

Chapter 1 Introduction

1.1 Background of the Problem

Object-Oriented Programming (OOP) and software case development techniques have opened a new avenue in the rapid prototyping and development of complex software. In the past five years OOP has been used to develop many of the software applications that are common in most of our standard desktop and workstation computers. New Operating Systems (OS) such as WINDOWS NT, NEXTStep, and Be among others have also employed OOP concepts in their development with the promise to simplify code maintenance and provide extensibility to the original OS for future enhancements. Unfortunately, concepts that have been embraced by computer scientists for over a decade have been slowly migrating to the engineering field. Today, few engineers use C++, Smalltalk, Java or other OOP languages to develop in-house applications or computational routines. Several authors point out that applications using at least some of the concepts of OOP techniques could save development time and effort by reducing coding times and making the programs more efficient to run [Hill, 1996]. Perhaps the best selling point in the use of OOP software development framework will be a substantial reduction in the maintenance cost of a real model. Booch [1994] estimates that between 45% and 65% of the cost to develop software is

needed to maintain a software model based in standard procedural programming techniques. The use of OOP promises a reduction to 25-40% instead.

In the specific area of computer simulation, several OOP simulation languages have been developed over the last fifteen years addressing the needs of every simulation analyst. Examples of new generation simulation languages using principles of OOP are: Modsim III, Simula 67, ROSS, Sim++, SIMEX 3.0 and QNAP2 among others. While some of these languages have gained widespread use in manufacturing and facilities planning sectors very few applications of these exist in the area of air traffic control and airport simulation. Today the best airport and airspace simulation models use procedural programming paradigms making models difficult to enhance, debug and maintain.

A typical example of a complex model structure is perhaps SIMMOD - the Federal Aviation Administration (FAA) airport and airspace simulation model. This software application comprises more than one hundred thousand lines of source files programmed in SIMSCRIPT II.5, a first generation simulation languages developed by CACI product in the early seventies [CACI, 1990]. The size of the model could have reached a half million lines of code had this model been coded in standard high level languages like FORTRAN or C. To give an idea of the complexity of this model its ground simulation engine comprises more than forty routines of various sizes that mimic all heuristics used by ATC personnel to control aircraft on a ground network. The model is refined enough to simulate aircraft push-back procedures from a gate and simulate blockages in adjacent taxi-lanes to other aircraft or to simulate the deicing procedures of aircraft during winter operations.

Unfortunately, large scale simulation models such as SIMMOD present a challenge to upgrade given the large amounts of dependencies between logical statements in procedural programming. While SIMMOD is just a point in case it is important to realize that over the

last twenty years airport engineers and operations analysts have relied on tools and models whose procedural programming internal structure has posed serious challenges to update and modify.

Another driver for OOP technologies is the in the area of real-time airport operations. Airport surface traffic control is an important element of the overall Air Traffic Control system (ATC). The smoothness of airport surface traffic flow is one of the major goals in the development of technology to help ATC personnel to manage traffic. Furthermore, the effectiveness of the airport surface traffic control is critical to aviation safety. In a previous study Hollister [1987] indicated that the potential for serious accidents on the airport surface increases with increased operations under low visibility. According to Federal Aviation Administration (FAA) statistics the number of runway incursions increased from 77 in 1984 to and 115 in 1986 and then on to 135 in 1994 [FAA, 1996]. A recent series of aircraft accidents on the ground has prompted the FAA to speed up the development of automated tools and procedures to further protect against aircraft collisions on the ground.

From the system-wide view point, an increase in capacity in the terminal airspace surrounding the airport facility will push the congestion to the runway, taxiway and gate levels. In fact, several of the most congested airports in the U.S. are limited by runway capacity and this trend is expected to continue. Runway capacity studies and proposed automation tools have been proposed in the literature [Trani et al., 1990; Gu et al., 1995]. An eventual solution to mitigate airport congestion could be the development of an Airport Surface Traffic Control System (ASTCS) to provide ATC personnel and pilots with optimal system-wide control strategies to manage aircraft traffic flows. At this stage one problem of concern to airport analysts is the complexity of the Air Traffic Management (ATM) decision making processes taking place in a real airport ground network. Dozens of heuristics are used everyday to manage aircraft traffic flows that would have to be incorporated into the ASTCS and

coupled with optimization flow algorithms to minimize delay or fuel consumption metrics. Detailed levels of simulation fidelity will be needed for such a system in order to depict minute state changes in the aircraft trajectory to assess and resolve possible ground collisions. Without any doubt this system will have to rely on augmented satellite positioning systems (i.e., differential global positioning system - GPS) or Airport Surface Detection Radar to achieve the desired position accuracy needed in real-time applications.

In order to study airport surface traffic flows, object oriented modeling coupled with information management capabilities could be a powerful tool set for detail system analysis and real time airport operation traffic management system.

1.2 Objective and Scope of This Study

Studies on airport operations have mainly addressed ground network traffic control, airport terminal operations, aircraft landing and takeoff distributions, and network flow management. Each facet of airport engineering has been typically studied using specific simulation models that seldom communicate among themselves. These models usually require several years of programming effort and months following their development to be validated. The objective of this study is to establish an experimental simulation framework (i.e., library using C++ and Java). To be used in a variety of airport operation analyses. The proposed framework will be flexible allowing quick modifications to represent various studies such as airfield capacity analysis, airport surface traffic safety evaluations, or airline flight scheduling. This study will emphasize the utilization of utility classes from standard C++ library or

Java (a new generation OOP language). Using these libraries as the foundation of the airfield simulation framework, the airfield traffic system modeling tool kit could be easily programmed as multi-platform application tool kit.

The scope of this study could be described as:

1. Analysis of airport surface traffic systems with objects such as aircraft, runway, taxiway and aprons. Extensions to the airspace network can also be done in the future.
2. Establish an experimental modeling library that could be used as a basic toolkit for researchers to study different ATC ground control strategies.
3. Provide the foundation for the next generation airport airspace simulation model.
4. Verify the platform independent modeling capabilities of the JAVA language in the context of an airport engineering application.

2.1 System Modeling Technologies

In recent years, computer software development technology is migrating to Object Oriented Programming (OOP). If the object oriented approach can be classified as a modern modeling technology, the following characteristics are probably the major reasons.

1. Computation speeds of commercial CPUs double every eighteen months and the unit cost for using a commercial CPU with same computation speed has been reduced.
2. Random Access Memory (RAM) prices have dropped drastically over the last two years making the development of memory intensive applications economically feasible.
3. Industrial managers and decision makers need management information systems (MIS) and decision support systems (DSS) that can process real time information and use the real time data to simulate system dynamics with details of all

system levels and to provide decision advisories.

4. System analysts need to use a variety of models to analyze different what-if cases with rapid modeling computer tools. These tools should have the flexibility to implement scenario-oriented system modeling.
5. The market demands force modelers to look for reusability modeling in their own research domains. Results or conclusions from their research should be readily reusable and redundant work among different research projects in model developments could be avoided. In other words, researchers can concentrate in problem depth rather than in the development of interfaces.

2.2 New Requirements of System Integration

System integration is defined as an approach to the organization and management a real operation system through the use of appropriate levels of computer and information technologies [Mize, 1992]. The nature of system integration requires system elements or sub models to be scalable to meet different requirements denoting clients' preferences. Also system integration is to interface different applications with each other. This nature of system integration makes object oriented modeling in high demand because OOM's late binding feature.

2.3 Object Oriented Programming(OOP) and Modeling (OOM)

Although object-oriented analysis and modeling are relatively new concepts in computer science, it is not difficult to understand that these concepts are identical to those meth-

ods used in modern automotive assembly lines. Parts and assembly are made or supplied from different manufacturers and sent to automotive assembly lines. Cars are assembled rather than made at a time. In object oriented analysis, objects are finite data models with functions representing states of a physical object in the real world. Today, the object oriented approach is considered to be the most appropriate method to model real world systems in a computer. The following examples may help illustrate some of the OOP concepts.

1. Physical objects:

Aircraft in an air traffic control system

Airports in a aviation traffic network

Cars in a car following simulation model

Runways in an airport ground network

Taxiway in an airport ground network

Terminals in an airport and gates in a terminal

Artificial links and nodes in a airport ground network

2. Data objects

Wind rose in a selected airport site

Schedules of airline services

Queue of arrival or departure aircraft

Runway configurations

Air traffic separation rules

3. Human entities

Pilots

Passengers

Class in OOP is defined as a collection of similar objects just as in our understanding of what a class is in the school. A component is defined as a collection of similar and well-

defined classes.

Comparing Different Object Oriented Methods

Object-Oriented Methods distinguish themselves by using different approaches to describe the target systems. They are usually categorized into several groups according to the ways they view the system. Fowler provides a very detailed comparison of different object-oriented modeling methods [Carmichael, 1995]. The following presents a summary of various idiosyncrasies adopted by mainstream object-oriented methods from Fowler.

Viewing as Data Types, Classes and their Association

This approach treats the structural view of a system as the types of objects in the system and various kinds of static relationships that exist between them. Three kinds of relationships are important: **associations** (e.g., a professor may have a number of students), **subtypes** (e.g., a B-727 is a type of aircraft belonging to approach category C), and **aggregation** (e.g., an engine is part of an aircraft). The various object-oriented methods all use different (and often conflicting) terminology for these concepts. Fowler recognizes this phenomena as extremely frustrating but inevitable. For people who like entity-relationship modeling, types correspond to entity types, associations to relationships, and sub-typing is the same. However the notion of types is common to most methods. In the entity-relationship paradigm models are represented as a box in a diagram. Martin and Odell, Figure 2.3, define object types that are conceptual and classes are part of their implementation. Coad-Yourdon, Figure 2.1 and 2.3, and Booch, Figure 2.2 and 2.3, distinguish in diagrams between abstract classes (called class) and nonabstract classes (class and object). The Booch notation also adds distinctions for metaclasses and parameterized classes.

Jacobson classifies types as entity objects, interface objects, or control objects as shown in Figure 2.1. Entity objects are used to hold stored information, interface objects are used to communicate with external actors (e.g., users), and control objects are used for processing. Associations represent relationships between instances of types (e.g., a person works for a company, a company has a number of offices). Fowler compared association to pointers in object-oriented languages, but points out the important distinction that they are usually considered to be bi-directional and can be navigated in either direction.

Attributes and operations are generally handled in a similar way. Methods show attributes and associations within the type box, but they are only appropriate for small models. For larger models, it is best to note them on a separate text list. Rumbaugh indicates the importance of putting the attribute type for attributes, and operation return type and arguments on operation definitions. Booch adds a very wide range of possible indications, including concurrence and persistence. An example of a Booch diagram is shown in Figure 2.4, which also shows the slightly awkward notation used. It is difficult to draw on a whiteboard or with a computer drawing package.

Viewing Systems by their Behaviors

In structured methods the comparative unanimity of the Entity-Relationship technique for modeling data has never had an equivalent for representing behavior. Various approaches have been used, many of which combined the notions of behavioral and architectural modeling. The most common approach for behavioral modeling is still the flow chart—or its nondiagrammatic equivalent of pseudocode. The defects of any of these procedure-based techniques for large, complex behaviors has been well documented. However, both techniques are probably still unrivaled for the description of a single algorithm, providing it is not too long or wide ranging .

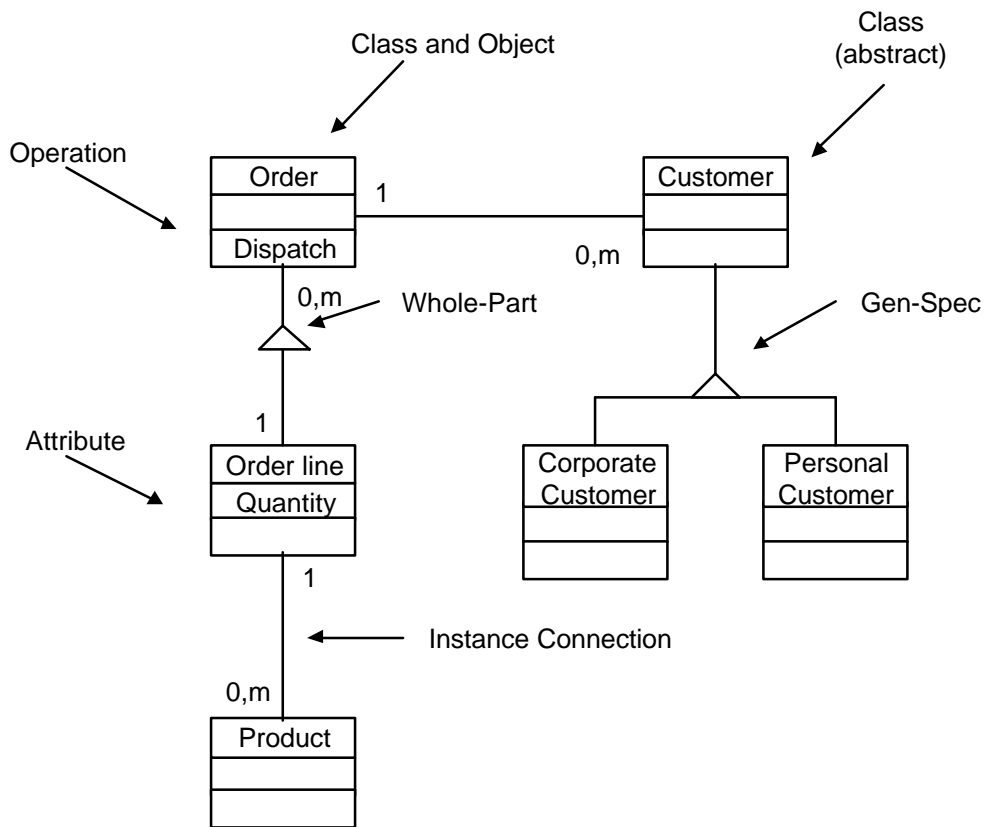


Figure 2.1 Coad/Yourdon object-oriented analysis (reproduced from Carmichael,1995).

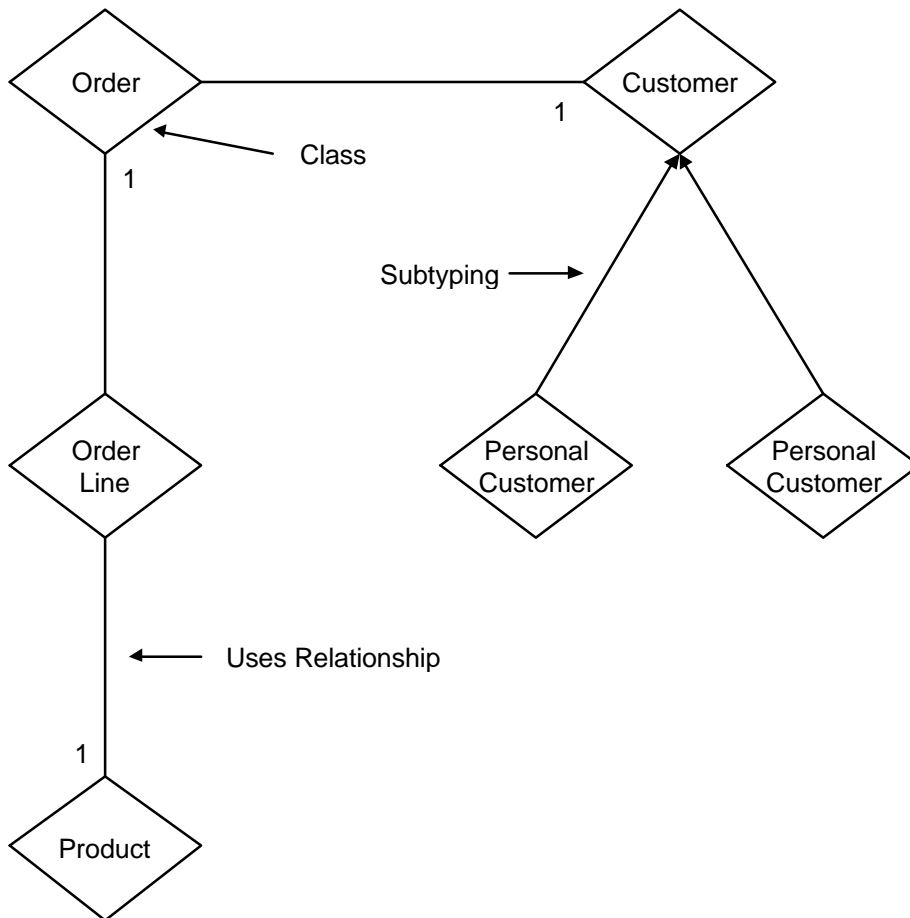


Figure 2.2 Booch class diagram (reproduced from Carmichael,1995).

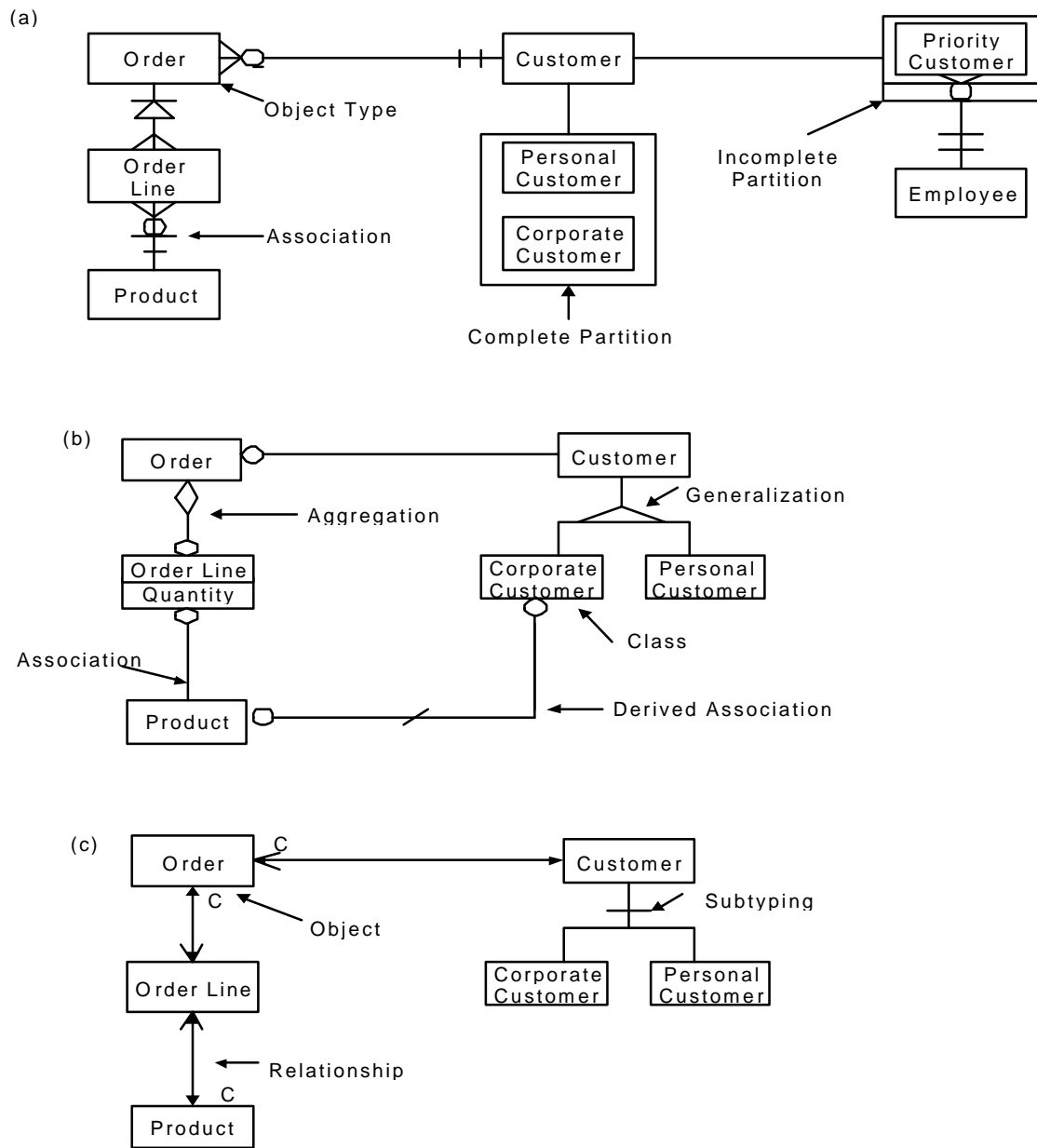


Figure 2.3 Three data views (a)Martin/Odell; (b)Rumbaugh; (c)Shlare/Mellon (reproduced from Camichael,1995).

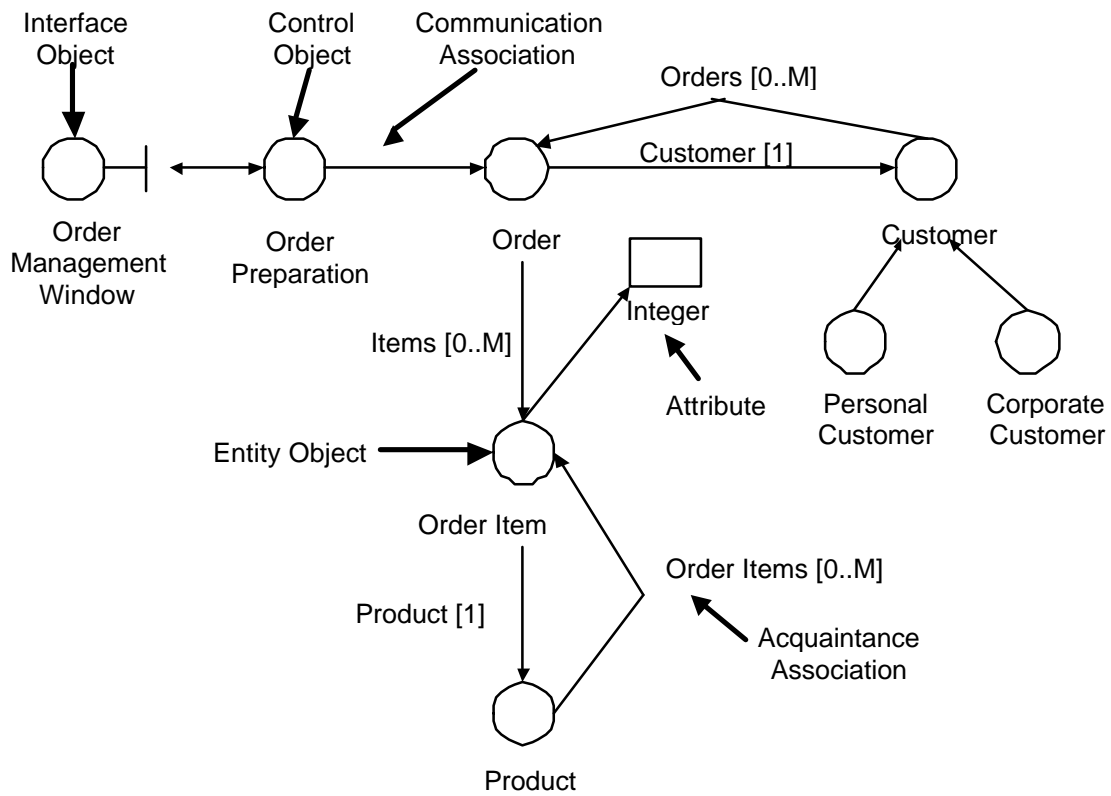


Figure 2.4 Jacobson's data view (reproduced from Carmichael, 1995).

One of the principal differences between various kinds of state transition diagrams is where actions fit into the technique. The actions are linked to transitions, so that when a transition occurs an action may be carried out. The system then waits in its various states. The two diagrams to consider initially are the Shlaer-Mellor and Booch versions shown in Figures 2.3 and 2.4, respectfully. Booch uses a Mealy model while Shlaer-Mellor uses a Moore model. The immediate pattern to notice is how the Shlaer-Mellor diagram introduces two extra states (Resuming and Initiating) to handle the actions associated with the resume and started transitions in Booch. The Booch Model requires the notion of conditional transitions, in that the resume event can lead to two different transitions from the state suspended. Each transition thus needs some form of condition attached to it to indicate which one should be called – the logic for this is captured explicitly in the Shlaer-Mellor diagram. Limiting state transition diagrams to single class succeeds in limiting the state explosion problem, but does generate a different problem. Although the behavior of a single class is easy to visualize, it is still difficult to visualize the behavior of a system of many different classes working together. One approach to solve this problem is the mechanism-based approach, which attempts to visualize the interclass behavior. Booch gives three ways of describing mechanisms:

1. To number the messages in sequence
2. To provide pseudo-code
3. The timing diagram that provides a diagrammatic version of mechanism 1

Viewing Systems as Architecture

The architectural view corresponds to breaking a large system up into components. Traditionally, this has been done functionally—a high-level function is broken into subfunctions that are further broken down until a sufficiently low level is reached. Since it is difficult to do this without some form of guidance. This allows the modeler to concentrate data flows in a

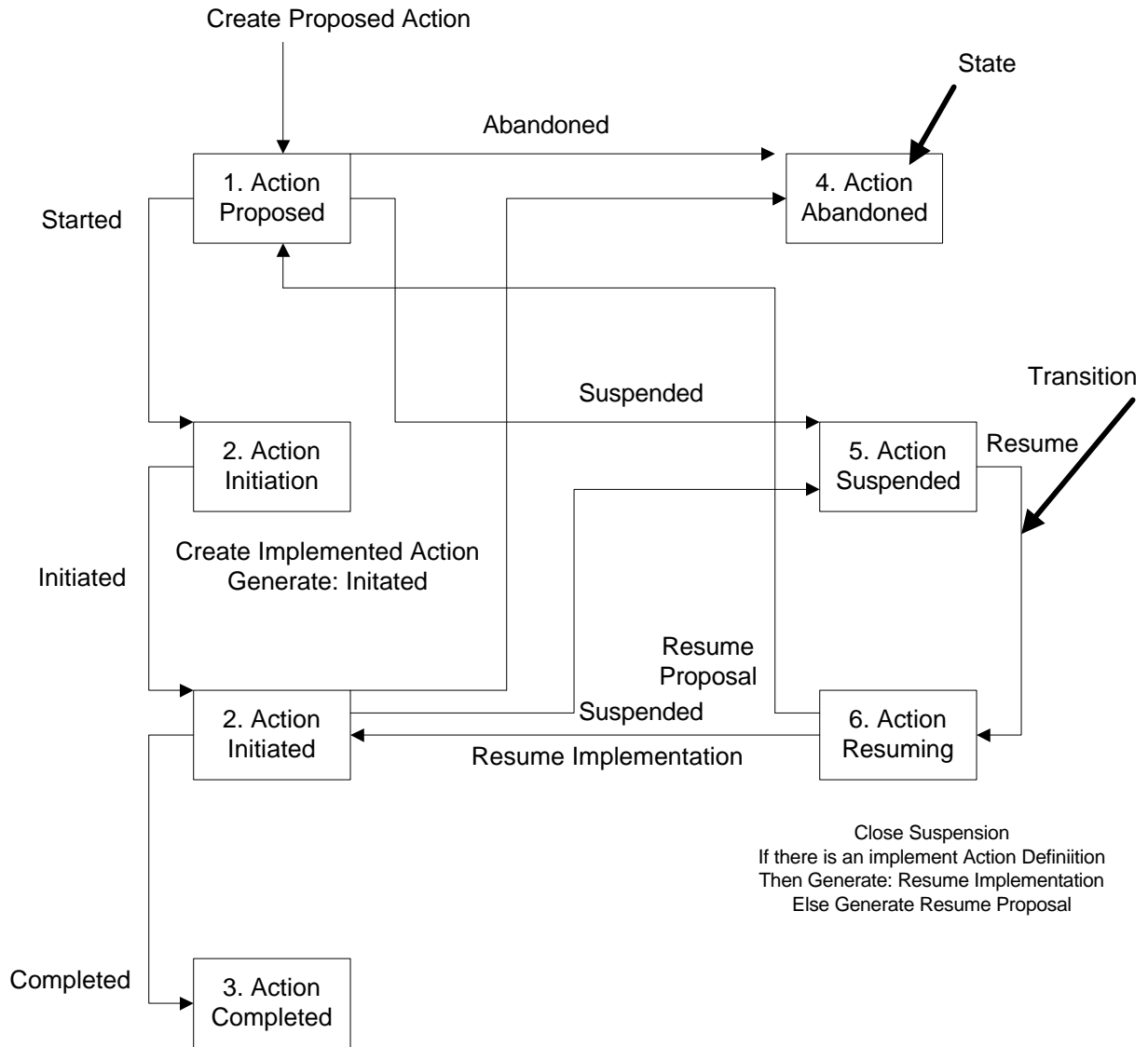


Figure 2.5 Shlare-Mellor State Transition Diagram (reproduced from Carmichael,1995).

particular model to reduce coupling.

Functional Decomposition

A system is described by functions and data stores with data flows connecting them. Subfunctions are further broken down until the bottom level is reached. At this bottom level, the connection with the other views is made: bottom-level functions correspond to operations on the data view.

Object Decomposition

The object decomposition approach suggests breaking systems down by use of higher-level objects rather than higher-level functions. These higher-level objects then communicate and use each other in roughly the same way that the lower-level objects do. Two principal issues are highlighted in this analysis: communication and visibility.

Method Summaries

The following summaries are given by Fowler [Carmichael, 1995]:

Booch Method

Entity-Relationship diagrams appear in the data view, here called class diagrams. Dynamics are shown with a state transition diagram within a class. Timing diagrams and object diagrams show interactions between objects. Logical architecture is shown with object diagrams and class categories that use object decomposition, and the physical design is shown with module and process diagrams. The object diagrams serve both to show object decomposition and to indicate the inter-object behavior.

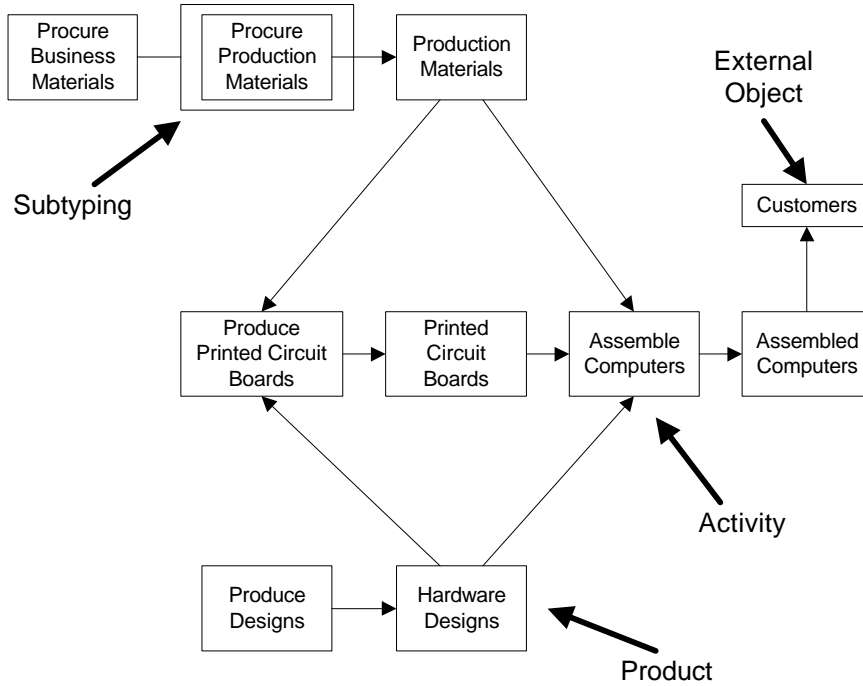


Figure 2.6 Martin/Odell s object flow diagram (reproduced from Carmichael, 1995).

In all, the wide range of techniques in Booch's method give it a lot of flexibility, particularly when dealing with concurrent systems. However, the variety of techniques also makes it difficult to use, since it is not at all apparent how to combine these to best effect for different kinds of problems.

Coad-Yourdon

The heart of this technique still lies in the data view with Entity-Relationship diagrams. These now show aggregation as well as associations and subtyping. Behavior is shown by naming operations on the classes and defining within the class by state-transition charts and flow charts (called service charts). Inter-object behavior is shown by putting message connections on the Entity-Relationship diagram. Architecture is defined by grouping classes into subjects. These groups are also shown on the Entity-Relationship diagram. Thus, nearly the whole system is described in a single diagram, which is considered to have several layers, so that the reader can choose which views of the system to see. This approach is simple, but may be difficult to work with in larger systems.

The technique for building up the architectural view is worth mentioning. The Entity-Relationship diagram is initially shown with the classes, subtyping, and aggregation relationships, but not with the associations. These relationships are then used to define the subject areas, essentially by equating the subjects with the classes at the top of the subtyping or aggregation trees.

Fowler pointed out simplicity as the greatest strength of the method. But simplicity is also its greatest weakness. A common feeling is that while it is an appropriate early start, it is not a method that would be effective in a sizeable

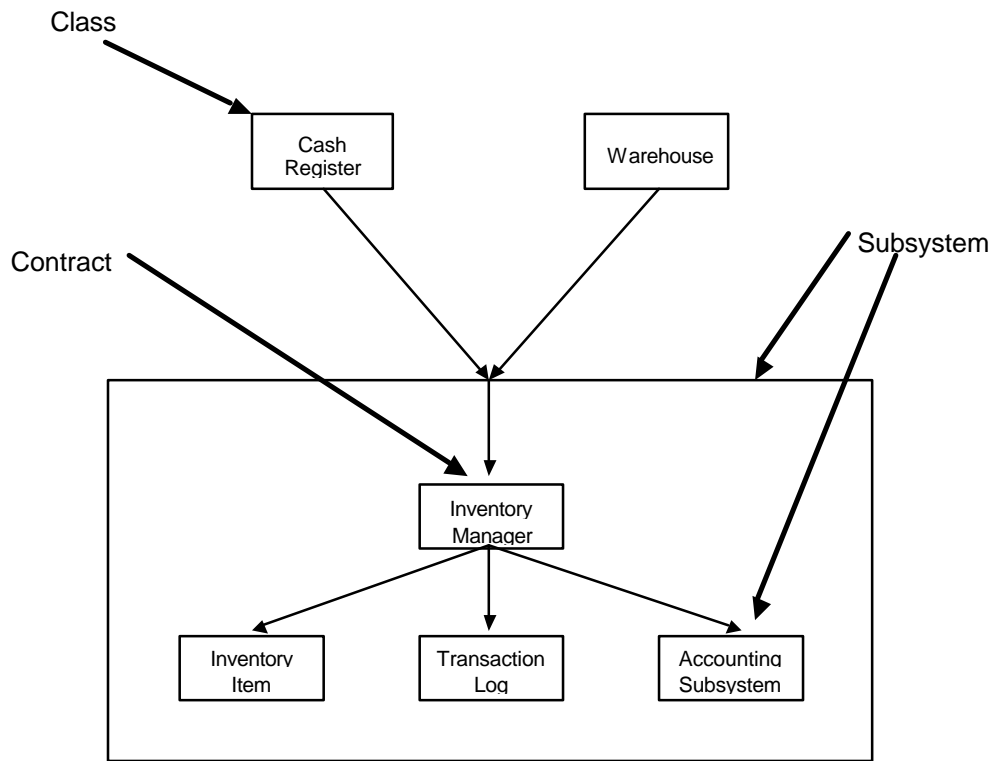


Figure 2.7 Wirfs-Brock collaboration graph (reproduced from Carmichael, 1995).

project.

Jacobson (OOSE)

The distinguishing aspect of Jacobson's method is the strong focus on use-cases, a scenario in which an actor (e.g., a user) makes use of the computer system. The analysis, design, and testing is entirely driven by the use-case. Upon this, Jacobson builds one of the more structured descriptions of the process of analysis and design.

Another unique feature is the classification of analysis objects into entity, interface, and control objects. This encourages a separated, yet well-integrated way of placing processing and user interfaces into the system design.

Martin-Odell (Ptech)

The data view stresses a binary rather than Entity-Relationship approach to data modeling. The method is significant in that it demonstrates a formal underpinning in class mathematics and a strong philosophical influence. This leads to its elegant way of handling subtyping, support for class expressions, derived associations, and dynamic classification (objects changing class during their lifetimes). Behavior is defined using event diagrams that can express system behavior in a compact diagram. The events are formally linked to the data view, and the CASE tools can compile the event diagrams into executable code. Architecture is handled by object-flow diagrams that use a functional decomposition driven by a value-added chain representation. This is a useful technique for strategic analysis, but does not provide the architectural modeling capabilities of other methods. The analysis approach is notable for encouraging an event-driven approach that identifies events first and then uses these events to define classes.

Rumbaugh (OMT)

Rumbaugh's Object Modeling Technique (OMT) was developed at General Electric's research labs in the United States. The method uses three techniques: 1) object diagrams (a dialect of Entity-Relationships diagrams) for the data view, 2) Harel statecharts for behavior, and 3) data flow diagrams for architecture. The Entity-Relationship technique is particularly rich, supporting partitioned and non-partitioned subtypes, derived objects and relationships, and aggregation. The statecharts provide the most powerful state transition modeling in the methods surveyed. The data flow diagrams stress the link to objects by decomposing, so that bottom level bubbles are operations on classes.

Shlaer-Mellor

In the Shlaer-Mellor method, classes are modeled in the data view, which provides associations and subtyping. Each class is then given a state diagram, which describes its life cycle. Actions defined on the state diagram are either described by pseudocode or an action data flow diagram that acts as a process precedence diagram. Message connections between classes, divided into asynchronous events and synchronous queries, are defined to begin the architectural view. These message-based models are used again to describe subsystems. At the highest level, domains are defined with visibility relationships (called bridges) between them. Another feature of Shlaer-Mellor is their recursive design notion. This encourages the development of design rules to formally turn a model into a working system.

Wirfs-Brock (CRC, RDD)

The Wirfs-Brock method (also referred to as the CRS method—classes, responsibilities, and collaborations— or as responsibility-driven design) is certainly the most unusual approach currently published. Unlike all the other methods, which rely on a strong data view, Wirfs-Brock completely abandons data modeling, using only a simple inheritance graph. The technique focuses on describing the system in terms of responsibility, which is taken on by classes. These responsibilities are used to describe collaborations between classes that are eventually formalized in terms of an architectural view. No behavioral view is used. The notion of responsibilities has been well received by most people in the field, and the initial class identification techniques of Wirfs-Brock are widely recommended. One of the key appeals of the technique is that the hiding of the data structure and the use of responsibilities carries the notion of encapsulation into the analysis and design activity. Other techniques see encapsulation more as a detailed design notion. Fowler questions that this approach could be impossible to effectively describe systems particularly larger systems, without defining a data model.

Current Industrial Applications in Java

Since the release of the Java Development Kit (JDK 1.0) in January 1996, over 200,000 commercial developers have started development in Java. As a matter of comparison, it took ten years to reach an installed base of 400,000 Windows API developers. Perhaps the most compelling reasons to use Java as a language are its transparent portability across all platforms and the explosive use of the Internet. After all Java has been touted as the only way to create dynamic web pages. In the field of transportation engineering, researchers are now actively trying to understand the various ways that Java can and should be used to provide

transportation services on the World Wide Web [www.itsonline.com].

Airline Travel Information System

SRI's Speech Technology and Research Laboratory developed a demonstration model that uses a speech recognition applet to answer questions about airline flights between selected cities, flight times, fares, etc)

Airport Pattern Simulator

Embry-Riddle Aeronautical University developed a demonstration of how aircraft enters the airspace pattern around an airport using Java.

Alternative Path Calculator

Princeton University has created a Java-based best path estimator between any two points on the Puerto Rico road network. This applet also calculates an alternative path. Specifically, it finds a path that has at most half of the links in common with the best path.

Automated Travel Assistant

A University of Washington Model mimics the dialogue between a travel agent and a client. This model uses real time airline flight data provided by the Internet Travel Network.

BART Simulation

A Java-based model developed by the Lawrence Berkeley Laboratory simulates the (scheduled) movements of Bay Area Rapid Transit (BART) vehicles.

Freeway Traffic Simulator

A traffic simulator model developed at San Jose State University moves vehicles along a discrete grid.

Interactive Routing for Minneapolis

University of Minnesota developed a shortest path calculator that includes the street network for the City of Minneapolis.

Interactive Subway Map

Transarc Corporation developed an applet that allows you to find the shortest route from one station to another in the New York City subway system. Routes are time-dependent (i.e., they account for the fact that the schedule varies during the day).

Interactive System Map

Fancois Vaillant developed a passenger information system that helps you find your way around in the Paris, Lyons, and Los Angeles subway systems.

Logistics Simulation

America Cargo Service developed an animated demonstration of the flow of documents, money and materials from the shipper to the consignee.

Network Traffic Simulator

Los Alamos National Labs developed a multithreaded applet that simulates vehicle movements on a simple traffic network that includes traffic signals under the user control.

Science and Technology of Decision-Making

Princeton developed a course designed to teach engineering to non-engineers. Several of the labs consider transportation topics. The Path Calculator allows you to digitize (small) networks and solve shortest path problems (or you can use the NJ highway network). The Congestion Pricing Evaluation Model solves traffic equilibrium problems on a small network, enabling students to determine the impacts of different toll policies. The Toll Plaza Simulator is an interesting (though not terribly realistic) simulation of a complex queuing system.

Traffic Control System Simulation

Philips developed a traffic control system simulation that allows you to simulate a traffic control system that includes four traffic lights.

Train Brain

Reynolds Transit Software Animation of various transit systems based on posted schedules.

University of Waterloo developed a long distance cyberspace course. Bullet proof developed an investment service that uses Java to fire-up online services for the Web. Kodak brings high quality editable images to the Web using Java (Photo CD on the Web).

Database Access via the World Wide Web

Donnelley developed WebDIRECT, a Java Powered Internet-based information distribution service that allows users to access online relational databases.

Weather Underground

University of Michigan scientists developed a Java Powered application that allows user to see instantly updated weather conditions throughout the nation.

CSX, an international transportation company, is the pioneer in transportation industry developed commercial Internet applications with Java. Their new TWS-net software is the interactive Shipment Tracking module with a Java graphical map that lets customers visually see where their goods are on the network. The first phase included shipment-status queries, E-mail, an address book, and account information in addition to the shipment tracking module. TSW-net provides customers the ability to monitor and track their shipments. It also provides them the ability to interact with us in terms of how those shipments should be managed more effectively as well. CSX Technology estimates that programming in Java will save the shipping firm \$5 to \$7 million dollars in development and maintenance costs on an annual basis.

2.4 Characteristics of Object Oriented Programming

There are four important characteristics of Object Oriented Programming: 1) data encapsulation, 2) polymorphism, 3) inheritance, 4) reusability of code and 5) model assembly. The following paragraphs briefly describe these characteristics in more detail.

1. Data Encapsulation

OOP provides special protection to data that belong to the object or a set of objects. This means an object can prevent some other unrelated part of the program from accidentally modifying it or from incorrectly using the private parts of the object [Shildt, 1991]. For

example, an aircraft object holds its aircraft type, weight, load factor, fuel consumption rate, flight schedule, flight number, landing runway, gate, etc. as private data. These data can only be changed or retrieved by authorized functions. It is impossible that, for example, the flight schedule in aircraft A is replaced by flight schedule for aircraft B accidentally.

2. Polymorphism

OOP languages support polymorphism. Polymorphism is characterized as “one interface, multiple methods.” One of the main purposes of polymorphism in OOP is to reduce the number of function names programmers or end users need to remember. For example, the advantage of polymorphism could be illustrated in the following examples:

Example No. 1:

In the equation
 $1 + 1$,
operator + means adding the quantities of two numbers.

Example No. 2:

In the equation
 $runway = node1 + link1 + node2 + link2 + node3$,
operator + is overloaded as a set of connecting actions that will connect a node object node1 to a link object.

Example No. 3:

In the operation
 $aircraft_A + flight_schedule_B$,
operator + is overloaded as assigning flight_schedule_B to aircraft_A.

The advantage of polymorphism in OOP is obvious. Hundreds of different methods or function calls can be managed by one interface, the amount of time saved in code writing and

logical management of data flow is significant.

3. Inheritance

Inheritance is the property that allows one object to acquire the properties from another one. For example, a B737 airplane is classified as a transport airplane, which in turn is part of the airplane class, which in turn belongs to the large class vehicle. Without the use of classification, an airplane would have to define explicitly with all its characteristics in every category. In the real world, researchers studying airport operations may need to know the time-space parameters of airplanes while an airline manager will be interested not only in the capacity-delay issues but also in the the load factor and fuel consumption of each aircraft. The inheritance mechanism makes it possible for one object to be an “instance “of a more general case.

4. Reusability

In OOP, just as the way cars are made, objects are classified, written, created, and debugged in advance, then, it can be distributed or used by other programmers in their own programs. Further more, because of inheritance, a programmer can take an existing class and, without modifying it, add additional features and capabilities to it. The marketing analogy of this is that, objects are “manufactured” and “supplied” to the other program “assemblers.” They can use these objects directly or make new objects from base objects and “sell” their new objects, in turn, to other users.

5. Model Assembly

In the procedural programming style, sub routines are usually developed to fit one or several functional needs. They are called in some designated places to perform certain previously defined processes. In OOM objects or class library are developed to mimic the behavior of real world physical objects before any program integration. Models are assembled

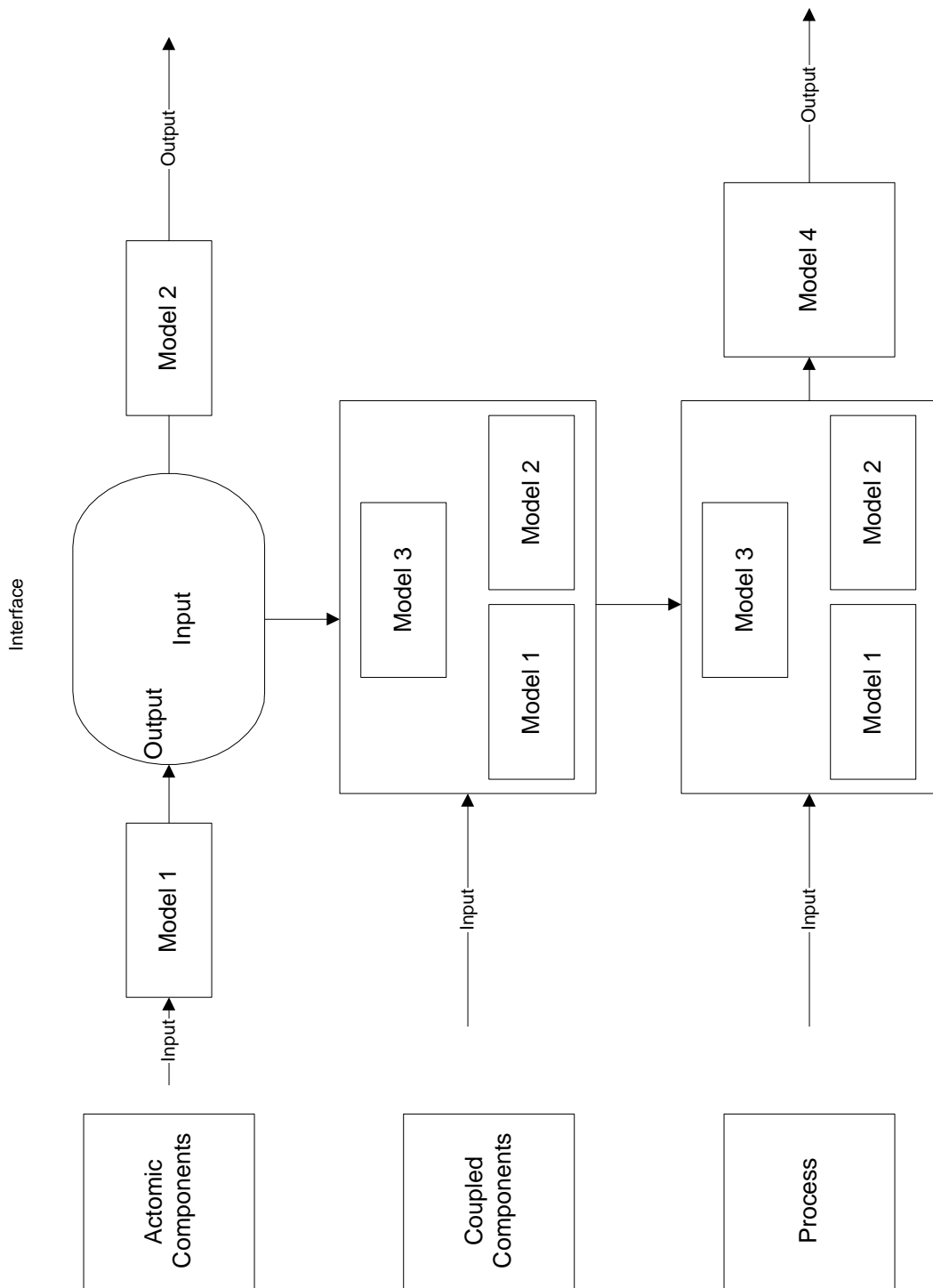
using different system configurations. A hierarchical modeling concept [AbouRizt, 1994] was presented in Figure 2.1.

2.5 Comparison of Traditional Modeling Paradigm and Object-Oriented Paradigm

Differences between traditional modeling paradigm and object-oriented paradigm are summarized by Mize et al [Mize, 1992]. Table 2.1 contains a summary of several factors on which object oriented paradigm is different from traditional modeling paradigm.

2.6 Current Applications of Object-Oriented Method in System Modeling

Object-oriented method is widely used in different facets of system modeling. Several related research papers are reviewed. Mize used an object-oriented approach in the modeling and simulation of complex manufacturing system within a computer integrated manufacturing framework [Mize, 1992]. Oloufa utilized object-oriented concepts in the modeling and simulation of construction operation regarding an earthmoving project; his model was implemented with MODSIM, an object-oriented simulation language [Oloufa, 1993]. In Siman's technical discussion [Siman, 1994], Luna presented a hierarchical modeling concept similar to that shown in Figure 2.1. This method uses two types of building blocks, the atomic model component and the coupled model component, as the basic building blocks in a simulation model. Reitsma presented an object-oriented modeling tool kit to implement river-basin models [Reitsma, 1994]. Application of object-oriented approach and learning automata routing techniques in telephone traffic routing problem was presented by Zgierski [Zgierski, 1994]. In Zgierski's approach, the telephone network is represented by objects characterizing nodes and trunk lines with each traffic routing method treated as an object. Therefore a traffic routing problem could be solved in two steps: 1) constructing the tele-



**Figure 2.8 Hierarchical Modeling Concepts Presented by Luna (1990),
reproduced from AbouRizk(1194)**

Table 2.1 Contrasting Traditional and Object-Oriented Paradigms for Simulation Modeling, (reproduced from [Mize, 1992]).

Factors	Traditional Modeling Paradigm	Object-Oriented Modeling Paradigm
Model Construction:		
Software	Simulation languages based on procedural programming style	Simulation environments based on object-oriented programming style
Translation into code	Process is abstract	Process is natural and intuitive
Interface	Usually textual	Usually graphical with icons and dialog boxes
Level of detail	Usually not much detail due to programming complexity	At user's discretion, but requires detailed object library
Treatment of distinct system elements	Different element types are not distinctly modeled; Aggregation to reduce program complexity	Physical, information, and decision/control elements are modeled distinctly and independently
Effort/time/cost	Moderate costs of model development, but a "throw-away" type	Initial cost of establishing detailed model is very high, but costs of subsequent reuse is relatively low
Model Attributes:		
Purpose	Usually a unique model is created for a specific purpose	More general models possible for multiple purposes
Usage	Single usage, throw-away models	Repeated usage and continuous refinement
Flexibility	Highly inflexible; changes almost always result in a complete rewrite of program	Highly flexible, due to the ability to modify fundamental building blocks; quick reconfiguration is possible
Accuracy	Useful for measuring relative differences in alternative configurations	With greater degree of detail and realism, can also estimate absolute performance with greater accuracy
System/Model Relationship	Not connected via data links	Detailed model can be imbedded in control structure of the firm, with linkages to databases; continuous model calibration and parameter updating

phone network with of node and trunk objects and 2) using different method objects to search the traffic routes. Lefrancois presented the concepts behind the implementation of behavioral and cognitive capabilities in objects used for controlling workstations in manufacturing systems. These objects intuitively play the role of human decision-makers in the system modeled [Lefrancois, 1994]. Hollister's report on airport surface traffic automation (ASTA) system is, in the best of my knowledge, the first technical document giving detailed conceptual description on the application of object-oriented approach to model the airport surface traffic automation system [Hollister, 1988]. Hollister's research presented in details the prototype of ASTA system. However his report did not discuss the details about object programming.

2.7 Current Airport Systems Simulation Tools

Simulation tools for airport systems can be classified into two categories. The first category includes commercial airport simulation packages such as SIMMOD [CACI, 1993] and The Airport Machine [Joline, 1993]. Both of these two simulation software are very popular in the airport engineering community. The advantages of these simulation tools are their proven performance in multiple airport configurations and their flexibility to model many airport network topologies. The major drawbacks of them are that these simulation packages use procedural programming styles. Therefore the real world systems are aggregated to reduce program complexity. This aggregation decreases the capability of the software for detailed system analyses. Furthermore these simulation packages are simulation oriented. It is difficult for an end user to retrieve the simulated data for specific purpose analyses, not to mention the possibilities to integrate the models developed from these packages with real time management or decision support systems. The second category includes generic simulation languages such as SIMSCRIPT, GPSS, MODSIM, and STELLA. Using

these generic simulation languages to model airport systems requires considerable amount of time and labor costs. All these languages require simulation engines or preprocessors which would make end users spend high costs in the model development phase.

2.8 A General Framework for Airport Operation Modeling

Object-oriented approach is widely adapted in different facets of system model and problem solving. OOA of airport operation modeling is currently being studied but very few research reports are available, especially in programming related issues. Furthermore, most of the studies used special OOP languages that are not easy to port the model onto different computer platforms.

A generic airport modeling library using object-oriented approach and written in C++/Java will significantly reduce the disadvantage associated with those commercial simulation packages mentioned above. C++ is an object oriented programming language with wide industrial support. Programs written in C++/Java are more portable between platforms and less affected by the companies that provide the C++/Java compilers. The creation of a generic airport simulation library will simplify the task of future model building as users do not need to reinvent the “wheel” [Lafore, 1995]. The late binding feature of OOM allows model builders assemble their own models with much fewer redundant efforts which are definitely needed if models are developed from scratch.

Chapter 3 Designing A Task Engine to Simulate Airfield Operations

3.1 Description of Airport Operation System Dynamics

System dynamics theories are widely used in system modeling. Object-oriented modeling technology provides analysts with detailed descriptions on how relationships among different system components interact. In an object modeling framework, the airport system can be divided into individual objects such as airplanes, runways, taxiway, and air traffic control centers. Each of these objects represents a physical object in the real world. The communications among airplane and air traffic controller objects become the links inside the airfield traffic system and therefore represent the dynamics of the system. In a task driven simulation environment, the communications between an airplane and air traffic controller objects could be regarded as message flows between two objects. Task objects are used to simulate or model the airfield system dynamics. For example an airplane asks an air traffic controller for permission to land on an assigned runway. To model this airport system activity, an airplane object must send a landing request to the air traffic controller object. Upon receiving the request, the air traffic controller object will query current position information of all the airplanes that could possibly be on the path of the incumbent aircraft and check the availability of the runway. If the runway is available, then a landing permission message is sent to the target airplane and the landing process starts.

Airport operation tasks are airplane oriented. Communications among objects are mainly between airplane and air traffic controller objects. For example, landing an airplane on a runway, several related tasks are involved. The creation of landing tasks follows a well-defined sequence. The simulation engine generates an arrival task, this task creates an airplane object when the global simulation clock time reaches the time mark when the airplane is supposed to arrive. The airplane sends a landing request task to the air traffic control tower; the air traffic control tower receives the landing request task and sends the task to the corresponding process; this process sends a task to the runway making a query of the current traffic situation; the runway object sends back the current traffic information; the control tower sends two tasks out, one tells all the other targeted airplanes to hold their position and the other tells the incumbent landing airplane to start the landing process; the airplane starts the landing process and sends an task to the runway to change its occupancy status; once the airplane finishes the landing process, it sends one task to tell the air traffic control tower object and another task to change the runway occupancy status again.

The discussion above is a simple description of airfield traffic system dynamics. Tasks for airfield operations vary in complexity depending on the purpose of the actual airfield traffic model. Because an object-oriented technique allows an object class to be inherited from its parent object, different tasks could be used to fit in the real situation and meet the functional requirements of the model. Nonetheless, a task management object is necessary for all the simulated situations. A task management object or task engine should be able to collect and dispatch all the tasks regardless of the actual supplement data attached to that particular task object.

3.2 Task Driven Simulation Engine

In airfield traffic modeling, tasks are organized with a queue object. All tasks generated queue in the task queue object. The task engine dispatches tasks one at a time until the simulation clock stops. All tasks shown in Figure 3.1 are arranged sequentially by the task engine which has the following main functions:

1. **Collecting tasks**

Whenever a task is created during the simulation, the task engine will add the task into the task queue.

2. **Dispatch tasks**

The task engine will dispatch the task into the task process channel whenever the global time matches the time mark of a task in the task queue.

3. **Branching tasks**

The task process channel will branch the task to the correspondent sub-channel.

4. **Pending tasks**

If the task cannot be executed at the moment and it is expected to be executed as soon as possible, the task can be pushed back into the task queue and its time mark set the same as that of the next available task.

5. **Deleting a task**

If a task is executed or need to be deleted, the task engine will remove and destroy the task.

3.3 Queuing with Priority

In an airfield traffic system simulation, tasks could be created early and executed late, such as scheduling a flight. On the other hand, tasks could be created late but required to be executed immediately after their creation, such as holding an airplane in a departure queue. If both tasks are created in a simulation, the first criterion will not be appropriate because the task created late should be executed first.

The priority queue object created in this library is a classical first in-first out queue. Before adding a task to the task queue, the task engine will first compare the priorities of all of the tasks in the queue with the priority of the newly created task, and then add the task into the queue at the right position according to the outcome of the priority comparison. The characteristics of priority queuing make the task engine be able to handle scenarios that are close to real airfield traffic situations. Priority queues are also modeled in runway operations because landing aircraft have priority over departures.

3.4 Interleaved Interpretation of Concurrency

Time could be thought in two different ways by most of system analysts. The interpretation is based on ordinary everyday experience whereas the second one is based on the idea of handling multitasks in a single control device, such as a computer. The fundamental difference is in the ways how concurrent events are handled. The simultaneous interpretation explains that two given events could occur at the same time. The interleaved interpretation explains that two events could occur at the same time slot, but one at a time within the same time slot. The interleaved interpretation of concurrency is supported by many multitask computer operating systems as well as this study. This is illustrated graphically in Figure 3.2.

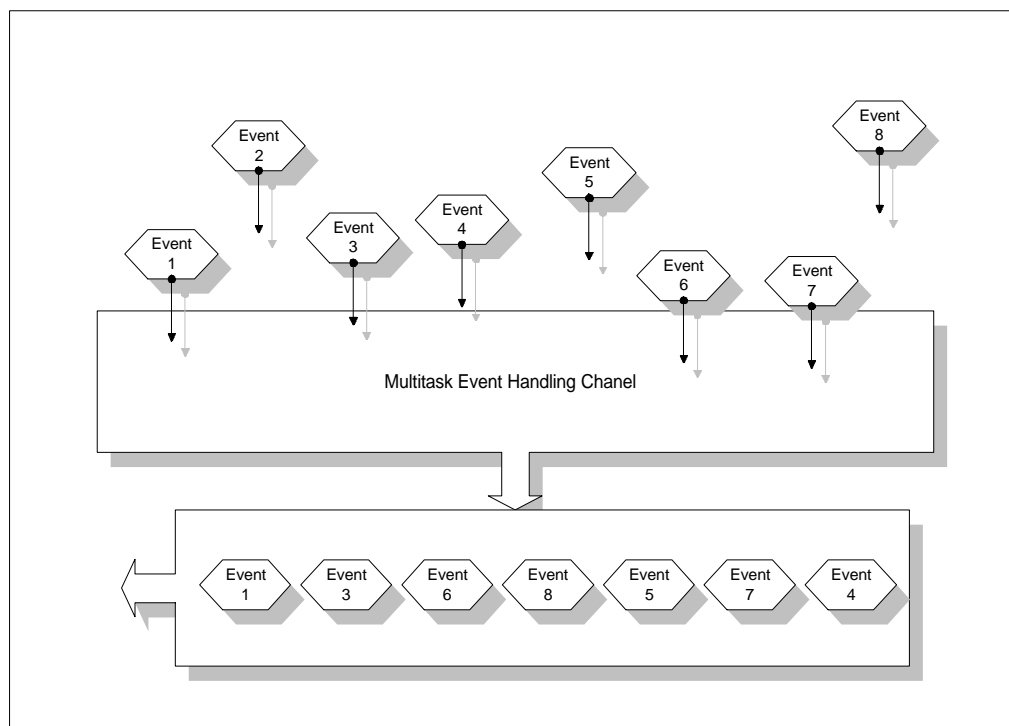


Figure 3.1 Multitask event (tasks) handling channel.

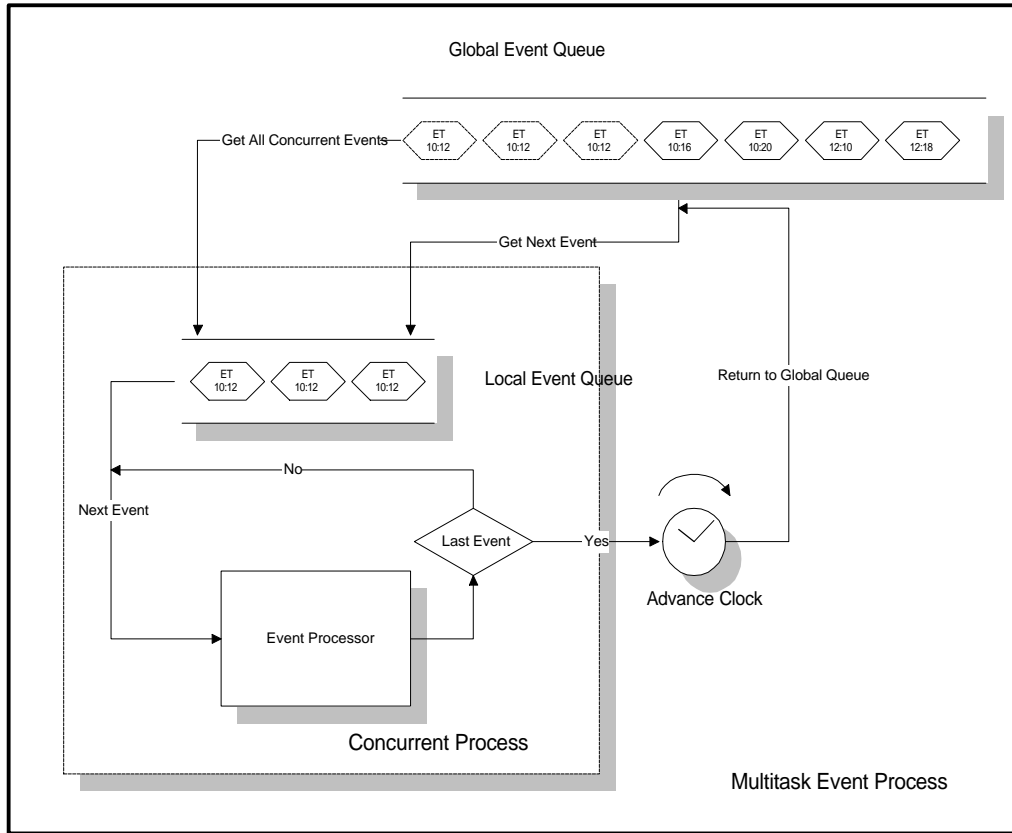


Figure 3.2 Illustration of interleaved concurrent process.

3.5 Basic Structure of the Task Class

A task is an abstraction of an incident in the real world that indicates when an object is moving to a new state. The actual prototype of a task could be different depending on the needs of the system. However, any given task must be equally handled by the task engine. This requirement for a task includes two pieces of information: an identifier and supplemental data. The task engine will generally handle the identifier and the supplemental data as sub-task process channels after the tasks are branched or within the targeted objects. In this study a generic task usually has both identifier and supplemental data.

3.5.1 Identifier data

Identifier data is a set of data that specifies which object is going to receive a task. Supplemental data is a set of attributes that tell the targeted object what is happening and usually what needs to be done. A task in any airfield simulation should have the following elements:

- a. **The primary task label:** Whenever a task is dispatched, the first task label which tells what task it is, a scheduling task, an airplane related task, or a shortest path searching task. According to the primary task identifier, the task dispatcher will send the task into corresponding task processing channel. The primary label is used in the primary branching process.

Example of primary labels:

```
enum {  
    AIRPLANE_EVENT;  
    RUNWAY_EVENT;  
}
```

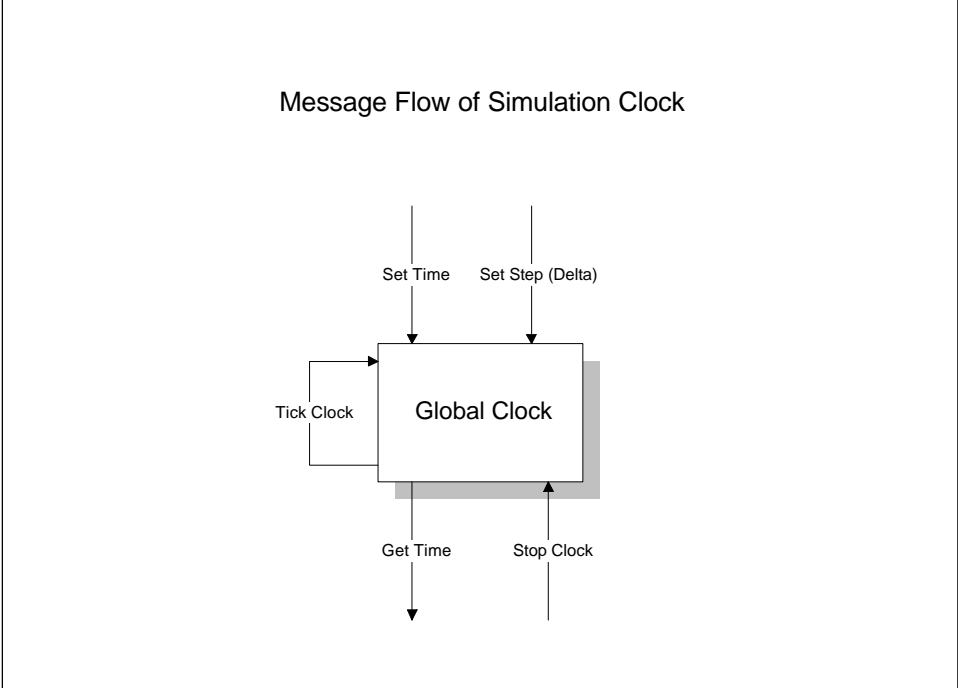


Figure 3.3 The message flow of the global simulation clock.

```
TAXIWAY_EVENT ;
GATE_EVENT ;
CONTROL_EVENT ;
FLIGHT_EVENT
};
```

- b. **The secondary task label:** Whenever a task is sent to the corresponding channel, a task is sent of the simulation task queue and into the target object, These objects could be an air traffic control processor or an airline task processor. Once the respective task processor receives it, the task will then be dispatched again to different process channels depending on its secondary task identifier. For example, the airplane task processor will dispatch an airplane task into the following built-in sub-task process channels such as, moving, holding, parking, or landing. The secondary label is used in the secondary branching process.

Example of secondary labels:

```
enum {
    MOVE_AIRPLANE_EVENT ,
    HOLD_AIRPLANE_EVENT ,
    PARK_AIRPLANE_EVENT ,
    LAND_AIRPLANE_EVENT ,
    DEPART_AIRPLANE_EVET ,
    NEXT_ACTION_EVENT
};
```

- c. **Priority data set:** tasks queue in the task engine according to their assigned priorities. Tasks are dispatched by in the order of their priorities other than the order that they are created. The built-in priority in this simulation engine is the time mark at which the task is supposed to be dispatched. Whenever a task is created, the time mark at which the task is to be dispatched is set. Whenever a task is added to the task queue, the simulation engine will automatically search for a right position in the task queue and add the task in that position. This

mechanism is very useful in a simulated air traffic control environment.

3.5.2 Supplemental Data

Task information set: The task information contained in all of the task objects is the information of time, task primary label, secondary label, and contents of actions. For example, a task that orders an airplane to move from point A to point B contains the following information: a primary label to indicate this is an airplane task, a time mark to tell when the airplane will move, a secondary task label to indicate that this is a moving task, and a set of instructions attached to the task object about information of point A and B as well as other necessary messages.

The following is an example piece of source code for a task definition:

Example: data structure for a flight schedule task

```
typedef struct
_flight_schedule_struct_def{
    int type;
    int from;
    int to;
    int direction;
} ZflightScheduleEvent;
```

Example: data structure for an airplane related task

```
typedef struct _AirplaneEvent {
    int type;
    int from;
    int to;
    int flow;
    int target_ID;
}ZairplaneEvent;
```

Example: data structure for a generic task

```
typedef struct _AsimEvent {
    int type;
    int hour;
    int minute;
    double second;
    ZairplaneEvent
    AirplaneEvent;
    ZrunwayEvent
    RunwayEvent;
    ZtaxiwayEvent
    TaxiwayEvent;
    ZgateEvent GateEvent;
    ZcontrolEvent
    ControlEvent;
    ZflightScheduleEvent
    FlightEvent;
} AsimEvent;
```

3.6 Structure of Simulation Clock Class

A simulation clock is a global time object which governs the simulation process preset time frame. A global clock must be able to communicate with other objects such as task, airplanes, and air traffic control. At the same time the global clock must be an independent object. A global clock consists of a global time structure which has hour, minutes, and seconds; and methods to advance the clock, set the time, and get the current time. Methods of getting and setting the global clock allow other objects to communicate with the clock, especially the task engine which dispatches a task according to the current clock time and the time mark set in the task objects.

Examples:

Create a simulation clock call global_clock:

```
time_class global_clock;
```

Set current time to 12:30:38:

```
global_clock.set_value(ZtimeClassNsecond,  
38);  
  
global_clock.set_value(ZtimeClassNminute,  
30);  
  
global_clock.set_value(ZtimeClassNhour,  
12);
```

Get current time:

```
int hour;  
int minute;  
double second;  
  
global_clock.get_value(ZtimeClassNsecond,  
&second);  
  
global_clock.get_value(ZtimeClassNminute,  
&minute);  
  
global_clock.get_value(ZtimeClassNhour,  
&hour);
```

Advance the simulation clock:

```
global_clock.tick();
```

3.7 Communications between the Task Queue and Global Clock

The communication between the priority task queue and the simulation clock can be established by using a function call. This communication can be executed in the simulation using the following steps.

Step one: initialize the global clock.

```
time_class global_clock;
```

Step two: initialize the task engine

```
EventProcess Engine;
```

Step three: get current time

```
int hour=0;
int minute=0;
double second=0.00;

global_clock.get_value(ZtimeClassNsecond,
&second);

global_clock.get_value(ZtimeClassNminute,
&minute);

global_clock.get_value(ZtimeClassNhour,
&hour);
```

Step four: send the current time to the priority task queue.

```
if(Engine.event_true() != 0)
Engine.run(&hour, &minute, &second);
```

Step five: update the current clock.

```
global_clock.set_value(ZtimeClassNsecond,second+delta);

global_clock.set_value(ZtimeClassNminute,
minute);

global_clock.set_value(ZtimeClassNhour,
hour);
```

Finally, the complete action could be implemented with an infinite loop. The following function call illustrates this idea.

```
void asim_start_mainloop()
{
int hour=0, minute=0;
double second=0.00;

while(!kbhit())
```

```

{
global_clock.get_value(ZtimeClassNsecond,
&second);

global_clock.get_value(ZtimeClassNminute,
&minute);

global_clock.get_value(ZtimeClassNhour,
&hour);

if(Engine.event_true() != 0)
Engine.run(&hour, &minute, &second);

global_clock.set_value(ZtimeClassNsecond,
second);

global_clock.set_value(ZtimeClassNminute,
minute);

global_clock.set_value(ZtimeClassNhour,
hour);

if(asim_advance_simclock())
break;
}
}

```

3.8 General Task Handling Mechanism

Airport activities are abstracted as task driven activities. These tasks could be scheduling a flight, assigning a path to the airplane, moving an airplane from point A to point X along a feasible path, changing the assigned path in the course of movement. Holding an airplane at point X for a period of time.

A simulation engine that can handle airport simulation tasks within a given simulation time frame should have the following characteristics:

a. Create a task process engine:

```
EventProcess Engine;
```

b. Accept a task to the task queue at any time.

```
Engine.AddEvent(task);
```

c. Run the task engine up to a given time mark (for example 10:20:30).

```
int hour = 10;  
int minute = 20;  
double second = 30.0;  
  
if(Engine.event_true() != 0)  
Engine.run(&hour, &minute, &second);
```

d. When a task is dispatched, the action should be finished or new task created.

e. At any time, the added task will be assigned at the right position according to the time at which it is supposed to be dispatched. In other words, the task queue will be preempted. All of the tasks are sorted when they are added to the task queue.

f. If the task can not be done, the task could be pushed back or a new task generated and added to the task queue for actions in the future.

3.9 Task Analysis of Airfield Traffic Operations

The following is a description of various tasks that have been used in the development of the airport simulation framework used in this research.

3.9.1 Basic Airplane Related Tasks

Landing Task:

Air traffic control tower sends the landing permission to an approaching airplane, upon receiving the landing permission, the airplane proceeds to land on the runway.

Destination: The targeted airplane.

Label: Land_an_Airplane_Task

Task Data: The identifier is airplane's ID and the supplemental data includes designated runway, exit and gate.

Exiting Task:

When an airplane leaves a network segment such as a runway or a taxiway, it sends a task to a network link to change the occupancy status of the link.

Destination: The targeted runway, taxiway or gate.

Label: Exit_a_Link_Task

Task Data: The identifier is the airplane ID and the supplemental data includes the runway IDs, taxiway IDs or Gate ID of current link and next link.

Entering Task:

When an airplane enters a network segment such as a runway or a taxiway, it sends a task to a network link to change the occupancy status of the link.

Destination: The targeted runway, taxiway or gate.

Label: Enter_a_Link_Task

Task Data: The identifier is airplane's ID and the supplemental data includes the runway IDs, taxiway IDs, or gate IDs of current link and next link.

3.9.2 Basic Traffic Network Related Tasks

Receiving an Object Task:

When a moving airplane enters a traffic network such as a runway, taxiway or gate, these network objects receive a signal that tells an airplane is moving into its internal queue.

Destination: The targeted traffic network object such as a runway, a taxiway and a gate.

Label: Receive_an_Object_Task

Task Data: The identifier is the network object ID and the supplemental data includes the entering airplane ID and time at which the airplane enters the object's internal queue.

Releasing an Object Task:

When an airplane leaves a traffic network object such as a runway, taxiway or gate, these network objects receive a signal that tells an airplane is moving out of the their internal queue.

Destination: The targeted traffic network object such as a runway, taxiway and gate.

Label: Release_an_Object_Task

Task Data: The identifier is the network object ID and the supplemental data include the airplane's ID and time at which the airplane leaves the object's internal queue.

Close_This_Link Task:

When a traffic network object such as a runway, taxiway, and gate is not avail-

able to any traffic and temporary storage, the network object is closed.

Destination: The targeted traffic network object.

Label: Close_This_Link_Task.

Task Data: The identifier is the targeted traffic network object's ID and the supplemental data includes the time at which the targeted traffic network object is closed.

Open_This_Link_Task:

When a traffic network object such as a runway, taxiway, and gate is available to serve any traffic flow and temporary storage, the network object is open.

Destination: The targeted traffic network element.

Label: Open_This_Link_Task.

Task Data: The identifier is the targeted traffic network element's ID and the supplemental data includes the time at which the targeted traffic network element is open.

3.9.3 Traffic Control Related Tasks

Moving an Airplane Task

The air traffic control tower sends a command to an airplane in the traffic network to move on, the task is triggered.

Destination: The targeted airplane.

Label: Move_an_Airplane

Task Data: The identifier is the airplane's ID and supplemental data include the original link and the next link, starting time, and ending time.

Holding an Airplane Task

The air traffic controller sends a command to an airplane in the traffic network to hold its position, the task is triggered.

Destination: The targeted airplane.

Label: Hold_an_Airplane_in_Position

Task Data: The identifier is the airplane ID and the supplemental data include the designated link and starting time.

Parking an Airplane Task

The air traffic control tower sends a command to an airplane in the traffic network to park at a gate, the task is triggered.

Destination: The targeted airplane.

Label: Park_an_Airplane.

Task Data: The identifier is the targeted airplane ID and the supplemental data includes the designated gate ID, and the starting time.

The following discussion explains in some detail how the air traffic components of the proposed simulation engine have been designed.

4.1 Airplane List Class

An airplane list object contains a list of airplane **object pointers** each of which points to the individual airplane object and several **methods** to execute various tasks. These methods are:

- a) a method to create an airplane for scheduling a arrival of departure
- b) a method to assign an airplane ID number to a newly created airplane
- c) a method to add an airplane into the list
- d) a method to remove an airplane from the list
- e) a method to sent a task to the target airplane with given airplane ID and,
- f) a method to get the target airplane object with a given airplane ID.

The airplane list object provides all the management functions to control every aircraft in the simulation. The following example illustrates the use of an aircraft list object. Figures

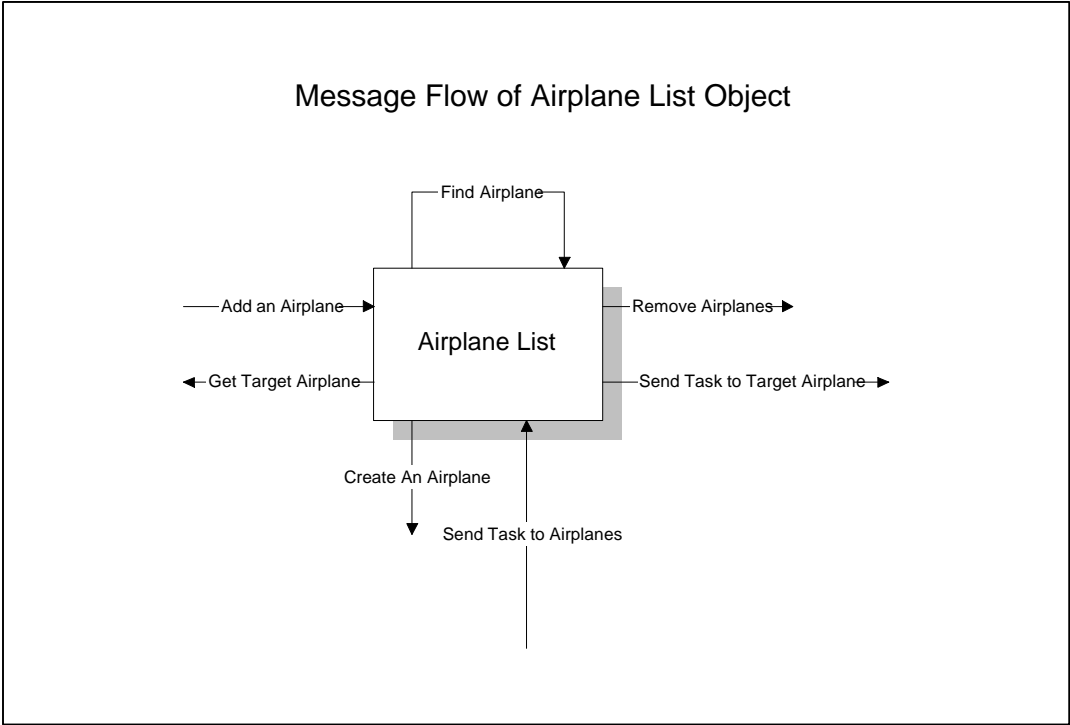


Figure 4.1 Message flow of airplane list class.

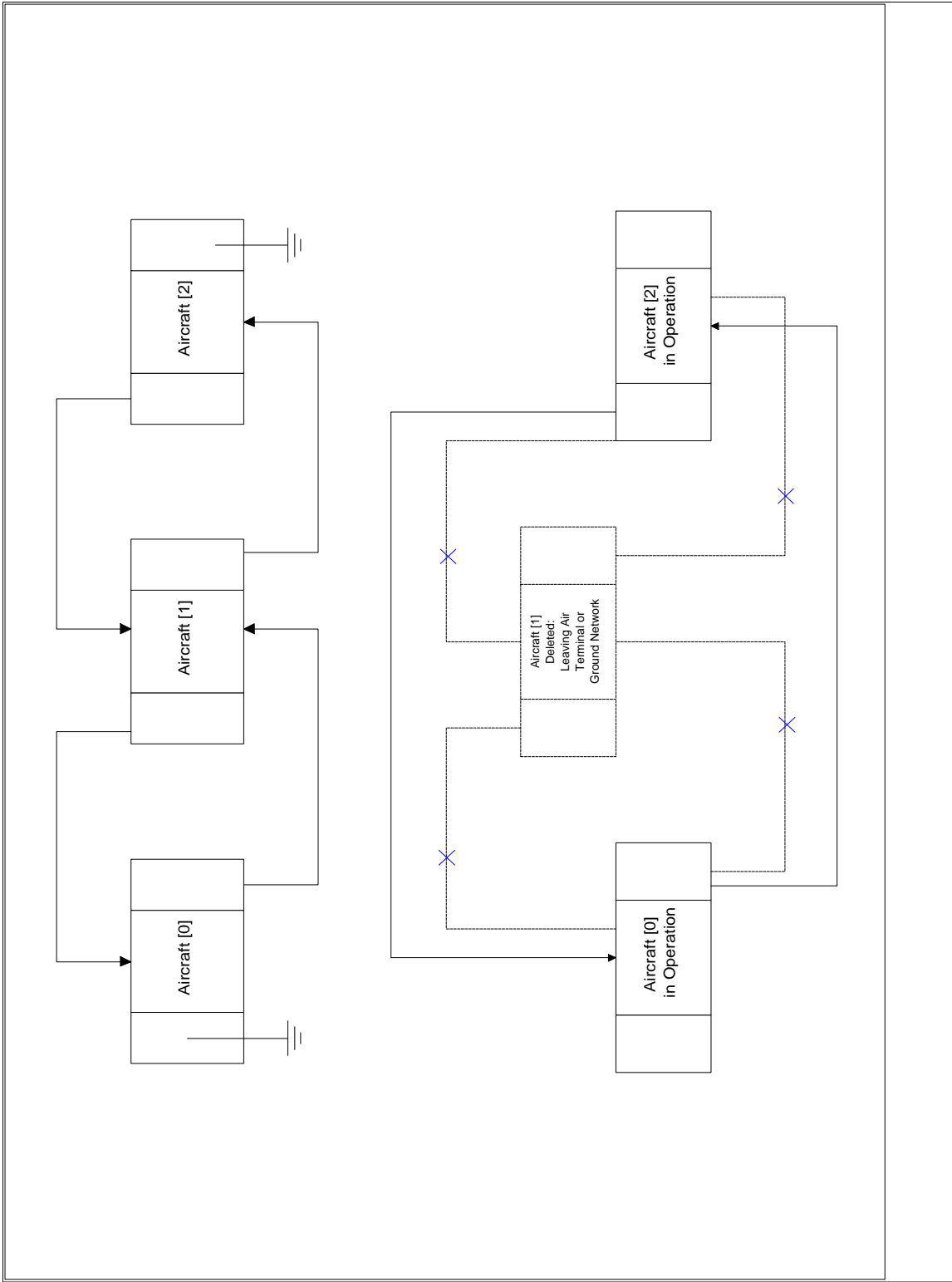


Figure 4.2 A linked list structure of the airplane list.

4.2 and 4.2 illustrate the list object concept graphically.

Examples:

1. Creating an airplane list object call `airplane_fleet`:

```
AcftList airplane_fleet();
```

2. Adding an airplane into the `airplane_fleet` with given orientation and destination:

```
int orientation;  
int destination;  
int airplane_id;  
airplane_id =  
airplane_fleet.add(orientation,  
destination);
```

3. Sending a task to the airplane:

```
airplane_fleet.send_event_to_target_airplane(task);  
where task is a task object;
```

4. Getting an airplane object with given airplane ID:

```
Airplane_class *this_airplane;  
this_airplane =  
airplane_fleet.get_target_airplane(ID);
```

5. Removing an airplane from the `airplane_fleet`:

```
airplane_class *removed_airplane;  
int ID = 10; // airplane No. 10  
removed_airplane =  
airplane_fleet.get_target_airplane(ID);  
airplane_fleet.remove_airplane(removed_airplane);
```

4.2 Airplane Class

4.2.1 Description of Airplane States during the Simulation

Behavior patterns in real world objects can be described with states and their transitions. Many objects, creatures and machines alike, go through various stages during their lifetimes in the systems where there are studied. For example an airplane can be parked at a gate, taxi to the runway and depart, or on the opposite direction, an airplane can land, taxi out of the airfield network, and park at the gate. The order in which an object progresses through its stages forms a characteristic pattern compliant with physical laws. A real-world entity is in exactly one stage of its behavior pattern at any given time. An airplane cannot be parked at a gate and taking off at the same time. Objects progress from one stage to another abruptly because of the ways their are perceived. We define takeoff to be a stage of the airplane behavior representing the complete acceleration maneuver ending when the wheels are off the ground. In a behavior pattern, not all progressions between stages are allowed. Some progressions are forbidden by physical constraints. For example, physical constraints prevent a fixed-wing airplane from progressing directly from the flying phase to a parking position at a gate.

There are incidents in the real world that cause entities to progress (or indicated that they have progressed) between stages. Starting the airplane engine causes the airplane to progress from parking at a gate to taxiing to runway.

As previously described, any real world objects could be represented with a set of abstract attributes and several stages. A life cycle is a set of all the stages that a real world object can take. Throughout the life cycle only one stage can be present at any one time.

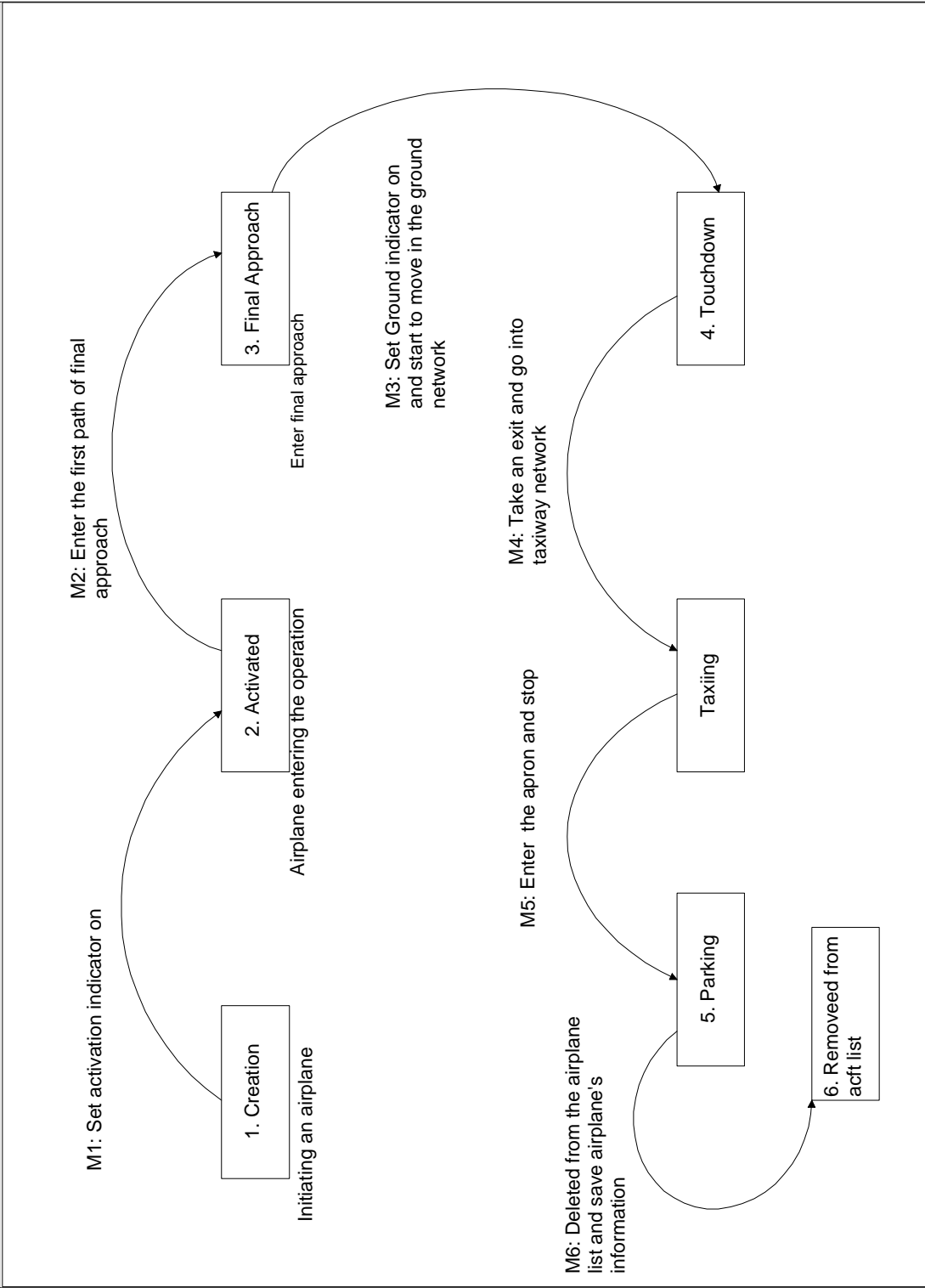


Figure 4.3 State model for a landing airplane

Therefore a life cycle is also called a state model in OOM. The state model is composed of the following:

1. A set of states: each state represents a stage in the life cycle of a typical instance of the object.
2. A set of tasks: Each task represents an incident or indication that a progression is happening.
3. Transition rule: A transition rule specifies what new state is achieved when an instance in a given state receives a particular task.
4. Actions: An action is an activity or operation that must be accomplished when an instance arrives in a state. One action is associated with each state. A state transition diagram is a diagrammatic form which represents a state model.

An airplane in the simulation will have a set of states regardless to its airplane type and airline ownership. Each instance of the airplane object will be in only one state at any time during the simulation.

An airplane object enters the simulation cycle from either a gate at the airport ground network or the final entry gate in the airport terminal airspace. An airplane leaves the simulation cycle from either a gate in the ground network or the final gate in the airport terminal airspace. Airplanes in most airport simulations are categorized into two flows, arrival and departure. An airplane in the arrival flow has valid states such as final approach, touch down, decelerating on the runway, network taxiing, holding, parking at the gate. An airplane in the departure flow has feasible states such as parking at the gate, taxiing on the airport ground

network, holding, taking off, moving in the air terminal and leaving the air terminal.

4.2.2 Airplane States

Creation: An airplane can be created in advance before it enters the simulation cycle or an airplane object can be created any time during the simulation. The following states apply to each aircraft. These states are illustrated graphically in Figure 4.3.

Air Phase: airplane flight in the air terminal; the airplane could be in an arrival or departure flow.

Touchdown: The arriving airplane lands on the runway. This state includes a generalized speed schedule on a runway— touchdown, free roll and braking.

Takeoff: The departing airplane accelerates on the runway and leaves the ground.

Taxiing: An airplane moving inside the ground taxiway network toward the runway end (taxiing out) or toward the gate (taxiing in).

Holding: An airplane holding its position in order to yield other traffic tasks and waiting for commands from the control tower.

Parking: An airplane parking at the gate for either uploading of downloading passengers.

Final State: An airplane object can be destroyed any time during the simulation,

when the airplane is parked at a gate.

In summary, an airplane will be in any one of the previously mentioned states.

4.2.3 Structure of Airplane Class

An object of the **airplane class** contains a set of variables: 1) airplane ID, 2) airplane orientation in the network, 3) the point at which the airplane enters the simulated airport traffic network, 4) airplane destination at which the airplane leaves the simulated traffic network, 5) the airplane current position, 6) next position, 7) total travel time or abstract type of total travel cost, 8) assigned best travel path, and 9) destination indicator. The basic attributes are illustrated in the following piece of source code.

```
class Airplane_class {
    // airplane ID
    int ID;
    //airplane orientation
    int orientation;
    //airplane destination
    int destination;
    //current starting point
    int from;
    //next point
    int to;
    //total travel cost
    int total_cost;
    //destination indicator
    int reach_destination;
    //best travel path
    char path[60];
    //time mark at which the air
    plane enter the simulation
    time_type start;
    //time mark at which the air
    plane reaches the
    //destination
    time_type end;
};
```

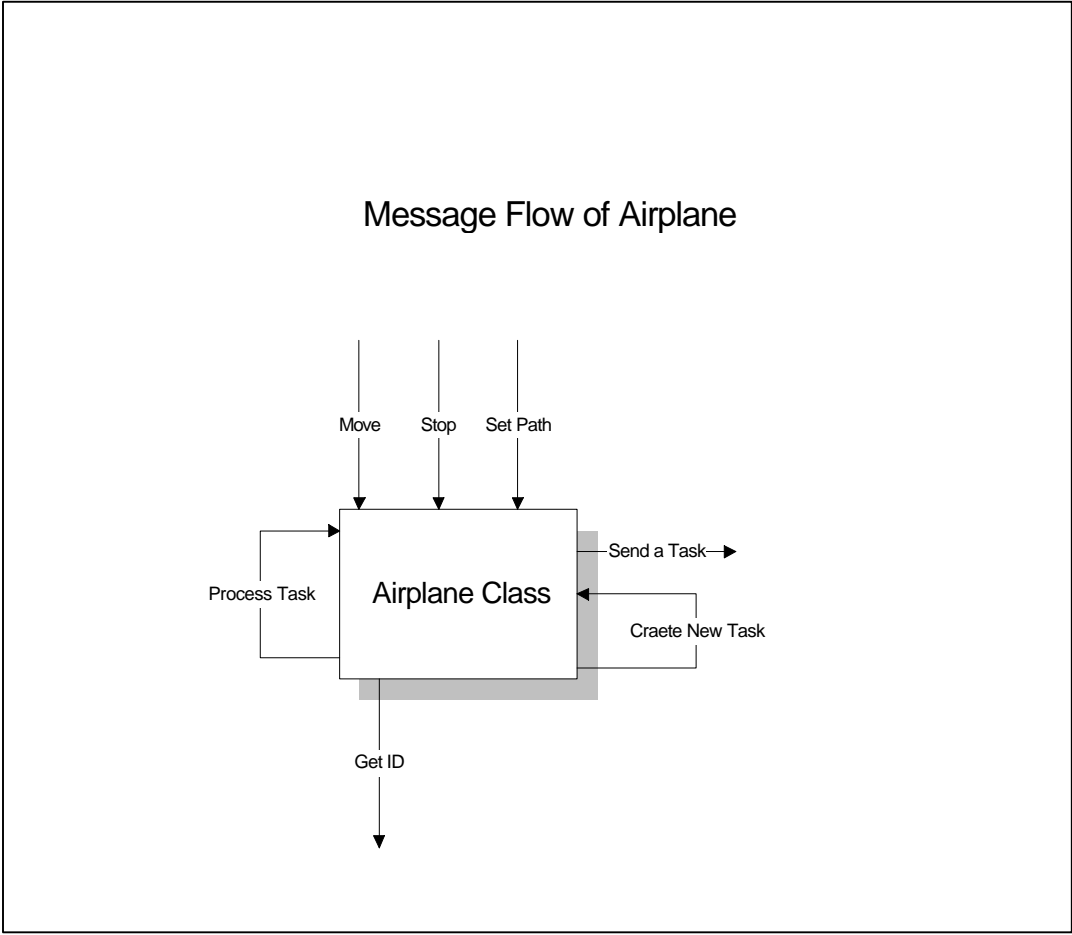


Figure 4.4 Message flow of airplane class.

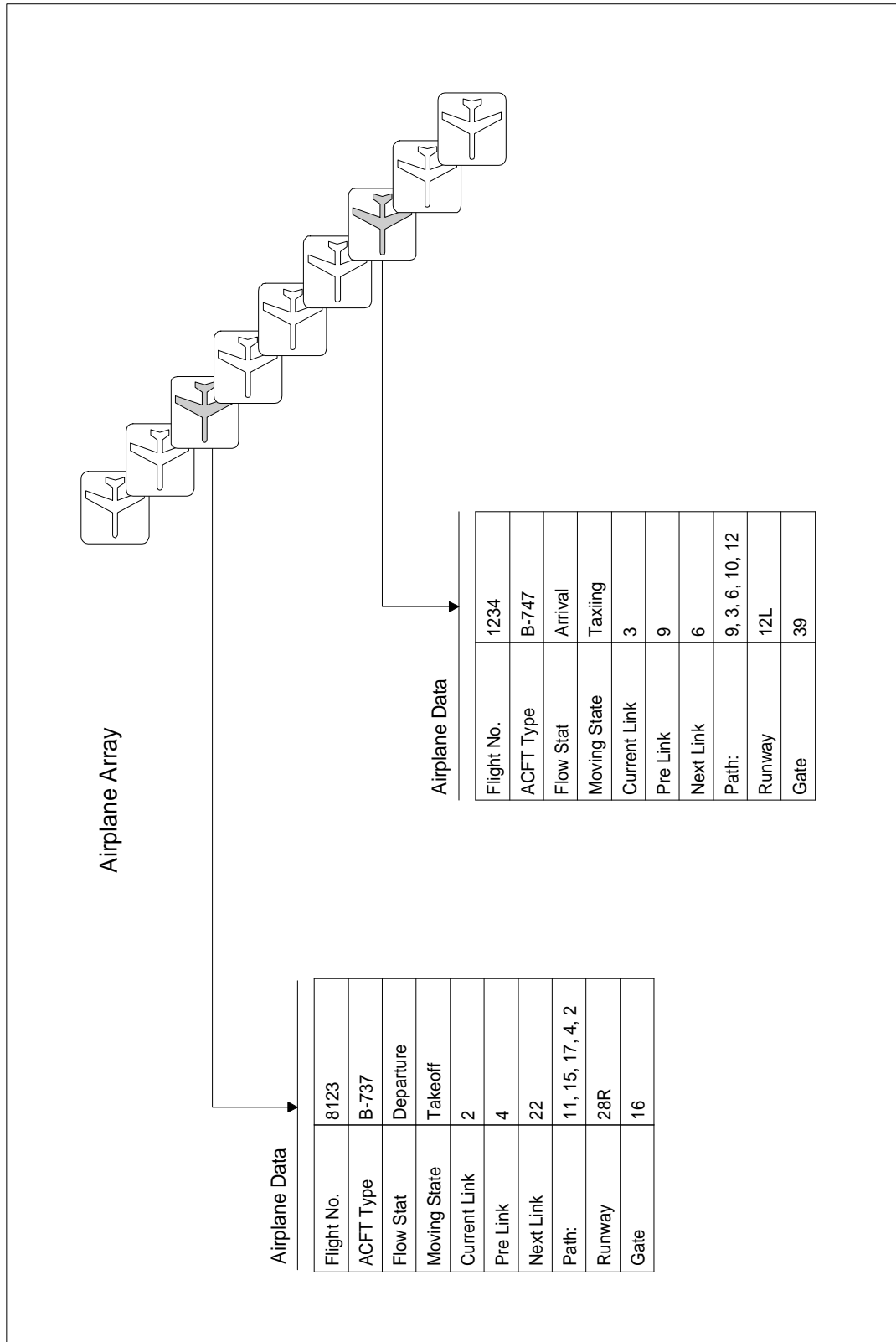


Figure 4.5 Data structure of airplane class.

There are several methods that facilitate handling of the airplane objects. The following examples illustrate the functions on the use of these methods. Figures 4.4 and 4.5 illustrate the point graphically.

Example:

1. Setting the best moving path:

```
// create an airplane with id equal
//to 1, moving from orientation A to
//destination B
Airplane_class my_airplane(1, (int)
'A', (int)'B');
//pseudo code: get the best travel
//path
char *best_path =
get_the_best_path(from_A, to_B);
//assign the best to the airplane
my_airplane.set_path(best_path);
```

2. Process an airplane related task

```
//pseudo code: obtain a task from the
//task engine
AsimEvent *this_task =
get_an_airplane_task_from_the_simulation_engine();
//process the task
my_airplane.process_event(*this_task);
```

3. Get the ID of the airplane

```
//obtain the ID from the get_ID
//function call
int ID=my_airplane.get_ID();
```

4.3 Airport Ground Network Class

Airport ground networks include runways, taxiways, and gates. During the simulation, airplanes move through the network following the rules of air traffic control and physical laws. In the airfield simulation framework, airport ground network objects are an abstract data type which provides ground network topology information to other objects such as

airplane list, shortest path processor, and traffic information collector objects. It can also be treated as a temporary storage of current traffic flow information during the simulation. Airport ground network objects could be easily implemented with relational databases. For example, whenever an airplane moves into a link in the network, the state of the link is changed from empty to occupied and the airplane ID is added into the airplane ID list associated with that link. A ground traffic flow information collector can query current traffic flow information to verify which link is empty and, if not, how many airplanes and which airplanes are there. By collecting all the needed traffic flow information, end users can easily apply their knowledge based traffic control rules or even optimization algorithms to control the airport ground traffic flow. Airport ground network serves as an information provider of both ground network topology and traffic flow. Figure 4.6 illustrates the message flow capabilities of the airport network class.

Definitions of States of Runways, Taxiway and Gates

Runways normally have two states. A runway can be opened or closed by the air traffic control tower. Physically runways have two directions of operation. Each direction could be set either open or close. If a runway is open, it will be further defined as occupied or available. Taxiways also have two states: opened or closed. Each taxiway can also be used from two directions. Each direction could be set either open or close.

If a taxiway is open, it will be further defined as occupied or available. Gates normally have two states. A gate can be opened or closed by the traffic control tower. If a gate is open, it will be further defined as occupied or available. The following definitions apply in the modeling framework.

Runway Open: When a runway is open, normally it will be available for the

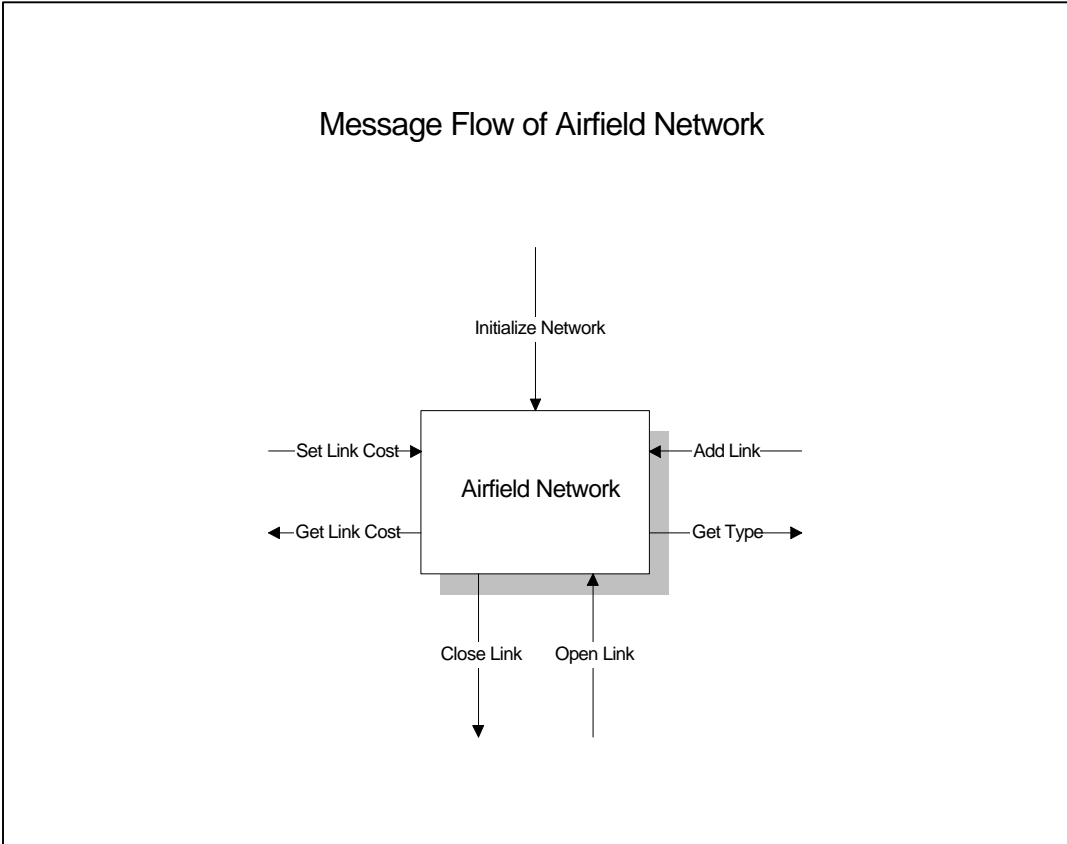


Figure 4.6 Message flow of airfield network class.

incumbent arriving or departing airplane.

Runway Close: When a runway is close, it could be occupied by an airplane or closed by air traffic control.

Taxiway Open: When a taxiway is open, normally it will be available for an airplane to pass.

Taxiway Close: When a taxiway is close, it could be occupied by other airplanes or closed by air traffic control.

Gate Open: When a gate is open, normally it will be available for an airplane to be parked.

Gate Close: When a gate is close, it could be occupied by an airplane or closed by the air traffic control tower.

Final State: When the simulation stops, the runway, taxiway, or gate objects become quiescent. They continue to exist, but have no subsequent dynamic behavior.

In summary, the runway open state must be associated with a direction indicator because an airplane can land on a runway in either direction. A closed runway is not necessary to be associated with a direction indicator. An open taxiway could be open in either or both directions. Furthermore, it is possible to hold or slowly move two or more airplanes on a taxiway, depending on the airplane ground separation criteria. An open gate is simply a gate that is available to upload or download cargo or passengers and a closed gate is either closed

or already occupied by an airplane.

4.4 Shortest Path Class

The shortest path method is an algorithm that can search a feasible path in a given abstract network with the minimum total travel cost. Although shortest path algorithm was originally designed for solving network travel cost problems, its theory could be applied to a variety of operational optimizations, such as a scheduling problem and other greedy algorithms. A slight modification of shortest path algorithm can be used to implement an algorithm for searching maximum network flows [Weiss, 1993]. With a set of objects for different optimization algorithm, end users can easily test different air traffic control methods to maximize the ground network throughput capacity and reduce delay.

The shortest path processor designed in this study is a generic shortest path algorithm that can search a shortest path between two points in a given network. The travel cost of any given link could be dynamically changed to represent the real time situations (such as traffic congestion). The basic rule or objective is to minimize the total travel cost between two given points. However, the minimum travel cost rules could be overwritten with other rules, such as maximizing the traffic flow between two given points.

Structure of the Shortest Path Processor

The structure of the shortest path processor includes the following elements. Figure 4.7 illustrates graphically the message flow elements in the shortest path processor.

1. A abstract network link structure which contains a orientation point from and a

Message Flow of Shortest Path Processor

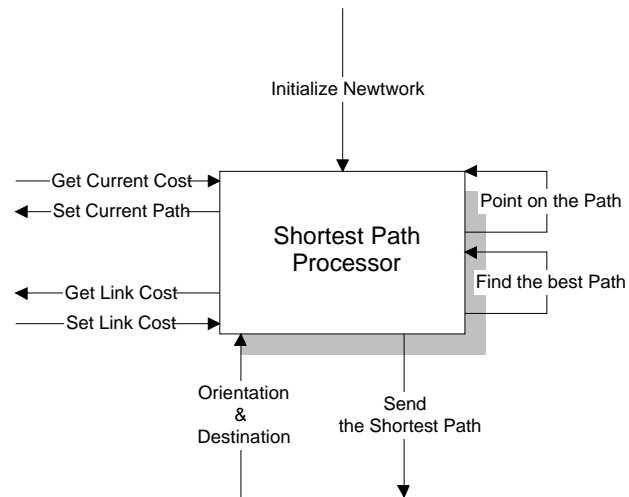


Figure 4.7 Message flow of shortest path processor.

- destination point to, the travel cost between two points cost, and the type of the link which is optional and is used to define whether a link is a runway or a taxiway.
2. the current total travel cost which is the total travel cost of the current travel path.
 3. the travel path which is the travel path being evaluated by the algorithm,
 4. the current best travel path which is the until now best travel path.
 5. an abstract network with an array of atom link structure, and
 6. the orientation and destination.

```

class Path_finder_class {
public:
typedef struct atom_struct_def {
                char from;
                char to;
                char type;
                int cost;
                } atom_path;

.....
private:
atom_path **traffic_net;
int n_max_segments;
int n_segments;
char *current_path;
int current_cost;
int num_points_on_path;
char *best_path;
long best_path_cost;
int
num_points_on_the_best_path;
char orientation;
char destination;
int head_tail_indicator;
};

```

The searching algorithm recursively searches all of the possible paths which start with the orientation point and end with the destination point. Constraints can be added into the objects to speed up the searching process. The internal method used to search the best travel

paths are briefly described in the next lines.

1. Method to initialize the network,

```
void  
initialize_traffic_net(link_struct  
**, int);
```

2. Method to set the link cost,

```
void set_path_segment_cost(char from,  
char to,  
  
int cost);
```

3. Method to get the link cost,

```
int get_path_segment_cost(char from,  
char to);
```

4. Method to check whether a point already used

```
int point_on_the_path(char  
this_point);
```

5. Method to find the best path recursively,

```
void find_the_best_path();
```

and

6. Method to get the shortest path cost,

```
char *get_the_best_path(char from,  
char to,int head_tail);
```

The shortest path algorithm is the default algorithm used in the airfield traffic simulation framework to control the movement of a given airplane and it is integrated with the air traffic control object.

4.5 Connecting Task Engine with Other Basic Components

The airfield traffic simulation framework is a framework which integrates the task engine and other basic airfield simulation objects. The connections between these basic objects

are a set of convenient functions which establish the links between objects to send and receive tasks and query network traffic information. To make the basic components reusable, all of the airfield simulation components follow the rule that member functions will not directly call each other. Components are connected through their inputs and outputs. For example the communication between the clock object and the task queue is through the functions to obtaining time from the global clock and advancing time in the global clock. Task queue accepts and release tasks through a function which will send a generic task into the task queue.

The connection between the air traffic control object and the airplane is a function call that will let the airplane send its own current situation to the air traffic control object and ask for future instructions. When the air traffic control object receives the request and the supplemental information, it will query the current traffic situation of the network and make a decision by sending the instruction to the target airplane by sending a task into the task queue. In this manner, continuous interactions between a moving airplane and the air traffic control can be simulated. The following functions serve as bridges between the task engine and other basic airfield simulation framework components. These functions are illustrated graphically in Figure 4.8.

1. Simulation loop function: a function call that activates the simulation by sending tasks to different basic objects.

```
void asim_start_mainloop();
```

2. Function call to create a task to an target airplane to stop the airplane.

```
AsimEvent_class  
*asim_create_airplane_stop_event(int  
ID,  
char from,  
char to);
```

3. Function call to initiate the airport ground network

```
void
```

```
    asim_initiate_airport_network(int);
```

4. Function calls to send a generic task to the task queue

```
void  
asim_send_event_to_queue(AsimEvent_class&  
event);  
void  
asim_send_event_to_queue(AsimEvent_class  
*event);
```

5. Function call to build the airfield network

```
void asim_build_airfield_net(int  
from, int to, int cost, int type);
```

6. Function call to process an airplane task

```
void  
asim_process_airplane_event(AsimEvent_class&  
theEvent);
```

7. Function call to get the travel cost of a link

```
int asim_get_link_travel_cost(int  
from, int to);
```

8. Function call to ask the traffic control object for future instructions

```
AsimEvent_class  
*asim_airplane_ask_controller_next_step(int  
ID, int from, int destination);
```

9. Function call to add a fly schedule into the task queue

```
void  
asim_add_flight_schedule_to_event_queue(  
airline_schedule_struct *schedule);
```

10. Function call to add a newly created airplane into the airplane list

```
void  
asim_add_new_airplane_into_acftlist(  
AsimEvent_class event);
```

11. Function call to create a task that moves an airplane from one point to another

point

```
AsimEvent_class  
*asim_create_moving_airplane_move_event(int  
ID, char from, char to);
```

12. Function call to get a new time mark with delay

```
int  
asim_get_new_time_with_given_delay(int  
*hour,  
int *minute, double *second,  
double delay_in_second);
```

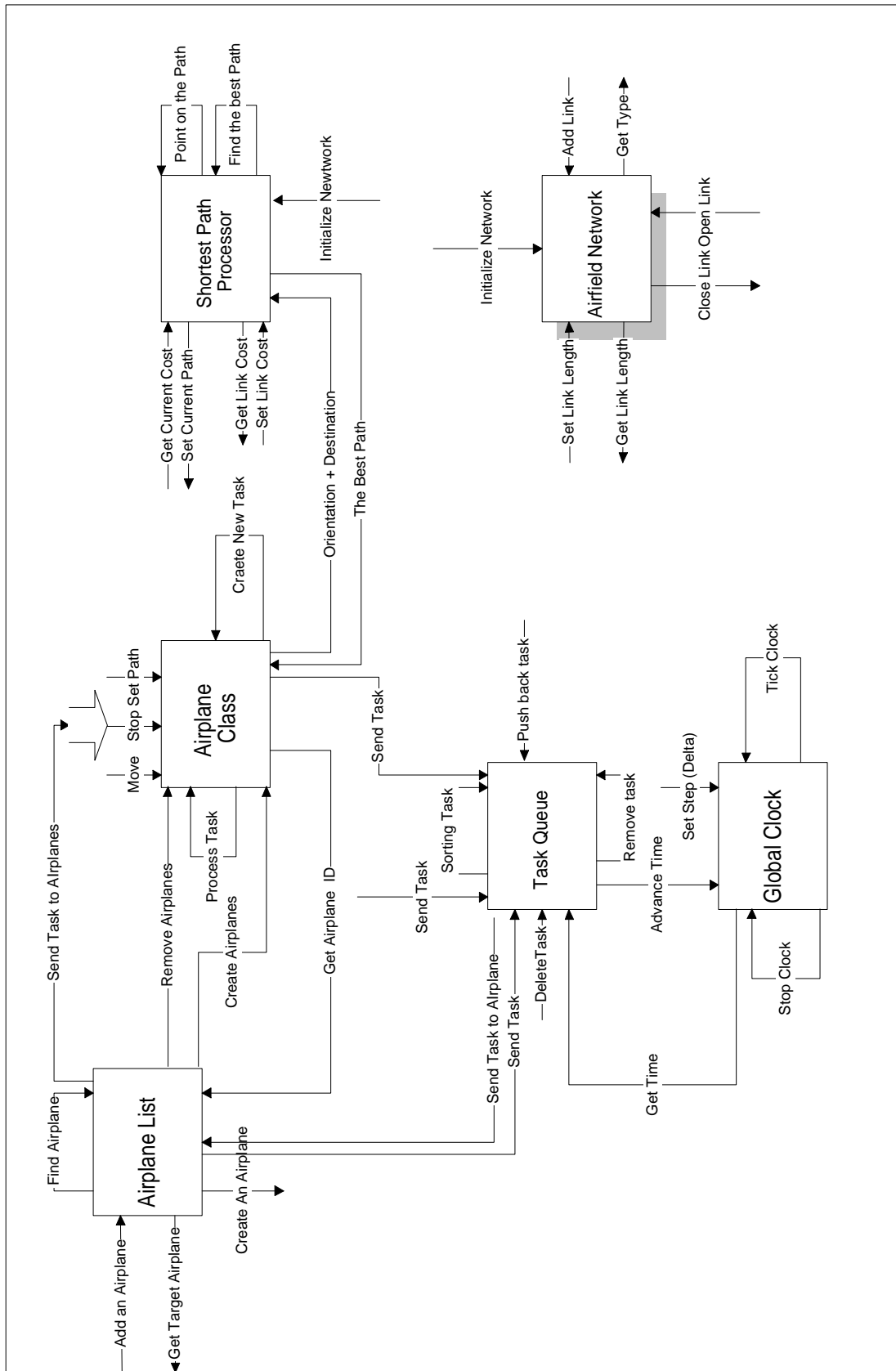


Figure 4.8 Message Flow of Airfield Simulation Framework

Chapter 5 Simple Applications of Airfield Traffic System Simulation Framework

5.1 Overview of the Simulation Framework

The advantages of object-oriented modeling to the end users are twofold: flexibility of detailed system modeling and reusability of the source code. In this Chapter, we apply the object-oriented modeling concept and use the class library developed in this study to model an airfield ground network. To mimic a real airport situation, an airfield ground network similar to that of Seattle-Tacoma International airport is used to demonstrate how to use the simulation frame work in airfield ground network modeling. This example will establish an airfield network with two closely parallel runways with segregated operations. Runway 16-R is used for landings whereas runway 16-L is reserved for departures. The operation assumes that departures can occur concurrently with arrivals to make the modeling process more complex.

Users can specify the aircraft mix or population by categories or by individual aircraft. A graphical animator is used to present the actual traffic flow during the simulation (see Figure 5.1). Each arriving or departing airplane is labeled in the simulation to make the processes easy to understand. User can visually monitor the runway usage, exit usage, and

Airport Ground Traffic Monitor

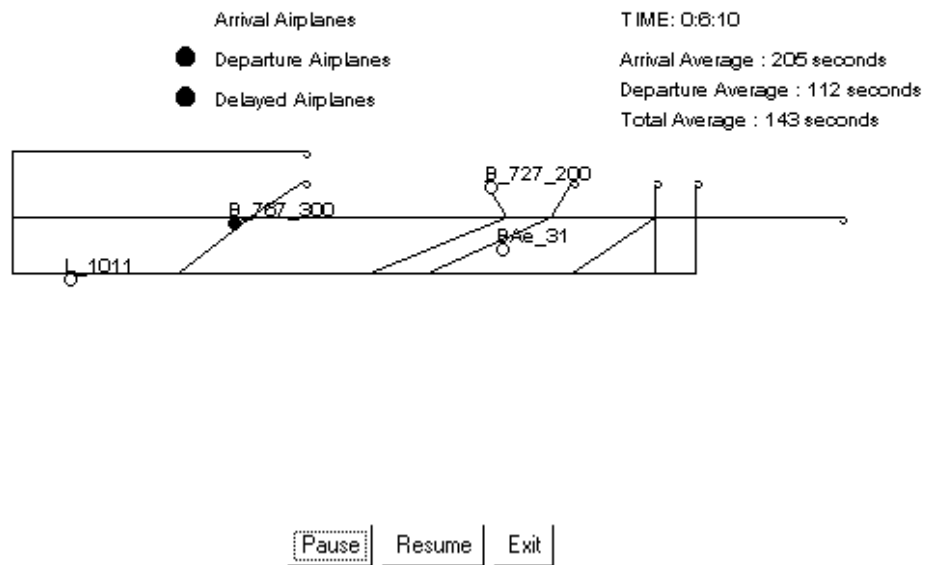


Figure 5.1 Interface of airport ground monitor

the traffic conflicts of arrivals and departures. Furthermore, the log file of the simulation will provide information that can derive important operational data such as network throughput sorted by aircraft categories, by aircraft types, and by exits. By using different color codes, users can even identify whether an arriving airplane is waiting in a link such as waiting for a departing airplane to finish its takeoff roll before crossing the runway. Several classes of the airport simulation library are used in this example. The documentation of about these classes is shown in full in Appendix A. The functions of Airport Traffic Monitor is described as follows.

This interface contains a customized built airport ground network. Airplanes move in the network when during landing or takeoff rolls. Departing airplanes are color coded blue. Arriving airplane are labeled green whereas red is used to identify delayed aircraft. In the graphic user interface a time label shows the simulated clock. The average arrival indicates the average time an arriving airplane takes from the time it is scheduled to land to the time it reaches its designated gate (i.e., airport terminal). The average departure time indicates the average time a departing airplane takes from the time it is scheduled to leave until the time it leaves the airport boundary. The average total label indicates the weighted average travel time of an airplane from its origin to its destination.

The first step in building the simulation framework is to initialize several basic components in the simulation. These include the following:

- a) **GroundMonitor object** - includes all the traffic conflicts of the airfield ground network
- b) **AirplaneFlightScheduler object** - creates the flight schedule for both arrivals and departures
- c) **GlobalClock object** — advances the simulation clock

- d) **AirfieldTrafficFlowAnimator** object -- graphically presents the movements of all airplanes in the airfield network
- e) **Airplane object** -- is the individual object that contains most of the data pertaining every aircraft
- f) **AirplaneFleet object** -- runs and moves all the aircraft
- g) **AirplaneQueue object** -- contains parameters and traffic situation of an individual link in the network
- h) **NetPath object** -- is a set of AirplaneQueue Objects that are in a traffic path in the ground network and implemented by the end users
- i) **ControlLogic object** -- contains all of the control logic that is implemented by the end users
- j) **TrafficParam object** — sets the parameters for the simulation including aircraft mix, category mix, runway occupancy time, exit choice distribution, and travel times for each taxiway segment in the ground network. The following Section illustrates how to initialize the simulation framework.

5.2 Initialization of the Airfield Ground Network

The selected ground network topology is an airfield ground network similar to that of the Seattle-Tacoma International Airport. The network contains two runways: 16L/34R and 16R/34L. The network is shown in Figures 5.1-2. To convert the ground network into a abstract data structure which can be efficiently used in the simulation, the airport ground network is first converted to an abstract network with each of its links represented as an AirplaneQueueObject. Runway occupancy times are stored in the TrafficParam object. Runway 16 R is used for arrivals and runway 16 L is used for departures.

The following example illustrates the initialization the ground network objects.

```
AirportGroundMap groundMap = new
AirportGroundMap();
where groundMap is the instance of
the airport ground map object.
```

The following example illustrates how to add a traffic path to the ground network objects. This path represents a landing traffic path following runway 16_R, exit C_3, taxiway B_2, and finally terminal TERMINAL_B_2. Tables 5.1 and 5.2 contain all the definitions of the ground network for this airport including runway exits and taxiway paths.

```
public static void
createAllNetPaths()
{
NetPath tempPath = new
NetPath("16_R_TO_C_3");
tempPath.addLinksAt("16_R", 0);
tempPath.addLinksAt("C_3", 1);
tempPath.addLinksAt("B_2", 2);
tempPath.addLinksAt("TERMINAL_B_2",
3);
AllNetPaths.addElement(tempPath);
tempPath = null;
}
```

The following piece of source code illustrates how to add an airplane queue into the ground monitor object:

```
queues.addElement(new
AirplaneQueue("16_R",new
Point(85,223),
new Point(439,223)));
where 16_R represents the runway
name; Point(85, 223) represents the
starting point of 16_R and Point(439,
223) represent the ending point of
16_R.
```

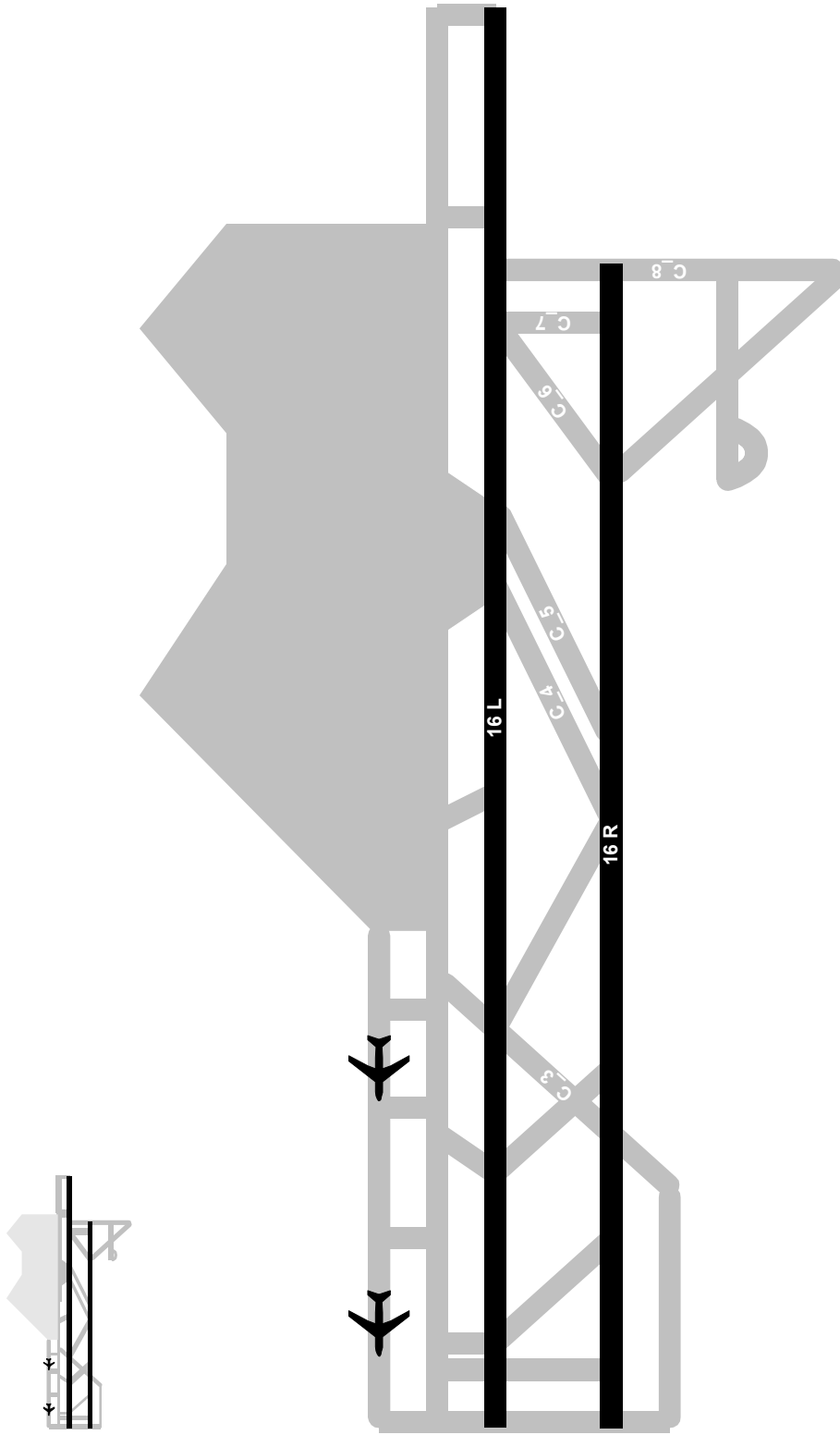


Figure 5.2 Airport ground network scenario.

In a similar fashion, the following piece of source code illustrates how to add an exit into the ground monitor object:

```
//insert exit name
insertExitName("C_3", 0.05, 35);
where C_3 is the exit name with over-
all 5 percent usage and a 35 second
runway occupancy time.
```

In the following lines we illustrate how to initialize the aircraft data base.

```
//insert category name
insertAirplaneCategoryName("CATEGORY_C",
0.40);
//where CATEGORY_C is the name for
//aircraft in category C and this
//category consists of 40 percent of
//the airplane.

//insert aircraft name
insertAircraftName("B_767_300", 7.8);
//where B_767_300 is the individual
//airplane name and it consists of
//7.8 percent of the aircraft mix.
```

The following function assigns aircraft B_767_300 to category D.

```
insertAircraftMasterData("B_767_300",
"CAT_D");
```

5.3 Initialization of Flight Schedule

The flight schedule in this example includes several arrivals and departures. Flights in the schedule are create by the AirlineScheduler object. Implemented by the end users, each traffic flow will have a list of corresponding arrivals or departures. In this example, we use a

scheduled flow for arrivals on runway 16R and the other scheduled flow for departures on 16L. It must be pointed out that if a different runway use is employed (i.e., different configurations) the traffic flow schedule could easily be varied. The following piece of code illustrates how to create an `AirplaneScheduler` object.

```
FlightScheduler = new
AirplaneScheduler();
AirplaneScheduler() {
AirplaneList = new Vector(100);
ArrivalRates = new Vector();
addRandomSchedule(0, 120);
addRandomSchedule(1, 120);
startScheduler();
}
```

5.4 Running the Multi-Threaded Simulation

As described above, the procedure to initialize the airfield simulation framework is to initialize a set of objects. After all of the necessary objects are created, the application will start four threads to finish different groups of tasks: 1) the `GlobalClock`, 2) `AirplaneScheduler`, 3) `AirplaneFleet`, and 4) `GraphicalAnimator`. Because of the preemptive multitask nature of Window 95/NT Operating System we could image that this multi-threaded application is to a certain extent equivalent to using four virtual computers simultaneously with each machine for an individual airport simulation. The following piece of code illustrates how create a `GlobalClock` object.

```
GlobalClock Clock = new
GlobalClock(10);
GlobalClock(int timeStep){
delta = timeStep;
time_base = 0;
hour = 0;
minute = 0;
second = 0;
```

```
subsecond = 0;
Thread ClockThread = new
Thread(this);
ClockThread.start();
}
```

The following piece of code illustrates how create a AirplaneFleet object.

```
Airline = new AirplaneFleet();
AirplaneFleet()
{
    if(runAirplane == null){
        runAirplane = new Thread(this);
        runAirplane.start();
    }
}
```

5.5 Implementation of the Airport Ground Network Control Logic

In a simulation model the air traffic control logic is usually implemented by the end users. In this example, a simple set of rules is applied to implement the control logic to illustrate how control actions are explicitly stated in the proposed framework.

Rule 1: Arrival aircraft has priority over departures.

Rule 2: An arriving aircraft will take the any available exit which yields the lowest runway occupancy time (ROT).

Rule 3: A departing airplane will check and make sure all the exits crossing 16 L are empty before taking off the runway.

5.6 Analyzing the Air Field Traffic Flows of the Given Configuration with Different Control Strategies

To illustrate the capability of the airport simulation frame work, we use the this example to perform a typical saturation capacity analysis of the given ground network configuration. By changing the interarrival times from 120 seconds to 40 seconds at a step of 10 seconds we can perform sensitivity analysis of the airport saturation capacity for a constant departure interval of 120 second between departures. Figure 5.3 illustrates the sensitivity of the interarrival times. In the same way, assuming a constant arrival rate at 120 seconds per arrival, we can vary the departure interval from 120 seconds to 60 seconds with a step of 10 seconds to perform sensitivity analysis when departure separation rules are modified. Figure 5.4 illustrates the sensitivity of the departure rate given a constant arrival rate. Figure 5.5 shows travel times on the network for the scenario simulated. Figure 5.6 shows average arrival delay with a given constant departure rate of 30 flights per hour.

Table 5.1 Exits for Runway 16 R

EXIT NAME	EXIT TYPE
C_3	45 degree
C_4	30 degree
C_5	30 degree
C_6	30 degree
C_7	90 degree
C_8	90 degree

Table 5.2 Traffic Paths in the Airfield Network

	Traffic Path
1	16_R -> C_3 -> B_2 -> B_2_TERMINAL
2	16_R -> C_4 -> B_7 -> B_7_TERMINAL
3	16_R -> C_5 -> B_9 -> B_8_TERMINAL
4	16_R -> C_6 -> B_10 -> B_9_TERMINAL
5	16_R -> C_7 -> B_10 -> B_10_TERMINAL
6	16_R -> C_8 -> B_11 -> B_11_TERMINAL
7	16_L -> 34_L_TERMINAL

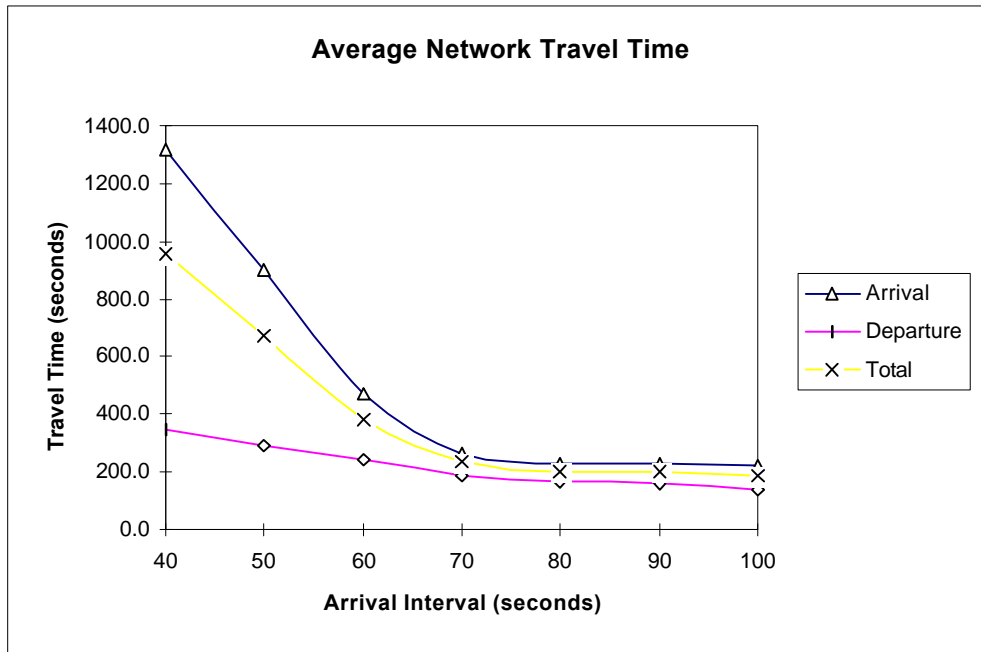


Figure 5.3 Average travel time for arriving aircraft.

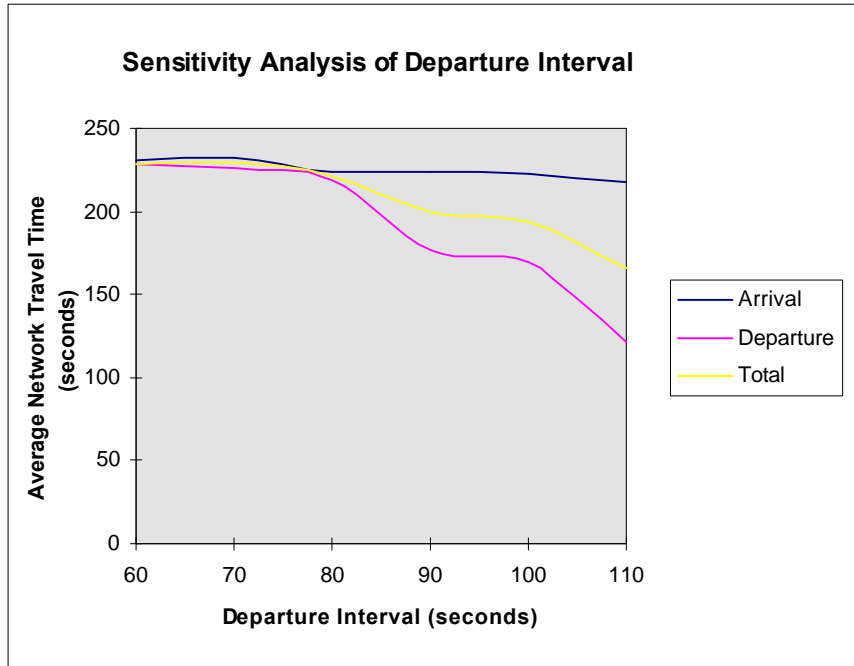


Figure 5.4 Average travel time for departing aircraft.

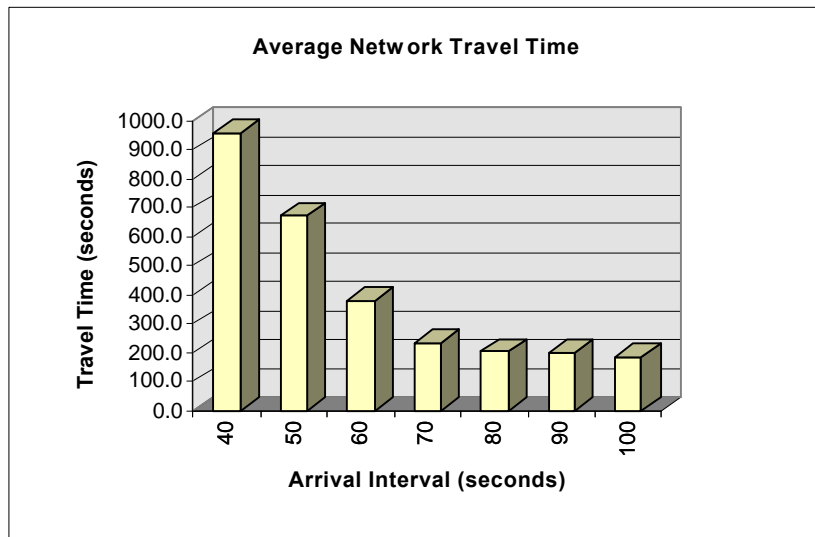


Figure 5.5 Average travel time in the network.

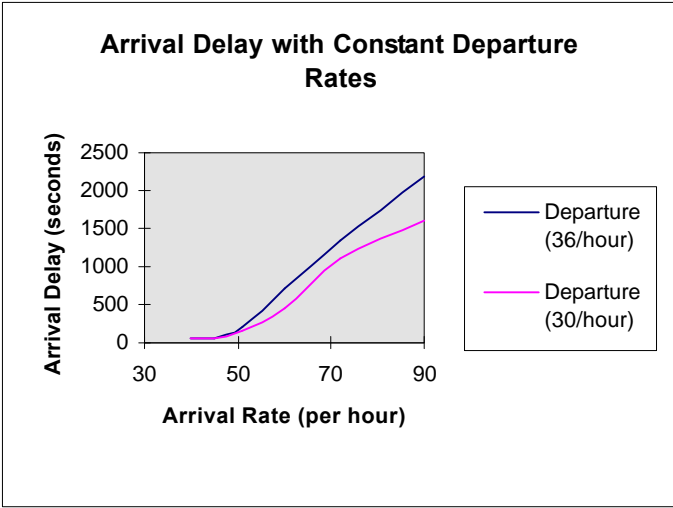


Figure 5.6 Average arrival delay

Arrival Delay with Constant Departure Rates

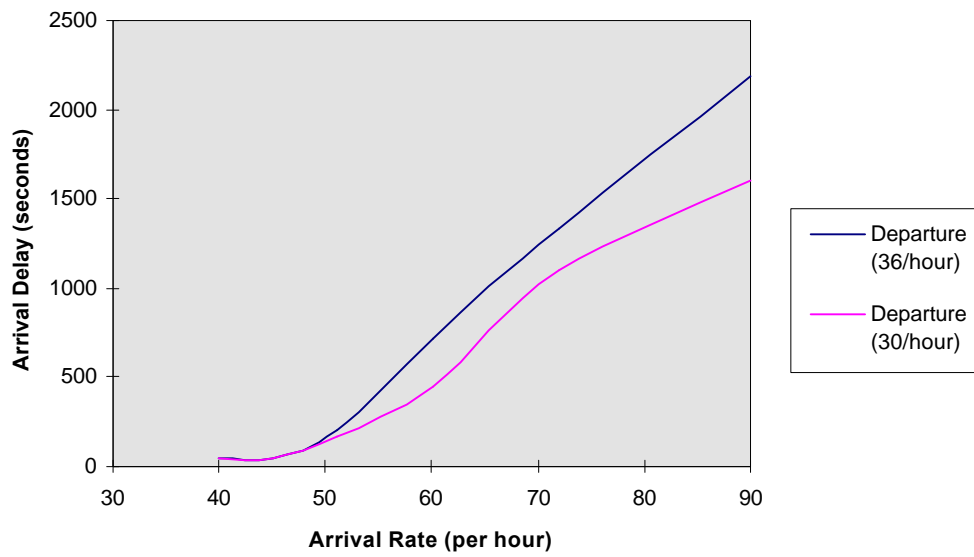


Figure 5.7 Average departure delay.

Chapter 6 Conclusions and Recommendations

Modeling real world systems with classes and components is the preferred modeling technique for new generation modeling software. This research proposes a new approach to simplify airfield simulation models using object-oriented modeling concepts. The language implementation of this project was carried out using JAVA a new generation programming language. An object class library was constructed from scratch allowing airport analysts to quickly assemble an airport scenario including control logic modeling. The library, while in the experimental stage, can easily be extended to model more complex processes. An important aspect of the modeling approach adopted in this research effort is that any model developed using Java offers clear opportunities of portability and offers an unlimited audience through the Internet. End users can use the proposed framework to develop distributed applications that will allow other users to download them from the World Wide Web. On the other hand, this library can be integrated with other airport engineering libraries to extend its reusability.

This study found that, JAVA, an object-oriented next generation computer language developed by SUN, is an ideal language for building simulation libraries to support airfield traffic simulations. JAVA offers several advantages over C++ such as a true cross-platform characteristic, it is network ready and has many easy to use features. JAVA is widely sup-

ported by computer industries and is slowly replacing C++ in the software engineering market.

In a not to distant future object-oriented modeling techniques will be commonplace in the engineering field. The advantages of the OOP paradigm are so numerous and compelling that engineering applications should move to this new way to structure simulation models soon. Our vision for the next generation air traffic control applications such as those required to support free flight and real-time airport automation includes heavy use of distributed, object-oriented modeling to ease development times, improve portability and avoid obsolescence. The framework proposed in this research includes true multi-threaded air traffic control for larger gains in computational speed.

The idea of a partially decentralized air traffic control system in the future using multi-thread networks will enhance the capacity of the National Airspace System (NAS). In a multi-threaded environment automatic traffic information will help pilots and air traffic controllers to communicate control orders and synchronize aircraft movements with optimal sequences to reduce traffic flow conflicts. The role of OOM will be crucial in the development of multi-agent strategies to recognize airspace and ground conflicts and to provide timely resolution strategies to improve safety in NAS.

While the application of OOM has been devoted to address a small element of the air transportation system the same modeling framework developed here could easily be ported to various disciplines within Transportation Engineering including Intelligent Transportation Systems (ITS). For example in an Advance Traveler Information System (ATIS) travelers can use a hand-held computer to obtain traffic information from Internet; the applications used for transmitting the real time traffic information must be light weight or require minimum data traffic among the network. These characteristics can only be achieved by OOM

modeling using state-of-the art programming languages that are truly portable across all platforms. In short, a generic and cross-platform network-oriented library for operational analysis could have obvious good benefits to the transportation engineering discipline in general.

The results obtained in the simple airport network analyzed in this research effort indicate that OOM modeling using JAVA is a powerful yet compact way to naturally simulate complex real-world behaviors. The results obtained in the simulations indicate that delay results for SEATAC airport operating under segregated runway conditions are comparable to those obtained using other models. The airport simulation results obtained also point out the benefits of JAVA in terms of computational efficiency and compactness. The SEATAC application developed used only 55 kilobytes of hard disk space and ran adequately in Intel-based 486 and P5 personal computers. The same source code was ported to a Macintosh PowerPC computer running an IBM/Motorola/Apple RISC microprocessor (604e) compiled and executed **without** modifications. The same source code will run on any computer that has a JAVA virtual machine.

Several recommendations can be set for further study:

- 1) The airport object library created in this research needs to be extended to include more complex aircraft and airfield behaviors such as de-icing, movement in the airspace including conflict resolution logic, fuel burn, etc.
- 2) A better graphic user interface should be developed for this framework modeling environment. A preprocessor and post-processor could be developed to enhance the use of an airport simulator model allowing users to interactively design airport and airspace networks and to edit all objects required for the

simulation.

- 3) Distributed airport-airspace application model should be tested to validate the efficiency of the JAVA code over large area networks. This approach should perhaps include the development of parallel computing strategies to reduce execution times over the network.

- 4) The framework should be implemented in real-time ATC control systems to demonstrate the capabilities of the JAVA language under mission-critical applications. The code for real-time applications will have to be fully tested and validated.

References

1. AbouRizk, S. "Modeling operational activities in object-oriented simulation," *Journal of Computing in Civil Engineering*, v. 8, n. 3, p. 392-395, July 1994.
2. CACI, Federal Aviation Administration, and Society for Computer Simulation, "Abstracts: Simulation conference 19," CACI Products Company, August 24-26, 1993.
3. Carmichael, Andy "Object Development Methods", SIGS BOOKS, Inc., 1994
4. Clymer, J. R., Corey, P. D., and Gardner, J. A. "Discrete event fuzzy airport control," *IEEE Transactions on Systems, Man, and Cybernetics*, v. 22, n. 2, p. 343-351, March/April 1992.
5. Hill, David R. C. "Object-Oriented Analysis and Simulation", Addison-Wesley Publishing Company, 1996
6. Hobeika, A. G., Trani, A. A., Sherali, H. D., Kim, B. J., and Sadam, C. K. "Runway exit designs for capacity improvement demonstrations (Phase 1)," Virginia Tech., Blacksburg, VA, Feb. 1990.
7. Hollister, W. M. "Airport surface traffic automation study," *DOT/FAA/PS-87/1*, 1988.
8. Horonjeff, R., and Mckelvey, F. X. *Planning and Design of Airports, Third*

- Edition*, McGraw-Hill Book Company, 1983.
9. Hughes, D. "New software boosts arrivals at logan," *Aviation Week & Space Technology*, v. 139, n. 11, p. 37-38, Sept 13, 1993.
 10. Joline, E. S. "Multipath runway exits and Taxiways,." *DOT/FAA/CT-92/19*, 1993.
 11. Kernighan, B. W., and Ritchie, D. M. *The C Programming Language, Second Edition*, PTR Prentice Hall, Englewood Cliffs, New Jersey, 1988.
 12. Lafore, R. *Object-Oriented Programming in C++, Second Edition*, Aaite Group Press, Corte Madera, California, 1995.
 13. Lefrancois, P., and Montreuil B. "An object-oriented knowledge representation for intelligent control of manufacturing workstations," *IIE Transactions*, v. 26, n. 1, p. 11-26, Jan. 1994.
 14. MacLean, R., Stepney, S., Smith, S., Tordoff, N., Gradwell, D., Hoverd, T., and Katz, S. *Analysing Systems: Determining Requirements for Object-Oriented Development*, Prentice Hall Internation (UK) Limited, 1994.
 15. Mize, J. H., Bhuskute, H. C., Pratt, D. B., and Kamath, M. "Modeling of integrated manufacturing systems using an object-oriented approach," *IIE Transactions*, v. 24, n. 3, p. 14-26, July 1992.
 16. Morlok, E. K. *Introduction to Transportation Engineering and Planning*,

- McGraw-Hill Book Company, 1978.
17. Nolan, M. S. *Fundamentals of Air Traffic Control*, Wadsworth Publishing Company, 1990.
18. Nussbaum, D. M. "Advanced ASDE provides new eyes and ears for controllers," *ICAO Journal*, p. 11-12, Feb. 1992.
19. Nussbaum, D. M. "Advanced ASDE radars planned for major U.S. airports," *ICAO Journal*, p. 41-43, Sept. 1987.
20. Oloufa, A. A. "Modeling operational activities in object-oriented simulation," *Journal of Computing in Civil Engineering*, v. 7, n. 1, p. 94-106, Jan. 1993.
21. Ott, R. L. *An Introduction to Statistical Methods and Data Analysis, Fourth Edition*, Duxbury Press, 1994.
22. Schildt, H. *C++: The Complete Reference*, Osborne McGraw-Hill, 1991.
23. Sproule, W. J. "Aviation crossroads: challenges in a changing world: proceedings of the 23rd International Air Transportation conference," *American Society of Civil Engineers*, 1994.
24. Stengel, R. F. "Toward intelligent flight control," *IEEE Transactions on Systems, Man, and Cybernetics*, v. 23, n.6, p. 1699-1717, Nov./Dec. 1993.
25. Trani, A. A., Hobeika, A. G., Kim, B. J., Nunna, V., and Zhong, C. "Runway

- exit designs for capacity improvement demonstrations: Phase II: Computer model development (Final Report),” Center for Transportation Research, Virginia Tech., Blacksburg, VA, Jan. 1992.
26. Trani, A. A., and Zhong, C. “Operational requirements to implement rapid runwayturnoffs at airports,” Dept. of Civil Engineering, Virginia Tech., Blacksburg, VA.
27. Venkatakrishnan, C. S., Barnett, A., and Odoni, A. R. “Landings at logan airport: Describing and increasing airport capacity,” *Transportation Science*, v. 27, n. 3, p. 211-227, August 1993.
28. Weiss, M. A. *Data Structures and Algorithm Analysis in C++*, The Benjamin/Cummings Publishing Company, Inc., 1994.
29. Wiederhold, Gio., Wegner, P., and Ceri, S. “Toward megaprogramming,” *Communications of the ACM*, v. 35, n. 11, p. 89-98, Nov. 1992.
30. Zgierski, J. R., and Oommen, B. J. “Seater: an object-oriented simulation environment using learning automata for telephone traffic routing,” *IEEE Transactions on Systems, Man, and Cybernetics*, v. 24, n. 2, p. 349-356, Feb. 1994.
31. <http://www.itsonline.com/>
32. <http://www.javasoft.com/>

Appendix A

Generic Airport Modeling Library in Java

```

import java.awt.*;
import java.util.*;
import GlobalClock;
import ControlLogic;
import GroundMonitor;

class airport extends Frame{
    public Panel Button_Panel;
    public Button Pause_Button;
    public Button Resume_Button;
    public Button Exit_Button;
    public Button Stop_Button;
    public TextField ArrivalRateTextField;
    public TextField DepartureRateTextField;
    AirplaneScheduler FlightScheduler;
    AirplaneFleet Airline;
    boolean current_state;

    GlobalClock Clock;
    MainCanvas GroundView;

    public airport(){
        super("Airport Simulation Example");

        Clock = new GlobalClock(10);
        GroundMonitor.createAllQueues();
        GroundMonitor.createAllNetPaths();
        TrafficParam.initiateTrafficParaClass();
        TrafficParam.setAllTrafficParameters();
        AirplaneScheduler FlightScheduler = new AirplaneScheduler();
        AirplaneFleet Airline = new AirplaneFleet();
        current_state = true;

        Button_Panel = new Panel();

        Pause_Button = new Button("Pause");
        Resume_Button = new Button("Resume");
        Stop_Button = new Button("Run");
        Exit_Button = new Button("Exit");
        Button_Panel.add(Pause_Button);
        Button_Panel.add(Resume_Button);
        //Button_Panel.add(Stop_Button);
        Button_Panel.add(Exit_Button);

        /*
        ArrivalRateTextField = new TextField(3);
        DepartureRateTextField = new TextField(3);
        Button_Panel.add(ArrivalRateTextField);
        Button_Panel.add(DepartureRateTextField);
        */

        GroundView = new MainCanvas(FlightScheduler, Airline);
        this.add("Center", GroundView);
        this.add("South", Button_Panel);
        this.resize(600, 400);
        this.show();
    }

    static public void main(String[] args) {

        airport myairport = new airport();

    }

    public boolean action(Event event, Object arg) {
        System.out.println("In Event action");
        if(event.target == Pause_Button)
        {
            System.out.println("Pasue");
            Clock.ClockThread.suspend();
            GroundView.AnimationEngine.suspend();
        }else if (event.target == Resume_Button)
        {
            System.out.println("Resume");
            GroundView.AnimationEngine.resume();
        }
    }
}

```

```

        Clock.ClockThread.resume();
    }else if (event.target == Exit_Button)
    {
        System.exit(0);
    }else if (event.target == Stop_Button){
        if(current_state == true){
            GroundView.AnimationEngine.stop();
            Airline.runAirplane.stop();
            //FlightScheduler.CreateFlightThread.stop();
            Clock.ClockThread.stop();
            GroundView.AnimationEngine= null;
            Airline.runAirplane = null;
            Clock.ClockThread = null;
            //FlightScheduler.CreateFlightThread = null;
            current_state = false;
        }
        else
        {
            System.out.println("re-activate thread");
            current_state = true;

            Clock.startClock();
            System.out.println("here1");
            //FlightScheduler.startScheduler();
            //Airline.startRun();
            //GroundView.startAnimation();
        }
    }
    return true;
}
}

class MainCanvas extends Canvas implements Runnable {
    public Font TitleFont = new Font("Helvetica", Font.BOLD, 18);
    public Font AirplaneFont = new Font("Helvetica", Font.PLAIN, 10);
    Image backGroundBuffer, workPadBuffer;
    Graphics backGroundGraphics, workPadGraphics;
    AirportGroundMap groundMap = new AirportGroundMap();
    AirplaneScheduler FlightScheduler;
    AirplaneFleet Airline;
    boolean upDateBackGround;
    Thread AnimationEngine;
    GlobalClock Clock = new GlobalClock();

    MainCanvas(AirplaneScheduler scheduler, AirplaneFleet fleet) {
        FlightScheduler = scheduler;
        Airline = fleet;
        AnimationEngine = new Thread(this);
        AnimationEngine.setPriority(6);
        AnimationEngine.start();
        upDateBackGround = true;
    }

    public void startAnimation(){
        AnimationEngine = new Thread(this);
        AnimationEngine.setPriority(6);
        AnimationEngine.start();
    }

    public void paint(Graphics g) {
        update(g);
    }

    public synchronized void update(Graphics g) {
        int w = bounds().width;
        int h = bounds().height;

        if (backGroundBuffer == null
            || backGroundBuffer.getWidth(null) != w

```

```

|| backGroundBuffer.getHeight(null) != h) {
backGroundBuffer = createImage(w, h);

if(backGroundBuffer != null) {
    if(backGroundGraphics != null) {
        backGroundGraphics.dispose();
    }
    backGroundGraphics = backGroundBuffer.getGraphics();
}

}

if (backGroundBuffer != null) {
    backGroundGraphics.setColor(Color.white);
    backGroundGraphics.fillRect(0, 0, w, h);
    groundMap.drawMap(backGroundGraphics);
}

    if(!FlightScheduler.AirplaneList.isEmpty())
for(int i = 0; i < FlightScheduler.AirplaneList.size(); i++){
// System.out.println("size = " + (int)FlightScheduler.AirplaneList.size());
Airplane tempAirplane = (Airplane) FlightScheduler.AirplaneList.elementAt(i);
tempAirplane.updateCurrentLocation();

if(tempAirplane.MovingTrack != null &&
tempAirplane.CurrentQueue != null &&
tempAirplane.crrntPoint != null)
{
    if(tempAirplane.delayIndicator == true)
        backGroundGraphics.setColor(Color.red);
    else
        if(tempAirplane.RoutName.compareTo("ENROUT") == 0)
            backGroundGraphics.setColor(Color.blue);
        else
            backGroundGraphics.setColor(Color.green);

    backGroundGraphics.fillOval(tempAirplane.crrntPoint.x,
tempAirplane.crrntPoint.y, 6, 6); //draw airplane
    backGroundGraphics.setColor(Color.black);
    backGroundGraphics.drawOval(tempAirplane.crrntPoint.x,
tempAirplane.crrntPoint.y, 6, 6); //draw airplane
    backGroundGraphics.setFont(AirplaneFont);
    backGroundGraphics.drawString(tempAirplane.getAircraftName(),
tempAirplane.crrntPoint.x,
tempAirplane.crrntPoint.y); //draw airplane

        backGroundGraphics.setColor(Color.black);
        tempAirplane.printTime(backGroundGraphics, 400, 120);
    }

if(tempAirplane.AirplaneFinalStop == true){
    FlightScheduler.AirplaneList.removeElement(tempAirplane);
}

}

backGroundGraphics.setFont(TitleFont);
backGroundGraphics.drawString("Airport Ground Traffic Monitor",
150,
60); //draw airplane
backGroundGraphics.setColor(Color.green);
backGroundGraphics.fillOval(170, 90, 10, 10);
backGroundGraphics.setFont(AirplaneFont);
backGroundGraphics.setColor(Color.black);
backGroundGraphics.drawString("Arrival Airplanes", 190, 100);

backGroundGraphics.setColor(Color.blue);
backGroundGraphics.fillOval(170, 110, 10, 10);
backGroundGraphics.setColor(Color.black);

```

```

        backGroundGraphics.drawString("Departure Airplanes", 190, 120);

        backGroundGraphics.setColor(Color.red);
        backGroundGraphics.fillOval(170, 130, 10, 10);
        backGroundGraphics.setColor(Color.black);

        backGroundGraphics.drawString("Delayed Airplanes", 190, 140);
        Clock.printTime(backGroundGraphics, 400, 100);

        g.drawImage(backGroundBuffer, 0, 0, this);
    }

    public void run() {
        while (true) {
            repaint();
            try{AnimationEngine.sleep(100);} catch (Exception e) {};
        }
    }
}

class Airplane extends Object {
    long freeFlowTime = 0;
    long delay;
    static long AvTotal;
    static long AvArrival;
    static long AvDepart;
    static long ArrivalTravelTime;
    static long DepartureTravelTime;
    static int numberOfArrival;
    static int numberOfDeparture;
    public void printTime(Graphics g, int x, int y){
        if(numberOfArrival > 0 && numberOfDeparture > 0){
            g.drawString("Arrival Delay : "+ AvArrival+" seconds", x, y);
            g.drawString("Departure Delay : "+ AvDepart+" seconds", x, y+15);
            g.drawString("Arrival:" + numberOfArrival+" Departure: "+numberOfDeparture,
                x, y+30);
        }
    }

    public String getAircraftName()
    {
        return AircraftName;
    }
    public boolean isCategory(String cat)
    {
        if(Category.equals(cat))
            return true;
        else
            return false;
    }

    public boolean delayIndicator;
    public synchronized void updateCurrentLocation()
    {
        if(MovingTrack != null && MovingTrack.Track.size() > 0)
        {
            crrentPoint = (Point) MovingTrack.Track.elementAt(0);
            MovingTrack.Track.removeElementAt(0);
        }
    }
    public synchronized void updateTrack()
    {
        if(CurrentQueue != null && NextQueue != null)
            MovingTrack = new LineObject(CurrentQueue.From, NextQueue.From);
        else
            MovingTrack = null;
    }
    public String RoutName;
    public void setAirplaneCategory(String cat_name)

```

```

{
    Category = new String(cat_name);
}
public void setAircraftName(String aircraft_name)
{
    AircraftName = new String(aircraft_name);
}
private String AircraftName;
private String Category;
int step_counter = 0;
public LineObject MovingTrack;
public Airplane(int flow)
{
    total++;
    ID = total;
    flow_ID = flow;
    startTime = Watch.getTime();
    resetTimer();
    setStateToApproach();
}
public synchronized void setAirplaneFinalStop()
{
    AirplaneFinalStop = true;
    finishTime = Watch.getTime();
    long throughput = finishTime - startTime;
    delay = throughput - freeFlowTime;
    System.out.println(ID()+" "+AircraftName+" "+RoutName+" follow "+pathName+
        " used "+throughput+" seconds"+" from "+startTime+
        " to "+finishTime);
    if("ENROUT".equals(RoutName))
    {
        //DepartureTravelTime +=throughput;
        DepartureTravelTime +=delay;
        numberOfDeparture++;
        if(numberOfDeparture > 0)
            AvDepart = DepartureTravelTime/numberOfDeparture;
    }
    else
    {
        //ArrivalTravelTime +=throughput;
        ArrivalTravelTime +=delay;
        numberOfArrival++;
        if(numberOfArrival > 0)
            AvArrival = ArrivalTravelTime/numberOfArrival;
    }

    if(numberOfArrival > 0 || numberOfDeparture > 0)
        AvTotal = (    ArrivalTravelTime +    DepartureTravelTime)/
                    (numberOfArrival + numberOfDeparture);
}
public boolean timerTimeOut()
{
    return (timer <= Watch.getTime())? true:false;
}
public long getTimer()
{
    return timer;
}
public void resetTimer()
{
    timer = 0;
}
public String pathName;
long timer;
public void setTimer(long second)
{
    if(second < 0)
    {
        System.out.println("exit here");
        System.exit(1);
    }
}

```

```

        timer = Watch.getTime() + second;
        freeFlowTime +=second;
        if(MovingTrack != null)
            MovingTrack.setTrack((int) second);
    }
    public static final int TAKEOFF = 4;
    public static final int APPROACH = 0;
    public static final int DEPART = 1;
    public static final int TAXI = 5;
    public static final int LAND = 3;
    public int State;
    public AirplaneQueue NextQueue;
    public AirplaneQueue CurrentQueue;
    // public ControlLogic TrafficGuid = new ControlLogic();
    static int total;
    private int ID;
    Point prevPoint = new Point(0, 0);
    public Point currntPoint;
    AirportGroundMap groundMap = new AirportGroundMap();
    public int currentLink = 0;
    public int flow_ID = 0;
    GlobalClock Watch = new GlobalClock();
    Point[] destination = new Point[2];
    public boolean AirplaneFinalStop = false;
    public long startTime;
    public long finishTime;
    SpeedController moveGuid = new SpeedController(Watch.interval);
    boolean moveFlag;
    public final static int airplaneTick = 100;

    Airplane(int setPath, int x, int y) {
        total++;
        ID = total;
        destination[0] = new Point(300, 100);
        destination[1] = new Point(380, 100);
        flow_ID = setPath;
        currntPoint.x = x;
        currntPoint.y = y;
        startTime = Watch.getTime();
        resetTimer();
        setStateToApproach();
    }

    public int getMovingStep(int TimeInSeconds) {
        return (int)(moveGuid.getFastMoveStep(TimeInSeconds, airplaneTick));
    }

    public void setSpeed(Point From, Point To, int deltaTime)
    {
        moveGuid.setSpeedProfile(From, To, startTime,
            startTime + deltaTime);
    }

    public int ID() { return ID;}

    public synchronized void setStateToApproach() {
        State = APPROACH;
    }
    public synchronized void setStateToLand() {
        State = LAND;
    }
    public synchronized void setStateToTaxi() {
        State = TAXI;
    }
    public synchronized void setStateToTakeoff() {
        State = TAKEOFF;
    }
    public synchronized void setStateToDepart() {
        State = DEPART;
    }

    public boolean isApproaching()

```

```

    {
        return (State == APPROACH? true:false);
    }
    public boolean isLanding()
    {
        return (State == LAND? true:false);
    }
    public boolean isTaxiing()
    {
        return (State == TAXI? true:false);
    }
    public boolean isTakeoff()
    {
        return (State == TAKEOFF? true:false);
    }
    public boolean isDeparting()
    {
        return (State == DEPART? true:false);
    }

    public boolean equals(Airplane comparedAirplane) {
        if(comparedAirplane.ID() == ID)
            return true;
        else
            return false;
    }
    public int random(int rd) {
        return (int) Math.floor(Math.random()*rd);
    }

    public void setTrackPoint(Point TrackPoint) {
        prevPoint.x = TrackPoint.x;
        prevPoint.y = TrackPoint.y;
    }
    public synchronized void run() {

        if(AirplaneFinalStop != true)
        {
            //Operation control
            if(timerTimeOut())
                ControlLogic.controlAllAirplanes(this);
        }
    }
}

class AirportGroundMap {
    public Point[] nodes = new Point[5];
    Point[] link_2 = new Point[2];
    Point[] link_3 = new Point[2];
    Point[] link_4 = new Point[2];
    Point[] terminalPoints = new Point[2];
    int delta = 4;

    AirportGroundMap(){
        nodes[0] = new Point(50, 300);
        nodes[1] = new Point(250, 300);
        nodes[2] = new Point(380, 300);
        nodes[3] = new Point(380, 100);
        nodes[4] = new Point(300, 100);

        terminalPoints[0]= new Point(300, 100);
        terminalPoints[1]= new Point(380, 100);
    }

    public void drawLinks(Graphics g) {
        /*
        g.drawLine(nodes[0].x, nodes[0].y,

```

```

        nodes[1].x, nodes[1].y);
g.drawLine(nodes[1].x, nodes[1].y,
           nodes[2].x, nodes[2].y);
g.drawLine(nodes[2].x, nodes[2].y,
           nodes[3].x, nodes[3].y);
g.drawLine(nodes[1].x, nodes[1].y,
           nodes[4].x, nodes[4].y);
*/
for(int i = 0; i < GroundMonitor.queues.size(); i++)
{
    AirplaneQueue tempQ = (AirplaneQueue)
        GroundMonitor.queues.elementAt(i);
    if(tempQ != null)
    {
        if(tempQ.From.equals(tempQ.To))
            g.drawOval(tempQ.From.x, tempQ.From.y, 3, 3);
        else
            g.drawLine(tempQ.From.x, tempQ.From.y,
                       tempQ.To.x, tempQ.To.y);
    }
}

}

public void drawPath(Graphics g, int Path) {

    if(Path == 1) {
        g.drawLine(nodes[0].x, nodes[0].y,
                  nodes[1].x, nodes[1].y);
        g.drawLine(nodes[1].x, nodes[1].y,
                  nodes[2].x, nodes[2].y);
        g.drawLine(nodes[2].x, nodes[2].y,
                  nodes[3].x, nodes[3].y);
    }

    if(Path == 2) {
        g.drawLine(nodes[0].x, nodes[0].y,
                  nodes[1].x, nodes[1].y);
        g.drawLine(nodes[1].x, nodes[1].y,
                  nodes[4].x, nodes[4].y);
    }
}

public void drawTerminal(Graphics g, int Terminal)
{
    for(int i = 0; i < GroundMonitor.queues.size(); i++)
    {
        AirplaneQueue tempQ = (AirplaneQueue)
            GroundMonitor.queues.elementAt(i);
        if(tempQ != null && tempQ.From.equals(tempQ.To))
            g.fillOval(tempQ.From.x, tempQ.From.y, 2, 2);

    }

    // g.fillOval(terminalPoints[Terminal].x,
    //            terminalPoints[Terminal].y, 20, 30);
}

public void drawMap(Graphics g) {
    g.setColor(Color.black);
    drawLinks(g);
    g.setColor(Color.green);
    drawTerminal(g, 0);
    g.setColor(Color.pink);
    drawTerminal(g, 1);
}

public void getNextPointToMove(Airplane TempAirplane,

```

```

        long LocalTime)
    {
        Point currentPoint = TempAirplane.crrntPoint;
        if(TempAirplane.flow_ID == 0) {
            //set flag runway
            if(currentPoint.x >=50 && currentPoint.x <380)
            {
                if(TempAirplane.moveGuid.flag != "runway")
                {
                    TempAirplane.moveGuid.flag = "runway";
                    TempAirplane.setSpeed(nodes[0], nodes[2], 50);
                    delta = TempAirplane.getMovingStep(50);
                    TempAirplane.moveGuid.setTimeSpaceTable(
                        nodes[0], nodes[2],
                        LocalTime, 50);
                }
                Point nextPoint =
                    TempAirplane.moveGuid.getNextCoordinate(LocalTime);
                if(nextPoint != null)
                    currentPoint.move(nextPoint.x, nextPoint.y);
            }

            //set flag taxiway
            if(currentPoint.x >= 380 && currentPoint.y <= 300
                && currentPoint.y > 100)
            {
                if(TempAirplane.moveGuid.flag != "taxiway")
                {
                    TempAirplane.moveGuid.flag = "taxiway";
                    TempAirplane.setSpeed(nodes[2], nodes[3], 100);
                    TempAirplane.moveGuid.setTimeSpaceTable(
                        nodes[2], nodes[3],
                        LocalTime, 100);
                }

                Point nextPoint =
                    TempAirplane.moveGuid.getNextCoordinate(LocalTime);
                if(nextPoint != null)
                    currentPoint.move(nextPoint.x, nextPoint.y);
            }
            //set flag terminal
            if(currentPoint.y <= 100)
            {
                if(TempAirplane.moveGuid.flag != "runway")
                {
                    TempAirplane.moveGuid.flag = "terminal";
                }
            }

            currentPoint.move(380, 100);
        }
    }

    if(TempAirplane.flow_ID == 1) {
        delta = 4;
        if(currentPoint.x >=50 && currentPoint.x <250)
            currentPoint.translate(delta, 0);

        if(currentPoint.x >=250 && currentPoint.y <= 300
            && currentPoint.y > 100)
            currentPoint.translate(
                Math.round((float)Math.cos(Math.atan(200/50))*delta),
                -Math.round((float)Math.sin(Math.atan(200/50))*delta));
    }
}

```

```

        if(currentPoint.y <= 100)
            currentPoint.move(300, 100);
    }
}

class AirplaneFleet implements Runnable {
    static Thread runAirplane;
    Airplane tempAirplane;

    AirplaneFleet()
    {
        if(runAirplane == null){
            runAirplane = new Thread(this);
            runAirplane.start();
        }
    }
    public void startRun(){
        runAirplane = new Thread(this);
        runAirplane.start();
    }
    public void run(){
        while(true){
            if(!AirplaneScheduler.AirplaneList.isEmpty())
                for(int i = 0; i < AirplaneScheduler.AirplaneList.size(); i++)
                {
                    tempAirplane = (Airplane) AirplaneScheduler.AirplaneList.elementAt(i);
                    tempAirplane.run();
                }

            try{runAirplane.sleep(30);} catch (Exception e) {};
        }
    }
}

```

```

class SpeedController extends Object {
    public int travelDistance;
    double deltaXSqt, deltaYSqt;
    Vector timeSpaceTable = new Vector(100);
    long deltaTime;
    long startTime;
    long finishTime;
    int clockInterval;
    String flag = new String("no flag");
    Point from = new Point(0,0);
    Point to = new Point(0,0);
    int prevTotal = 0;
    int n_segs = 30;

    SpeedController(int Interval) {clockInterval = Interval;}

    public void setSpeedProfile(Point From, Point To, long StartTime, long FinishTime)
    {
        deltaXSqt = Math.pow((double)(From.x -To.x), 2);
        deltaYSqt = Math.pow((double)(From.y -To.y), 2);
        System.out.println("from:" + From.x + ":" + To.x
            + "time: " + StartTime + "-> " + FinishTime);
        travelDistance = (int) Math.round(Math.sqrt(deltaXSqt + deltaYSqt));
        deltaTime = FinishTime - StartTime;
        this.startTime = StartTime;
        this.finishTime = FinishTime;
        from.x = From.x; from.y = From.y;
        to.x = To.x; to.y = To.y;
    }

    int getDistance ( Point From, Point To)
    {
        double deltaX = Math.pow((double)(From.x -To.x), 2);
    }
}

```

```

double deltaY = Math.pow((double)(From.y - To.y), 2);
int distance = (int) Math.round(Math.sqrt(deltaX2 + deltaY2));
return distance;
}

public int getSpeedProfile(Point CurrentPoint, long FutureTime)
{
    int total = (int)((long)travelDistance)/deltaTime)*(FutureTime - startTime);
    int distance = total - prevTotal;
    prevTotal = total;
    return Math.min(travelDistance, distance);
}

public int getFastMoveStep(long TimeLimit, int AirplaneTick){
    return (int) Math.round(travelDistance/TimeLimit);
}

public void setTimeSpaceTable(Point from, Point to,
    long startTime, int deltaTime)
{
    if(from.x == to.x)
    {
        int length = to.y - from.y;

        double delta = length/n_segs;
        timeSpaceTable.removeAllElements();
        for(int i = 0; i < n_segs; i++)
        {
            if(i < n_segs - 1) {
                timeSpaceTable.addElement(
                    new MovingRecord((long)(startTime + (long)i*deltaTime/n_segs),
                    new Point(from.x, (int) Math.round(i*delta) + from.y)));
            }
            else
            {
                timeSpaceTable.addElement(
                    new MovingRecord((long)(startTime + i),
                    new Point(to.x, to.y)));
            }
        }
        return;
    }

    if(from.y == to.y)
    {
        int length = to.x - from.x;
        double delta = length/n_segs;
        timeSpaceTable.removeAllElements();
        for(int i = 0; i < n_segs; i++)
        {
            if(i < n_segs-1) {
                timeSpaceTable.addElement(
                    new MovingRecord((long)(startTime + (long)i*deltaTime/n_segs),
                    new Point((int) Math.round(i*delta)+from.x, to.y)));
            }
            else
            {
                timeSpaceTable.addElement(
                    new MovingRecord((long)(startTime + i),
                    new Point(to.x, to.y)));
            }
        }
        return;
    }
}

public Point getNextCoordinate(long timeAt) {

```

```

        int counter =0;
        MovingRecord tempRecord;
        while(counter < timeSpaceTable.size())
        {
            tempRecord =(MovingRecord)timeSpaceTable.elementAt(counter);
            if(tempRecord.timeAt >= timeAt )
                return tempRecord.xyCoordination;
            counter++;
        }

        return
        ((MovingRecord)timeSpaceTable.elementAt(
        timeSpaceTable.size() - 1)).xyCoordination; // always return last point
    }
}

```

```

class MovingRecord extends Object
{
    Point xyCoordination;
    long timeAt = 0;

    MovingRecord(long time, Point At)
    {
        timeAt = time;
        xyCoordination = new Point(At.x, At.y);
        // System.out.println("create:" + At.x + "," + At.y);
    }
}

```

```

class AirplaneDataSet
{
    public boolean equals(AirplaneDataSet temp_data)
    {
        if(this.name.equals(temp_data.name))
            return true;
        else
            return false;
    }
    public AirplaneDataSet(String name, String category)
    {
        this.name = new String(name);
        this.category = new String(category);
    }
    public String category;
    public String name;
}

```

```

import java.awt.*;
import java.util.*;

```

```

public class AirplaneScheduler implements Runnable {
    static Vector AirplaneList;
    static Vector ArrivalRates;
    static Thread CreateFlightThread;
    GlobalClock Watch = new GlobalClock();
    int delta = 120;
    int prevTime = 0;
    long currentTime;

    AirplaneScheduler() {
        // System.out.println("start initiating Scheduler");
        startScheduler();
        // System.out.println("finished initiating Scheduler");
    }
}

```

```

public synchronized void startScheduler() {
    System.out.println("here");
    AirplaneList = null;
    ArrivalRates = null;
}

```

```

        AirplaneList = new Vector(100);
        ArrivalRates = new Vector();
        addRandomSchedule(0, 120);
        addRandomSchedule(1, 60);
        CreateFlightThread = new Thread(this);
        CreateFlightThread.start();
    }

    public void createAirplane(int Path) {
        // AirplaneList.addElement(new Airplane(Path, 50, 300));
        AirplaneList.addElement(new Airplane(Path));
    }

    // add random schedule for the Path
    public void addRandomSchedule(int Path, int ArrivalRate) {
        ArrivalRates.addElement(
            (new ArrivalGenerator(Path, ArrivalRate)));
    }

    public synchronized void run() {
        while(true) {
            // control the total simulated airplane fleet size
            if(AirplaneList.size() < 200)
            {
                //get current time
                currentTime = (long) Watch.getTime();
                //create arrival for each path
                for(int i = 0; i < ArrivalRates.size(); i++)
                {
                    //System.out.println("Arrival Path size: " + ArrivalRates.size());
                    ArrivalGenerator newRate =
                        (ArrivalGenerator) ArrivalRates.elementAt(i);
                    //check if the interarrival interval reached
                    if(newRate != null &&
                        newRate.nextTime <= currentTime)
                    {
                        // create time mark for next arrival
                        newRate.setTimeMarkForNextArrival((int)currentTime);
                        // System.out.println("Create ACFT at: " + currentTime);
                        // create this arrival
                        createAirplane(newRate.path);
                    }
                    newRate = null;
                }
                try{CreateFlightThread.sleep(50);} catch (Exception e) {};
            }
        }
    }
}

class ArrivalGenerator extends Object{
    int path;
    int averageInterval;
    int nextTime;
    int minInterval;

    ArrivalGenerator(int Path, int AverageInterval) {
        minInterval = 20;
        this.path = Path;
        this.averageInterval = AverageInterval - minInterval;
        this.nextTime = randomInterval();
    }

    int randomInterval(){
        return( (int)(Math.floor(Math.random()*averageInterval))
            + minInterval);
    }
}

```

```

public void setTimeMarkForNextArrival(int currentTime) {
    nextTime = currentTime + randomInterval();
    //System.out.println(nextTime);
}

}

public final class ControlLogic extends Object
{
    public static int getCategoryByAircraft(Airplane airplane)
    {
        double tempCat = Math.random()*100;
        double sum_1 = 0, sum_2 = 0;
        String tempAcName;

        for(int i = 0; i < TrafficParam.AIRCRAFT_MIX.size(); i++)
        {
            if(i > 0)
                sum_1 +=((Double)TrafficParam.AIRCRAFT_MIX.elementAt(i-1)).doubleValue();
            sum_2 +=((Double)TrafficParam.AIRCRAFT_MIX.elementAt(i)).doubleValue();
            if(tempCat >= sum_1 && tempCat < sum_2)
            {
                tempAcName = (String) TrafficParam.AIRCRAFT_NAME.elementAt(i);
                airplane.setAircraftName(tempAcName);
                airplane.setAirplaneCategory(TrafficParam.getCategoryByAircraft(tempAcName));
                if(airplane.isCategory("CAT_A"))
                    return 1;
                if(airplane.isCategory("CAT_B"))
                    return 2;
                if(airplane.isCategory("CAT_C"))
                    return 3;
                if(airplane.isCategory("CAT_D"))
                    return 4;
            }
        }

        return 0;
    }

    public static long getTravelTime(String queue_name, Airplane airplane)
    {
        long time;
        if(airplane.CurrentQueue.name.equals("I6_R") ||
            airplane.CurrentQueue.name.equals("I6_L")){

            time = TrafficParam.getROT(queue_name, airplane);
            // System.out.println("queue: "+queue_name+"time: "+time);

        }
        else
            time = TrafficParam.getTaxiwayTravelTime(queue_name, airplane);
        return time;
    }

    public static int getCategory()
    {
        double tempCat = Math.random();
        double sum_1 = 0, sum_2 = 0;

        for(int i = 0; i < TrafficParam.CATEGORY_MIX.size(); i++)
        {
            if(i > 0)
                sum_1 +=((Double)TrafficParam.CATEGORY_MIX.elementAt(i-1)).doubleValue();
            sum_2 +=((Double)TrafficParam.CATEGORY_MIX.elementAt(i)).doubleValue();
            if(tempCat >= sum_1 && tempCat < sum_2)
                return i+1;
        }
    }
}

```

```

    return 0;
}
public static int getNextAvailableExitIdForCat_D()
{
    String tempQname;
    for(int i = 3; i < TrafficParam.EXIT_NAME.size(); i++)
    {
        tempQname = (String) TrafficParam.EXIT_NAME.elementAt(i);
        if(GroundMonitor.airplaneQueueIsEmpty(tempQname))
            return i+1; // indx starting with 0
    }
    return 6;
}
public static int getNextAvailableExitIdForCat_C()
{
    String tempQname;
    for(int i = 1; i < TrafficParam.EXIT_NAME.size(); i++)
    {
        tempQname = (String) TrafficParam.EXIT_NAME.elementAt(i);
        if(GroundMonitor.airplaneQueueIsEmpty(tempQname))
            return i+1; // indx starting with 0
    }
    return 6;
}
public static int getNextAvailableExitIdForCat_B()
{
    String tempQname;
    for(int i = 1; i < TrafficParam.EXIT_NAME.size(); i++)
    {
        tempQname = (String) TrafficParam.EXIT_NAME.elementAt(i);
        if(GroundMonitor.airplaneQueueIsEmpty(tempQname))
            return i+1; // indx starting with 0
    }
    return 6;
}
public static int getNextAvailableExitIdForCat_A()
{
    String tempQname;
    for(int i = 0; i < TrafficParam.EXIT_NAME.size(); i++)
    {
        tempQname = (String) TrafficParam.EXIT_NAME.elementAt(i);
        if(GroundMonitor.airplaneQueueIsEmpty(tempQname))
            return i+1; // indx starting with 0
    }
    return 6;
}
public static void setTrafficLightGreen(String queue_name)
{
    ((AirplaneQueue)
    GroundMonitor.getAirplaneQueueObject(queue_name)
    ).TrafficSignal.setGreen();
}
public static void setTrafficLightRed(String queue_name)
{
    ((AirplaneQueue)
    GroundMonitor.getAirplaneQueueObject(queue_name)
    ).TrafficSignal.setRed();
}
public static void controlAllAirplanes(Airplane airplane)
{
    airplane.delayIndicator = false;
    if(airplane.flow_ID == 0)
        controlDepartAirplanes(airplane);
    else
        controlArriveAirplanes(airplane);
}
public static void controlDepartAirplanes(Airplane airplane)

```

```

{
    getCategoryByAircraft(airplane);
    airplaneFollowPath_16_L_TAKEOFF(airplane);
}
public static void airplaneFollowPath_16_L_TAKEOFF(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_L_TAKEOFF");
        airplane.RoutName = new String("ENROUT");
        // System.out.println("set Path name from airplane" + airplane.ID());
    }

    if(airplane.getTimer() == 0)
    {
        airplane.setTimer(10);
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("16_L");
        // System.out.println("set timer in approach for airplane: " + airplane.ID());
        return;
    }

    //enter runway 16_L
    if(airplane.NextQueue.name.equals("16_L"))
    {
        if(airplane.timerTimeOut() &&
            GroundMonitor.airplaneQueueIsEmpty("16_L")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_3")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_4")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_5")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_6")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_7")
            &&
            GroundMonitor.airplaneQueueIsEmpty("C_8")
        )
        {
            // System.out.println("Airplane: " + airplane.ID() +"enter runway 16_L");
            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("TERMINAL_34_R");
            GroundMonitor.enterAirplaneIntoQueue("16_L", airplane);

            airplane.setTimer(TrafficParam.getROT("16_L",airplane));
            return;
        }
    }

    //enter runway C_6
    if(airplane.NextQueue.name.equals("TERMINAL_34_R"))
    {
        GroundMonitor.moveAirplaneOutFromQueue("16_L", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        // GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_34_R", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
    }
}

```

```

        airplane.delayIndicator = true;
        return;
    }
    public static void airplaneFollowPath_16_R_TO_C_3(Airplane airplane)
    {
        if(airplane.pathName == null)
        {
            airplane.pathName = new String("16_R_TO_C_3");
            // System.out.println("set Path name from airplane" + airplane.ID());
        }

        if(airplane.getTimer() == 0)
        {
            airplane.setTimer(120);
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("16_R");
            // System.out.println("set timer in approach for airplane: " + airplane.ID());
            return;
        }

        //enter runway 16_R
        if(airplane.NextQueue.name.equals("16_R"))
        {
            if(airplane.timerTimeOut() &&
                GroundMonitor.airplaneQueueIsEmpty("16_R"))
            {
                setTrafficLightGreen("16_R");

                airplane.CurrentQueue = airplane.NextQueue;
                airplane.NextQueue =
                    GroundMonitor.getAirplaneQueueObject("C_3");
                GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);

                // System.out.println("Airplane: " + airplane.ID() +"enter runway 16_R");

                airplane.setTimer(getTravelTime("C_3", airplane));
                airplane.setStateToLand();
                return;
            }
            else
                setTrafficLightRed("16_R");
        }

        //enter runway C_6
        if(airplane.NextQueue.name.equals("C_3"))
        {
            if(GroundMonitor.airplaneQueueIsEmpty("C_3"))
            {
                setTrafficLightGreen("C_3");

                GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
                // System.out.println("Airplane: " + airplane.ID() +" leaves runway 16_R");

                airplane.CurrentQueue = airplane.NextQueue;
                airplane.NextQueue =
                    GroundMonitor.getAirplaneQueueObject("B_2");
                GroundMonitor.enterAirplaneIntoQueue("C_3", airplane);
                // System.out.println("Airplane: " + airplane.ID() +" enter exit C_3");

                airplane.setTimer(getTravelTime("C_3", airplane));
                airplane.setStateToTaxi();
                return;
            }
            else
                setTrafficLightRed("C_3");
        }

        if(airplane.NextQueue.name.equals("B_2"))

```

```

    {
        if(GroundMonitor.airplaneQueueIsEmpty("B_2")
            &&
            GroundMonitor.airplaneQueueIsEmpty("16_L"))
        {
            setTrafficLightGreen("B_2");

            GroundMonitor.moveAirplaneOutFromQueue("C_3", airplane);
            // System.out.println("Airplane: " + airplane.ID() +"leaves exit C_3");

            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("TERMINAL_B_2");
            GroundMonitor.enterAirplaneIntoQueue("B_2", airplane);
            // System.out.println("Airplane: " + airplane.ID() +"enter taxiway B_2");

            airplane.setTimer(getTravelTime("B_2", airplane));
            return;
        }
        else
            setTrafficLightRed("B_2");
    }

    if(airplane.NextQueue.name.equals("TERMINAL_B_2"))
    {
        GroundMonitor.moveAirplaneOutFromQueue("B_2", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        // GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_2", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
    }

    airplane.delayIndicator = true;
    return;
}

public static void airplaneFollowPath_16_R_TO_C_4(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_R_TO_C_4");
        // System.out.println("set Path name from airplane" + airplane.ID());
    }

    if(airplane.getTimer() == 0)
    {
        airplane.setTimer(120);
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("16_R");
        // System.out.println("set timer in approach for airplane: " + airplane.ID());
        return;
    }

    //enter runway 16_R
    if(airplane.NextQueue.name.equals("16_R"))
    {
        if(airplane.timerTimeOut() &&
            GroundMonitor.airplaneQueueIsEmpty("16_R"))
        {
            setTrafficLightGreen("16_R");

            airplane.CurrentQueue = airplane.NextQueue;

```

```

        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("C_4");
        GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);

// System.out.println("Airplane: " + airplane.ID() +"enter runway 16_R");

        airplane.setTimer(getTravelTime("C_4", airplane));
        airplane.setStateToLand();
        return;
    }
    else
        setTrafficLightRed("16_R");
}

//enter runway C_6
if(airplane.NextQueue.name.equals("C_4"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("C_4"))
    {
        setTrafficLightGreen("C_4");

        GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
// System.out.println("Airplane: " + airplane.ID() +" leaves runway 16_R");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("B_7");
        GroundMonitor.enterAirplaneIntoQueue("C_4", airplane);
// System.out.println("Airplane: " + airplane.ID() +" enter exit C_8");

        airplane.setTimer(getTravelTime("C_4", airplane));
        airplane.setStateToTaxi();
        return;
    }
    setTrafficLightRed("C_4");
}

if(airplane.NextQueue.name.equals("B_7"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("B_7")
        &&
        GroundMonitor.airplaneQueueIsEmpty("16_L"))
    {
        setTrafficLightGreen("B_7");

        GroundMonitor.moveAirplaneOutFromQueue("C_4", airplane);
// System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("TERMINAL_B_7");
        GroundMonitor.enterAirplaneIntoQueue("B_7", airplane);
// System.out.println("Airplane: " + airplane.ID() +"enter taxiway B_11");

        airplane.setTimer(getTravelTime("B_7", airplane));
        return;
    }
    else
        setTrafficLightRed("B_7");
}

if(airplane.NextQueue.name.equals("TERMINAL_B_7"))
{
    GroundMonitor.moveAirplaneOutFromQueue("B_7", airplane);
// System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

    airplane.CurrentQueue = airplane.NextQueue;

```

```

        // GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_7", airplane);
//      System.out.println("Airplane: " + airplane.ID() +"enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
    }

    airplane.delayIndicator = true;
    return;
}

public static void setDynamicControl()
{
    controlMethod = DYNAMIC_CONTROL;
}
public final static int DYNAMIC_CONTROL = 1;
public final static int STATIC_CONTROL = 0;
public static void setStaticControl()
{
    controlMethod = STATIC_CONTROL;
}
static int controlMethod;
public static void airplaneFollowPath_16_R_TO_C_5(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_R_TO_C_5");
//      System.out.println("set Path name from airplane" + airplane.ID());
    }

    if(airplane.getTimer() == 0)
    {
        airplane.setTimer(120);
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("16_R");
//      System.out.println("set timer in approach for airplane: " + airplane.ID());
        return;
    }

//enter runway 16_R
    if(airplane.NextQueue.name.equals("16_R"))
    {
        if(airplane.timerTimeOut() &&
            GroundMonitor.airplaneQueueIsEmpty("16_R"))
        {
            setTrafficLightGreen("16_R");

            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("C_5");
            GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);

//      System.out.println("Airplane: " + airplane.ID() +"enter runway 16_R");

            airplane.setTimer(getTravelTime("C_5", airplane));
            airplane.setStateToLand();
            return;
        }
        else
            setTrafficLightRed("16_R");
    }
}

//enter runway C_6
    if(airplane.NextQueue.name.equals("C_5"))
    {
        if(GroundMonitor.airplaneQueueIsEmpty("C_5"))
        {

```

```

        setTrafficLightGreen("C_5");

        GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
//      System.out.println("Airplane: " + airplane.ID() + " leaves runway 16_R");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("B_9");
        GroundMonitor.enterAirplaneIntoQueue("C_5", airplane);
//      System.out.println("Airplane: " + airplane.ID() + " enter exit C_8");

        airplane.setTimer(getTravelTime("C_5", airplane));
        airplane.setStateToTaxi();
        return;
    }
    else
        setTrafficLightRed("C_5");
}

if(airplane.NextQueue.name.equals("B_9"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("B_9")
        &&
        GroundMonitor.airplaneQueueIsEmpty("16_L"))
    {
        setTrafficLightGreen("B_9");

        GroundMonitor.moveAirplaneOutFromQueue("C_5", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("TERMINAL_B_9");
        GroundMonitor.enterAirplaneIntoQueue("B_9", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "enter taxiway B_11");

        airplane.setTimer(getTravelTime("B_9", airplane));
        return;
    }
    else
        setTrafficLightRed("B_9");
}

if(airplane.NextQueue.name.equals("TERMINAL_B_9"))
{
//      GroundMonitor.moveAirplaneOutFromQueue("B_9", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
//      GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_9", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
    }

    airplane.delayIndicator = true;
    return;
}

public static void airplaneFollowPath_16_R_TO_C_6(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_R_TO_C_6");
//      System.out.println("set Path name from airplane" + airplane.ID());
    }
}

```

```

}

if(airplane.getTimer() == 0)
{
airplane.setTimer(120);
airplane.NextQueue =
    GroundMonitor.getAirplaneQueueObject("16_R");
// System.out.println("set timer in approach for airplane: " + airplane.ID());
return;
}

//enter runway 16_R
if(airplane.NextQueue.name.equals("16_R"))
{
    if(airplane.timerTimeOut() &&
        GroundMonitor.airplaneQueueIsEmpty("16_R"))
    {
        setTrafficLightGreen("16_R");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("C_6");
        GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);

        // System.out.println("Airplane: " + airplane.ID() + "enter runway 16_R");

        airplane.setTimer(getTravelTime("C_6", airplane));
        airplane.setStateToLand();
        return;
    }
    else
        setTrafficLightRed("16_R");
}

//enter runway C_6
if(airplane.NextQueue.name.equals("C_6"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("C_6"))
    {
        setTrafficLightGreen("C_6");

        GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
        // System.out.println("Airplane: " + airplane.ID() + " leaves runway 16_R");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("B_10");
        GroundMonitor.enterAirplaneIntoQueue("C_6", airplane);
        // System.out.println("Airplane: " + airplane.ID() + " enter exit C_8");

        airplane.setTimer(getTravelTime("C_6", airplane));
        airplane.setStateToTaxi();
        return;
    }
    else
        setTrafficLightRed("C_6");
}

if(airplane.NextQueue.name.equals("B_10"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("B_10")
        &&
        GroundMonitor.airplaneQueueIsEmpty("16_L"))
    {
        setTrafficLightGreen("B_10");

        GroundMonitor.moveAirplaneOutFromQueue("C_6", airplane);
        // System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");
    }
}

```

```

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("TERMINAL_B_10");
        GroundMonitor.enterAirplaneIntoQueue("B_10", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "enter taxiway B_11");

        airplane.setTimer(getTravelTime("B_10", airplane));
        return;
    }
    else
        setTrafficLightRed("B_10");
}

if(airplane.NextQueue.name.equals("TERMINAL_B_10"))
{
    GroundMonitor.moveAirplaneOutFromQueue("B_10", airplane);
//    System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");

    airplane.CurrentQueue = airplane.NextQueue;
//    GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
    GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_10", airplane);
//    System.out.println("Airplane: " + airplane.ID() + "enter taxiway TERMINAL_B_11");

    airplane.setAirplaneFinalStop();
    return;
}

    airplane.delayIndicator = true;
    return;
}
public static void airplaneFollowPath_16_R_TO_C_7(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_R_TO_C_7");
//      System.out.println("set Path name from airplane" + airplane.ID());
    }

    if(airplane.getTimer() == 0)
    {
        airplane.setTimer(120);
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("16_R");
//      System.out.println("set timer in approach for airplane: " + airplane.ID());
        return;
    }

//enter runway 16_R
    if(airplane.NextQueue.name.equals("16_R"))
    {
        if(airplane.timerTimeOut() &&
            GroundMonitor.airplaneQueueIsEmpty("16_R"))
        {
            setTrafficLightGreen("16_R");

            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("C_7");
            GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);

//      System.out.println("Airplane: " + airplane.ID() + "enter runway 16_R");

            airplane.setTimer(getTravelTime("C_7", airplane));
            airplane.setStateToLand();
            return;
        }
    }
}

```

```

else
    setTrafficLightRed("16_R");
}

//enter runway C_8
if(airplane.NextQueue.name.equals("C_7"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("C_7"))
    {
        setTrafficLightGreen("C_7");

        GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
//      System.out.println("Airplane: " + airplane.ID() + " leaves runway 16_R");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("B_10");
        GroundMonitor.enterAirplaneIntoQueue("C_7", airplane);
//      System.out.println("Airplane: " + airplane.ID() + " enter exit C_8");

        airplane.setTimer(getTravelTime("C_7", airplane));
        airplane.setStateToTaxi();
        return;

    }
    else
        setTrafficLightRed("C_7");
}

if(airplane.NextQueue.name.equals("B_10"))
{
    if(GroundMonitor.airplaneQueueIsEmpty("B_10")
        &&
        GroundMonitor.airplaneQueueIsEmpty("16_L"))

    {
        setTrafficLightGreen("B_10");

        GroundMonitor.moveAirplaneOutFromQueue("C_7", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("TERMINAL_B_10");
        GroundMonitor.enterAirplaneIntoQueue("B_10", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "enter taxiway B_11");

        airplane.setTimer(getTravelTime("B_10", airplane));
        return;
    }
    else
        setTrafficLightRed("B_10");
}

if(airplane.NextQueue.name.equals("TERMINAL_B_10"))
{
//      GroundMonitor.moveAirplaneOutFromQueue("B_10", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
//      GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_10", airplane);
//      System.out.println("Airplane: " + airplane.ID() + "enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
}

```

```

    }

    airplane.delayIndicator = true;
    return;
}
public static void airplaneFollowPath_16_R_TO_C_8(Airplane airplane)
{
    if(airplane.pathName == null)
    {
        airplane.pathName = new String("16_R_TO_C_8");
        // System.out.println("set Path name from airplane" + airplane.ID());
    }

    if(airplane.getTimer() == 0)
    {
        airplane.setTimer(120);
        // System.out.println("set timer in approach for airplane: " + airplane.ID());
        // airplane.CurrentQueue = null;

        airplane.NextQueue =
            GroundMonitor.getAirplaneQueueObject("16_R");
        return;
    }

    //enter runway 16_R
    if(airplane.NextQueue.name.equals("16_R"))
    {
        if(airplane.timerTimeOut() &&
            GroundMonitor.airplaneQueueIsEmpty("16_R"))
        {
            setTrafficLightGreen("16_R");

            // System.out.println("Airplane: " + airplane.ID() +"enter runway 16_R");
            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("C_8");
            GroundMonitor.enterAirplaneIntoQueue("16_R", airplane);
            airplane.setTimer(getTravelTime("C_8", airplane));
            airplane.setStateToLand();
            return;
        }
        else
            setTrafficLightRed("16_R");
    }

    //enter runway C_8
    if(airplane.NextQueue.name.equals("C_8"))
    {
        if(GroundMonitor.airplaneQueueIsEmpty("C_8"))
        {
            GroundMonitor.moveAirplaneOutFromQueue("16_R", airplane);
            // System.out.println("Airplane: " + airplane.ID() +" leaves runway 16_R");
            setTrafficLightGreen("C_8");

            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("B_11");

            GroundMonitor.enterAirplaneIntoQueue("C_8", airplane);
            // System.out.println("Airplane: " + airplane.ID() +" enter exit C_8");
            airplane.setTimer(getTravelTime("C_8", airplane));
            airplane.setStateToTaxi();
            return;
        }
        else
            setTrafficLightRed("C_8");
    }
}

```

```

    }
    if(airplane.NextQueue.name.equals("B_11"))
    {
        if(GroundMonitor.airplaneQueueIsEmpty("B_11")
            &&
            GroundMonitor.airplaneQueueIsEmpty("16_L"))
        {
            setTrafficLightGreen("B_11");

            GroundMonitor.moveAirplaneOutFromQueue("C_8", airplane);
            // System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

            airplane.CurrentQueue = airplane.NextQueue;
            airplane.NextQueue =
                GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
            GroundMonitor.enterAirplaneIntoQueue("B_11", airplane);
            //System.out.println("Airplane: " + airplane.ID() +"enter taxiway B_11");

            airplane.setTimer(getTravelTime("B_11", airplane));
            return;

        }
        else
            setTrafficLightRed("16_R");
    }

    if(airplane.NextQueue.name.equals("TERMINAL_B_11"))
    {
        GroundMonitor.moveAirplaneOutFromQueue("B_11", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"leaves exit C_8");

        airplane.CurrentQueue = airplane.NextQueue;
        // GroundMonitor.getAirplaneQueueObject("TERMINAL_B_11");
        GroundMonitor.enterAirplaneIntoQueue("TERMINAL_B_11", airplane);
        // System.out.println("Airplane: " + airplane.ID() +"enter taxiway TERMINAL_B_11");

        airplane.setAirplaneFinalStop();
        return;
    }

    airplane.delayIndicator = true;
    return;
}

//customized design for specific ground network configuration
public static void controlArriveAirplanes(Airplane airplane)
{
    setDynamicControl();

    if(controlMethod == DYNAMIC_CONTROL && airplane.pathName == null)
    {
        airplane.RoutName = new String("ARRIVE");
        // int cat =getCategory();
        int cat = getCategoryByAircraft(airplane);

        switch(cat) {
            case 1:
                airplane.flow_ID = getNextAvailableExitIdForCat_A();
                break;

            case 2:
                airplane.flow_ID = getNextAvailableExitIdForCat_B();
                break;
        }
    }
}

```

```

        case 3:
            airplane.flow_ID = getNextAvailableExitIdForCat_C();
            break;

        case 4:
            airplane.flow_ID = getNextAvailableExitIdForCat_D();
            break;

        default:
            airplane.flow_ID = getNextAvailableExitIdForCat_B();
            break;
    }
}

switch(airplane.flow_ID) {
    case 6: //C_8
        airplaneFollowPath_16_R_TO_C_8(airplane);
        break;
    case 5: //C_7
        airplaneFollowPath_16_R_TO_C_7(airplane);
        break;
    case 4: //C_6
        airplaneFollowPath_16_R_TO_C_6(airplane);
        break;
    case 3: //C_5
        airplaneFollowPath_16_R_TO_C_5(airplane);
        break;
    case 2: //C_4
        airplaneFollowPath_16_R_TO_C_4(airplane);
        break;
    case 1: //C_3
        airplaneFollowPath_16_R_TO_C_3(airplane);
        break;

    default:
        break;
}
}
}

/*      System.out.println("enter controlArriveAirplane Path 0: "+
        GroundMonitor.queueHasAirplane("16_R") +
        "timer: " + airplane.getTimer() +
        "time: " + airplane.Watch.getTime());
*/

import java.awt.*;

public final class GlobalClock implements Runnable{
    private static long time_base;
    private static int delta;
    private static int hour;
    private static int minute;
    private static int second;
    private static int subsecond;
    public static final int interval = 500;
    Thread ClockThread;

    GlobalClock(int timeStep){
        delta = timeStep;

        time_base = 0;
        hour = 0;
        minute = 0;
        second = 0;
        subsecond = 0;
        ClockThread = new Thread(this);
        // (new Thread(this)).start();
        ClockThread.setPriority(ClockThread.MIN_PRIORITY);
        ClockThread.start();
    }
}

```

```

    }
    GlobalClock(){};
    public void startClock(){
        ClockThread = new Thread(this);
        ClockThread.start();
    }
    public void pause_clock(){
        ClockThread.suspend();
    }

    public void resume_clock(){
        ClockThread.resume();
    }
    public long convertTime(int hour, int minute, int second) {
        if(hour < 0 || minute < 0 || second < 0)
        {
            System.out.println("Wrong time format");
            System.exit(1);
        }

        return (hour*3600 + minute * 60 +second * 60);
    }
    public long getTime() {
        return (this.time_base);
    }

    public int checkTime(long time) {
        if(time_base > time)
            return (1);
        if(time_base == time)
            return (0);
        if(time_base < time)
            return (-1);

        return (100);
    }

    public void printTime(Graphics g, int x, int y){
        g.drawString("TIME: "+ hour+"." +minute+"." +second, x, y);
    }

    public synchronized void run() {
        while(true) {
            // System.out.println("Clock Advanced "+time_base);
            time_base = time_base+delta;
            /*
            subsecond +delta;
            if(subsecond >= 10){
                second++;
                subsecond -= 10;
            }
            */
            second += delta;
            if(second >= 60){
                minute++;
                second -= 60;
            }
            if(minute >= 60){
                hour++;
                minute -= 60;
            }

            if(hour >=24)
                System.exit(0);

            try{Thread.sleep(interval);} catch (Exception e) {};
        }
    }

```

```

    }
}
}

import java.awt.*;
import java.util.*;

public final class GroundMonitor
{
    public static NetPath getNetPathObject(String path_name)
    {
        NetPath tempP;
        for(int i = 0; i < AllNetPaths.size(); i++)
        {
            tempP =(NetPath) AllNetPaths.elementAt(i);
            if(tempP.name.equals(path_name)){
                return tempP;
            }
        }
        System.out.println("wrong path_name in getNetPathObject");
        return null;
    }
    boolean DATA_INITIALIZED = false;
    static Vector queues = new Vector(30);
    public static Vector AllNetPaths = new Vector();

    public static void createAllNetPaths()
    {
        System.out.println("Start createAllNetPath in Ground Monitor");
        NetPath tempPath = new NetPath("16_R_TO_C_3");
        tempPath.addLinksAt("16_R", 0);
        tempPath.addLinksAt("C_3", 1);
        tempPath.addLinksAt("B_2", 2);
        tempPath.addLinksAt("TERMINAL_B_2", 3);
        AllNetPaths.addElement(tempPath);
        tempPath = null;

        tempPath = new NetPath("16_R_TO_C_4");
        tempPath.addLinksAt("16_R", 0);
        tempPath.addLinksAt("C_4", 1);
        tempPath.addLinksAt("B_7", 2);
        tempPath.addLinksAt("TERMINAL_B_7", 3);

        AllNetPaths.addElement(tempPath);
        tempPath = null;

        tempPath = new NetPath("16_R_TO_C_5");
        tempPath.addLinksAt("16_R", 0);
        tempPath.addLinksAt("C_5", 1);
        tempPath.addLinksAt("B_9", 2);
        tempPath.addLinksAt("TERMINAL_B_9", 3);

        AllNetPaths.addElement(tempPath);
        tempPath = null;

        tempPath = new NetPath("16_R_TO_C_6");
        tempPath.addLinksAt("16_R", 0);
        tempPath.addLinksAt("C_6", 1);
        tempPath.addLinksAt("B_10", 2);
        tempPath.addLinksAt("TERMINAL_B_10", 3);

        AllNetPaths.addElement(tempPath);
        tempPath = null;

        tempPath = new NetPath("16_R_TO_C_7");
        tempPath.addLinksAt("16_R", 0);
        tempPath.addLinksAt("C_7", 1);
        tempPath.addLinksAt("B_10", 2);
        AllNetPaths.addElement(tempPath);
        tempPath.addLinksAt("TERMINAL_B_10", 3);

        tempPath = null;
    }
}

```

```

tempPath = new NetPath("16_R_TO_C_8");
tempPath.addLinksAt("16_R", 0);
tempPath.addLinksAt("C_8", 1);
tempPath.addLinksAt("B_11", 2);
tempPath.addLinksAt("TERMINAL_B_11", 3);
AllNetPaths.addElement(tempPath);
tempPath = null;

tempPath = new NetPath("16_L_TAKEOFF");
tempPath.addLinksAt("16_L", 0);
tempPath.addLinksAt("TERMINAL_34_R", 1);
AllNetPaths.addElement(tempPath);
tempPath = null;

System.out.println("end create AllNetPath in Ground Monitor");
}

// get next airplanequeue object given the path name and the
// current queue name
public static synchronized AirplaneQueue getNextAirplaneQueueObject
(String path_name, String current_queue)
{
    NetPath tempP = getNetPathObject(path_name);

    if(tempP != null)
        return tempP.getNextLink(current_queue);
    System.out.println("wrong path_name in getNextAirplaneQueueObject");
    return null;
}

//check wether an airplanequeue is empty
public static boolean airplaneQueuesEmpty(String queue_name)
{
    if(queueHasAirplane(queue_name) > 0)
        return false;
    else
        return true;
}

//get an airplanequeue object given airplanequeue name
public static synchronized AirplaneQueue getAirplaneQueueObject(String queue_name)
{
    AirplaneQueue tempQ;
    for(int i = 0; i < queues.size(); i++)
    {
        tempQ =(AirplaneQueue) queues.elementAt(i);
        if(tempQ.name.equals(queue_name)){
            // System.out.println("GetAirplaneQueueName:" + tempQ.name);
            return (AirplaneQueue) tempQ;
        }
        // System.out.println("AirplaneQueueName:" + tempQ.name + "target name" +
queue_name);
    }
    System.out.println("wrong queue_name in getAirplaneQueueObject");
    return null;
}

//get the first airplane in a given airplane queue
public Airplane theFirstAirplane(String queue_name)
{
    AirplaneQueue tempQ;
    for(int i = 0; i < queues.size(); i++)
    {
        tempQ =(AirplaneQueue) queues.elementAt(i);

```

```

        if(tempQ.name == queue_name){
            return (Airplane) tempQ.queue.firstElement();
        }
    }
    System.out.println("wrong AirplaneQueue name; program terminated 4");
    return null;
}

//get the last airplane from the given airplane queue
//without remove the airplane
public Airplane theLastAirplane(String queue_name)
{
    AirplaneQueue tempQ;
    for(int i = 0; i < queues.size(); i++)
    {
        tempQ =(AirplaneQueue) queues.elementAt(i);
        if(tempQ.name == queue_name){
            return (Airplane) tempQ.queue.lastElement();
        }
    }
    System.out.println("wrong AirplaneQueue name; program terminated 1");
    return null;
}

//check if the airplane queue has airplane
public static synchronized int queueHasAirplane(String queue_name)
{
    AirplaneQueue tempQ;
    for(int i = 0; i < queues.size(); i++)
    {
        tempQ =(AirplaneQueue) queues.elementAt(i);
        if(tempQ.name.equals(queue_name)){
            return tempQ.queue.size();
        }
    }
    System.out.println("wrong AirplaneQueue name; program terminated 2");
    return (0);
}

// create all of the airplanequeues
public static void createAllQueues(){
    System.out.println("Start creating ground links");
    queues.addElement(new AirplaneQueue("16_R", new Point(85,223), new Point(439,223)));
    queues.addElement(new AirplaneQueue("16_L", new Point(85,195), new Point(514,195)));
    queues.addElement(new AirplaneQueue("A_1", new Point(85,177), new Point(85,162)));
    queues.addElement(new AirplaneQueue("A_2", new Point(85,162), new Point(236,162)));

    queues.addElement(new AirplaneQueue("C_1", new Point(85,223), new Point(85,195)));
    queues.addElement(new AirplaneQueue("C_2", new Point(85,223), new Point(439,223)));//
not implemented
    queues.addElement(new AirplaneQueue("C_3", new Point(171,223), new Point(208,195)));
    queues.addElement(new AirplaneQueue("C_4", new Point(270,223), new Point(341,195)));
    queues.addElement(new AirplaneQueue("C_5", new Point(300,223), new Point(364,195)));
    queues.addElement(new AirplaneQueue("C_6", new Point(375,223), new Point(418,195)));
    queues.addElement(new AirplaneQueue("C_7", new Point(418,223), new Point(418,195)));
    queues.addElement(new AirplaneQueue("C_8", new Point(439,223), new Point(439,195)));
    queues.addElement(new AirplaneQueue("B_1", new Point(85,195), new Point(85,177)));
    queues.addElement(new AirplaneQueue("B_2", new Point(208,195), new Point(236,177)));
    queues.addElement(new AirplaneQueue("B_3", new Point(85,223), new Point(439,223)));//
not implemented
not implemented
    queues.addElement(new AirplaneQueue("B_4", new Point(85,223), new Point(439,223)));//
not implemented
    queues.addElement(new AirplaneQueue("B_5", new Point(85,223), new Point(439,223)));//
    queues.addElement(new AirplaneQueue("B_6", new Point(85,223), new Point(439,223)));//
    queues.addElement(new AirplaneQueue("B_7", new Point(341,195), new Point(330,177)));
    queues.addElement(new AirplaneQueue("B_9", new Point(364,195), new Point(375,177)));
    queues.addElement(new AirplaneQueue("B_10", new Point(418,195), new Point(418,177)));
    queues.addElement(new AirplaneQueue("B_11", new Point(439,195), new Point(439,177)));
    queues.addElement(new AirplaneQueue("TERMINAL_B_2", new Point(236,177), new
Point(236,177)));
    queues.addElement(new AirplaneQueue("TERMINAL_B_7", new Point(330,177), new
Point(330,177)));
}

```

```

        queues.addElement(new AirplaneQueue("TERMINAL_B_9", new Point(375,177), new
Point(375,177)));
        queues.addElement(new AirplaneQueue("TERMINAL_B_10", new Point(418,177), new
Point(418,177)));
        queues.addElement(new AirplaneQueue("TERMINAL_B_11", new Point(439,177), new
Point(439,177)));
        queues.addElement(new AirplaneQueue("TERMINAL_34_R", new Point(514,195), new
Point(514,195)));
        queues.addElement(new AirplaneQueue("TERMINAL_A_2", new Point(236,162), new
Point(236,162)));

        System.out.println("finish creating ground links");
    }

    //add an airplane into the airplane queue
    public static boolean enterAirplaneIntoQueue(String queue_name, Airplane airplane)
    {
        AirplaneQueue tempQ;
        for(int i = 0; i < queues.size(); i++)
        {
            tempQ =(AirplaneQueue) queues.elementAt(i);
            if(tempQ.name.equals(queue_name)){
                tempQ.enterAirplane(airplane);
                return true;
            }
        }
        return false;
    }

    //remove an airplane out of a given airplane queue
    public static synchronized boolean moveAirplaneOutFromQueue(String queue_name, Airplane
airplane)
    {
        AirplaneQueue tempQ;
        for(int i = 0; i < queues.size(); i++){
            tempQ =(AirplaneQueue) queues.elementAt(i);
            if(tempQ.name.equals(queue_name)){
                tempQ.departAirplane(airplane);
                return true;
            }
        }
        return false;
    }
}

import java.util.*;
import java.awt.*;

class AirplaneQueue {
    AirplaneQueue(String QueueName, Point from, Point to)
    {
        this.name = new String(QueueName);
        From = from;
        To = to;
        TrafficSignal = new TrafficLight();
    }
    public void setFromPointToPoint(int from_x, int from_y, int to_x, int to_y)
    {
        From = new Point(from_x, from_y);
        To = new Point(to_x, to_y);
    }
    public Point To;
    public Point From;

    public TrafficLight TrafficSignal;
    Point from;

```

```

Point to;
Vector queue = new Vector(10);
String name;

AirplaneQueue(String QueueName)
{
    this.name = new String(QueueName);
}

public boolean equals(AirplaneQueue airplane_queue)
{
    if(this.name.equals(airplane_queue.name))
        return true;
    else
        return false;
}

public void unholdFirstAirplane()
{
    TrafficSignal.setGreen();
}

public void holdFirstAirplane()
{
    TrafficSignal.setRed();
}

public void enterAirplane(Airplane airplane)
{
    queue.addElement(airplane);
    //set airplane's current track for animation
    airplane.updateTrack();
}

public void departAirplane(Airplane airplane){
//    if(TrafficSignal.isGreen())
        queue.removeElement(airplane);
}

public void departAirplane()
//    if(TrafficSignal.isGreen())
        queue.removeElementAt(0);
}

class TrafficLight
{
    public TrafficLight()
    {
        signal = green;
    }
    public static final int green = 2;
    public static final int yellow = 1;
    public static final int red = 0;
    private int signal;

    public synchronized boolean isGreen()
    {
        return (signal == green? true:false);
    }
    public synchronized boolean isYellow()
    {
        return (signal == this.yellow? true:false);
    }
    public synchronized boolean isRed()
    {
        return (signal == red? true:false);
    }
    public synchronized void setGreen()
    {
        signal = this.green;
    }
}

```

```

    }
    public synchronized void setYellow()
    {
        signal = this.yellow;
    }

    public synchronized void setRed()
    {
        signal = this.red;
    }
}

class NetPath extends Object
{
    public AirplaneQueue getFirstAirplaneQueue()
    {
        if(!Links.isEmpty())
            return (AirplaneQueue) Links.firstElement();
        else
            return null;
    }
    public AirplaneQueue getLastQueueInPath()
    {
        if(!Links.isEmpty())
            return (AirplaneQueue) Links.lastElement();
        else
            return null;
    }
    public String name;
    Vector Links;
    int number_links;

    public NetPath(String name)
    {
        this.name = new String(name);
        Links = new Vector();
        number_links = 0;
    }

    public boolean isLastAirplaneQueue(AirplaneQueue airplanequeue)
    {
        if(airplanequeue == (AirplaneQueue) Links.lastElement())
            return true;
        else
            return false;
    }

    public boolean hasAirplaneQueue(String queue_name)
    {
        if(Links.contains(GroundMonitor.getAirplaneQueueObject(queue_name)))
            return true;
        else{
            System.out.println("wrong queue_name in hasAirplaneQueue");
            return false;
        }
    }

    public void addLinksAt(String queue_name, int position)
    {
        AirplaneQueue TempQ = GroundMonitor.getAirplaneQueueObject(queue_name);
        if(TempQ != null && !Links.contains(TempQ)){
            Links.insertElementAt(TempQ, position);
            number_links = Links.size();
        }
    }

    public AirplaneQueue getNextLink(String queue_name)
    {
        AirplaneQueue CurrentQ = GroundMonitor.getAirplaneQueueObject(queue_name);

```

```

        if(CurrentQ != null){
            for(int i = 0; i < Links.size(); i++)
                if(CurrentQ == (AirplaneQueue) Links.elementAt(i) && i < Links.size() - 1)
                    return (AirplaneQueue) Links.elementAt(i+1);
            else
                return null;
        }
        return null;
    }
}

import java.util.*;
import java.awt.*;

class AirplaneQueue {
    AirplaneQueue(String QueueName, Point from, Point to)
    {
        this.name = new String(QueueName);
        From = from;
        To = to;
        TrafficSignal = new TrafficLight();
    }
    public void setFromPointToPoint(int from_x, int from_y, int to_x, int to_y)
    {
        From = new Point(from_x, from_y);
        To = new Point(to_x, to_y);
    }
    public Point To;
    public Point From;

    public TrafficLight TrafficSignal;
    Point from;
    Point to;
    Vector queue = new Vector(10);
    String name;

    AirplaneQueue(String QueueName)
    {
        this.name = new String(QueueName);
    }

    public boolean equals(AirplaneQueue airplane_queue)
    {
        if(this.name.equals(airplane_queue.name))
            return true;
        else
            return false;
    }

    public void unholdFirstAirplane()
    {
        TrafficSignal.setGreen();
    }

    public void holdFirstAirplane()
    {
        TrafficSignal.setRed();
    }

    public void enterAirplane(Airplane airplane)
    {
        queue.addElement(airplane);
        //set airplane's current track for animation
        airplane.updateTrack();
    }

    public void departAirplane(Airplane airplane){
//        if(TrafficSignal.isGreen())
            queue.removeElement(airplane);
    }
}

```

```

    public void departAirplane()
    {
        // if(TrafficSignal.isGreen())
        queue.removeElementAt(0);
    }
}

class TrafficLight
{
    public TrafficLight()
    {
        signal = green;
    }
    public static final int green = 2;
    public static final int yellow = 1;
    public static final int red = 0;
    private int signal;

    public synchronized boolean isGreen()
    {
        return (signal == green? true:false);
    }
    public synchronized boolean isYellow()
    {
        return (signal == this.yellow? true:false);
    }
    public synchronized boolean isRed()
    {
        return (signal == red? true:false);
    }
    public synchronized void setGreen()
    {
        signal = this.green;
    }
    public synchronized void setYellow()
    {
        signal = this.yellow;
    }

    public synchronized void setRed()
    {
        signal = this.red;
    }
}

class NetPath extends Object
{
    public AirplaneQueue getFirstAirplaneQueue()
    {
        if(!Links.isEmpty())
            return (AirplaneQueue) Links.firstElement();
        else
            return null;
    }
    public AirplaneQueue getLastQueueInPath()
    {
        if(!Links.isEmpty())
            return (AirplaneQueue) Links.lastElement();
        else
            return null;
    }
    public String name;
    Vector Links;
    int number_links;

    public NetPath(String name)
    {
        this.name = new String(name);
        Links = new Vector();
    }
}

```

```

    number_links = 0;
}

public boolean isLastAirplaneQueue(AirplaneQueue airplanequeue)
{
    if(airplanequeue == (AirplaneQueue) Links.lastElement())
        return true;
    else
        return false;
}

public boolean hasAirplaneQueue(String queue_name)
{
    if(Links.contains(GroundMonitor.getAirplaneQueueObject(queue_name)))
        return true;
    else{
        System.out.println("wrong queue_name in hasAirplaneQueue");
        return false;
    }
}

public void addLinksAt(String queue_name, int position)
{
    AirplaneQueue TempQ = GroundMonitor.getAirplaneQueueObject(queue_name);
    if(TempQ != null && !Links.contains(TempQ)){
        Links.insertElementAt(TempQ, position);
        number_links = Links.size();
    }
}

public AirplaneQueue getNextLink(String queue_name)
{
    AirplaneQueue CurrentQ = GroundMonitor.getAirplaneQueueObject(queue_name);
    if(CurrentQ != null){
        for(int i = 0; i < Links.size(); i++)
            if(CurrentQ == (AirplaneQueue) Links.elementAt(i) && i < Links.size() - 1)
                return (AirplaneQueue) Links.elementAt(i+1);
        else
            return null;
    }

    return null;
}

}

import java.awt.*;
import java.util.*;

class LineObject
{
    LineObject(Point from, Point to)
    {
        Track = new Vector(30);
        setFromPointToPoint((new Point(from.x, from.y)),
            (new Point(to.x, to.y)));
    }
    public double tangent;
    public void setTrack(int time)
    {
        int step;

        if(time <=0)
            delta = length;
        else
            delta = (int)(length/time);
        if(To.y == From.y)
            step = 30;
        else
            step = 5;
    }
}

```

```

    if(From != null && To != null){
        for(int i = 0; i < step; i++){
            Point tempMidPoint= new Point(
                (int)((To.x - From.x)/step*i + From.x),
                (int)((To.y - From.y)/step*i + From.y));
            Track.addElement(tempMidPoint);
        }
        Track.addElement(To);
    }

}

/*
// horizontal line;
// this will also handle situation when From.equals(To)== true;
if(From.y == To.y)
{
    for(i = 0; i < 9; i++)
    {
        delta *= (To.x > From.x? 1:(-1));
        Track.addElement(new Point(From.x + delta*i, From.y));
    }
    // System.out.println("create track for: "+From.x+","
    // +From.y+"-> "+To.x+","+To.y);

    //last point is To
    Track.addElement(new Point(To.x, To.y));
    return;
}

//virtical line
if(From.x == To.x)
{
    for(i = 0; i < 9; i++)
    {
        delta *= (To.y > From.y? 1:(-1));
        Track.addElement(new Point(From.x, From.y + delta*i));
    }
    //last point is To.
    Track.addElement(new Point(To.x, To.y));
    return;
}

//has a tangent
tangent = (To.y - From.y)/(To.x - From.x);

for(i = 0; i < 9; i++)
{
    Track.addElement(new Point(From.x + (int)(delta*i/tangent),
        From.y + (int)(delta*i*tangent)));
}
//last point is To.
Track.addElement(new Point(To.x, To.y));
*/
return;

}

public void setFromPointToPoint(Point from, Point to)
{
    From = from;
    To = to;

    if(From == null || To == null){
        System.out.println("setFromPointToPoint in LineObject: null Points");
        System.exit(2);
    }

    length = (int) Math.sqrt((from.x - to.x)^2+(from.y - to.y)^2);
}

```

```

}
LineObject(int from_x, int from_y, int to_x, int to_y)
{
    Track = new Vector(30);
    setFromPointToPoint((new Point(from_x, from_y)),
        (new Point(to_x, to_y)));
}
public Vector Track;
public int delta;
public int length;
public Point To;
public Point From;
}

import java.util.*;

public final class TrafficParam
{
    public static String getCategoryByAircraft(String ac_name)
    {
        AirplaneDataSet tempData;

        for(int i = 0; i < MASTER_ACFT_DATA.size(); i++){
            tempData = (AirplaneDataSet) MASTER_ACFT_DATA.elementAt(i);
            if(tempData.name.equals(ac_name))
                return tempData.category;
        }

        return null;
    }

    public static void insertAircraftMasterData(String name, String category)
    {
        AirplaneDataSet tempData = new AirplaneDataSet(name, category);
        if(!MASTER_ACFT_DATA.contains(tempData))
            MASTER_ACFT_DATA.addElement(tempData);
    }

    public static Vector MASTER_ACFT_DATA;
    public static long getTaxiwayTravelTime(String taxiway_name, Airplane airplane)
    {
        if(airplane.RoutName.compareTo("ARRIVE") == 0)
        {
            for(int i = 0; i < TAXIWAY_NAME.size(); i++)
            {
                if(((String) TAXIWAY_NAME.elementAt(i)).equals(taxiway_name))
                    return ((Long) TAXIWAY_TRAVEL_TIME.elementAt(i)).longValue();
            }
        }

        System.out.println("AirplaneQueue name:"+taxiway_name);
        return -1;
    }

    public static void insertTaxiwayName(String taxiway_name, long travel_time)
    {
        if(!TAXIWAY_NAME.contains(taxiway_name)){
            TAXIWAY_NAME.addElement(new String(taxiway_name));
            TAXIWAY_TRAVEL_TIME.addElement(new Long(travel_time));
        }
    }

    public static Vector TAXIWAY_TRAVEL_TIME;
    public static Vector TAXIWAY_NAME;
    public static long getROT(String exit_name, Airplane airplane)
    {
        // System.out.println("RoutName: "+ airplane.RoutName);
        if(airplane.RoutName.compareTo("ARRIVE") == 0)
        {
            for(int i = 0; i < EXIT_NAME.size(); i++)
            {
                if(((String) EXIT_NAME.elementAt(i)).equals(exit_name))
                    return ((Long) ROT_BY_EXITS.elementAt(i)).longValue();
            }
        }
    }
}

```

```

    }
}

if(exit_name.compareTo("16_L") == 0
    &&
    airplane.RoutName.compareTo("ENROUT") == 0)
// hard coded for departure from 16_L
return (long) 70;

//wrong exit name or rout_name
return (-1);
}
public static void setAllTrafficParameters()
{
//insert exit name
insertExitName("C_3", 0.05, 35);
insertExitName("C_4", 0.20, 30);
insertExitName("C_5", 0.30, 38);
insertExitName("C_6", 0.30, 65);
insertExitName("C_7", 0.05, 70);
insertExitName("C_8", 0.10, 75);

//insert category name
insertAirplaneCategoryName("CATEGORY_A", 0.05);
insertAirplaneCategoryName("CATEGORY_B", 0.25);
insertAirplaneCategoryName("CATEGORY_C", 0.40);
insertAirplaneCategoryName("CATEGORY_D", 0.30);

//insert aircraft name
insertAircraftName("PA_38_112", 1.0);
insertAircraftName("CE_208", 1.0);
insertAircraftName("CE_402C", 4.0);
insertAircraftName("EMB_120", 2.0);
insertAircraftName("SA_227", 12.0);
insertAircraftName("BAe_31", 10.2);
insertAircraftName("DHC_7", 4.0);
insertAircraftName("CE_550", 4.1);
insertAircraftName("A_300_600", 1.4);
insertAircraftName("B_767_300", 7.8);
insertAircraftName("B_727_200", 19.0);
insertAircraftName("B_737_300", 12.4);
insertAircraftName("MD_83", 12.0);
insertAircraftName("B_747_200B", 3.0);
insertAircraftName("L_101P", 3.1);
insertAircraftName("DC_8_73", 3.0);

checkParameterIntegrity("AIRCRAFT");
checkParameterIntegrity("CATEGORY");

insertTaxiwayName("C_3", 15);
insertTaxiwayName("C_4", 15);
insertTaxiwayName("C_5", 15);
insertTaxiwayName("C_6", 16);
insertTaxiwayName("C_7", 20);
insertTaxiwayName("C_8", 20);
insertTaxiwayName("B_2", 20);
insertTaxiwayName("B_7", 20);
insertTaxiwayName("B_9", 20);
insertTaxiwayName("B_10", 20);
insertTaxiwayName("B_11", 20);

insertAircraftMasterData("PA_38_112", "CAT_B");
insertAircraftMasterData("CE_208", "CAT_B");
insertAircraftMasterData("CE_402C", "CAT_B");
insertAircraftMasterData("EMB_120", "CAT_B");
insertAircraftMasterData("SA_227", "CAT_B");
insertAircraftMasterData("BAe_31", "CAT_C");
insertAircraftMasterData("DHC_7", "CAT_B");

```

```

insertAircraftMasterData("CE_550", "CAT_B");
insertAircraftMasterData("A_300_600", "CAT_D");
insertAircraftMasterData("B_767_300", "CAT_D");
insertAircraftMasterData("B_727_200", "CAT_C");
insertAircraftMasterData("B_737_300", "CAT_C");
insertAircraftMasterData("MD_83", "CAT_C");
insertAircraftMasterData("B_747_200B", "CAT_D");
insertAircraftMasterData("L_101I", "CAT_D");
insertAircraftMasterData("DC_8_73", "CAT_D");
}
public static void insertAircraftName(String airplane_name, double percent)
{
    if(!AIRCRAFT_NAME.contains(airplane_name)){
        AIRCRAFT_NAME.addElement(new String(airplane_name));
        AIRCRAFT_MIX.addElement(new Double(percent));
    }
}

public static Vector AIRCRAFT_NAME;
public static Vector CATEGORY_MIX;
public static double doubleSum(Vector v)
{
    double sum = 0;

    for(Enumeration e = v.elements();
        e.hasMoreElements();
        sum +=((Number)e.nextElement()).doubleValue());
    // System.out.println("double_sum = "+sum);
    return sum;
}

public static Vector CATEGORY_NAME;
public static void insertAirplaneCategoryName(String cat_name, double percent)
{
    if(!CATEGORY_NAME.contains(cat_name)){
        CATEGORY_NAME.addElement(new String(cat_name));
        CATEGORY_MIX.addElement(new Double(percent));
    }
}

public static void insertExitName(String exit_name, double percent, long rot)
{
    if(!EXIT_NAME.contains(exit_name)){
        EXIT_NAME.addElement(new String(exit_name));
        EXIT_USAGE.addElement(new Double(percent));
        ROT_BY_EXITS.addElement(new Long(rot));
    }
}

public static Vector EXIT_NAME;
public static void checkParameterIntegrity(String type)
{
    // System.out.println("wrong exit usage total = " + doubleSum(EXIT_USAGE));
    double sum = doubleSum(EXIT_USAGE);
    if(Math.abs(sum - 1.00) > 0.001)
    {
        System.out.println("wrong exit usage total = " + doubleSum(EXIT_USAGE));
        System.exit(4);
    }

    if(type.compareTo("AIRCRAFT") == 0)
    if(doubleSum(AIRCRAFT_MIX) != 100){
        System.out.println("wrong aircraft mix total");
        System.exit(4);
    }

    if(type.compareTo("CATEGORY") == 0)
    if(doubleSum(CATEGORY_MIX) != 1.00){
        System.out.println("wrong category mix total");
        System.exit(4);
    }
}

```

```

    }
    public static void setExitUsage(int exit, double percentage)
    {
        if(exit < 0 || exit > 5)
            System.exit(3);
//    EXIT_USAGE[exit] = percentage;
    }
    public static int NUM_OF_CATEGORIES = 4;
    public static int NUM_OF_EXITS = 6;
    public static void initiateTrafficParaClass()
    {
        EXIT_NAME = new Vector();
        CATEGORY_NAME = new Vector();
        AIRCRAFT_NAME = new Vector();
        EXIT_USAGE = new Vector();
        AIRCRAFT_MIX = new Vector();
        CATEGORY_MIX = new Vector();
        ROT_BY_EXITS = new Vector();
        TAXIWAY_NAME = new Vector();
        TAXIWAY_TRAVEL_TIME = new Vector();
        MASTER_ACFT_DATA = new Vector();
    }

    public static Vector ROT_BY_EXITS;
    public static Vector EXIT_USAGE;
    public static Vector AIRCRAFT_MIX;
}

```

Vita

Mr. Caoyuan Zhong obtained his BS degree in Mechanical Engineering from South China University of Technology in 1983. He worked as an engineer in the field of transportation information management in Institute of Transportation Research of Guangdong, China from 1983 to 1990. He started his graduate study in transportation engineering in Virginia Polytechnic Institute and State University in 1990. He obtained his MS degree in 1992 and finished his dissertation in 1997. Mr. Zhong will continue his effort in applications of computer technology in transportation engineering and system management.