CHAPTER 4
OBJECT-ORIENTED DESIGN METHODS FOR SOFTWARE IMPLEMENTATION

4.1 **Development of an Interactive Network Optimization Environment**

Current incident-management and network optimization software usually operate on a stand-alone basis, mainly using non-recyclable procedural code, and usually having no in-built GUIs (graphical user-interfaces) for interactive decision making. Also, while dealing with a complex open-ended problem such as traffic management, it is not always possible to work with one fixed scenario or prescribe one set of decisions that can be used to optimally solve a particular transportation engineering problem. For example, the variation of user-behavior and time-dynamic variation in state variables should always be taken into consideration. The decision maker must have a means by which he/she can (by a process of trial-and-error, analytic or heuristic methods) converge to the "right" set of decisions that he/she must take to generate (predict) a favorable outcome. In most cases, the decision maker may not be interested in the internal mechanisms of the tool and will be guided mostly by a common-sense approach, (whether the optimization problem solution makes sense or not, for example). Hence, he or she should not be burdened with having to know the inner details of the programs being used. (The decision maker should work at a higher level of abstraction of the problem.) The decision maker should understand how algorithm TD-*k*SP works, not by having to look at the source code, but by seeing how it works in reality and how it can be beneficial in dynamic traffic routing. In many cases, the results of the algorithm should be used simply to corroborate intuitive (or semi-analytic) findings that were obtained by looking at various possible scenarios.

Keeping the ideas described above in mind, this section outlines the framework of an integrated transportation management software system that has the seamless capability of combining a wide range of analysis and optimization methods within an interactive environment. This can be accomplished using languages like Java and C++. Such languages are object oriented in nature and can be used to run remote applications. In our case, we are interested in dynamic network optimization applications such as TDSP or DTA. Such applications are ultimately useful in practice if they are provided with real-time data which is updated frequently. The size of such data is voluminous and require high-speed computer links. Future technology like asynchronous transfer mode (ATM) networks can potentially revolutionize such ITS applications.

The study of algorithms such as TDSP, DTA, etc., is not complete unless we can develop a framework within which such algorithms can be applied to solve practical problems. Traffic congestion and management is a complex problem which requires an entire array of analytical tools. Methods such as demand forecasting, system dynamics/simulation, traffic flow modeling and network optimization must all be integrated to create a seamless software system without affecting the overall structure of any pre-existing software elements. Object-oriented design can be an effective technique toward achieving this goal.

The life cycle of object-oriented software has four phases. They are analysis, design, evolution, and maintenance [Broadwater, 1995]. The analysis phase requires the following tasks to be accomplished.

1. Understand and model a real-world problem domain.
2. Identify the key objects in the problem domain.
3. Set up a client-server model. The objects must provide a set of services to clients, or request a set of services from servers, or both.

The design phase includes the following tasks.

1. Identification of subsystems in the domain, where each subsystem provides a set of services and has a well defined interface.
2. Decomposition of the system into subsystems.
3. Identification of high level objects in subsystem.
4. Decide on what kind of program is required - procedure driven, event driven, or concurrent (several independent programs control execution).
5. Conduct scenario case studies (the sequence of operations that are performed).

At this stage, the program is rarely so complete that coding becomes mechanical. Note that one would usually not be able to tell from design what language is to be used. The next phase is one of evolutionary development. The development of the software designed is continued. The salient features of this phase are shown below.

1. Application is implemented gradually through successive refinement (the prototype implements the basic functionality).

2. Development is performed in an iterative fashion.

3. Development is incremental because each iteration builds on previous iterations.

The final phase is that of maintenance and is a continuation of the evolutionary phase with hopefully a stable architecture. Maintenance and upgrading is much easier because of the ease of understanding the structure of the software. This is because the object oriented paradigm closely maps the conceptual model to the implementation model. The development of CASE (computer aided software engineering) tools in popular languages like C++ help in documentation apart from other features. The most common CASE tools that are used are the class library, the class browser, and the graphic designer.

The choice of when to use an object-oriented is still open to debate. A structured or functional approach emphasizes functionality, and does not stress the nature or structure of the data. The object oriented approach includes both functionality as well as data. If the data is likely to change, then using an object oriented approach is natural. However, if the functionality is likely to change, then the choice of the software modeling approach is open to question. At the point where the objects are identified, the functional approach is used within the implementation of each object. Before we present the object-oriented software model, we need to define some data structures that are necessary in our software design. In particular, the data structures used in algorithm TD-*k*SP will be presented.

4.2 Dynamic Data Structures and Shortest Path Classes

The key elements of any object-oriented software design is the use of structured, efficient data structures, referred to as objects. These objects are implemented in practice using *class* definitions. One advantage of using objects to model the problem domain is that they mirror the physical reality. The three main features of an object are listed below.

1. **Abstraction** - the ability to protect (encapsulate) data from tampering.
2. **Inheritance** - the ability to partly or fully emulate other classes.
3. **Polymorphism** - The ability to perform different tasks while retaining the same object structure (via virtual functions).

The basic building block of networks is the linked list. The linked-list class is a self-referential data structure. The size of the list class can be varied dynamically by introducing a new link, each time a new piece of data is attached to the list. Consider the construction of a network class. A network class contains node and arc classes. An arc class contains link-layer level data (depending on the actual ITS scenario). In particular they contain reference "keys". These keys act as pointers to locate the head and tail nodes of the arc. Initially, we create $|A|$ arc classes. Next, to transform the network data to the forward-star format, the following procedure has to be carried out.

1. Node coordinates are transformed from global (original) to local coordinates. This is done by scanning the array of arc classes and incrementing a node count if a node has a node number not encountered before. This can be carried out in $O(n^2)$ or $O(n \log n)$ time using any sorting/inserting technique.

2. Once we obtain the number of nodes, $|N|$ node classes are created. The arc array is scanned and the arc numbers are appropriately inserted in an "adjacency" list for nodes, so that each node class now contains keys to the arcs in its forward star. Thus a total of $|N| + 2\,|A|$ units of memory is needed to store network data in the forward star format.

The network class can also carry other data types like trees, queues and even other network classes if necessary. If a simulation analysis needs to be performed, vehicle classes can be created along with any other enhancements to node and arc classes. The implementation of the functions (behavior) for these classes must mirror the actual implementation in the real world, and the classes must also be constructed to facilitate such a model. A C++ implementation of these classes is presented toward the end of this section. The object-oriented analysis (OOA) and design (OOD) technique is one way of achieving this goal. This method is language independent and the focus is purely on "software modeling". The interactive network optimization environment is designed using this method. The main features of the OOD methodology is presented next.

4.3 **Object Oriented Analysis**

4.3.1 **Transforming the Problem Domain into a Conceptual Model**

For the present application, we have to route traffic for each time period from the origin node(s) in the network to the destination node(s) via the current shortest path(s) from the origin to the destination, such that an objective function for a given portion of the network is optimized. We have to perform all calculations necessary and monitor the network performance. Simulation models can also be built using suitable control laws. Also, we must be able to perform calculations for various scenarios to help decide on optimal traffic management decisions. Toward this end, the services of the following objects are available.

1. The Optimizer, which contains mathematical programming, network optimization and shortest path routines. In addition, we could also include an additional object for stochastic optimization models (queuing theory based models, for example).

2. The Path Finder, which performs optimal path calculations.

3. The Simulator/Controller, which maintains control, and simulation routines based on traffic flow and car-following equations.

4. The Rectifier, which provides error diagnostics on feasibility and optimality of decisions made by the user.

5. The Editor which includes the interface between the user and the workstation and helps construct various scenarios (for example a GIS software element).

Another object that can form an interface with the database uses open database connectivity (ODBC). Here, the data may reside in a computer that may be located elsewhere in the network (remote site). This database object would provide an interface for ODBC. Note that we are interested in building a shell for a proposed optimization workstation, rather than developing an entire software package from the scratch. The approach used here can use pre-existing software packages as a core for the shell.

The topology of the system to be analyzed is modeled as a network. In this particular case, roads in the network can be modeled as arcs and intersections can be modeled as nodes. Vehicles are created at origin nodes and are destroyed at destination nodes. They queue up on the road, wait till they reach the end of the arc, where they get information as to which of the roads emanating from that intersection should be taken. No overtaking is allowed (optional) and the vehicles must adhere to traffic flow equations.

Problem domain Objects Selected from the Network Description

By inspection, the following list of readily visible objects is created.
Network, Sub-Network, Route, Road, Intersection, Origin_Node, Destination_Node, Road emanating from an intersection, Vehicle, Shortest_Path_tree, and Queue. These objects describe the topology.

A second set of objects can also be identified as shown below.
Optimizer, Analyzer, Rectifier, Prompter, Editor. These objects act as servers to the client (network).

4.4 **Object Oriented Design**

4.4.1 **Software Design Layout**

We can look at the software design separately at three levels of complexity. At a macroscopic (design) level**,** we can see only the client (the network model) and a set of servers. The client's aim is to achieve an optimal network performance value (*Z*). All that the client knows is that a network has arcs and nodes, and an associated performance measure *Z*. Servers and clients communicate with each other via messages. At an intermediate (analysis) level, the servers, who know a bit more than the client about the problem look at the network more closely. Each server is fully responsible for its body and interface. At a microscopic (functional) level, the history of each vehicle, link, etc., is studied and a record is kept of what event takes place at every point in time (the bulk of the procedural code). The macro model is described first, followed by the analysis level model. We shall study the software design at the first two levels only.

4.4.2 **Message Map and Class Relationships at the Macro Level**

The cardinality of all class relationships is 1:1. Servers contact each other using a key in a global array of pointers (gap) to servers (gap_servers[*S*]) and another key for the service function to be used. Each server contains a set of service functions and we have similar global arrays of pointers for their service functions. At any point, each server can provide or obtain one service function. The message map diagram is shown in Figure x. The arrows drawn between the objects represent "object relationship" type Booch notations [Booch, 1994]. The dashed arrows in the figure indicate that the message is actually meant for object indicated. For example, the rectifier sends an "Infeasibility/Optimality " message to the network

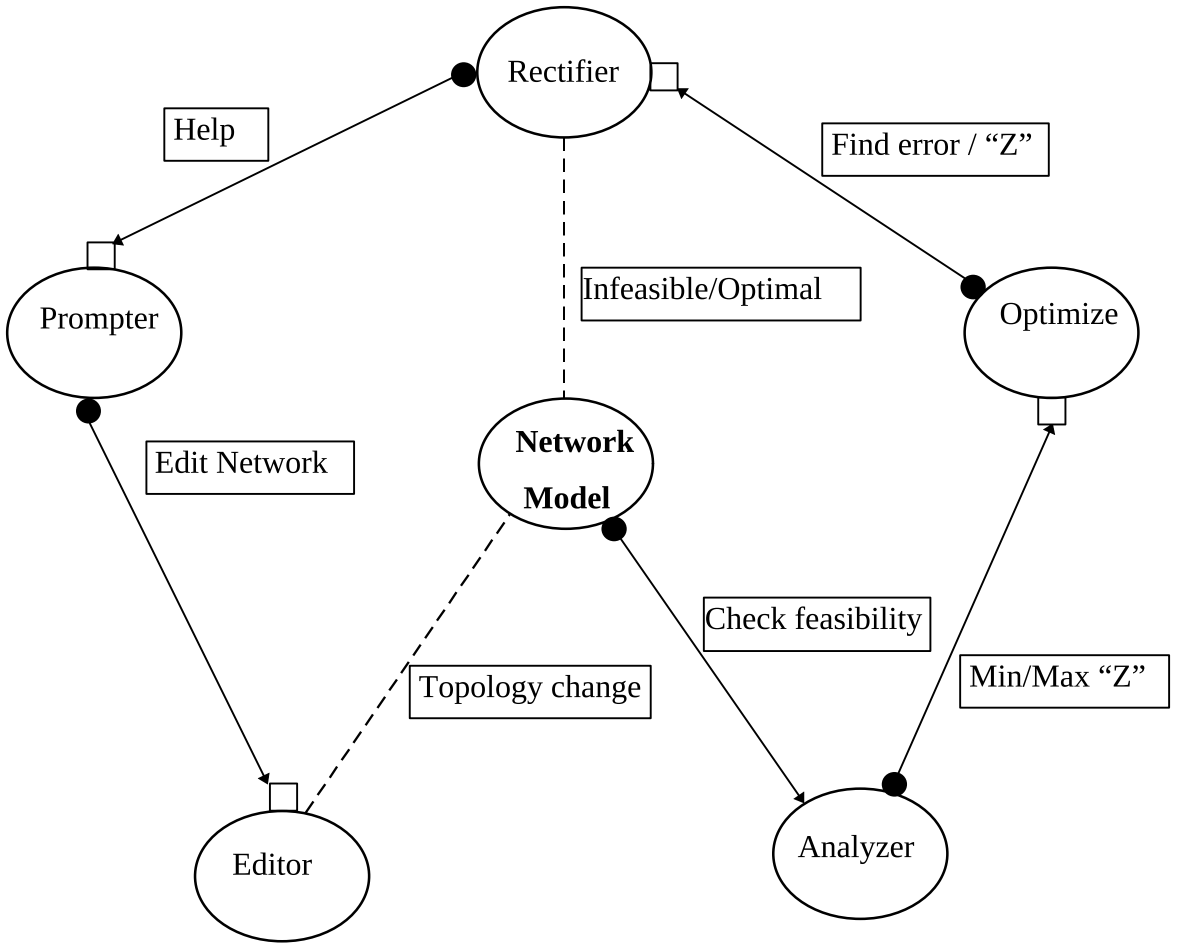


Figure 7. Message Map Diagram for the Client-Server Model.

4.4.3 **Class behaviors, responsibilities and definitions for design level model**

A generic network object has a set of arcs, nodes and network attributes. These can be stored in linked lists. They also include adjacency and connectivity data. The network object defines the topology of the model. The flow in the network is through these arcs and nodes. The topology can be changed only by the editor. Messages *originate* when a new scenario is created in the model and *terminated* when a decision is taken based on it (edit, stop, etc.) The efficiency of the network performance is assumed to be described by a measure of effectiveness (*Z*). Our aim is to bring the model to a state that optimizes *Z. A message is generated from the network model whenever a change in* *topology takes place*. The network object understands and responds to three kinds of messages (direct or relayed) from other servers as shown in Table 4.

Table 3. Messages Processed by the Network Object.

Incoming Message	**Outgoing Message**
Z is feasible (the scenario is acceptable).	Optimize *Z*.
Z is optimal (the scenario is the best possible).	Terminate - end task (send to itself).
The network has been edited (the scenario has changed).	Is *Z* feasible ?

Note that the messages received are not service messages, but rather, are instructions that were sent to the network, itself based on events and results. The network is connected only to the analyzer to which it provides data in its outgoing messages.

4.4.4 **Server Objects**

The server objects respond to messages that originate from a client or server. Each server can communicate with exactly one server or client. The contents of these messages (usually a reference to a network type object) may get modified when it reaches a server. The servers have a set of functions that perform calculations and pass a message on to the next server. Each server performs a special task.

The Analyzer

This object has a set of functions that operate on the topology of the network and check if the design is feasible or not. The message it can respond to is: Is the design feasible ? It can send the following messages after performing analysis tasks.

1. If the design is infeasible, locate the point of infeasibility in the network.
2. If the design is feasible for the current setting, calculate *Z*.

The Optimizer

This object has a set of functions that perform various optimization procedures for a given *feasible* network setting (if the client has received an "On feasible" message). It responds only to optimization queries. For example, it finds optimal flow settings and sends a (trivial) "find *Z*" message.

The Rectifier

The Rectifier has a set of diagnostic functions that help it pinpoint infeasible arcs or nodes that cause infeasibility. If it gets a "find *Z*" message, it calculates *Z*; otherwise it routes the diagnostics to the prompter. If the Rectifier is unable to arrive at an analytical solution to move toward feasibility, it sends a help message to the prompter, along with the closest answer it could find (perhaps in the form of a subnetwork).

The Prompter

The prompter has a set of string functions that provide qualitative (or even semi-quantitative) help to the designer. It can process only help messages. It has a set of logical "if-then-else" rules that provide passive help to the user. These rules operate on the local network topology and flows and decides on what "edit" or "display" messages to send.

The Editor

This server contains editing functions like add, insert, delete or modify arcs/nodes in the network, that help in scenario changes. This is the object that *interactively* executes the instructions it receives from either the rectifier or the prompter. This object passes a "topology change" message to the network.

4.4.5 **The Service_Message (SM) object**

It is a message created whenever a change in network topology takes place. The message is assigned a server, along with contents of the instructions for that server. We can thus see that at a macro level, the model is generic, i.e., it can be applied to any network optimization problem. Here, we consider the special case of traffic routing, and shall later discuss what happens to our software design when we solve a building design optimization problem, for example. The structure of the classes needed at the design level is shown in Table 4. Note the complete absence of any reference to the type of optimization performed.

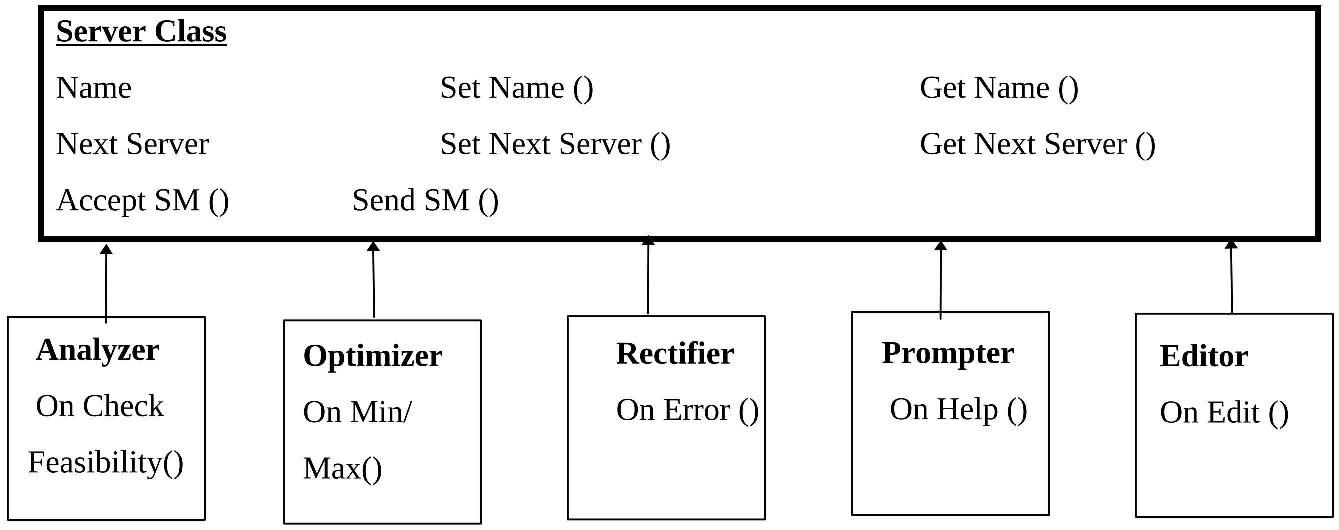
Table 4. Class Structure for the Network, Message and Server Classes

Network	**Service_Message**
Node, Arc	Server
Next Server	Service
Z	Set Server()
Set Node, Arc No()	Set Service()
GetNode, Arc No()	Get Server ()
Set *Z* (), Get *Z* ()	Get Service()
Send, Create SM ()	SM()
Set Next Server ()	~SM()
Get Next Server ()	
OnNetChange()	
OnFeasible()	
OnOptimal()	

Analyzer	**Optimizer**	**Rectifier**
Name	Name	Name
Next Server	Next Server	Next Server
Set Name ()	Set Name ()	Set Name ()
Get Name ()	Get Name ()	Get Name ()
Set Next Server ()	Set Next Server ()	Set Next Server ()
Get Next Server ()	Get Next Server ()	Get Next Server ()
Send SM ()	Send SM ()	Send SM ()
Accept SM()	Accept SM()	Accept SM()
OnCheck Feas ()	On Min Max ()	On Error ()
		On Find *Z* ()

Table 4. Cont'd

Prompter	**Editor**
Name	Name
Next Server	Next Server
Set Name ()	Set Name ()
Get Name ()	Get Name ()
Get Next Server()	Get Next Server()
Send SM ()	Send SM()
Accept SM()	Accept SM()
On Help ()	On Edit ()



Class Scenario Diagram

Consider the following sequence of instructions carried out by the workstation. "On analysis, a feasible decision is detected. The Optimizer calculates optimal settings. The Rectifier finds the optimal *Z*". No further change in topology is allowed for the network. This is achieved via two message cycles in the model as shown in the Figure 6.

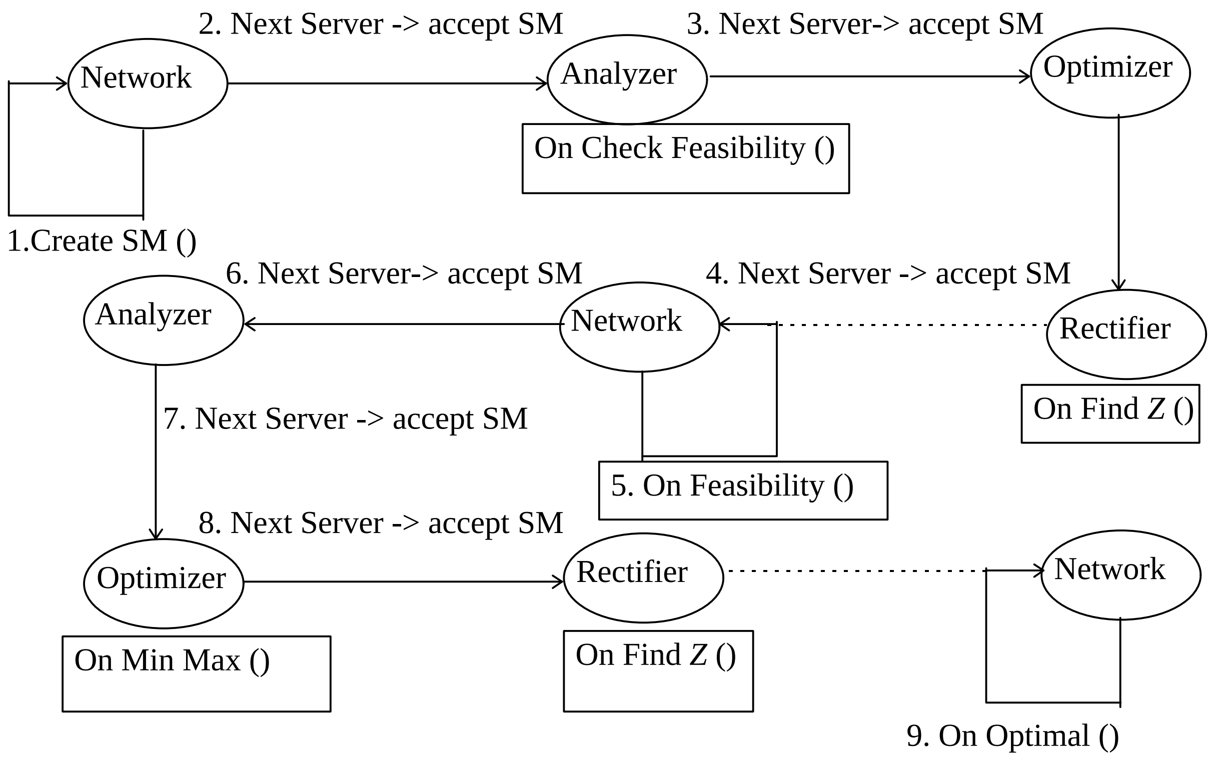


Figure 8. A Class Scenario Diagram that Illustrates Message Passing.

4.4.6 Class Behaviors at the Analysis Level

At the analysis level, the main calculations are performed by the Analyzer and Optimizer classes (objects). Here, the Optimizer acts as a server for the Analyzer (client). Calculation of system cost (*Z*) is a simple example. If we are considering a routing problem, the sum of all travel (flow) costs for all vehicles is calculated at their destinations (for the entire time period under study). This sum has to be minimized. The problem of minimizing *Z* is tackled by the Optimizer. Usually, we have typical network constraints.

1. Flow on any arc should not exceed the capacity.
2. Flows should follow Kirchoff's laws (conservation of flow).

The user can add other rules, depending on the problem. For example, the user can stipulate that the system cost imposed by vehicles on any link equals the waiting time in queue plus the service time. Sometimes, probabilistic or simulation analysis capabilities must be used to analyze problems having stochastic data or low tractability. An example for a simulation analysis is shown below. Assume that we are simulating for a total time *T* (sub-periods t_1, t_2,..t_n). For each sub-period, the following calculations are performed.

1. Find shortest path trees using costs from the current and previous sub-periods (an optimization module).

2. Route traffic along trees (use a queuing model and a distributed/centralized implementation for information dispersal in the network).

3. Check for feasibility (check flow rules and measure costs).

4. Compute costs and *Z*, if the network state is feasible. Otherwise, detect source of error.

5. If a feasible network state is achieved, calculate the optimal *Z* for the current scenario.

4.4.7 **Class Definitions and Responsibilities for a Simulation/Optimization Problem**

Tree - An object that is rooted at an origin and spans all nodes, connecting each destination to the origin via an unique route. For example, we consider shortest cost path trees, which may keep changing with time.

Arc - This object consists of a head node and a tail node, which specify its location. It has a vehicle capacity and a queue. Capacity is limited. Arcs are responsible for inserting a vehicle in its queue after checking the capacity.

Node - A junction of arcs. There are three types of nodes. They are origin nodes, destination nodes and transfer nodes. Nodes regulate vehicles along arcs, according to the destination of the vehicle. It sends a message if there is any infeasibility.

Origin Nodes - These nodes create vehicles in time according to some given rules and store this information in the vehicle.

Destination nodes - These nodes destroy a vehicle if it is that vehicle's destination and store the travel cost, travel time, etc. and other data accumulated for that vehicle.

Transfer nodes - These nodes simply remove vehicles from one arc and transfer them to another arc and update travel costs for vehicles. Note that, depending on the actual ATIS scenario and the actual means of information transfer, the responsibilities for these nodes can be modified to realistically model the physical reality (the nature of the transfer protocol).

Vehicle - This originates at an origin node. It is assigned a destination node that it reaches (in an optimal manner) with the help of information from nodes it encounters. It carries travel cost information, that gets updated at every node it encounters. It is destroyed at the destination node. If a vehicle is unable to reach its destination before a threshold time, it delivers that information to the node where the bottleneck was encountered.

Queue - There is one for every arc. Delay/cost due to a vehicle entering a queue depends on the current queue length, capacity, etc. Vehicles enter at the end of a queue, exit from the front and follow FIFO (or non-FIFO) rules. It stores all functions needed to perform queuing calculations/operations. Table 6 shows a class prototype for actual (minimum) responsibility requirements for each object.

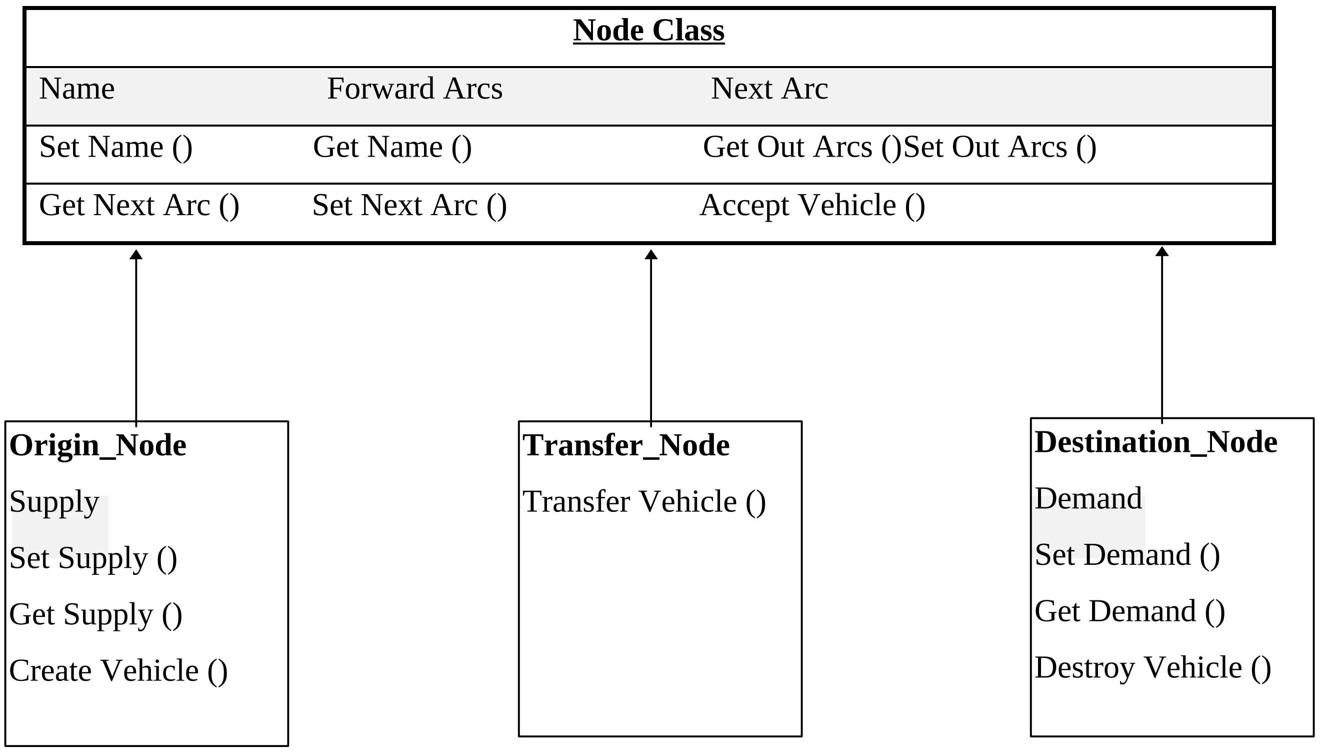
Keys - Nodes refer to arcs via arc numbers. Each origin node has exactly one tree. Destination nodes refer to vehicles via their destination number. Arcs and trees refer to nodes via node numbers. Queues use vehicles and do not use keys to refer to them. There is one queue for every arc. The class and inheritance structure and for these objects are shown in Table 5.

Table 5. Class Structure for the Vehicle Routing Analysis Problem.

Vehicle	**Origin Node**	**Dest Node**	**Transfer Node**
Origin	Name	Name	Name
Destination	Forward Arcs	Forward Arcs	Forward Arcs
Origin Time	Next Arc	Next Arc	Next Arc
Dest Time	Supply	Demand	Set Name ()
Travel Time	Set Name ()	Set Name ()	Set Out Arcs ()
Service Time	Set Out Arcs ()	Set Out Arcs ()	Set Next Arc ()
Set/Get Origin ()	Set Next Arc ()	Set Next Arc ()	Get Name ()
Set/Get Dest ()	Set Supply ()	Set Demand ()	Get Out Arcs ()
Set Origin Time ()	Get Name ()	Get Name ()	Get Next Arc ()
Set Dest Time ()	Get Out Arcs ()	Get Out Arcs ()	Accept Vehicle ()
Get Origin Time ()	Get Next Arc ()	Get Next Arc ()	Transfer Vehicle ()
Get Dest Time ()	Get Supply ()	Get Demand ()	
Set Travel Time ()	Accept Vehicle ()	Accept Vehicle ()	
Get Travel Time ()	Create Vehicle ()	Destroy Vehicle ()	
Set Service Time ()	Build Tree ()		
Get Service Time ()	Destroy Tree ()		
Vehicle (),~Vehicle ()			

Table 5. Cont'd.

Arc	**Queue**	**Tree**
Name	Member	Root
Head	Length	Children
Tail	Av Wait Time	Parent
Queue	Max Length	Set root ()
capacity	Get position ()	Set children ()
Set/Get Name ()	Set position ()	Set parent ()
Set/Get head ()	Enque, Deque Vehicle ()	Get root ()
Set/Get tail ()	Set/Get Length ()	Get children ()
Set/Get Capacity ()	Set/Get Av Time ()	Get parent ()
Accept Vehicle()	Set/Get Max Length ()	Tree ()
Create Queue()	Get Head /Tail ()	~Tree ()
Destroy Queue ()	Queue (), ~Queue ()	



Class Scenario Diagram

"Given a single time period, a single origin, transfer node, destination, route a single vehicle to its destination". If flow exceeds queue capacity, the non-feasibility of the problem is detected.

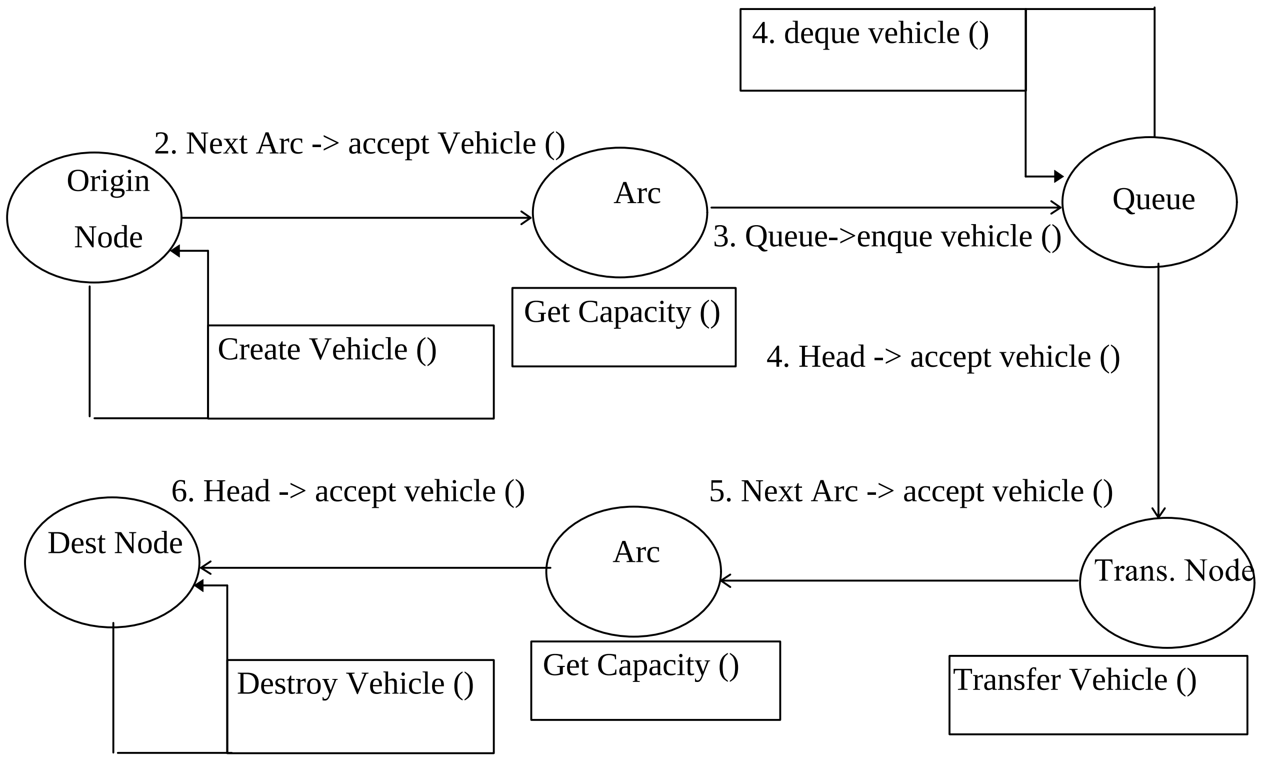

4. deque vehicle ()
2. Next Arc -> accept Vehicle ()
Origin Node
Arc
Queue
3. Queue->enque vehicle ()
Get Capacity ()
Create Vehicle ()
4. Head -> accept vehicle ()
6. Head -> accept vehicle ()
5. Next Arc -> accept vehicle ()
Dest Node
Arc
Trans. Node
Destroy Vehicle ()
Get Capacity ()
Transfer Vehicle ()

Figure 9. A Class Scenario Diagram for a Vehicle Routing Problem.

4.5 C++ Class Implementations

Finally, we shall now see some important features of the network, arc, server classes using an implementation of the C++ language. Terms prefixed by * represent "pointers" to the object (or variable) they represent.

```
Server *gap_servers; Analysis *gap_analysis ;
```

```
 // gap_analysis [A], gap_optimization [O], gap_rectification [R], etc.

network {
 arc * m_pArcList; node * m_pNodeList;      // embedded arcs and nodes
 double m_Z ; char * m_name;                            // network attributes
 long S ;                                                // server index
 long A;                                                 // analysis index
          -------                                        // other variables / functions
          }

 arc {
                  int m_arc_no;                   // doubly linked arc list
          int m_nhead_no, m_ntail_no ;           // node index
          queue *m_vehicle_que;                   // embedded queue
 arc *m_next, *m_previous;
 -------                                 //functions and other variables
 }                                       // the class structure for a node is similar
```

For any generic server X-er (where X could be any specific server type), we can define the following class. This generic class serves as a template or a “base class” from which all specific server classes can “inherit” properties and functions.

```
X_er: public Server{
                  // a server inherits from this generic server class
          .---------                      //server functions
 long S ;                        // index values for various servers
 long A, R,...                   // index values for functions from other servers
                  }
```

4.6 **Changing Scenarios, Analysis, and Client Needs.**

If we want conduct an analysis that maximizes Z instead of minimizing it, we simply have to set the function index corresponding to the maximization routine. If we want to only analyze a problem and are not interested in an iterative decision making process, for example, we can stop after we exit from the analyzer. We can also intervene and edit the network whenever we want to, via the interactive editor. We can add new functions, algorithms, etc. in the corresponding servers, and do not have to change anything else in other servers, or the client network. We can also use the model for simply performing optimization calculations. Hence, the model can be used in whole or in part. The model is such that developers can work on different parts of the software design independently, and come up with servers, functions, etc., independent of others. Some additional features are given below.

1. Adding /removing a server.

Suppose that we want to perform econometric calculations, such as benefit-cost analysis, we can create a new class derived from the base server class, and then add within this whatever functions we desire. Removing servers is trivial.

2. Solving a different problem.

This is a significant feature of the model. Suppose that we want to design a building structure such that the weight of structure is a minimum. Then we can specify Z equal to the sum of the weight of the beams and columns in the building. We can model beams and models as arcs, etc. Here, we have to change some of the analysis functions and specific node /arc attributes. Again, the overall model is left unchanged, as <u>the basic model is independent of the analysis being performed.</u> In general, any network model (another example would be an electrical circuit) can be designed using this model.

3. Developing an expert system shell.

The prompter can be made more powerful by converting it or linking it to a knowledge based expert system shell, which can be improved with time. For complex applications and designs, a neural network could be used.

4. The rectifier could be developed into an error analysis tool by adding more functions and algorithms.

4.7 **Integration of Shortest Path Algorithms within WAIMSS**

The Center for Transportation Research at Virginia Tech has been developing a Wide-Area Incident Management Support System (WAIMSS) for the past four years (1994-1997). The core element of this system is a Geographical Information System (GIS) software package ARC/INFO. The secondary modules use ARC/INFO to obtain network data and display graphical results. ARC/INFO is an excellent candidate for acting as the editor object of the interactive environment. Further, one can use programming languages such as C++ and *JAVA* to generate messages and create the interface and network classes. Optimization software packages such as CPLEX, simulation software such as INTEGRATION, and transportation engineering software such as HCS can be used to develop the server classes. WAIMSS already has an expert system shell (NEXPERT) that can be used as a prompter class to generate heuristics.

As a beginning, algorithm TD-*k*SP was integrated into WAIMSS using a C programming “bridge”. This was done using the set of C functions available in ARC/INFO (*infolib.c*) as a part of an AML (*Arc Macro Language*) module. The user can specify the origin and destination nodes, the value of *k*, etc., via pop-up menus. The user can also specify a subset of the network that he or she wants to consider. This data is written to a file. Algorithm TD-*k*SP reads this data, performs node-renumbering, calculates the desired shortest paths, and finally writes out the (original) numbers of the links in the shortest path(s) to a file. WAIMSS then reads these link numbers from the file and changes the color attributes of these links to finally display these *k* shortest paths on the screen.

In another test, the user can also input the location of an incident that reduces the capacity of a particular stretch or freeway or road. The algorithm sets the link delay for this path to infinity for the period of closure and calculates multiple diversion routes using *k*-shortest paths. WAIMSS is used to display these dynamic diversion routes. It must be mentioned here that loopless versions *k*-SP algorithms are essential if they are to be used in practice. This is because in a few cases, most of the *k*-shortest paths generated were practically useless since they differed only by small circuits consisting of dangling links and dead-ends. This is because of the nature of the digitized coverage maps that are available. Therefore, any algorithm that uses such maps needs to have automated pre-processing features such as link and node elimination routines. Another useful method would be to use partially of fully disjoint paths [Sherali et al., 1996] for static and dynamic routing. Future applications of algorithm TD-*k*SP with WAIMSS involve the display of routes for emergency vehicles from their depots to the incident sites.

Until now, we have studied the problem formulation and algorithmic procedures for algorithm TD-*k*SP. We then discussed practical software implementation details, programming considerations, and the integration of the algorithm into an object-oriented, interactive network-optimization environment. If this algorithm is to be used for real-time implementation, then it is essential that it be computationally efficient in terms of speed and storage requirements. The next chapter describes the computational performance tests conducted followed by a regression analysis that can be used to predict the computational speed of Algorithm TD-*k*SP.