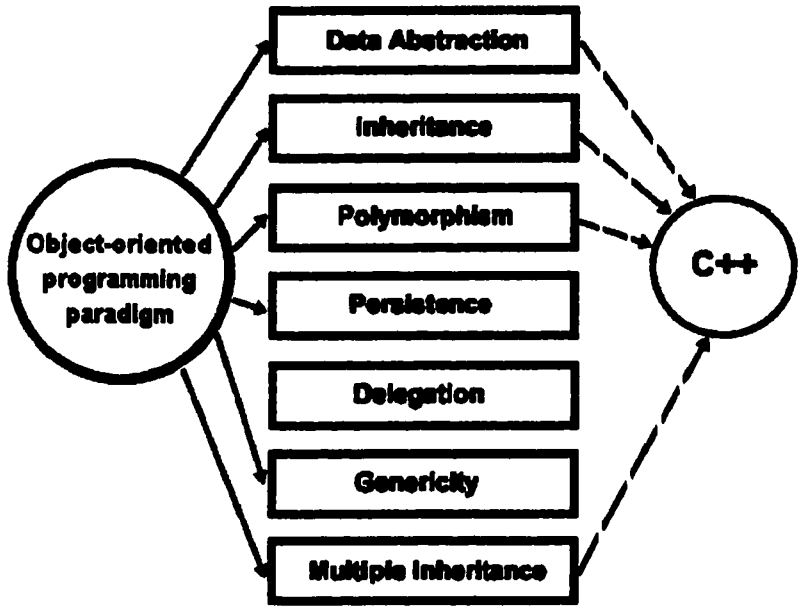


Fig. 2.1: Object-Oriented Programming Language Requirements

Although most programming languages do not exhibit all the requirements, some languages such as C++, Smalltalk etc. are capable of displaying many of these features and hence are described as object-oriented programming languages.

3. Principal Features of Object-Oriented Programming

3.1. Objects

The concept of an object in an object-oriented programming language (e.g. C++, Simula) is that it acts as a mechanism for modeling the problem domain of real world situations. The entities in the problem domain are modeled as objects and the operations pertaining to real world situations are modeled as methods for those objects.

An object can appropriately change its state, be manipulated, or relate to other objects in the system according to the integrity, operations and invariant properties defined for that object in relation to other components in the system.

Booch [1991] has defined an object as an entity that can be abstracted, classified or categorized by its type, i.e. an object is an instance of a class. The class of an object has associated with it a set of operations that are allowed on the objects belonging to that class. An object is denoted by a specific name that provides a unique identity for the object and allows the user to distinguish it from other objects.

3.2. Classes

In object-oriented terms, each object in the system is viewed as an instance of a class.

A class is defined as a collection of objects possessing similar attributes used to determine their compatibility in a system. It is an implementation of an abstract data type (ADT) that characterizes the behavior of an object as a set of properties and a list of operations that can manipulate these properties. A class exhibits the property of data encapsulation when the data in an object cannot be manipulated directly by other objects in the system.

According to Henderson-Sellers [1992], a class is composed of features that can be either attributes or methods. Attributes are defined as the class' variables that represent the characteristics of the class rather than its behavior. Methods can be divided into functions which query the state of the object, and procedures which change the state of the object. The figure below illustrates the composition of a class.

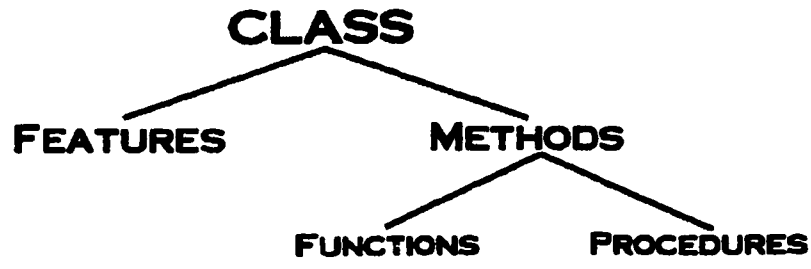


Fig. 3.1: Composition of a Class

A class has two views associated with it, an external view and an internal view. The external view refers to the class specification and is comprised of interface functions which define the operations that can be performed on instances of the class. The internal view represents the implementation of the class and is comprised of details pertaining to data representation and implementation of the interface functions. It is quite possible for a programmer to create more than one version of a class which implements the same ADT, since the ADT is the specification and the class is the implementation of the ADT.

In an object-oriented programming language, the concept of a class is supported as one of the basic constructs in the language for the implementation of an object. The purpose of class creation in an object-oriented programming environment is to reduce the complexity of a large system through the use of different techniques.

Through the principle of modularity, a problem can be partitioned into smaller manageable tasks. Independent entities are created and can be developed separately, thus providing some sort of framework for the creation of software components that can be easily modified and extended without a major redesign process. Also, by using classes within a given application, software components are created that can be reused. This may reduce the overall cost of the software product if a series of similar or related products are constructed.

The abstract properties of an object and its operations are implemented as variables and functions in the programming language and are defined in the class interface. A class's variables and functions are called *member variables* and *member functions* to differentiate them from other variables and functions that are not part of a class in C++. A syntax for class specification is as follows:

class class_name

{ *private*: declarations of the private members of the class

public: declarations of the public members of the class

};

In the class definition, any member functions declared after the keyword "private" are accessible only to other members of that class. The member functions declared after the keyword "public" are the interface functions or methods for the users of the class and are accessible from outside the class. Fig. 3.2 illustrates these concepts.

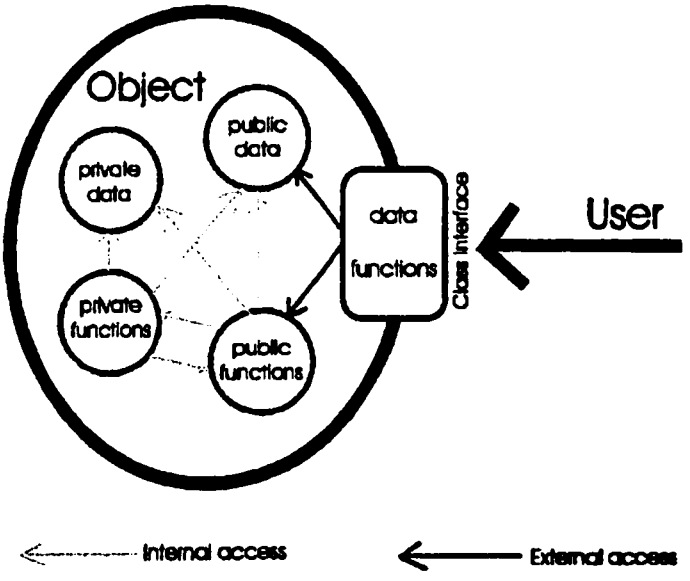


Fig. 3.2: Data Encapsulation

3.3. Abstract Classes

Abstract classes are used for the purpose of deriving other classes, a method called subclassing in object-oriented programming. By using object-oriented concepts, the subclassing approach implies that new classes of objects can be defined through the process of inheritance, encapsulation and behavior modification of an existing class. Classes can also be derived from concrete classes.

An important point to note is that a class acts as a pattern for objects whereas an abstract class acts as a pattern for classes. Abstract classes do not have implementation details and thus cannot be instantiated. By identifying common behavior and characteristics between various classes, a hierarchy of classes can be formed with an abstract class as the root. Fig. 3.3 shows how abstract classes called Shape, Vehicle, Bank Account were used to derive concrete classes such as Square, Rectangle, Polygon, Car, Bus, Checking Account, Savings Account, etc.

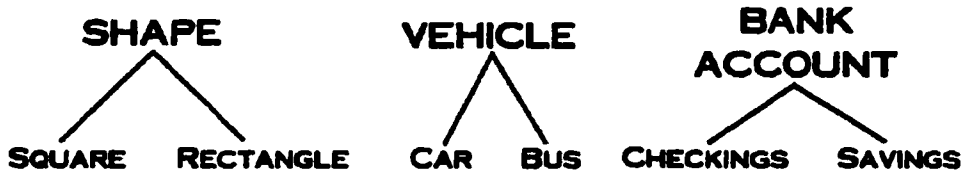


Fig. 3.3: Abstract Classes

Other characteristics of object-oriented programming include abstraction, encapsulation, inheritance, polymorphism and dynamic binding.

3.4. Abstraction

One of the fundamental ways to cope with complexity is through the process of abstraction which is a high level description of a complex entity. Abstraction can be defined as follows:

> "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." [Booch 1991]

Data abstraction allows for the definition of abstract operations on abstract data types. Objects are instances of a class and they encapsulate the data called the instance variables which are accessible only through the interface functions. Thus data abstraction promotes modular programming where classes are encapsulated modules and the internal states of their instances are manipulated only through accessible operations.

3.5. Data Encapsulation

Encapsulation is concerned with information hiding, which is the ability to hide implementation details while providing a public interface to the "client". Various application modules using the objects are loosely coupled, and changes to the implementation detail of the object can be made without major modifications rippling through the entire system (which is often the case in structured programming). Without changing their interfaces, encapsulated objects can be reused in different systems.

3.6. Inheritance

Through the process of inheritance, new classes can be derived from existing ones. The existing class that serves as a mechanism for inheritance is called a base class or superclass, and the new class that is derived from the base class is called the derived class or subclass.

If a subclass is derived from one superclass, it is called single inheritance. If a subclass is derived from more than one superclass it is referred to multiple inheritance. Also, there can be more than one derived class from one base class.

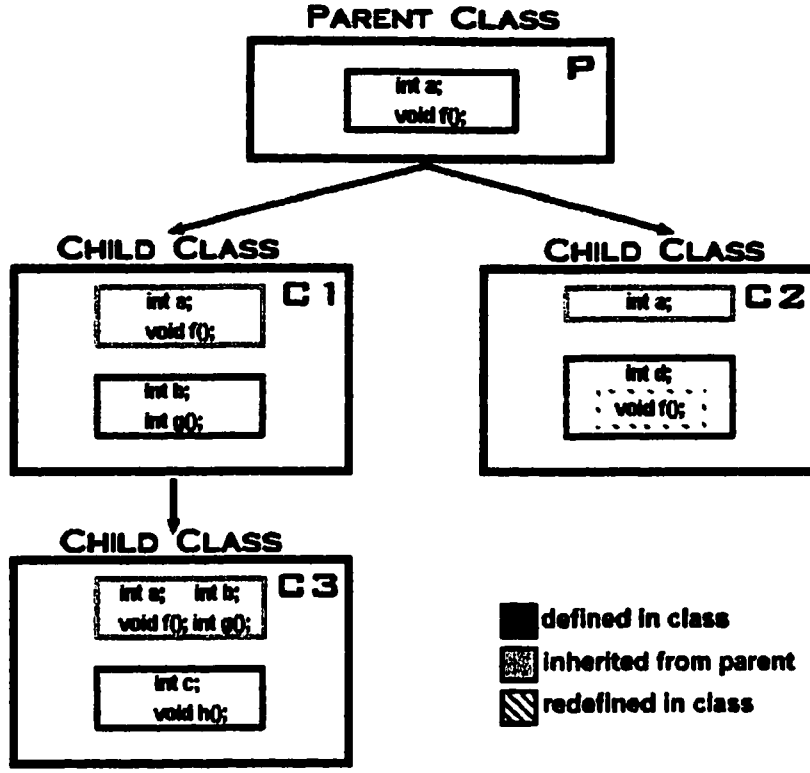


Fig. 3.4: Inheritance and Polymorphism

Usually, a base class defines a generic class of objects while a derived class defines a specific class of objects. Thus one can create a hierarchy of related types where code and interface can be shared. The derived class inherits the properties of the base class and is a technique used for coping with complexity. Within a class hierarchy, one can identify specialization and generalization processes in moving from leaf to root. Specialization is the process of creating new specialized subclasses from the existing class and generalization is the process of creating a superclass or parent class (see Fig. 3.4).

3.7. Polymorphism and Binding

Polymorphism enables one method name to be associated with many different actions. This phenomenon occurs when a concrete operation inherits its definition and properties from a generic operation, such as a name denoting objects of many different classes all related by a common base class. Polymorphism allows a programmer to provide the same interface to different objects. According to Rao [1993], there are three different varieties of polymorphism:

> "Inclusion polymorphism: An object may belong to many different types that need not be disjoint. The object type may include one or more related types, as found in subtyping. In the class hierarchy, objects belonging to a class in the hierarchy are manipulatable as belonging not only to that type, but also to its supertypes. Thus, certain operations on objects can work not only on objects of the subclasses but also on objects of the superclasses.

> "Parametric polymorphism: An implicit or explicit type parameter is used to determine the actual type of argument required for each of the polymorphic applications. Thus, the same operation can be applied to arguments of different types.

"Ad hoc polymorphism: When a procedure works or appears to work on several types, it is called ad hoc polymorphism. It is similar to overloading, and not considered to be a true polymorphism."

Depending upon the processing time, polymorphism can be either static (resolved at compile-time), or dynamic (resolved at run-time).

> "Polymorphism at compile-time is a form of overloading and allows a derived class to re-implement a service already implemented in its parent class; objects of the derived class will then execute the derived class' version of that service, while objects of the base class still act according to the original definition of the service. At runtime, polymorphism allows an object to reference objects of varying class without having to specify the exact class of the referenced object." [Wieland 1990]

Consider the example illustrated in Fig. 3.5.

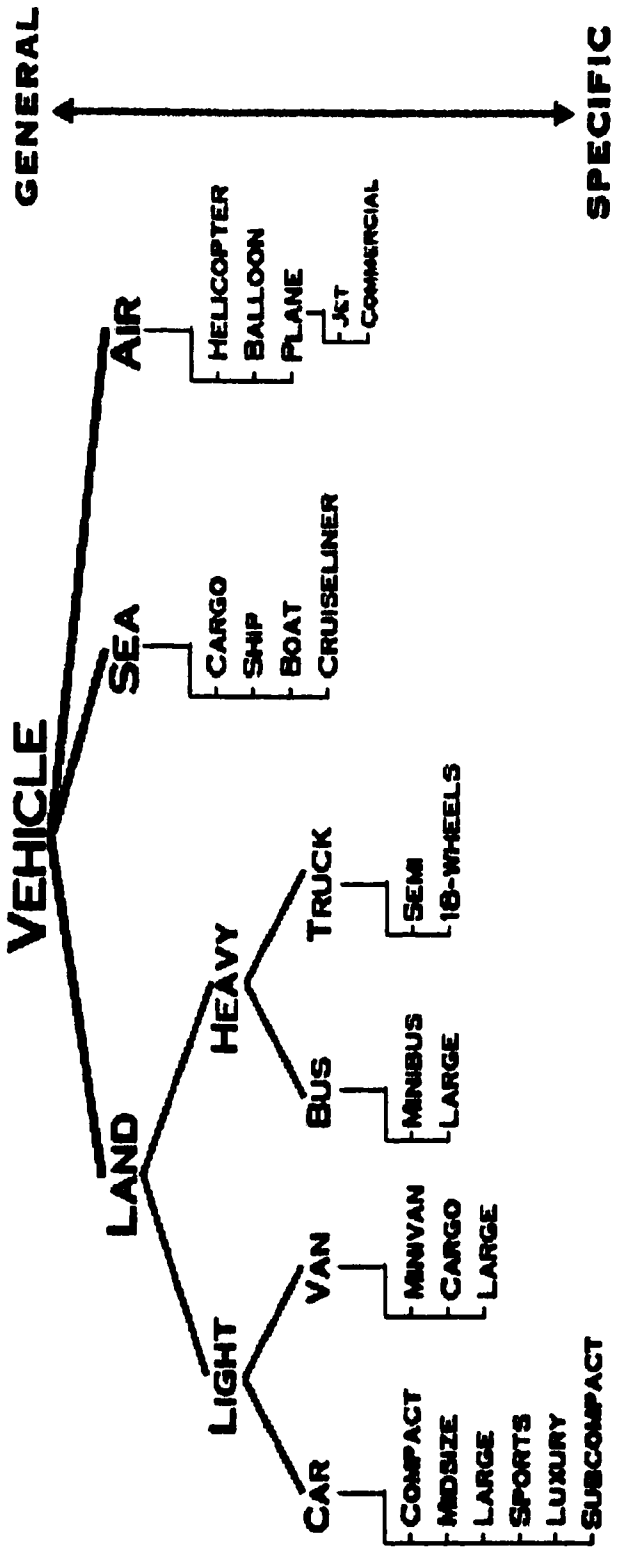


Fig. 3.5: Class Hierarchy, Inheritance and Polymorphism

Vehicles can be classified into light and heavy vehicles. Light vehicles can be car or van. Similarly, heavy vehicles can be bus or truck. For instance, an operation such as *start automobile* may be defined for all vehicles. Thus by inclusion, *start automobile* will be applicable to car, van, bus or truck. But an operation such as *accelerate* may be defined for car and bus. In this case, there is no concept of inclusion between car and bus. Thus, the *accelerate* operation must be interpreted in the context of each call. By using parametric polymorphism, the *accelerate* operation could be interpreted generally to apply to all automobiles.

Thus inclusion polymorphism and parametric polymorphism can be applied to these scenarios depending upon the definition of the operation which lies in the hands of the programmer developing the application.

Another characteristic feature of object-oriented programming is binding. Binding refers to the time at which a decision is made concerning what function to execute. There are two forms of binding--static and dynamic.

In static binding the decision as to which function to be called is made by the compiler at runtime and depends mainly upon the scope rules of the language. In dynamic binding the decision of which function to apply to an object is delayed until the last possible moment namely to the time of execution of a program when a function is actually called and this is dependent upon the class of the object itself. [Wieland 1990]

4. Benefits of Object-Oriented Programming

The key elements of object-oriented programming are object, class and inheritance. Using these concepts give rise to certain benefits over traditional programming methodologies.

Object-oriented programming can be viewed as one of the latest tools in the software methodology that help the programmer to accomplish the task of problem solving. Object-oriented programming exhibits different mechanisms that support the following features:

- A high level of modularity for the support of loose coupling methods
- Encapsulation of data to support information hiding techniques
- Data abstraction for object creation
- Classification and inheritance mechanisms that provide reusability and extendibility.

For a given application, by concentrating on data and behavior through the process of modularity, different components can be identified as self-contained entities, each consisting of a set of data and operations to manipulate it. Each of these components are described as an object, an instance of a class. The advantage of using objects is that the data is encapsulated and implementation details are hidden from outside manipulation except through defined operations at a given interface level. Data

abstraction helps the programmer to represent changes to objects and their behavior without affecting a drastic change to the entire system.

The characteristics of polymorphism along with dynamic binding allow a programmer to concentrate on the operations performed on a data object without worrying about the precise effect of the operation at runtime. Thus an application is flexible in case there are later changes to be made to it. In addition, any changes made to a class specification is limited to one place, namely the class itself, and do not cause a ripple effect throughout the entire program wherever an object of that class has been used.

The "class" construct in object-oriented programming supports abstraction and modularity of an application by serving as the unit for modules. Through the use of classes, it becomes easier to tackle complex problems by splitting them up into smaller modules.

Through the process of inheritance, relationships and commonalties between classes can be brought out in the class framework through related class hierarchies where classes inherit features from their superclasses. Also, there is support for differential programming where a new class of objects can be created by making few changes to an existing class. Any enhancements made to the base class are propagated automatically to its subsequent derived classes.

Object-oriented techniques provide a framework to define a class hierarchy where the higher levels define general properties such as abstract concepts, common protocols, shared code, and the lower levels define specific properties such as concrete implementations of applications for complex systems.

There is the benefit of reusability in object-oriented programming. It seems more feasible to write code once and have it used again and again rather than starting from scratch or "reinventing the wheel" every time an application is developed. This can be achieved by placing more emphasis on creating reusable classes that are well tested and documented for future applications.

Inheritance, encapsulation, polymorphism and dynamic binding enable the reuse and sharing of code between related classes and support flexibility, refinement and modification of existing applications and systems.

5. Frameworks in Object-Oriented Programming

5.1. Definition of a Framework

A framework is a collection of concrete and virtual classes that defines a software architecture intended to be reused for a specific purpose by different users and applications (see Fig. 5.1).

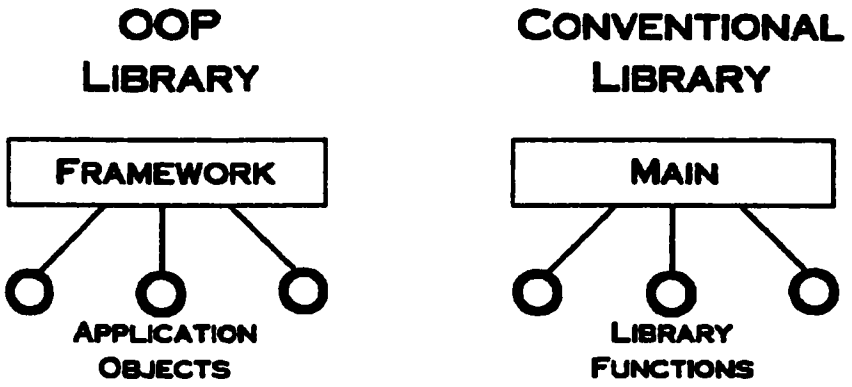


Fig 5.1: Contrast between OOP and Conventional Libraries

5.2. An Example: MVC Framework.

The Model-View-Controller (MVC) framework was designed as a tool to help programmers develop graphical user interfaces (GUI) through the reuse of software components paying particular attention to the presentation and interaction aspects of the application. An important point is that the Model, View and Controller are independent.

One can improve software productivity by reusing user interfaces for different applications and providing support for multiple ways of viewing and interacting with the application model. For example, for a given model, some users may prefer to view their data in the form of tables while other users may prefer viewing their data as graphs. Different views can be mapped onto one model for a given application. Through this process, the development cost involved in implementing the model is incurred only once and different models can be presented with similar user interfaces.

The Model-View-Controller principle allows for the systematic development of an interactive application. Any interface within this type of framework consists of three components:

Model: A collection of objects representing the application domain of the user interface.

View: A collection of objects that contribute to the user interface. It is a specification of how various aspects of the model are presented to the user.

Controller: A collection of objects that control the flow of information between the model and the view. It specifies how the user can interact with the application by requesting changes either in the view or the model.

These three components constitute the MVC framework. The controller "C" is responsible for handling the user input and communicates with the model "M" and view "V" via message passing. Fig. 5.2 illustrates the MVC framework.

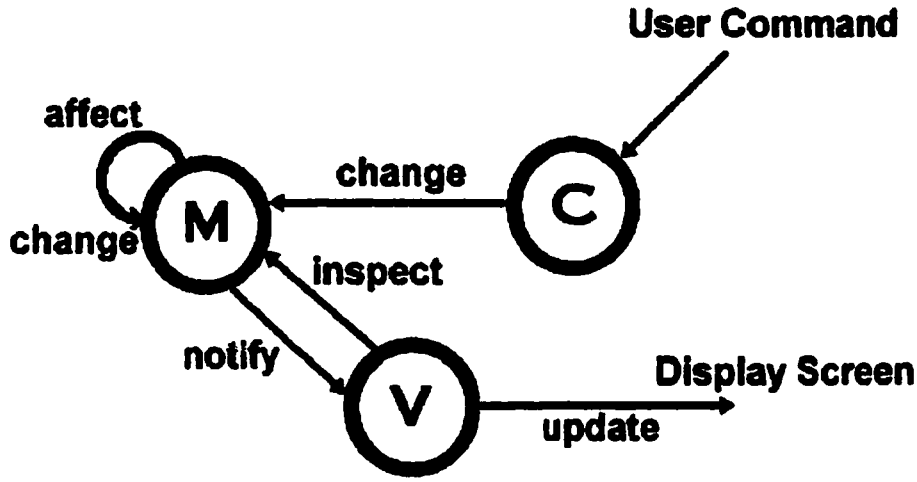


Fig. 5.2: MVC Framework

For example, a user issues a command to change data. The controller receives the user's command and sends a message to the model requesting a change. The model executes the appropriate procedures pertaining to the user's command and then notifies the view that it has changed. As a result, the view inspects the current state of the model and updates or redisplays the effects of a change in state to the user on the screen. A key point to note is that the Model and View are independent.

A typical process for developing a software application under the MVC framework is for the software developer to focus on the design and implementation

components with respect to the model. After completion of the model, the developers search their library for reusable user interface components and select the appropriate ones according to the user's requirements. Thus by varying the user interface, different implementations are produced quickly.

In order to use the MVC approach, the system is decomposed into three categories of objects, namely Model, View and Controller objects. Objects described in the Model form the main composition of the system and are responsible for informing Views and Controllers of changes in their state. Objects defined in the View are responsible for graphical representations of a Model as defined in the specification of the system. Objects defined in the Controller are responsible for controlling the flow of information between Model and View, handling user interaction, etc.

5.3. Advantages of the MVC framework

Using the MVC framework offers the advantage of multiple viewing, development productivity and quality for the development of an application or system. [Goldberg 1990]

Multiple Viewing

As the Model and View are independent, a model can be mapped onto several interfaces. For example, a cruise control system model can have different graphical user interface such as analog, digital, graph etc.

Development Productivity

While retaining the model, it is possible to create new views and controllers through the refinement of existing ones. Thus the reusability of existing MVC framework components can significantly reduce the amount of programming required in order to develop a system.

Quality

Through the reuse of existing components the quality of these components improves as they undergo a refinement process under different circumstances and environments.

6. An Application of Frameworks Control Systems

(Used in simulation of engines, heating systems, etc.)

Given an application for a control system, a framework can be constructed and then object-oriented concepts applied to develop the system in an object-oriented methodology. But do control systems have enough common features for a framework? To answer this question, two simplified versions of real world applications are discussed--an automobile cruise control system and a home heating system. The application is described in terms of the MVC framework which is good for simulation.

6.1. Application: Automobile Cruise Control System

An automobile cruise control system tries to maintain a cruising speed set by the driver over varying terrain. There are external and internal factors affecting the cruise control system. Some external factors are those pertaining to the environment such as weather conditions, gradient of the road, condition of the road surface, friction, and temperature of the environment. Internal factors are those that relate to the system itself such as speed, accelerator pedal, brake pedal, vehicle weight, gear position etc.

6.1.1. Model

A model is defined by those classes which are specific to the components to which it belongs. For example, the model associated with the cruise control system is an instance of the World, Car and Cruise classes, which are all subclasses of the Model class.

A model of the cruise control system can be constructed by examining the features that affect the system.

External factors

World Conditions

Gradient - The gradient of the road surface (e.g. 35deg)

Weather - The driving weather conditions (e.g. clear, foggy, snow, rain)

Road surface - The condition of the road surface (e.g. smooth, icy, wet)

Temperature - The temperature of the environment (e.g. 20C, -30C)

Friction - The force acting against the car

Internal Factors

Car Behavior

Set speed - The speed set by the driver (e.g. 100km/h)

Pedal Change -The pedal change determines whether the car is braking or accelerating to meet the set speed of the vehicle depending upon the external factors of the environment.

Gear - The gear position of the car (e.g. 1,2,3,4)

Mass - The weight of the car (e.g. 1000kg)

Acceleration needed = Set speed - Current speed

The behavior of the car is simulated according to the environment factors and tries to meet the desired speed as set by the user. The operations performed on the model are Update World and Update Car.

6.1.2. View

The view constitutes the visual interface between the user and data. It describes the various displaying formats the user can choose to view the data of the application. For example the view can be in the following formats: digital, graph, analog etc.

6.1.3. Controller

The controller is responsible for controlling the flow of information between the user and the application. The main operations in control systems are usually update operations that produce a change on the data as desired by the user of the application. In our example, the controller is responsible for updating the Model and View.

6.2. Application: Home Heating System

The function of a home heating system is to regulate the flow of heat in the home. The user sets the desired temperature of a room in the home. Depending upon an external factor such as environment temperature, heat dissipates accordingly to the environment. By sensing the amount of heat required, the system controls heat flow to a room.

6.2.1. Model

A model is defined by those classes which are specific to the components to which it belongs. For example, the model associated with the heating system is an instance of the World, Room and Thermostat classes, which are all subclasses of the Model class.

A model of the heating system can be constructed by examining the features that affect the system.

External factors

Temperature - The external temperature of the environment

Internal factors

Temperature - The internal temperature of the room

Furnace - The component that generates heat to the room

Volume - The capacity of the room

Thermostat - Sets the desired temperature of the room

Heat needed = Set temperature - Internal temperature

The behavior of the heater is simulated according to the environment factors and tries to meet the desired heat in a room as set by the user. The operations performed on the model are Update World and Update Room.

6.2.2. View

The view constitutes the visual interface between the user and data. It describes the various displaying formats the user can choose to view the data of the application. For example the view can be in the following formats: digital, graph, analog etc.

6.2.3. Controller

The controller is responsible for controlling the flow of information between the user and the application. The main operations in control systems are usually update operations that produce a change on the data as desired by the user of the application. In our example, the controller is responsible for updating the Model and View.

In describing these control systems, the common features are identified. The common feature of control systems is that they control the behavior of the system by

monitoring variables in an environmental setting and attempt to achieve the desired condition according to the user's request.

For example, a cruise control system controls the car's behavior by maintaining the user's desired speed (100km/h) while being affected by external factors. Similarly, in the case of the home heating system, the room's behavior is controlled by maintaining the user's desired temperature (30C) while being affected by external factors.

In the model, the common features are external ones that pertain to the environment conditions and internal ones that pertain to the behavior of the system under these conditions. The operations on the model are update external and internal features of the system. For example, abstract classes can be defined for control systems such as World, System. Depending upon the application, concrete classes such as cruise control, heat control, navigation control, etc., can be developed accordingly (see Fig 6.1).

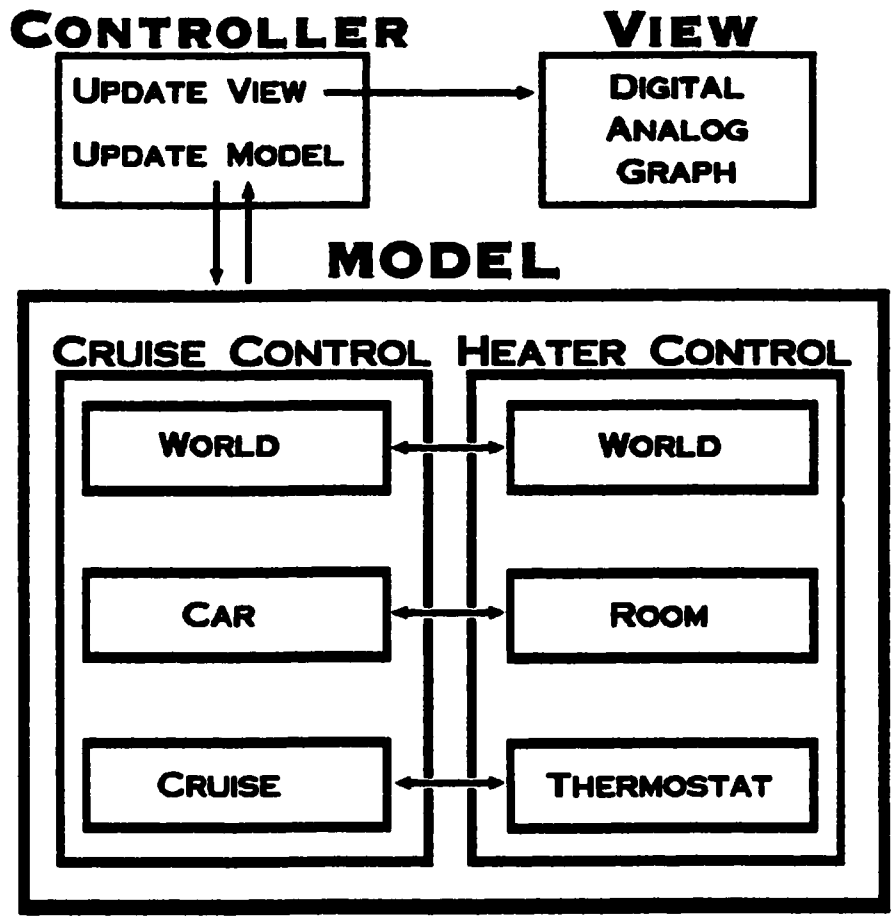


Fig. 6.1: Common Features of MVC Application Examples

7. Results, Problems Encountered and Solutions Found

As a result of this exercise, it can be seen that control systems used in the simulation of engines, heating systems, etc., do have enough common features to constitute a framework.

The following classes can be constructed: a Model that represents the application domain of the system, a View that deals with displaying information to the user and a Controller that acts as the interface between the user and the system through a set of operations such as Update Model and Update View. Abstract MVC frameworks can be constructed for control systems using their common features and specific applications can be derived from them. These common features are central to control systems and thus an architecture such as the MVC framework serves as an initial guide and the development of control systems as the final objective.

The MVC framework can be used to simulate control systems, for graphical user interface representations, incorporation of different Views and Controllers without affecting a drastic change on the Model, etc. The MVC triple is easier to understand and programmers can make necessary changes to an application by concentrating on the respective category Model, View or Controller. The key to reusability in software methodologies lies within the object-oriented paradigm and the MVC framework.

Control systems are unique in the sense that the behavior can be captured by controlling various factors affecting the system. In the case of the cruise control system, the desired speed is fixed while other factors change the state of the control system accordingly. In the case of the home heating system the desired room temperature is fixed while external factors affect the control system and change its state accordingly. The behavior of these systems can be studied by controlling various factors respectively. Using the MVC framework for control systems helps to understand the system easily and facilitate necessary changes in moderation without major complications throughout the entire system.

7.1. Problems Encountered in Using Object-Oriented Programming to Develop a Framework for Control Systems

Some of the problems encountered in developing this exercise for control systems are described below:

Minor problems:

- Understanding the concepts of an object-oriented language such as C++
- Describing the model in object-oriented concepts.
- Trying to identify the objects in the system.

Moderate problems:

- How to separate the objects to fit the MVC framework.
- For a given class, what attributes and methods to associate with that class.
- Redundancy of class methods.

Major problems:

- Describing abstract classes for the system.
- Changes to a class led to problems that propagated to subsequent classes causing a rippling effect throughout the model.
- Incorporating the notion of reuse into the application.

The solutions to these problems were obtained by decomposing the entire system into subsequent modules and tackling each one separately. For example, the cruise system can be broken down into objects such as world, cruise, car, etc. each with their own attributes and methods. The system was matched on to the MVC framework by describing the decomposed modules within the appropriate triple--Model, View and Controller category.

By finding solutions for smaller modules, the problems that initially came about in developing the system were resolved. This feature of object-oriented programming helps to think of the application at hand without worrying about unnecessary details that hinder programming techniques.

Other problems that can be foreseen in the development of large scale applications of systems are:

- Inheritance and polymorphism techniques can add to the complexity of a software product thus making its maintenance more difficult and costly.
- A class hierarchy can become quite complex, leading to difficulty in locating subclasses within a particular hierarchy. A solution to this problem is the requirement for a browser that can rapidly locate the hierarchy or path of an entity contained in a low level subclass.
- However, the compilation times may be significantly longer and the use of dynamic binding may take extra execution time compared to the use of more traditional languages.

Nonetheless, the benefits of object-oriented programming far outweigh the problems.

7.2. Future Prospects of Programming Paradigms in Software Engineering

Looking back to the history of programming languages, there is a progression from assembly languages to procedure oriented languages to object-oriented languages. But as we approach the 21st century, what language comes after object-oriented languages? Is this the end of the line for programming languages? Wegner [1990] views object-oriented programming as the ultimate buzzword.

> "According to this view, object-oriented programming is the culmination of an evolutionary process, there is no need for further evolution, and we have reached the end of history."

As is discussed below, this is not entirely true.

Fig. 7.1 shows the progression of programming languages and possible scenarios for the future of programming languages.

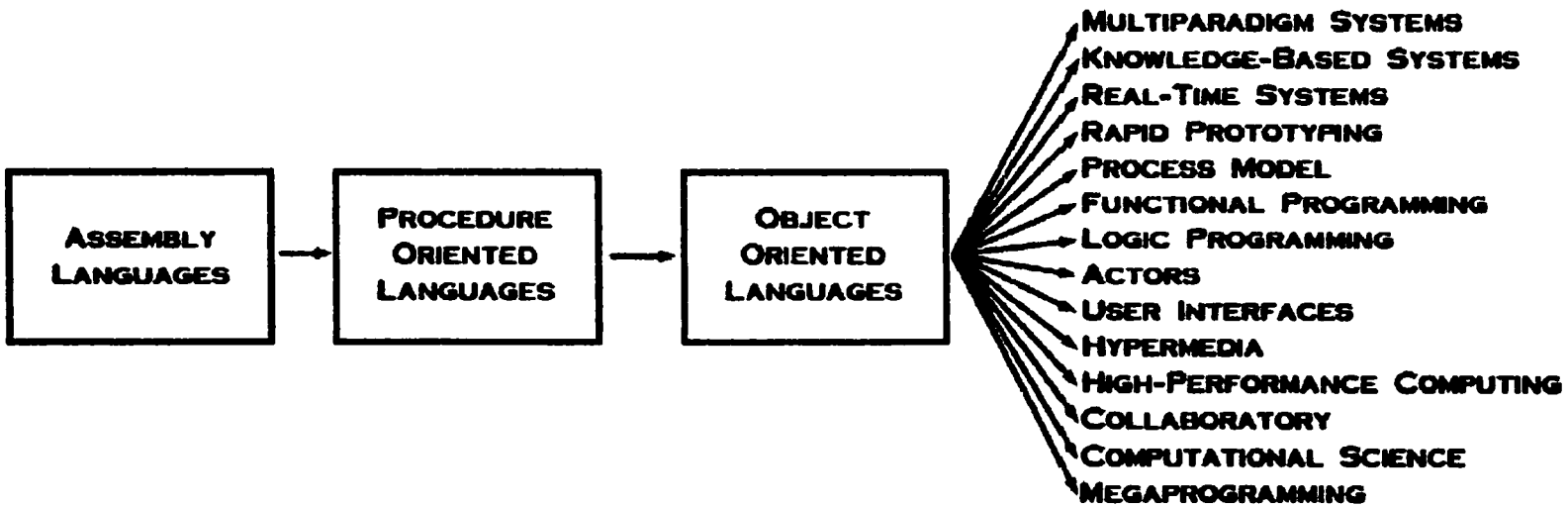


Fig. 7.1: Possible Scenarios for the Future of Programming Languages

8. Conclusions

Programming languages have progressed a long way since their invention. Today we find that object-oriented programming is the preferred choice for programming in the 1990s and perhaps even for the beginning of the twenty-first century. This is due to the fact that this paradigm with its unique style of programming offers significant advantages over the conventional paradigms in software engineering.

Through the use of object-oriented concepts, the program application models the problem domain more closely. The data and behavior of a system are captured through encapsulation, abstraction, and information hiding techniques. Inheritance relationships, polymorphisms, and dynamic binding help to provide greater flexibility, interoperability, extendibility and reusability which are all important goals in the software engineering industry. Software developers are concerned with maintenance costs and reliability of systems in the long run.

Although today, object-oriented programming is popular, other styles of programming will not become extinct. Object-oriented programming is not suitable for all applications since, like all programming paradigms, it too has some limitations. Programmers are brainwashed into believing that object-oriented programming can be used for all purposes in solving problems, but that is not the case. Sometimes the

advantages of object-oriented programming are exaggerated to give the notion that this method of programming possesses power over all programming paradigms.

In using this form of programming we have to take advantage of the benefits offered but pay close attention to the reusability feature. The development of object-oriented software libraries can help in the process of reusing pieces of code from the library for the development of a new application. The concept of reuse is easy to understand but the practical application of reusing object-oriented programs for software system development is quite difficult.

The MVC framework serves as a guide for control system development. They have enough features to constitute a framework. This type of framework provides an excellent interface between user and application. The Controller interacts with the user's commands by appropriately notifying the Model and View of desired changes made to the application. The control applications described were meant to show that in real world situations, the commonality of systems is an important feature to address when developing applications for systems.

Wegner's claim that object-oriented programming is the ultimate buzzword in software engineering methodology can be disputed. When procedure oriented languages were developed, programmers thought that it was the final solution to all programming paradigms. But now we see that this is not the case.

Similarly, object-oriented programming might seem like the fad for now, but one should not rule out the possibility that another paradigm more powerful than object-oriented programming may emerge from object-oriented programming concepts and enter the domain of programming languages.

The object-oriented programming paradigm is a very interesting concept in the programming methodology. Many programmers are used to the traditional top down style of programming and do not know how to apply the methodology of bottom up object-oriented programming, which can be a deterrent to the development of efficient programming techniques.

As the software industry accepts this style of programming as an important key to the development of complex software systems, programmers and users once educated and exposed to this methodology will begin to appreciate object-oriented programming more readily along with its benefits as one complete package.

Much of computer programming is aimed at the simulation of complex physical and abstract systems. In order to design, construct and communicate these complex systems to the users, appropriate tools are required for the development of successful systems. The object model is a useful concept and tool, providing guidelines to categorize the different entities (abstract and concrete) appropriately in order to develop the system efficiently.

In summing up what it means to use object-oriented programming,

> "Writing a program in classical languages has been compared to writing an essay. Writing an object-oriented program is then comparable to writing a drama. Every object has its own behavior and acts in accord with it. So every object-oriented program becomes a simulation of the real world, an executable logical system that imitates and reflects real life objects. Object-oriented programming is the best way to program because the gap between reality and object-oriented programs is the least possible." [Savic 1990]

References

Atkinson, C. *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley Publishing, 1991.

Blair, Gordon S., John J. Gallagher, and Javid Malik. "Genericity vs Inheritance vs Delegation vs Conformance vs...", *J. Object-Oriented Programming*, vol. 2, no. 3, pp. 11-17, September/October 1989.

Booch, G. "Object-Oriented Development", *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 211-221, February 1986.

Booch, G. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

Ege, K. R. *Programming in an Object-Oriented Environment.* Academic Press, 1992.

Goldberg, A. "Information Models, Views, and Controllers", Dr. Dobb's Journal, vol. 15, no. 7, pp. 54-174, July 1990.

Henderson-Sellers, B. "Object-Oriented Information Systems: An Introductory Tutorial", *The Australian Computer Journal*, vol. 24, no. 1, pp. 12-24, February 1992.

Lorenz, M. *Object-Oriented Software Development: A Practical Guide.* Prentice Hall, 1993.

Masini, G., A. Napoli, and D. Colnet. *Object-Oriented Languages.* Academic Press, 1991.

Meyer, B. "Reusability: The Case for Object-Oriented Design", *IEEE Software*, pp. 50-64, March 1987.

Nielsen, K. *Object-Oriented Design with Ada*. Bantam books, 1992.

Nygaard, K. "Basic Concepts in Object-Oriented Programming", *ACM SIGPLAN Notices*, vol. 10, no. 2, pp. 128-132.

O'Shea, T., K. Beck, D. Halbert, and K. Schmucker. "The Learnability of Object-Oriented Programming Systems", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 502-504, September 1986.

Rao, B.R. *C++ and the OOP Paradigm*. McGraw-Hill Inc., 1993.

Savic, D. *Object-Oriented Programming with Smalltalk/V*. Ellis Horwood Ltd., 1990.

Shaw, M. "Abstraction Techniques in Modern Programming Languages", *IEEE Software*, pp. 10-26, October 1984.

Smith, D.N. *Concepts of Object-Oriented Programming*. McGraw Hill Inc., 1991.

Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

Wegner, P. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger*, vol. 1, no. 1, pp. 7-87, August 1990.

Wieland, T. *An Object-Oriented Discrete-Event Simulation System with a Graphical User Interface*. Master's Thesis, Concordia University, April 1990.

Appendix A. Cruise Control System

This appendix shows relationship diagrams for class definition, state transition and a scenario for the cruise control system. The source code listing is also provided in later sections for the classes used in this application.

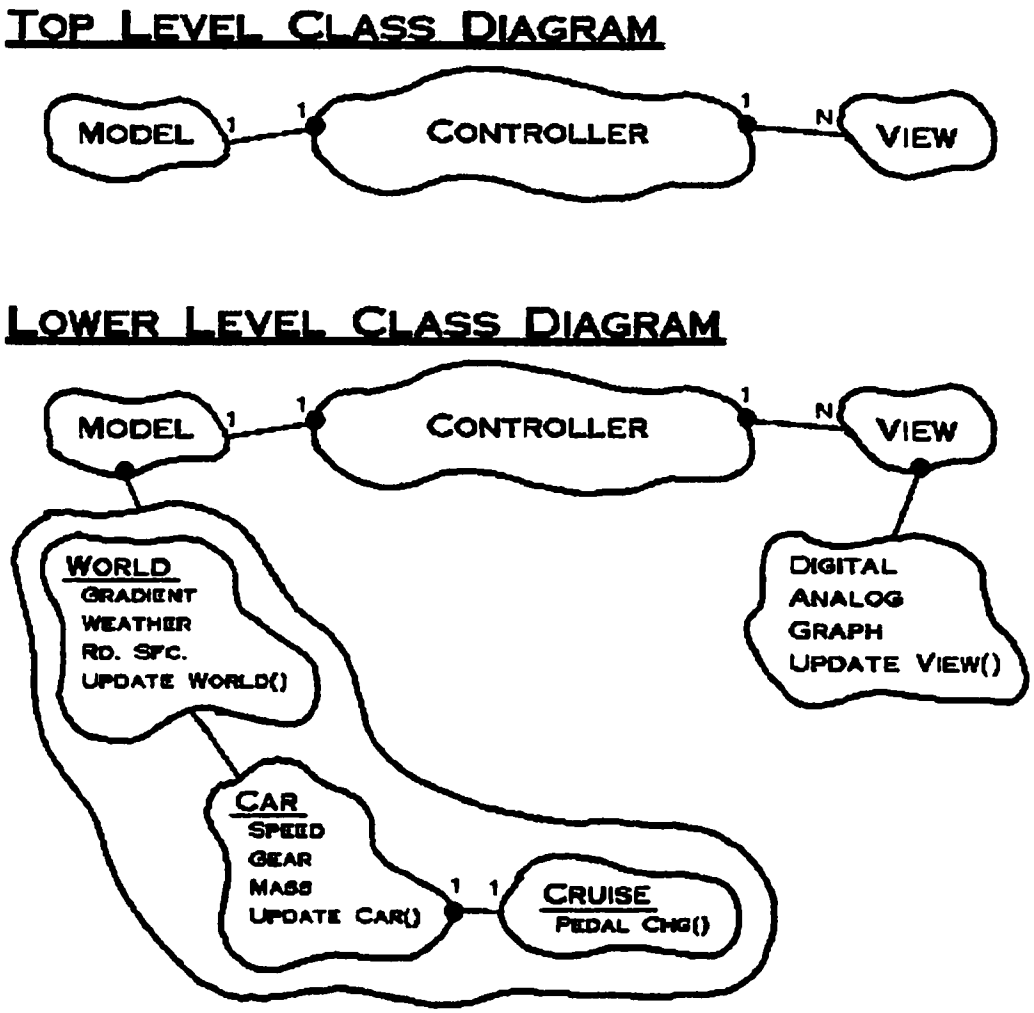


Fig. A.1: Cruise Control Class Definition

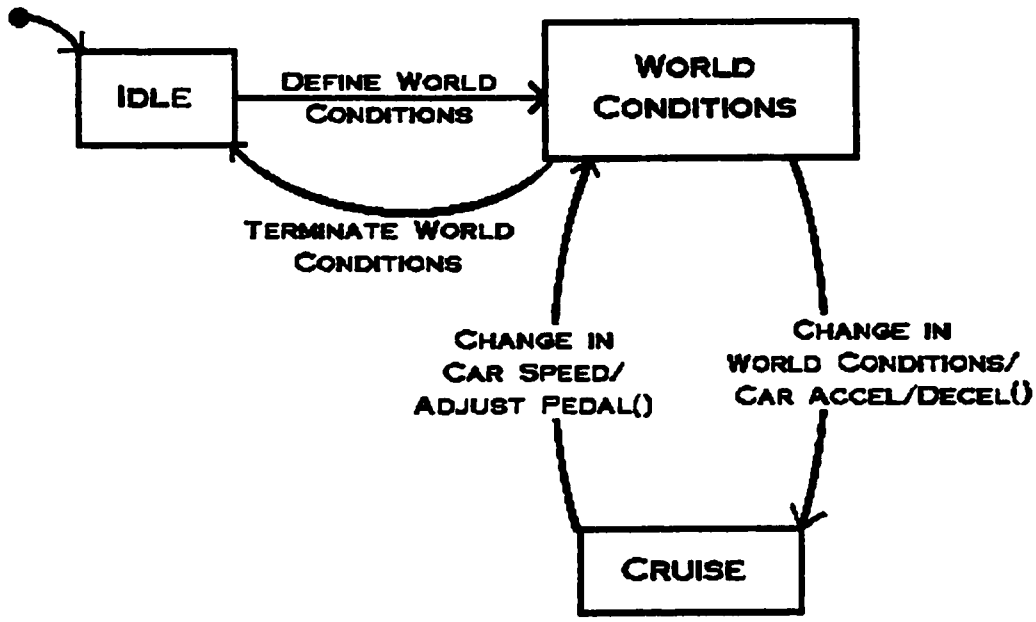


Fig. A.2: Cruise Control State Transition

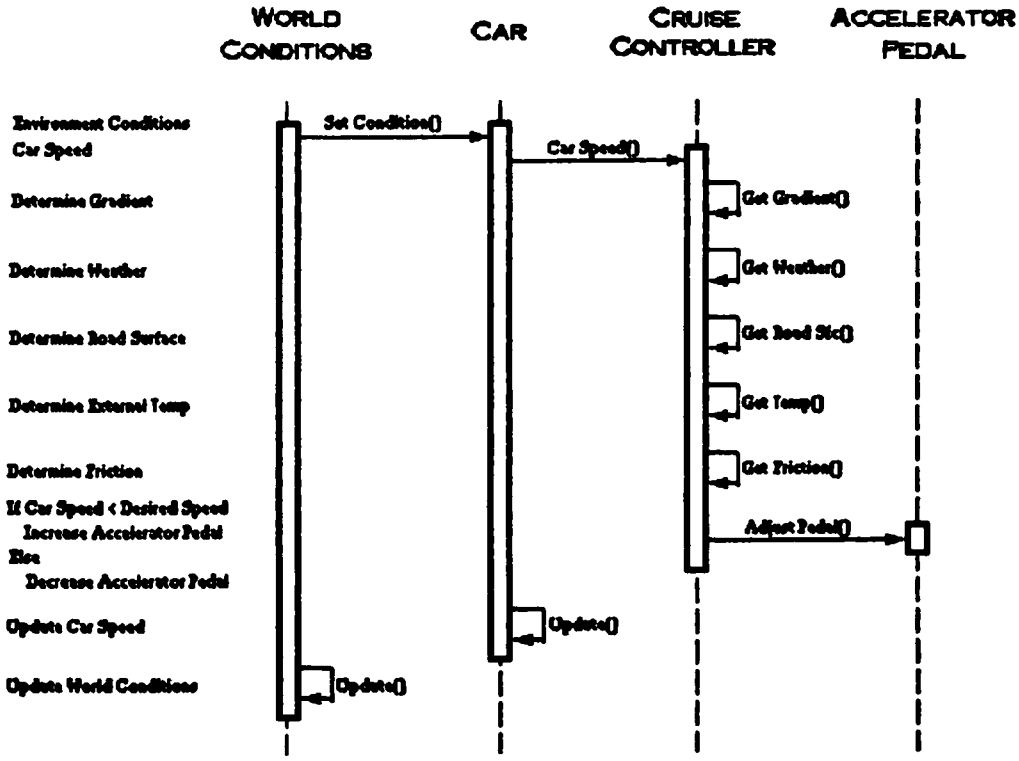


Fig. A.3: Cruise Control Scenario

A.1. Code Listing for Variable Definition

The variables used for the cruise control system application are defined below.

```
#define SPEED 100                    // Set speed by the driver

#define INITMINGRAD -25              // Initialize minimum gradient
#define INITMAXGRAD 25               // Initialize maximum gradient
#define INITMINWTH 0                 // Initialize minimum weather condition
#define INITMAXWTH .05               // Initialize maximum weather condition
#define INITMINRSRF 0                // Initialize minimum road surface condition
#define INITMAXRSRF .05              // Initialize maximum road surface condition
#define INITMINTEMP -10              // Initialize minimum environment temerature
#define INITMAXTEMP 30               // Initialize maximum environment temperature

#define CHNGGRAD .001                // Factor change in gradient
#define CHNGWTH .001                 // Factor change in weather
#define CHNGRSRF .001                // Factor change in road surface
#define CHNGTEMP .001                // Factor change in temperature

#define MINGRAD -25                  // Minimum gradient
#define MAXGRAD 25                   // Maximum gradient
#define MINWTH 0                     // Minimum weather
#define MAXWTH 1                     // Maximum weather
#define MINRSRF 0                    // Minimum road surface
#define MAXRSRF 1                    // Maximum road surface
#define MINTEMP -50                  // Minimum external temperature
#define MAXTEMP 50                   // Maximum external temperature
#define MINFRICT 0                   // Minimum friction
#define MAXFRICT 1                   // Maximum friction

#define INITMINSPEED 50              // Initialize minimum car speed
#define INITMAXSPEED 200             // Initialize maximum car speed
#define INITMINPEDAL -10             // Initialize minimum pedal change
#define INITMAXPEDAL 10              // Initialize maximum pedal change
#define INITMINMASS 1000             // Initialize minimum car mass
#define INITMAXMASS 5000             // Initialize maximum car mass

#define GEAR1 0                      // Set gear change according to car speed
#define GEAR2 5
#define GEAR3 25
#define GEAR4 75
#define GEAR5 120
```

```
#define MINSPEED 0                    // Minimum car speed
#define MAXSPEED 200                  // Maximum car speed
#define MINPEDAL -100                 // Minimum pedal change
#define MAXPEDAL 100                  // Maximum pedal change


struct DataField                      // Data Field
{
  char Name[80];
  float Val;
  float MaxVal, MinVal;
  int XCoord1, YCoord1;
  int XCoord2, YCoord2;
} Data[MAXFIELDS];
```

//

A.2. Code Listing for Class Definition

The classes for the MVC framework are defined: Model, View and Controller.

A.2.1. Model

The Model class has three subclasses: World, Car and Cruise.

```
class Model                           // Class Model
{
```

A.2.1.1. World Class

```
  class World                         // External factors
  {
    private:
    public:
      float gradient;                 // -25 upto +25
      float weather;                  // 0-fair upto 0.99-stormy
      float RoadSurface;              // 0-highway upto 0.99-dirt road
      float temperature;              // Temperature in centigrade (-50 upto +50)
      World()                         // Generate random environment conditions
      {
        gradient = Func.RandNum(INITMINGRAD, INITMAXGRAD);
        weather = Func.RandNum(INITMINWTH, INITMAXWTH);
        RoadSurface = Func.RandNum(INITMINRSRF, INITMAXRSRF);
        temperature = Func.RandNum(INITMINTEMP, INITMAXTEMP);
      }
```

```
    void Update(World *w)                    // Update class World

    {
      w->gradient = w->gradient + Func.RandNum(-CHNGGRAD, CHNGGRAD);
      if (w->gradient < MINGRAD) { w->gradient = MINGRAD; }
      if (w->gradient > MAXGRAD) { w->gradient = MAXGRAD; }

      w->weather = w->weather + Func.RandNum(-CHNGWTH, CHNGWTH);
      if (w->weather < MINWTH) { w->weather = MINWTH; }
      if (w->weather > MAXWTH) { w->weather = MAXWTH; }

      w->RoadSurface = w->RoadSurface + Func.RandNum(-CHNGRSRF,
            CHNGRSRF);
      if (w->RoadSurface < MINRSRF) { w->RoadSurface = MINRSRF; }
      if (w->RoadSurface > MAXRSRF) { w->RoadSurface = MAXRSRF; }

      w->temperature = w->temperature + Func.RandNum(-CHNGTEMP,
            CHNGTEMP);
      if (w->temperature < MINTEMP) {w->temperature = MINTEMP; }
      if (w->temperature > MAXTEMP) {w->temperature = MAXTEMP; }
    }

    float friction(World *w)                 // Calculate friction acting on the car
    {
      float frict;
      w->Update(w);
      w->temperature = (w->temperature < 0) ? w->temperature = -(w->temperature) :
            w->temperature;
      frict = ( ( (abs(w->temperature - 20) / 1000) + (sin(w->gradient*PI/180)) +
            w->weather + w->RoadSurface ) / 4) ;
      return(frict);
    }
  };
  World Montreal;                            // An instance of class World
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

A.2.1.2. Car Class

```
class Car                                   // Internal factors
{
  private:
  public:
    float speed;
    int gear;
    float pedal;
    float mass;

    Car()                                   // Generate random internal conditions
    {
      speed = Func.RandNum(INITMINSPEED, INITMAXSPEED);
      pedal = Func.RandNum(INITMINPEDAL, INITMAXPEDAL);
      mass = Func.RandNum(INITMINMASS, INITMAXMASS);
    }
    void SetGear(Car *c)
    {
      c->gear = 0;                          // Initialize gear

      if (c->speed >= GEAR1) { c->gear++; }
      if (c->speed >= GEAR2) { c->gear++; }
      if (c->speed >= GEAR3) { c->gear++; }
      if (c->speed >= GEAR4) { c->gear++; }
      if (c->speed >= GEAR5) { c->gear++; }
    }

    void Update(Car *c, World *w)           // Update Car and World classes
    {
      float accel;
      SetGear(c);                           // Set gear

      accel = ((exp((4.5) * log (c->gear)) * c->pedal) / c->mass) * 2;
      c->speed = c->speed + accel;
      c->speed = c->speed - (w->friction(w) * c->speed);
      if (c->speed <= MINSPEED) { c->speed = MINSPEED + 0.001; }
    }
};
Car Honda;                                  // Instance of class Car
```

A.2.1.3. Cruise Class

```
class Cruise                                    // Cruise control class
{
  private:
  public:
    float SetSpeed;
    Cruise()
    {
      SetSpeed = SPEED;                         // Set speed
    }

    void PedalChange(Car *c)                    // Calculate pedal change (brake or
                                                      accelerate)
    {
      int AccelNeeded;

      AccelNeeded = (SetSpeed - c->speed);      // Calculate acceleration needed
      c->SetGear(c);
      c->pedal = (c->mass * AccelNeeded) / exp((4.5) * log (c->gear));
      if (c->pedal < MINPEDAL) { c->pedal = MINPEDAL; }
                          if (c->pedal > MAXPEDAL) { c->pedal = MAXPEDAL; }
    }
};
Cruise Mobile;                                  // Instance of class Cruise
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
 private:
 public:

   Model()
   {
   }

   void UpdateDataBlock(struct DataField Data[])        // Update DataBlock
   {
   }
   void Update()                                // Update necessary classes
   {
       Montreal.Update(&Montreal);
       Honda.Update(&Honda, &Montreal);
       Mobile.PedalChange(&Honda);
   }
};
Model CarSimul;                                 // Instance of Model
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

A.2.2. View

```
class Display                                   // Class Display
{
  private:
  public:
    Display()
    {
     }

    void Digital()                              // View in digital format
        {
         Func.DrawBox();
         Func.DrawBox();
         CarSimul.UpdateDataBlock(Data);
               for (int i = 0; i < MAXFIELDS; i++)
               {
                        gotoxy(Data[i].XCoord1, Data[i].YCoord1);
                        printf("%s", Data[i].Name);
                        if (Data[i].XCoord2 != -1)
                        {
                                 gotoxy(Data[i].XCoord2, Data[i].YCoord2);
                                 printf("%.2f", Data[i].Val);
                        }
               }
        }

    void BarGraph()                             // View in graph format
        {
         float Scale;
          CarSimul.UpdateDataBlock(Data);
         for (int i = 0; i < MAXFIELDS; i++)
          {
               gotoxy(Data[i].XCoord1, Data[i].YCoord1);
               printf("%s", Data[i].Name);
               if (Data[i].XCoord2 != -1 && Data[i].XCoord2 != -2)
               {
                        Scale = 47 / (Data[i].MaxVal - Data[i].MinVal);
                        for (int j = 15; j < 79; j++)  printf(" ");
                        gotoxy(Data[i].YCoord1);
                        printf("%.0f", Data[i].MinVal);
                        gotoxy(Data[i].YCoord1);
                        printf("%.0f", Data[i].MaxVal);
                        Data[i].Val *= Scale;
               }
```

```
        }
      }
  void Update(char choice)              // Update user selection
    {
      switch(choice)
     {
       case '1':
              Digital();
              break;
       case '2':
              BarGraph();
              break;
       default:
              cout << "Unknown choice!";
     }
    }
};
Display View;                           // Instance of Display
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

A.2.3. Controller

```
class Controller                        // Class Controller
{
  private:
  public:
    Controller()
    {
    }

    void UpdateModel()                  // Update Model
    {
      CarSimul.Update();
    }

    void UpdateView(char choice)        // Update View
    {
      View.Update(choice);
    }
};
Controller CC;                          // Instance of Controller
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Appendix B. Heater Control System

This appendix shows relationship diagrams for class definition, state transition and a scenario for the heater control system. The source code listing is also provided in later sections for the classes used in this application.

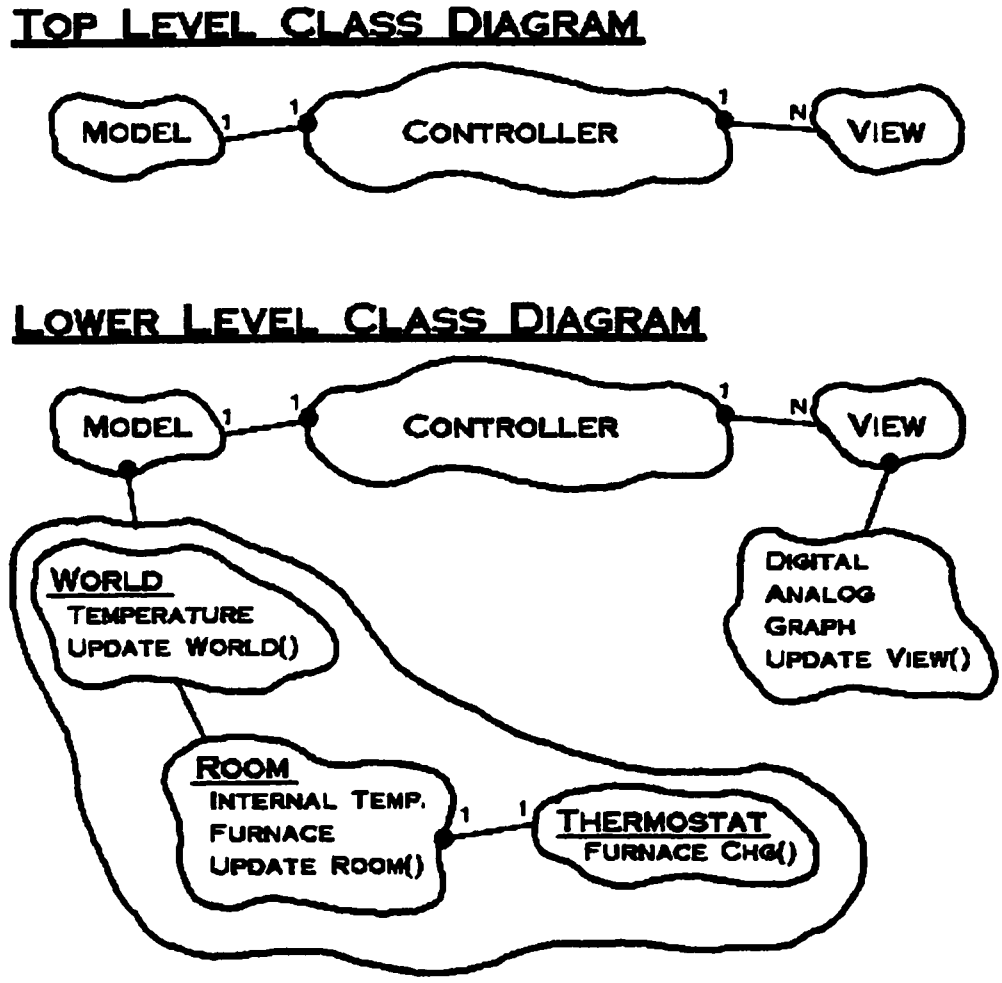


Fig. B.1: Heater Control Class Definition

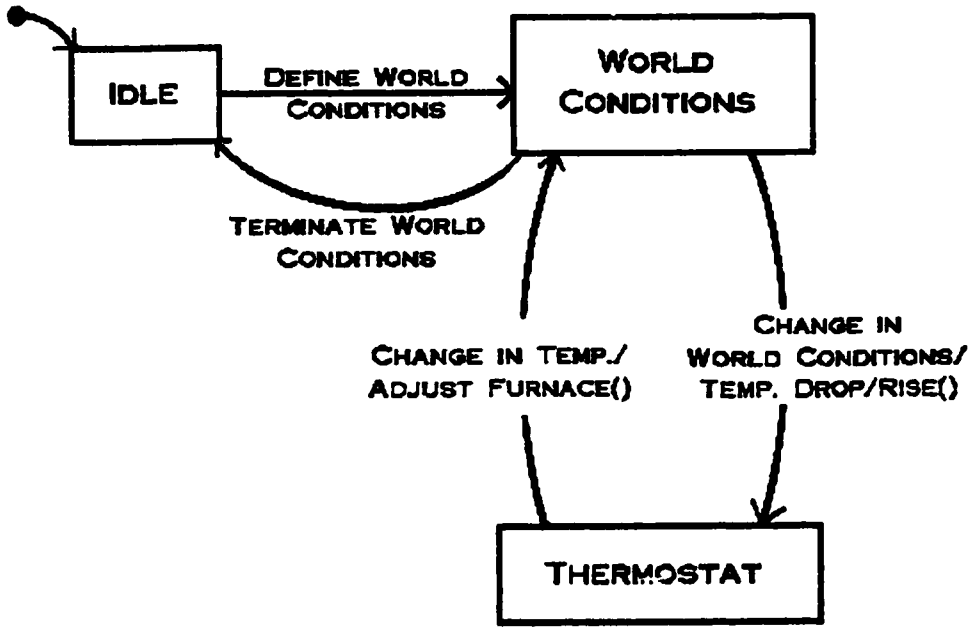


Fig. B.2: Heater Control State Transition

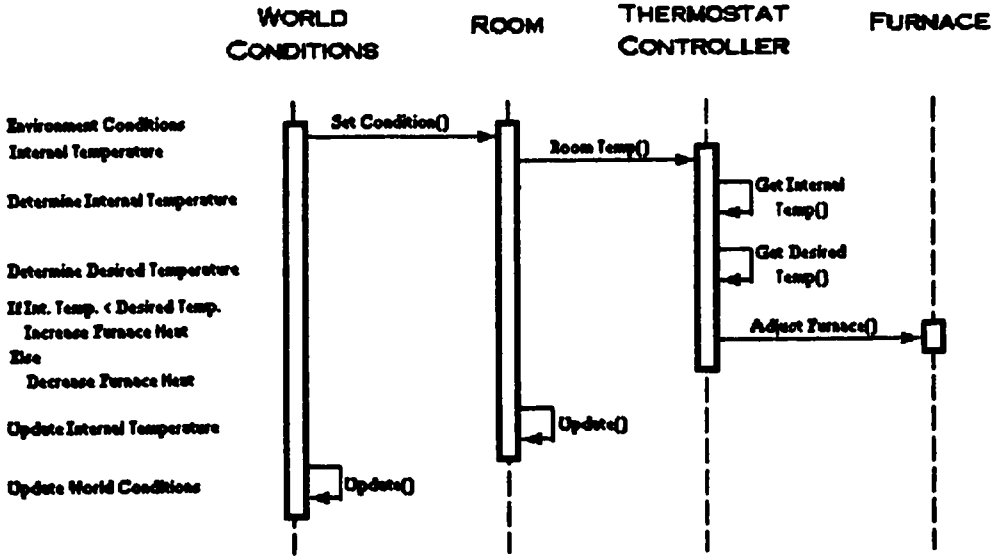


Fig. B.3: Heater Control Scenario

B.1. Code Listing for Variable Definition

The variables used for the heater control system application are defined below.

```
#define MAXFIELDS 7                          // Maximum number of fields

#define TEMP 15                              // Set temperature of the room
#define TEMPFACTOR 20                        // Temperature factor
#define VOLFACTOR 2500                       // Capacity of the room
#define INITMINEXTTEMP -10                   // Initialize minimum external
                                                temperature
#define INITMAXEXTTEMP 30                    // Initialize maximum external
                                                temperature
#define CHNGEXTTEMP .1                       // Factor change in external
                                                temperature
#define MINEXTTEMP -50                       // Minimum external temperature
#define MAXEXTTEMP 50                        // Maximum external temperature

#define INITMININTTEMP -10                   // Initialize minimum internal temp
#define INITMAXINTTEMP 30                    // Initialize maximum internal temp
#define INITMINFURNACE 0                     // Initialize minimum furnace heat
#define INITMAXFURNACE 0                     // Initialize maximum furnace heat
#define INITMINVOLUME 1000                   // Initialize minimum room volume
#define INITMAXVOLUME 5000                   // Initialize maximum room volume

#define MININTTEMP -50                       // Minimum internal temperature
#define MAXINTTEMP 50                        // Maximum internal temperature
#define MINFURNACE -5                        // Minimum furnace factor
#define MAXFURNACE 5                         // Maximum furnace factor


struct DataField                             // Data Field
{
  char Name[80];
  float Val;
  float MaxVal, MinVal;
  int XCoord1, YCoord1;
  int XCoord2, YCoord2;
} Data[MAXFIELDS];
```

//

B.2. Code Listing for Class Definition

The classes for the MVC framework are defined: Model, View and Controller.

B.2.1. Model

The Model class has three subclasses: World, Room and Thermostat.

```
class Model                                    // Class Model
{
```

B.2.1.1. World Class

```
  class World                                  // External conditions
  {
    private:
    public:
        float ExternTemp;                      // ExternTemp in centigrade (-50 upto +50)

      World()
      {
        ExternTemp = Func.RandNum(INITMINEXTTEMP, INITMAXEXTTEMP);
      }

      void Update(World *w)                    // Update class World
      {
              w->ExternTemp = w->ExternTemp +Func.RandNum(-CHNGEXTTEMP,
                    CHNGEXTTEMP);
              if (w->ExternTemp < MINEXTTEMP) {w->ExternTemp =
                    MINEXTTEMP; }
              if (w->ExternTemp > MAXEXTTEMP) {w->ExternTemp =
                    MAXEXTTEMP; }
      }
  };
  World Montreal;                              // Instance of World class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

B.2.1.2. Room Class

```
class Room                                  // Internal conditions
  {
    private:
    public:
        float InternTemp;
        float furnace;
        float volume;

        Room()
        {
                InternTemp = Func.RandNum(INITMININTTEMP,
                        INITMAXINTTEMP);
                furnace = Func.RandNum(INITMINFURNACE, INITMAXFURNACE);
                volume = Func.RandNum(INITMINVOLUME, INITMAXVOLUME);
        }

        void Update(Room *c, World *w)
        {
                float TempChange;
                TempChange = ((c->InternTemp - w->ExternTemp) / TEMPFACTOR) +
                        (c->volume / VOLFACTOR);
                TempChange = (c->InternTemp > w->ExternTemp) ? TempChange =
                        -TempChange : TempChange;
                TempChange += c->furnace;
                c->InternTemp = c->InternTemp + TempChange;
                if (c->InternTemp <= MININTTEMP) { c->InternTemp = MININTTEMP
                        + 0.001; }
        }
  };
Room LivingRoom;                                    // Instance of Room class
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

B.2.1.3. Thermostat Class

```
class Thermostat                                    // Thermostat class
  {
    private:
    public:
        float SetTemp;
        Thermostat()
        {
                SetTemp = TEMP;                     // Set temperature of room
         }

        void FurnaceChange(Room *c)
        {
                float HeatNeeded;                   // Calculate heat needed
                HeatNeeded = (SetTemp - c->InternTemp);
                c->furnace = (HeatNeeded > MAXFURNACE) ? MAXFURNACE :
HeatNeeded;
                if (c->furnace < MINFURNACE) { c->furnace = MINFURNACE; }
                if (c->furnace > MAXFURNACE) { c->furnace = MAXFURNACE; }
        }
  };
Thermostat Sensor;                                  // Instance of Thermostat class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
private:
public:

    Model()
    {
    }

    void UpdateDataBlock(struct DataField Data[]) // Update Data Block
    {
    }
    void Update()                                   // Update necessary classes
    {
        Montreal.Update(&Montreal);
        LivingRoom.Update(&LivingRoom, &Montreal);
        Sensor.FurnaceChange(&LivingRoom);
    }
};
Model RoomSimul;                                    // Instance of Model class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

B.2.2. View

```
class Display                                   // Display class
{
  private:
  public:
    Display()
    {
    }

        void Digital()                          // View output in digital format
        {
                Func.DrawBox();
                Func.DrawBox();

                RoomSimul.UpdateDataBlock(Data);
                for (int i = 0; i < MAXFIELDS; i++)
                {
                        gotoxy(Data[i].XCoord1, Data[i].YCoord1);
                        printf("%s", Data[i].Name);
                        if (Data[i].XCoord2 != -1)
                        {
                                gotoxy(Data[i].XCoord2, Data[i].YCoord2);
                                printf("%.4f", Data[i].Val);
                        }
                }
        }

        void BarGraph()
        {
                float Scale;
                int Origin;

                RoomSimul.UpdateDataBlock(Data);

                for (int i = 0; i < MAXFIELDS; i++)
                {
                        gotoxy(Data[i].XCoord1, Data[i].YCoord1);
                        printf("%s", Data[i].Name);
                }
        }
```

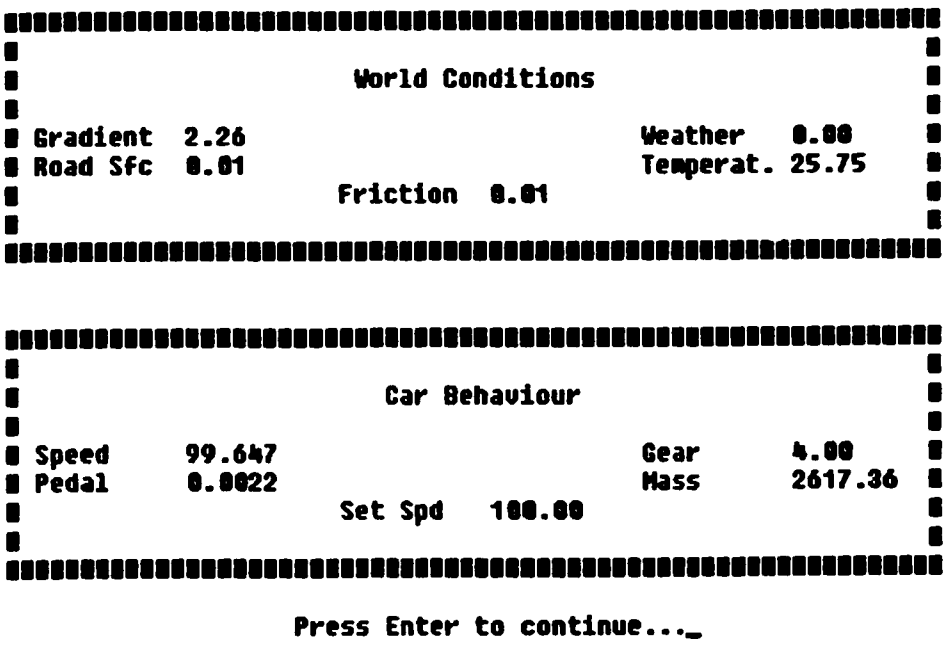
```
    void Update(char choice)                    // Update output format
    {
      switch(choice)
      {
        case '1':
                Digital();
                break;
        case '2':
                BarGraph();
                break;
        default:
                cout << "Unknown choice!";
      }
    }
};

Display View;                                   // Instance of Display class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

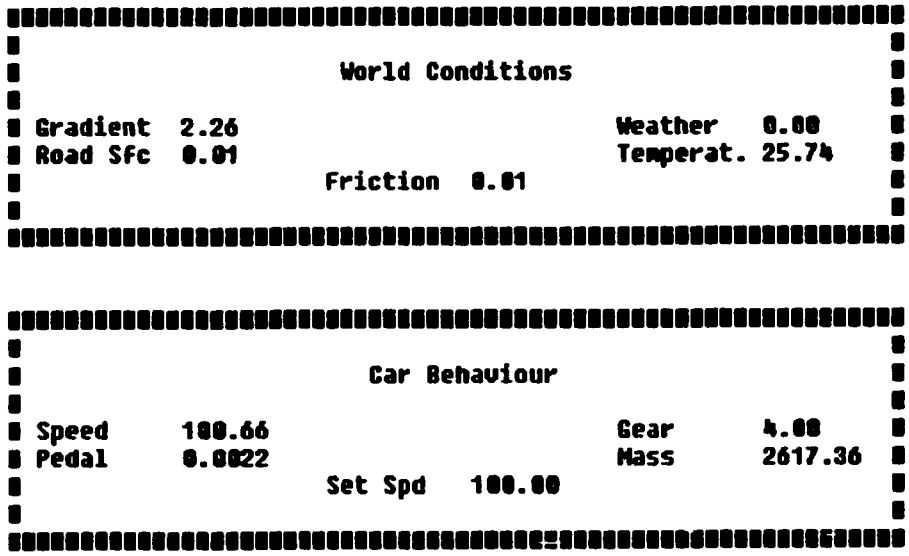
B.2.3. Controller

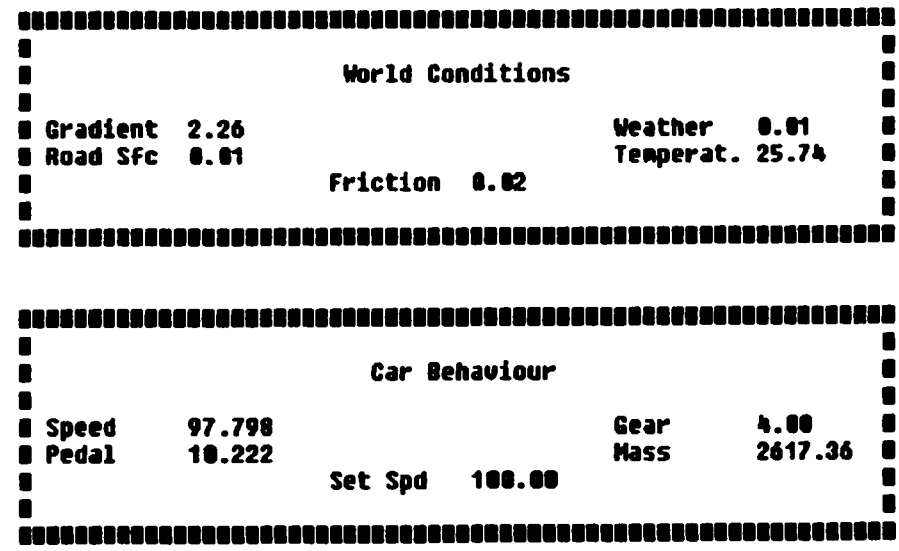
```
class Controller
{
  private:
  public:
    Controller()
    {
    }

    void UpdateModel()                          // Update Model
    {
      RoomSimul.Update();
    }

    void UpdateView(char choice)                // Update View
    {
      View.Update(choice);
    }
};
Controller CC;                                  // Instance of Controller class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Appendix C. Results Generated from Cruise Control System Simulation

The results generated from the cruise control system are shown below. Under different conditions, the controller tries to achieve and maintain the set speed desired by the driver.




World Conditions
Gradient 2.26
Road Sfc 0.01
Weather 0.00
Temperat. 25.74
Friction 0.01
Car Behaviour
Speed 100.66
Pedal 0.0022
Gear 4.00
Mass 2617.36
Set Spd 100.00
Press Enter to continue...


World Conditions
Gradient 2.26
Road Sfc 0.01
Weather 0.01
Temperat. 25.74
Friction 0.02
Car Behaviour
Speed 97.798
Pedal 10.222
Gear 4.00
Mass 2617.36
Set Spd 100.00
Press Enter to continue...

Appendix D. Results Generated from Heater Control System Simulation

The results generated from the heater control system are shown below. Under different conditions, the controller tries to achieve and maintain the set temperature desired in a room.

```
World Conditions

Extern.Temp.        -8.1427


Room Behaviour

Intern.Temp.   12.9532        Set Temp.      15.0000
Furnace        2.0468         Volume         2480.1479

Press Enter to continue...
```

```
                    World Conditions

               Extern.Temp.        -8.3428


                    Room Behaviour

 Intern.Temp.   12.9436         Set Temp.      15.0000
 Furnace        2.0564          Volume         2400.1479

```

Press Enter to continue..._

```
                    World Conditions

               Extern.Temp.        -9.0071


                    Room Behaviour

 Intern.Temp.   12.9883         Set Temp.      15.0000
 Furnace        2.0917          Volume         2400.1479

```

Press Enter to continue...