INFORMATION TO USERS

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

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

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

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

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

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

Extendable Design of a Cellular Network Simulator

Bamdad Ghafari

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June 2001

Abstract

Extendable Design of a Cellular Network Simulator

Bamdad Ghafari

Cellular Network Simulator (CNS) is a stand-alone application, intended to study the behavior and compare the performance of frequency channel assignment and handoff algorithms in different types of cellular networks. A key requirement of such a simulator is that it should be easy to implement and integrate new types of cells or networks, methods for mobility, and algorithms for channel assignment and handoff. Thus the design of CNS was required to be extendable, and particularly in the specific ways mentioned above.

CNS was designed and implemented by using object-oriented design and programming techniques. The design of CNS utilized both *Unified Modeling Language (UML)* and design patterns, while it was implemented in Java. CNS provides a convenient graphical user interface as well as a command line interface. CNS was designed and implemented in a manner to completely separate its GUI from the rest of the application. This enables CNS to run without need of any human intervention when this feature is needed. CNS is a multi-platform application that could run on any standard platform (Unix, Windows, and DOS). CNS also provides a record and playback mechanism in order to save and re-run a simulation. An in-depth description of the basic technology used in the design and implementation of CNS is provided in this report.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to my supervisor, professor Lata Narayanan, for her support throughout my studies at Concordia University. Without her extraordinary guidance, this work would not have been done.

My special thanks are also due to Asmaa Alsumait. She joined me later in the project and made a big contribution to the project, by implementing methods and algorithms for frequency channel assignment, handoff, heap structure, mobile movement and also by helping me during the integration phase. Finally, thanks to all the professors and staff in the department of computer science, especially Halina Monkiewicz, for their excellent support during my studies.

Table of Contents

1 INTRODUCTION 1

1.1 Introduction to Cellular Networks 1

1.2 Frequency Channel Assignment Problem 3

1.3 The Motivation for CNS 3

1.4 Related Work 4

2 SYSTEM REQUIREMENTS 8

2.1 General Requirements 8

2.2 Hardware Requirements 10

2.3 Software Requirements 10

3 SYSTEM ANALYSIS AND DESIGN 11

3.1 Design Patterns 11

3.2 Use Cases 13

3.3 Preliminary Analysis 14

3.4 Use of Design Patterns 15

3.5 System Design 17

3.6 Detailed Class Diagrams 21

3.7 Sequence Diagrams 35

4 INTERFACE DESIGN 39

4.1 Graphical User Interface 39

4.2 Command Line Interface 42

5 CONCLUSIONS 43

5.1 Design Lessons 43

5.2 Future Enhancements 44

5.3 Extendability 44

APPENDIX A: Design Patterns 49

APPENDIX B: Command Line Interface 56

REFERENCES 57

1 Introduction

This report will describe the detailed design and implementation of a *Cellular Network Simulator (CNS).* CNS is a utility which simulates mobile traffic and management of calls arising from this traffic in a cellular network. CNS can be used by researchers and scientists who study the behavior of different types of cellular networks, and algorithms for frequency channel assignment and handoff. In this chapter, we give a brief introduction to cellular networks, channel assignment algorithms and the motivation to develop CNS, as well as review some related work.

1.1 Introduction to Cellular Networks

Cellular telephone systems are a way of providing portable telephone services. Each telephone is connected by a radio link to a base station, which is linked to the landline telephone network. A cellular mobile communications system uses a large number of low-power wireless transmitters to create *cells*. A cell is the basic geographic service area of a wireless communications system. Variable power levels allow cells to be sized according to the subscriber density and demand within a particular region. In order to simplify planning and designing of a cellular network a hexagonal type of cell is usually used. Hexagons tile the plane without any overlap and provide a close enough approximation to the circular shape that best describes the coverage area of a transmitter. But it is important to understand that the hexagonal shaped cells are artificial and cannot be generated in the real world and also they are not suitable for planning all kinds of networks. For instance, if a cellular network is supposed to give coverage to a city, a grid of rectangular cells may be more suitable.

In order to increase capacity, the key concepts used by cellular networks are *frequency reuse* and *cell splitting.* Frequency reuse refers to the idea that the same frequency channel can be reused in different cells of the network. However, if the same channel is used in adjacent cells, this can lead to radio interference, called *co-channel interference.* Thus, there is a limitation in the extent of frequency reuse. Generally, frequency channels may not be reused in adjacent cells. Cell splitting refers to the idea that a cell may be split into several smaller cells, with lower power transmitters in each, in order to increase the amount of frequency reuse. Cells can also be added to accommodate growth, creating new cells in unserved areas, or overlaying cells in existing areas. Usually a base station covers a cell and is responsible for managing ongoing calls in the cell. When a mobile user tries to initiate a call, it requests a frequency channel from the closest base station which assigns a frequency channel to the mobile. Each mobile uses a separate, temporary radio channel to talk to the base station. However, the base station talks to many mobiles at once, using one frequency channel per mobile. It is common to pre-assign a set of frequency channels to each base station in advance, where the assignment is pre-computed to avoid interference with neighboring cells [2,4,5,6]. It is important to do this in such a way that each cell site can support the maximum possible number of mobile users. Then, when a call arrives, the base station has to make a decision about which of its available channels to assign to the call.

The mobility of users adds an extra complication to channel assignment strategies. In particular, users may move between several cells during the duration of a call. When moving to an adjacent cell a mobile would generally lose contact with the original base station. Thus, the call is *"handed off"* to the base station in the adjacent cell. If the call

were to continue on the original channel despite moving to a new cell this could cause *co-channel interference*. Thus the channel is usually released, and the new base station assigns a new channel to the call, if one is available.

A successful handoff is subject to the availability of a frequency channel in the adjacent cell. Some channel assignment algorithms give priority to handoff calls (ongoing calls) rather than new calls [1,7]. This can be done by reserving frequency channels for handoff calls. Base stations can estimate the likelihood of a handoff based on cell size, speed and direction of mobile stations. More details on cellular networks can be found in [2].

1.2 Frequency Channel Assignment Problem

Mobile telecommunication systems establish a large number of communication links with a limited number of available frequency channels. Furthermore, reuse of the same frequency channels in neighboring cells causes interference. The task of assigning frequency channels with minimal interference is the channel assignment problem. The channel assignment problem is usually treated as a *"graph multicoloring"* problem where the number of colors is minimized [6]. The phenomenal growth of wireless services has fuelled the demand for efficient assignment of frequency channels, and thus, more sophisticated algorithms are required.

1.3 The Motivation for CNS

By implementing several types of frequency channel assignment algorithms that could be run on various types of cellular networks, CNS becomes a very useful tool to study the behavior and efficiency of each channel assignment algorithm.

CNS has been designed in a manner to keep the door open for new implementations of new frequency channel assignment algorithms, network types and mobile movements. In particular, CNS has been designed with an open architecture so that any researcher with some knowledge of computer programming can add his or her own implementations of frequency channel assignment algorithms, network types or mobile movements to CNS.

1.4 Related Work

Like any dynamic simulation architecture, CNS models real world objects such as mobiles, base stations, and the geographic network area. Thus, it is important to: (a) identify the actors who represent the real world objects (b) identify discrete events (c) identify continuous dependencies, and (d) provide adequate performance [8]. In many simulators, entities communicate with each other by sending messages through a centralized messaging/events distributor object. They also create different processes or threads for each simulator object in order to improve the performance and to simulate concurrent events. In this section, we will provide several examples of such simulators. We will briefly describe each simulator and will compare it with CNS.

Simjava is a series of libraries that can be used to write stand-alone simulations developed by Institute for Computing Systems Architecture at the University of Edinburgh [10]. Simjava is a process-based discrete event simulation package for Java. A simjava simulation is a collection of entities each running in its own thread. These entities are connected together by ports and can communicate with each other by sending and receiving event objects. A central system class controls all the threads, advances the

simulation time, and delivers the events. The progress of the simulation is recorded through trace messages produced by the entities and saved in a file. CNS uses the same techniques. CNS objects communicate with each other through a centralized event distributor object. CNS also creates new threads for those objects that need extended CPU power. For example, a new thread is created for each object representing a mobile unit. CNS has different functionality compared to SimJava. SimJava is a collection of general-purpose libraries that can be used to develop a simulator, while CNS is a special-purpose simulator for cellular networks. While Simjava could have been used to develop our cellular network simulator, one of the purposes of this project was to understand how design patterns could be applied in the design of such a simulator. Therefore, we chose not to use Simjava but instead to do the design "from scratch."

MobSim++ is an object-oriented cellular network simulator, developed by the Department of Teleinformatics at the University of Stockholm [11]. MobSim++ runs on top of a parallel simulation executive, the Parallel Simulation Kernel (PSK). PSK is an implementation of the Time Warp mechanism for optimistic synchronization of Parallel Discrete Event Simulations and is also written in C++. MobSim++ like CNS simulates mobile movement, hand-off and also collects statistics for the simulation but MobSim++ does not implement any specific network type, mobile movement or frequency channel assignment algorithms. Thus MobSim++ does not meet the desired objectives of our project.

CellSim is a tool to simulate GSM and CDMA mobile cellular networks [12]. It is developed by Australia's Commonwealth Scientific and Industrial Research Organisation (CSIRO). CellSim performs static GSM or CDMA simulations generating

multiple snapshots in time. Each snapshot represents a particular arrangement of mobiles in cells and thus a particular realization of signal propagation arising from base station and mobile transmission and reception powers. Thousands of snapshots can be quickly run, with mobile positions varying to simulate full cell coverage. Histograms of Carrier-to-Interference ratios accumulated over the snap-shots reveal network performance. CellSim also offers dynamic simulations that are event-driven. A run represents a period of time in which mobiles may enter the network, initiate calls, handover to adjacent cells, terminate calls or leave the network. Thousands of such events can be quickly simulated representing full network operation with network performance measures accumulated. Unlike CNS that is not tied to any specific wireless technology, CellSim is specific to GSM and CDMA networks. CellSim is also a commercial application, and the source code is not freely available.

NS is a discrete event simulator developed in the Department of Computer Engineering at Bogazici University [13]. It provides support for wired/wireless networking with multicast capabilities and satellite networks. Mobility features include node movement, periodic position updates, and maintenance of topology boundary. These are implemented in C++. Plumbing of network components like classifiers, dmux, Link Layer, Media Access Control, channel within Mobile node have been implemented in OTCL. NS focuses on the impact of the mobile movement on routing algorithms in a large network combined from wired, wireless and satellite nodes, while CNS focuses on frequency channel assignments.

As shown in the above examples, although CNS is in many respects similar to other simulators, it provides a unique functionality tailored to our needs. In general, we emphasize three points:

(a) At the time that we started the development of CNS, there were no publicly available cellular network simulators. They were restricted to the private sector.

(b) Even now, there are several network simulators are available, but we have not found any that satisfy the requirements for CNS. Most of these products are for commercial use with a focus on simulating the network traffic. The main purpose of these products is to provide a test tool for network engineers to evaluate the capacity of the network. This differs from the CNS objective to provide a research tool to evaluate algorithms for frequency channel assignment.

(c) One of the objectives of CNS was to study and understand the use of design patterns in this application domain.

2 System Requirements

This section will describe the general, hardware, and software requirements of the cellular network simulator.

2.1 General Requirements

First, we describe the domain-related requirements, and then the user related requirements. CNS should be able to simulate relevant events in the geographic area that is covered by a cellular network. In a real life situation, usually the geographic area has been divided into a network of cells. Depending on the distribution and movement of mobile traffic, the network will be assembled in various ways and by using various cell shapes. For instance in a city, it is desired to have rectangular cells between street blocks and buildings; so a network of rectangular cells should be used. In a remote area, a hexagonal type of cell is more desirable; so a network of hexagonal cells should be used. In order to have a realistic simulation, CNS should be able to simulate different types of networks such as hexagonal and rectangular networks.

One of the main characteristics of a cellular network is that its users are not stationary. In particular, during the period of a call, a user may move to several different cells. The way that a mobile user moves can follow different patterns and could depend on several parameters such as network type. For instance in a city, a rectangular type of network is usually used and the mobile can change direction only at the intersections and in a very specific way. On the other hand, a hexagonal type of network is more likely to cover areas including highways and the mobile movement usually follows a straight line

pattern. It is important that CNS should be able to simulate different type of mobile movements.

Another important aspect of a cellular network is to understand when a mobile user tries to use the system and for how long. Mobile arrivals are generally modeled by a Poisson process, while the duration of a call generally follows an exponential distribution. CNS should follow the same conventions for call arrivals and duration.

One of the main reasons behind developing CNS is to design a tool to simulate different frequency channel assignment algorithms, in order to compare their performance in a specific type of network. Thus, it is obvious that CNS should be able to simulate different types of frequency channel assignment algorithms. Furthermore, it should be easy to add new algorithms at a later date. Finally, the channel assigned to a call may change while crossing a cell boundary. Thus, the appropriate base station must be notified when this occurs so that a handoff procedure may be called into effect.

The main motivation behind the development of CNS is to simulate channel assignment algorithms, in order to study their behavior and compare their performance. Thus appropriate statistics should be collected, and CNS must provide a mechanism to perform this function.

Next, we describe the requirements that arise from the types of expected users. CNS is expected to have two classes of users. The first set may use CNS to demonstrate channel assignment algorithms, for example, for teaching purposes. Thus, a good user interface will be required for this set of users. The second set is expected to use CNS to experiment with newly designed algorithms. This will require stepping through a simulation and probably a record mechanism. To do performance evaluations, the same

set of users is expected to set up and run several simulations to gather data. In this anticipated use, it is desired to provide a means of running CNS without a graphical user interface.

Finally, CNS should be accessible and portable. The best way to make a software tool accessible is to make it Web-enabled. In order to achieve the portability, CNS should be able to run on both Windows and Unix platforms.

2.2 Hardware Requirements

CNS can run on any standard UNIX or PC station, but due to the demanding nature of any graphical application, it is highly recommended to run CNS on a powerful station with at least 32-MB of RAM.

2.3 Software Requirements

CNS is a Java applet-based application, which operates in two modes, with graphical user interface or in the background and without graphical user interface. In both modes, Java Virtual Machine is required (VM). VM is part of JDK (Java Development Kit); to run CNS, JDK version 1.1.3 or higher is required. To run an applet, Java delivers in two ways: using appletviewer, included in JDK, or a Java-enabled web browser (for example, Netscape's Navigator or Microsoft Explorer version 4 or later). Theoretically, other Java-enabled web browsers could be used, but they have not been tested.

3 System Analysis and Design

The main objective of the design of CNS was to create a reusable extendable system with an open architecture for future enhancements. This leads CNS 's designers with no choice rather than practicing object-oriented modeling. The next section gives a brief introduction to the ideas we use, including UML and design patterns. Next, we give some use cases for our application, followed by a preliminary analysis of the system and our use of pertinent design patterns.

3.1 Design Patterns

Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structures and behavior in a single entity. Object-oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation, and designing software [8].

In late 1994, Grady Booch, Jim Rumbaugh, and Ivar Jacobson decided to join forces and create a standard set of graphical notations for object-oriented design. The result was the *Unified Modeling Language (UML)* [9]. UML is a commonly understood notation for describing object-oriented systems. UML notation has been used to describe the relationship between objects throughout this report.

Designing object-oriented software is hard, and designing reusable object oriented software is even harder. Finding pertinent objects, factoring them into classes at the right granularity, defining class interfaces and establishing key relations among them is a

tough job to do. Many experts feel that getting a reusable design right the first time is an extremely difficult task [3]. At the same time, several design problems tend to recur in applications, and experienced designers have found good solutions for them. These solutions ought to be reusable. This point of view has been explored by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in their book [3]. In their words,

> *"Design patterns make it easier to reuse successful designs and architecture. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster."*

Several design patterns were studied for this project, but only those which were found applicable, have been described in this report (see Appendix A).

3.2 Use Cases

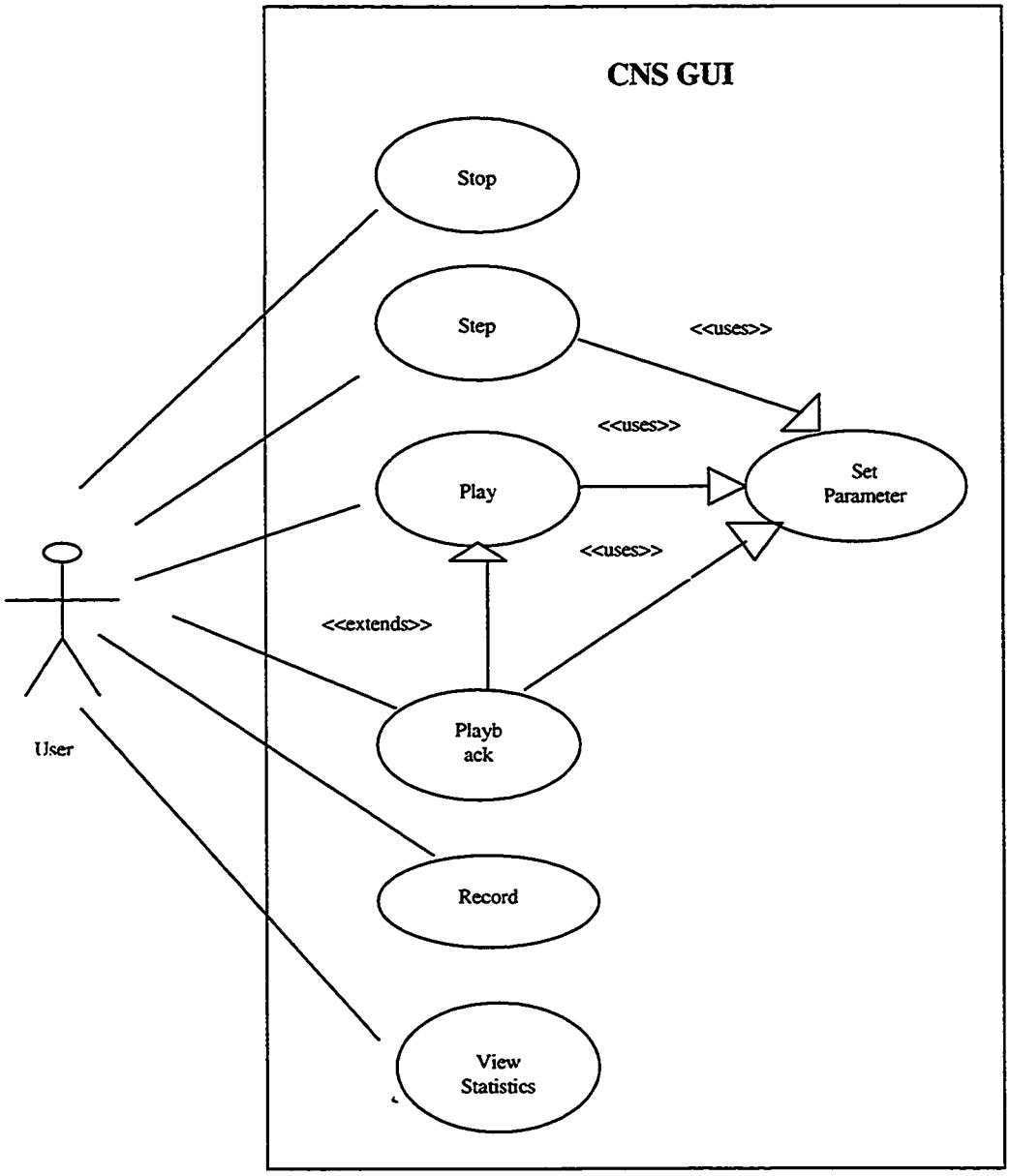


Figure 3.1

Figure 3.1 describes the available CNS's functionality from its GUI interface. The <<uses>> relationships from "step", "play", and "playback" to "set parameter" indicate

that they use the simulation values set by the "set parameter" use case. The <<extend>> relationship from "playback" to "play" indicates that "playback" is a specific case of "play". The "playback" task is a "play" of the previous simulation.

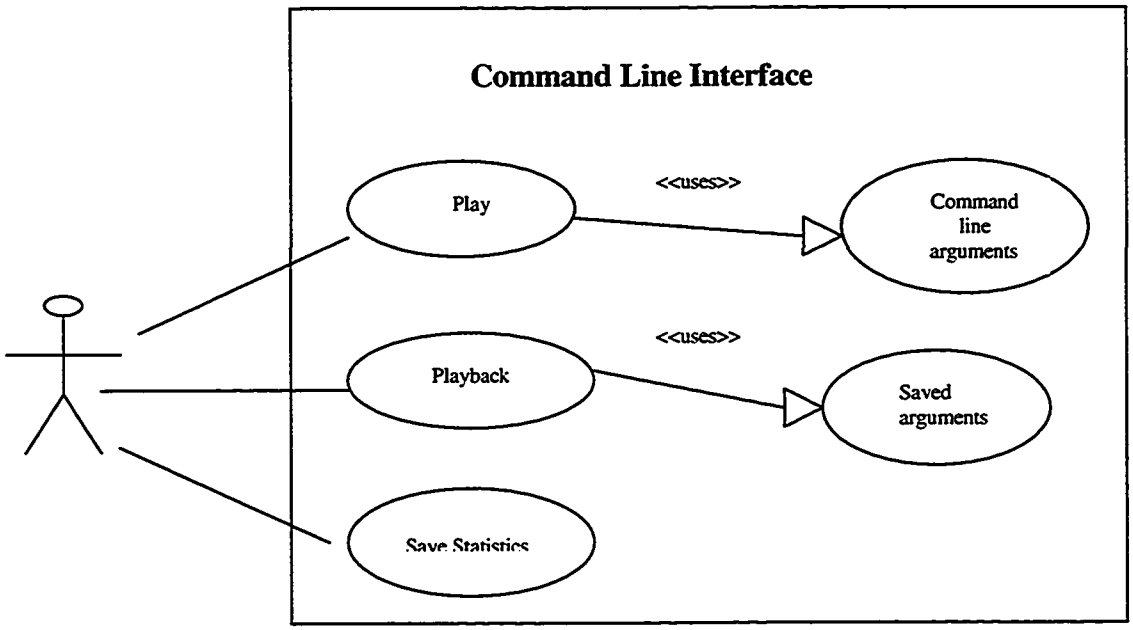


Figure 3.2

The above diagram describes the CNS functionality available from its command line interface. The <<uses>> relationships from "play", to "Command line arguments" indicate that the "play" use case uses the command line arguments values. The <<uses>> relationships from "playback" to "Saved arguments" indicate that the "playback" use case uses the simulation values saved in a simulation file. The use case also indicates that the user can save the simulation statistics.

3.3 Preliminary Analysis

One of the first steps in designing CNS was to identify the objects that can describe the characteristics of the cellular network simulator as well as establish the relationship between those objects.

From the requirements, three fundamental objects appear to be: An object representing a cell (***Cell***), an object representing a group of cells (***Network***) and an object in charge of the simulation (***Sim***). In addition to those objects there is an obvious need for an object representing a mobile station (***Mobile Station***). A base station object (***Station***) assigns frequency channels for mobile stations.

To run the simulation itself, using a scheduler that extracts simulation events from a priority queue looked reasonable; this necessitates a scheduler object (***Scheduler***), an event object (***Event)*** and a heap manager object (***Event Queue***) .

Figure 3.3 gives an idea of the objects identified so far and their relationships.

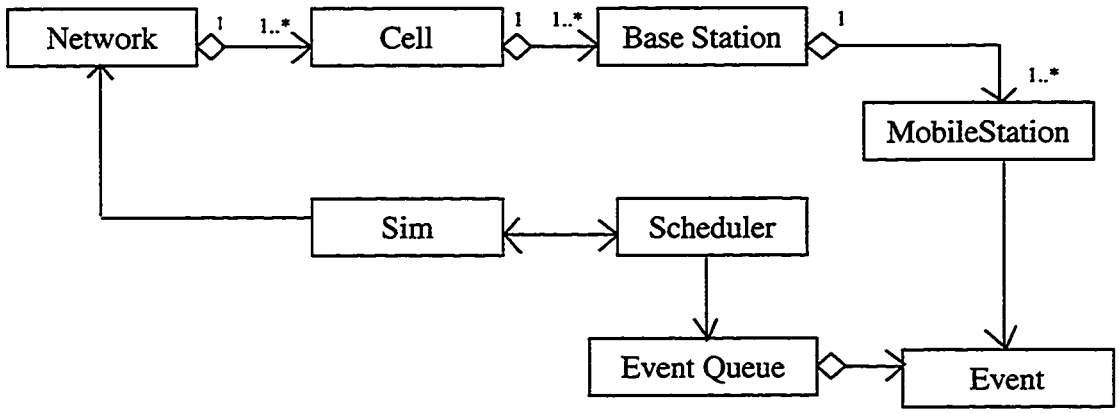


Figure 3.3

3.4 Use of Design Patterns

In this section we identify the design patterns applicable to our problem. CNS should be able to create different types of networks by using different type of cells, or using the same cell type but assembling them in a different fashion. This is exactly the job of the

Builder pattern. Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. For instance the only difference between the hexagonal overlap network and hexagonal network is the way that hexagonal cells will be assembled together. Thus building the CNS network is a very good candidate for applying the builder pattern.

Some CNS objects should only have one instance such as ***Event Queue*** object, which collects all the events. In order to make sure a class has only one instance, the **Singleton** pattern is used. This can also be used to implement objects such as ***Sim*** and ***Scheduler.***

One of the key requirements is that CNS should be able to use different types of frequency channel assignment algorithms and should be able to implement different types of movements. All frequency channel assignment algorithms are related, just as all of the mobile movement algorithms are related; they only differ in their behavior. The **Strategy** pattern is useful for those classes because implementations of different algorithms are needed.

When a mobile station changes its states (created, moves, etc.) it will have an impact on other objects such as interface object and the object that collects statistics. The mobile object need not be aware of how many other objects needed to be changed. Also the mobile station objects should not be attached to the interface object in order to enable the design of dual mode operation (with or without GUI Interface). The ***Observer*** pattern provides a solution for this situation.

3.5 System Design

In this section, we will provide a graphical notation of CNS classes and their relationship with each.

Figure 3.4 describes the generalization relationship between the ***Network*** classes. All ***Network*** classes share the same methods and properties. They only differ in the way that they construct a network. Each class implements its own version of the *Build Network* method by overwriting its parent *Build Network* method.

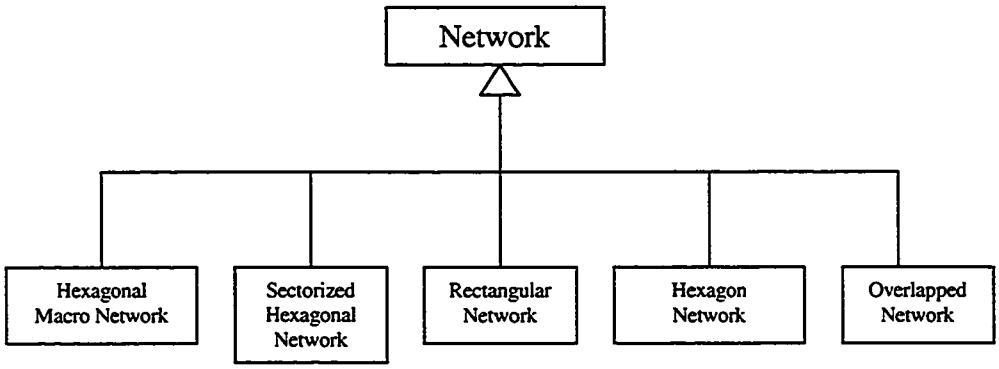


Figure 3.4

Figure 3.5 describes the generalization relationship between the ***Cell*** classes. All ***Cell*** classes share the same methods and properties. They only differ in the way that they draw themselves. Each class implements its own version of the *paint* method by overwriting its parent *paint* method.

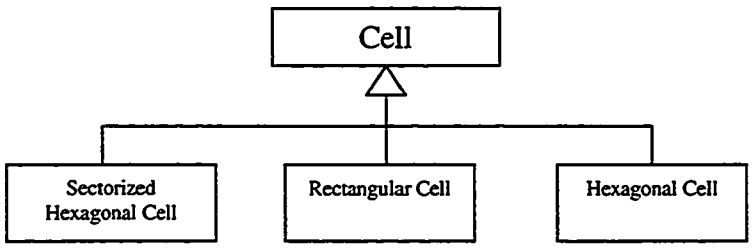


Figure 3.5

Figure 3.6 describes the generalization relationship between the ***Channel Assignment*** classes. They only share the properties and methods related to spectrum setup. Each class has its own implementation of channel assignment.

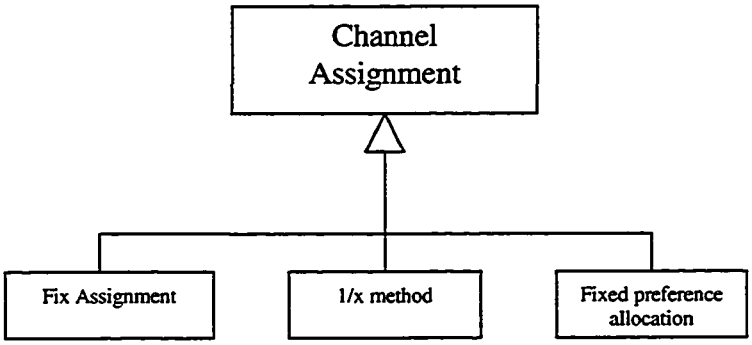


Figure 3.6

Figure 3.7 describes the generalization relationship between the ***Mobile Movement*** classes. All ***Mobile Movement*** classes share the same methods and properties. They only differ in the way that they set the direction of the next move *setDirection* and the way that they calculate the next mobile location *moveLocation.*

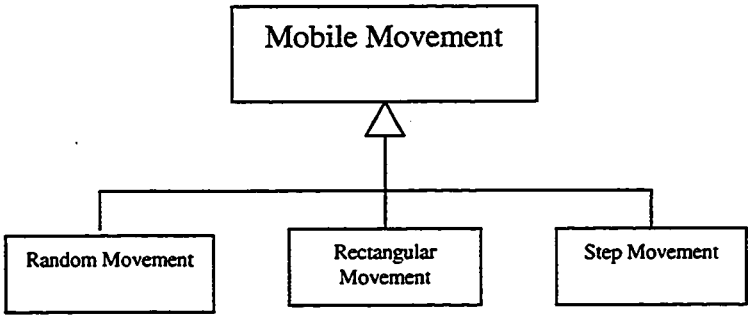


Figure 3.7

Figure 3.8 describes the relationship between the most important objects of CNS. The objects in CNS fall in four major categories: objects implementing the geographical simulation area, objects implementing frequency channel assignment, objects implementing mobility, and objects managing the simulation.

The geographical simulation area has been implemented by using the ***Network*** (which is a base class for different type of networks) and the ***Cell*** objects (Cell is a base class for different type of cells). Each network consists of several cells. There are different types of cells, and additionally there are different types of networks. This means that a network could be constructed by different types of cell objects and assembled in various ways to represent different type of networks. In order to separate construction of a network from its representation and provide a generic solution for assembling various types of networks the **Builder** pattern is used. Each Network class has a method called *BuildNetwork* which constructs a representation of the network geographic area for a specific network. The network object is also an aggregation of mobile objects. This is to enable the network object to keep track of the status of each mobile station.

Each cell has a ***Base Station*** (Base Station is a class representing a base station and is responsible for assigning frequency channels to mobile stations). Each base station has a frequency assignment algorithm. The ***Channel Assignment*** object (Channel Assignment is a base class for different type of frequency channel assignment algorithms) is responsible to find an available frequency channel and then the frequency channel will be assigned to a ***Mobile Station*** by the ***Base Station*** object. It is desired to use a single interface to access various channel assignment algorithms. The **Strategy** pattern is used

to fullfill this goal. The Channel Assignment class provides a single interface for different implementations of channel assignment algorithms.

A mobile is randomly instantiated somewhere in the network geographical area. It finds its base station and requests a frequency channel from its base station. Then it starts moving within the network. As with the channel assignment, the **Strategy** pattern is used to provide a single interface for various algorithms that implement the mobile movement. In this case, the ***Mobile Movement*** class provides this interface. If a mobile moves to a new cell, a handoff procedure is initiated and a new frequency channel will be assigned to the mobile. Change of states within a ***Mobile Station*** object will have an impact on the objects that have dependencies on it such as the ***Interface*** object and the object responsible for collecting statistics. In order to notify and update those objects automatically, the **Observer** pattern is used. The Java built-in ***Observable*** and ***Observer*** classes are used for the implementation of the **Observer** pattern.

Each action from the beginning until the end of simulation is an event. ***Events*** are instantiated by CNS's objects and managed by a priority queue. ***Heap Manager*** is a class implementing a heap data structure. The ***Scheduler*** object retrieves events from the priority queue and assigns them to the appropriate object to handle them. The ***Sim*** object is responsible for setting up, starting, and stopping the simulation.

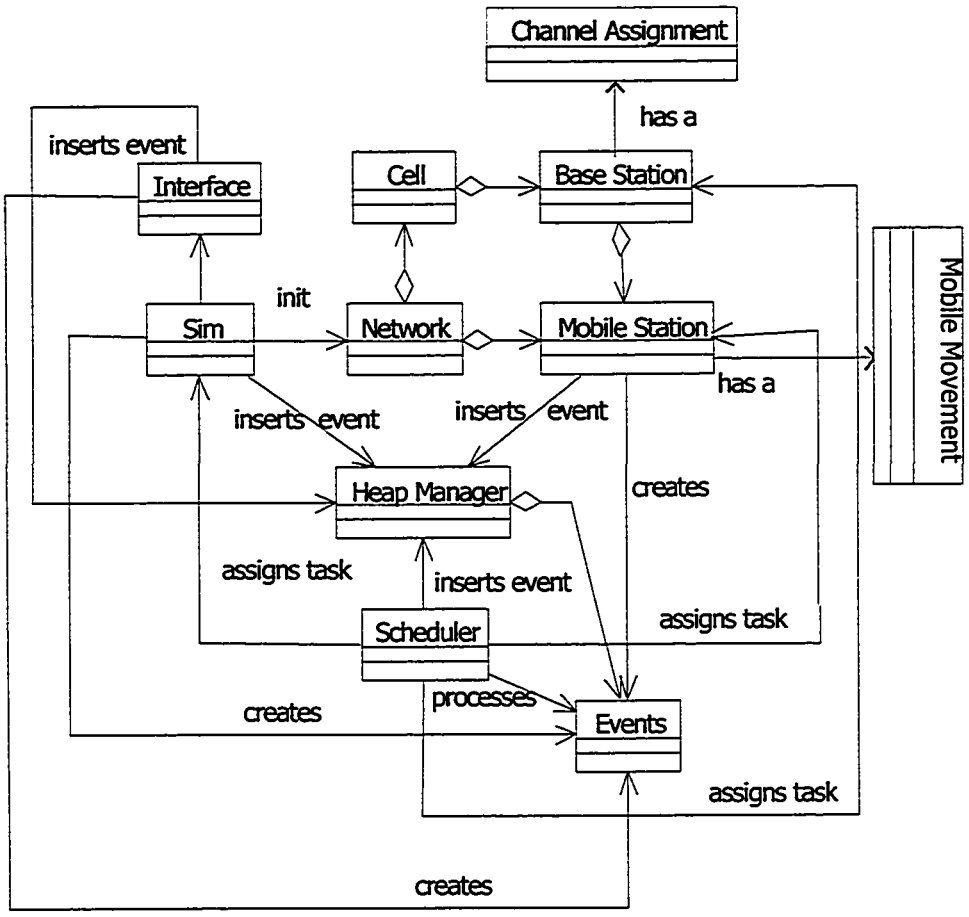


Figure 3.8: Class Diagram

3.6 Detailed Class Diagrams

The following diagrams will provide the detailed design of CNS's classes.

3.6.1 Sim

The Sim object is the CNS's main class. It is responsible for setting up and starting the simulation and stopping it. It also provides methods (*getTime* and *setTime*) for other objects to synchronize the event creation activities. Events are processed based on their event time and their priorities. It is very important for all objects to use the same clock source.

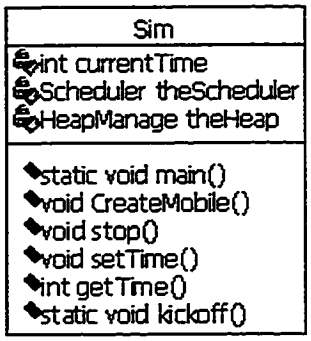


Figure 3.9

3.6.2 Cell

Cell is a base class for different types of cells. A cell has a station and it is identified by its Id (CellId). A ***Cell*** object is able to draw itself by using the *paint* method. The *paint* method would be implemented differently in different cell types. A ***Cell*** object also can determine if it contains a specific point by using its *contains* method. This is used by mobile objects to identify the cell id of their current location.

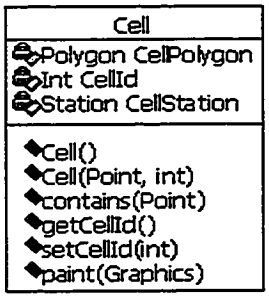


Figure 3.10

3.6.3 Hexagonal Cell

HexagonalCell is derived from the Cell class and implements a hexagonal cell (see Figure 3.12). This type of cell can be used in a buildup process of any network that uses

hexagonal cells. ***Hexagonal Cell*** inherits all properties and methods of its parent class ***Cell*** and only the *paint* method is overwritten.

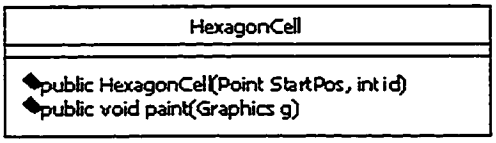


Figure 3.11

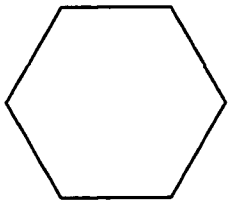


Figure 3.12

3.6.4 Rectangular Cell

RectangularCell is derived from the ***Cell*** class and implements a rectangular cell (see Figure 3.14). This type of cell can be use in a buildup process of any network that uses rectangular cells. ***Rectangular Cell*** inherits all properties and methods of its parent class ***Cell*** and only the *paint* method is overwritten.

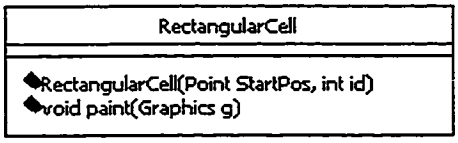


Figure 3.13

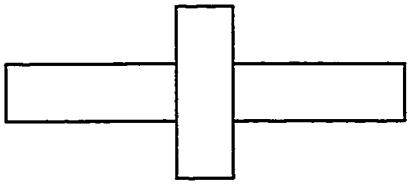


Figure 3.14

3.6.5 Network

Network is the base class for different types of networks (see Figure 3.4). It is an aggregation of cells and mobile stations. The **Builder** pattern has been used to implement different types of networks. The *BuildNetwork* method assembles cells together and builds different types of networks. The ***Network*** object also keeps track of all mobile stations within the network, and calculates the number of active mobile stations and number of hand off. There are methods for adding (*AddMobileToNetwork*), deleting (*DelMobileInNetwork*) and also locating (*FindIndexOfMobile*) mobile objects. The ***Network*** class also has a *ResetNetwork* method to destroy the network in order to build a new one.

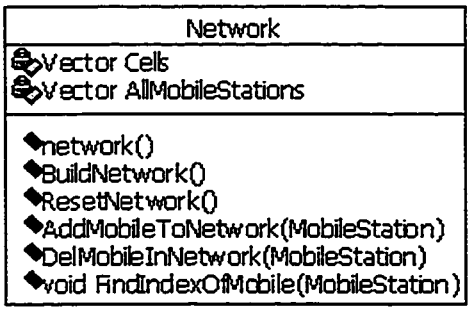


Figure 3.15

3.6.6 HexagonalNetwork

HexagonalNetwork is derived from the ***Network*** class and implements a hexagonal network by assembling non-overlapping hexagonal cells together. ***Hexagonal Network*** inherits all properties and methods of its parent class ***Network*** and only the *BuildNetwork* method is overwritten. This is where it constructs a hexagonal type of network.

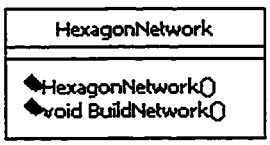


Figure 3.16

3.6.7 RectangularNetwork

RectangularNetwork is derived from the ***Network*** class and implements a rectangular network by assembling rectangular cells together, in a non-overlapping fashion. ***Rectangular Network*** inherits all properties and methods of its parent class ***Network*** and only the *BuildNetwork* method is overwritten. This is where it constructs a rectangular type of network.

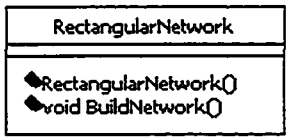


Figure 3.14

3.6.8 Base Station

Base Station objects are responsible for assigning channels to mobile stations within the cell area that they are located. A ***Base Station*** object has a ***Channel Assignment Algorithm*** object; which in different simulations, can be instantiated to different types of channel assignment algorithms. Station color is an attribute used by certain channel assignment algorithms, and is set by the network. A base station also maintains a list of all the mobile stations it is currently handling using *AddMobile* and *DelMobile*. A station object maintains a list of available, used and reserved frequency. channels using *AddToFrequencyList* can be used to add frequency channel to any of the frequency lists (ListOfAvailabeFrequency,ListOfReservedFrequency,ListOfUsedFrequency).

RequestFrequency returns an available frequency channel to a mobile request. *FrequencyRelease* is to delete a frequency channel from a frequency list. Reserved frequency channels may be used for handoff purposes because it is generally more desirable to drop a new call rather than terminate an ongoing call. *AddHandOff* will add

the time of the next hand-off to the HandOffQueue List. NewCallQueue maintains a list of new calls using *AddNewCall* .The ***Base Station*** object will decide to drop or accept a new call by consulting with these two lists. When a ***Base Station*** object is instantiated, it sets all its available frequency channels using its frequency channel assignment algorithm. When a mobile requests a frequency channel, the base station assigns a channel to the mobile using its channel assignment algorithm. If there are no more available frequency channels, the call will be dropped.

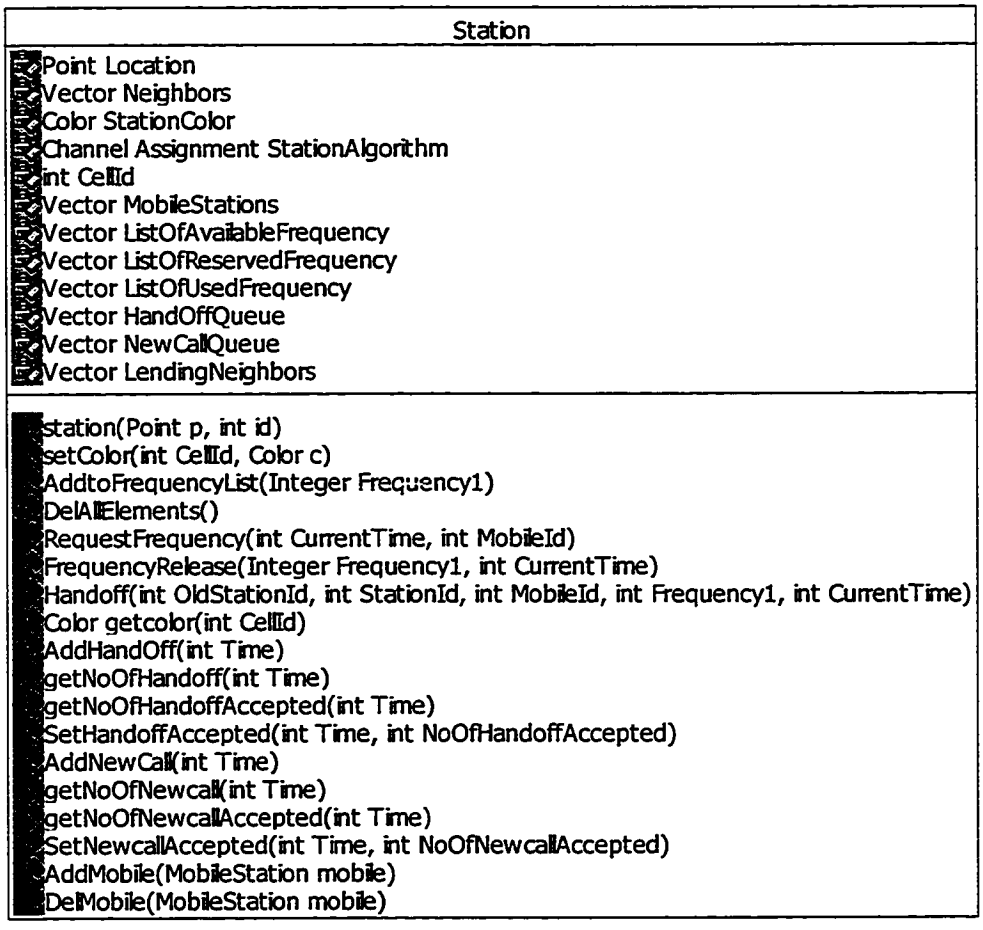


Figure 3.18

3.6.9 Mobile Station

Mobile Station objects simulate the mobility in the system. They are randomly instantiated in a location within the network geographical area. The mobile inter-arrival time is based on a Poisson distribution and its duration is based on an exponential distribution. The parameters for these can be set by the user. Upon creation, a mobile station tries to find the nearest base station by scanning the network's cells using *FindStation* method. After determining the base station, it requests a frequency channel from the station and sets its frequency channel using *setFrequency* then starts to move. The move algorithm is based on network type and also simulator parameters. When the mobile duration is finished, the mobile releases its frequency channel using the *Endcall* method. If the mobile wants to move to a new cell, it notifies the neighboring cell in advance. This information may be used to reserve a frequency channel for future handoff. The creation of a mobile station sets up the creation of the next mobile as well as all the mobile's activities during its lifetime by inserting appropriate events in the event queue. ***Mobile Station*** also implements the **Observer** pattern by using the Java built in *Observer*. When a ***Mobile Station*** changes its states (move, hand-off, end of call, etc.), it has to notify the ***Interface*** object and the object in charge of collecting statistics. This is automatically done by implementing the **Observer** pattern. The ***Mobile Station*** adds all objects that have to be notified of its state's change to its list of observers and then notifies every object that exists on that list. The ***Mobile Station*** provides several interfaces to enable other CNS objects to retrieve the mobile's run time parameters. Those interfaces are *getTime, getDuration, getMobileId, getStation and getOldStation.*

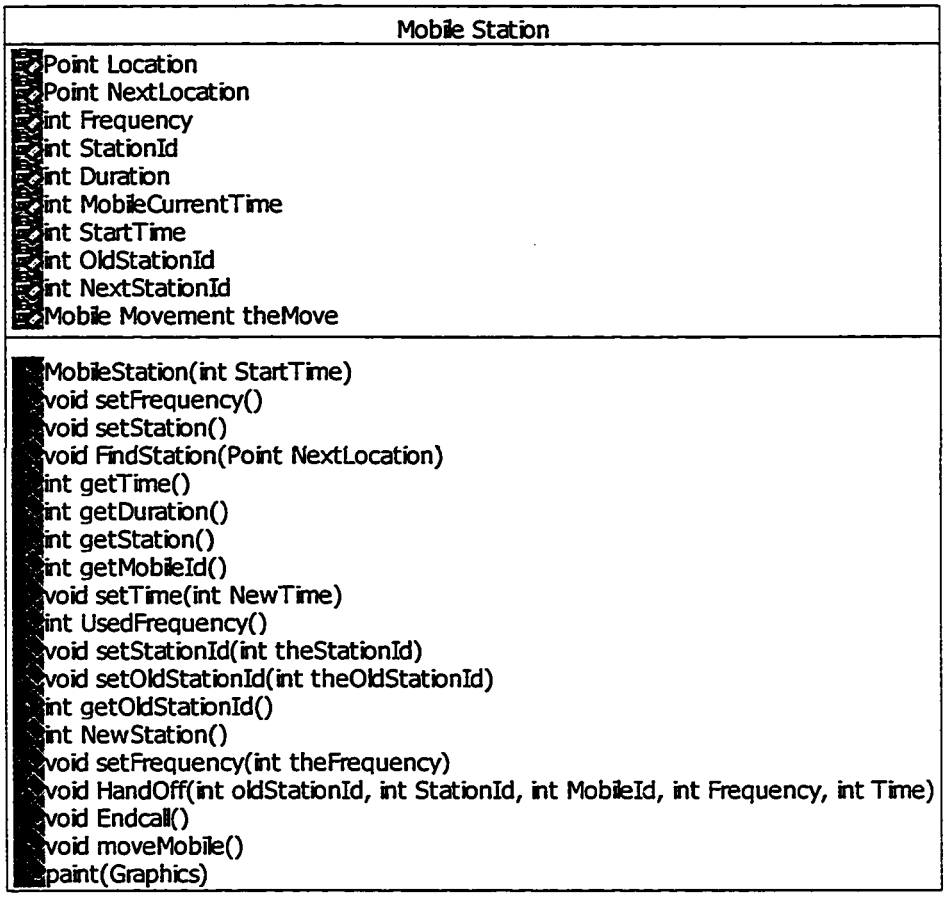


Figure 3.19

3.6.10 Channel Assignment

This is a base class for different frequency channel assignment algorithms. Each ***Channel Assignment*** has a spectrum which is a list of its available frequency channels. The ***Channel Assignment*** class is responsible to set up the spectrum. It performs this function by analyzing the network structure and considering its own specific implementation. The ***Channel Assignment*** class is implemented by using the **Strategy** pattern. The ***Channel Assignment*** class is an abstract class, meaning there is no object instantiated by this class and no method that is actually implemented. Different ***Channel Assignment*** classes are

derived from this class (see Figure 3.6). Those classes include the implementations of the channel assignment algorithms.

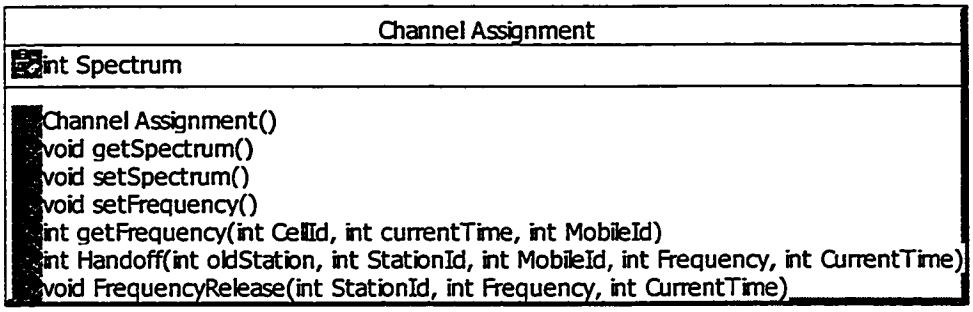


Figure 3.20

3.6.11 Fixed Assignment

This is derived from the ***Channel Assignment*** class and implements the Fixed channel Assignment. This algorithm is described in [4].

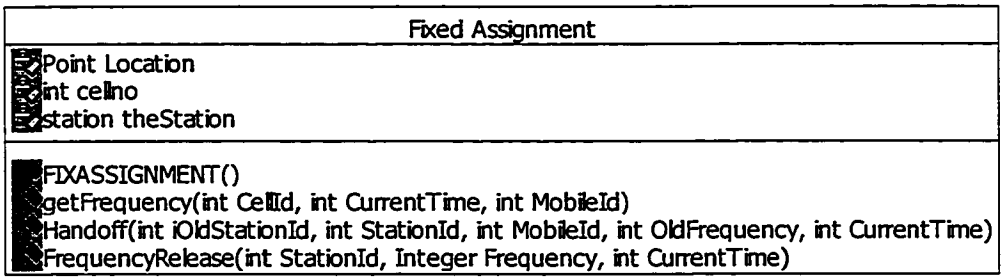


Figure 3.21

3.6.12 Mobile Movement

This is the base class for different types of movement algorithms. Mobile Stations can move in different patterns. For each specific type of movement, one sub-class is inherited from the Mobile Movement class (see Figure 3.7). Mobile Movement

implements the *getLocation* and *move* methods that are common to all movement algorithms. The implementation of the rest of methods is left for the sub-classes.

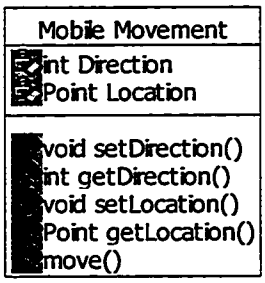


Figure 3.22

3.6.13 Random Movement

RandomMovement is derived from ***Mobile Movement*** and it is used in the ***Hexagonal Network.*** In this method a mobile randomly chooses a direction at the start of its movement and keeps moving in the same direction for the rest of its duration.

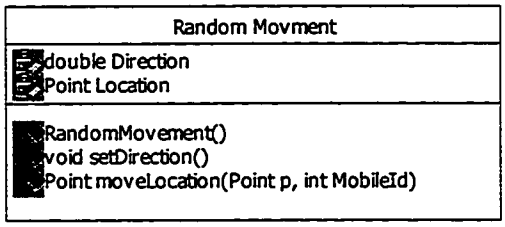


Figure 3.23

3.6.14 Rectangular Movement

RectangularMovment is derived from ***Mobile Movement*** and it is used in the ***Rectangular Network.*** In this type of movement, a mobile first examines its location within the cell. If the mobile is located in the vertical part of the cell it moves either upward or downward; otherwise it moves either right or left. When the mobile crosses an intersection, it randomly decides to move up, down, right or left. Unlike ***Random***

Movement, in this type of movement, a mobile has to examine its location after each step to check for an intersection.

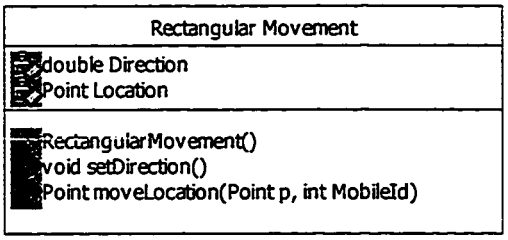


Figure 3.24

3.6.15 Step Movement

StepMovement is derived from *Mobile Movement* class and it is used in a hexagonal type of network. It is similar to *Random Movement*, the only difference is that the mobile station changes its direction after a number of steps specified by the user.

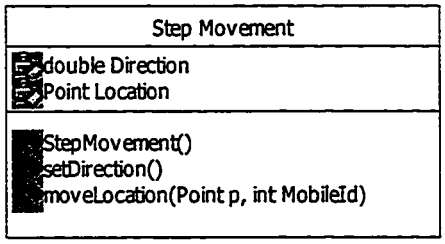


Figure 3.25

3.6.16 Event

Events for CNS are as fuel for a car. *Event* objects are created and handled based on the event map described in Table 3.1. Each event has a creator, who creates the event, a handler which is the object that performs the action expected by the event, a time when it must be executed, and a priority. *Events* are created and inserted into the priority queue by *Mobile Station* objects and the *Sim* object. The *Scheduler* object removes them from

the priority queue and assigns them to an appropriate object to handle the event. Table 3.1 describes different types of events and their attributes.

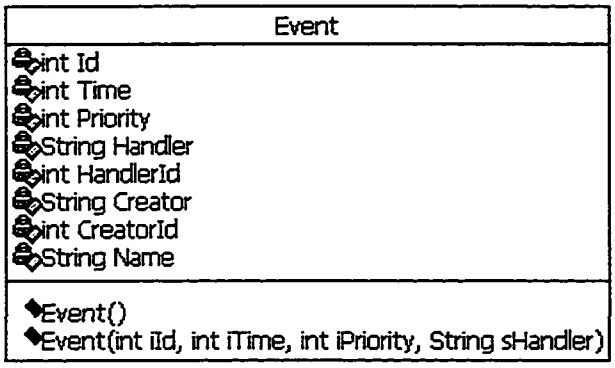


Figure 3.26

Event Name	Creator	Event Id	Handler	Priority	Time
Start	Interface	1	Simulator	1	
Stop	Interface	2	Simulator	1	
Step	Interface	3	Simulator	1	
Create Mobile	Simulator	4	Mobile Station	2	
Request Frequency	Mobile Station (I)	5	Base Station(J)	4	
Move	Mobile Station (I)	6	Mobile Station (I)	4	
End	Mobile Station (I)	7	Base Station(J)	2	
Hand Off	Mobile Station (I)	8	Base Station(K)	2	
Reserve Frequency	Mobile Station (I)	9	Base Station(K)	3	

Table 3.1

3.6.17 Heap Manager

The ***HeapManager*** object implements a heap data structure and it provides methods to store, retrieve and manage the events based on their priority and the time.

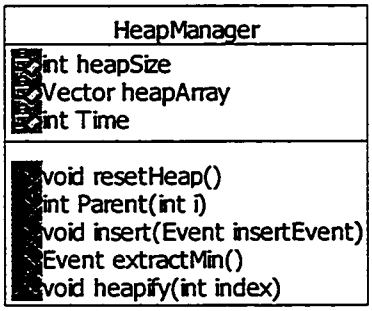


Figure 3.27

3.6.16 Scheduler

The ***Scheduler*** object is responsible for retrieving the events from ***HeapManager***, analyzing them and assigned them to a proper handler object. Each ***Event*** object has an Id and also the identity of who is responsible for this event. The ***Scheduler*** object retrieves this information from the event object and then based on the event Id calls the appropriate method of the handler object. For more information please look at Table 3.1.

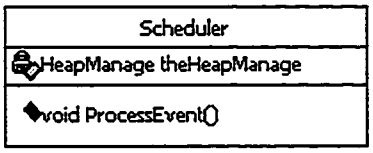


Figure 3.28

3.6.18 Interface

The ***Interface*** object provides the graphical user interface for CNS. Besides providing the standard graphical user interface functionality such as push buttons and panels, CNS is also able to respond to user requests while proceeding with the simulation. This means that CNS should have multithreading functionality. In addition, the user interface also needs to keep track of all mobile objects all the time in order to reflect a valid view of the

simulation. In order to achieve the above-mentioned goals, CNS uses Java built-in ***Thread*** and ***Observer*** functionality.

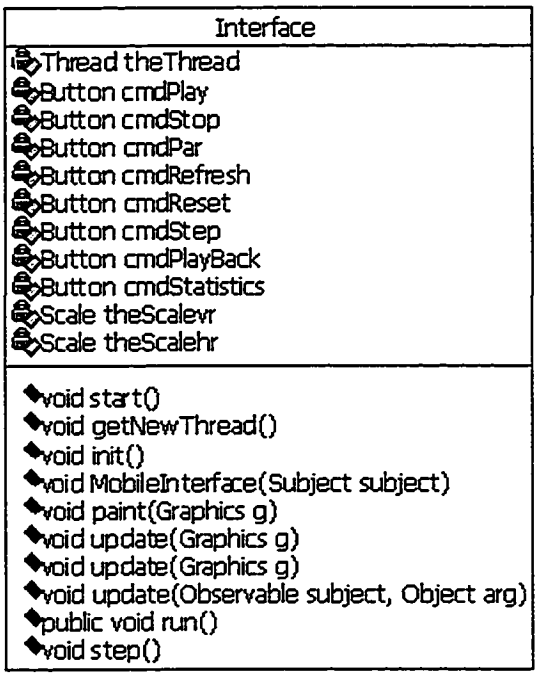


Figure 3.30

3.6.18 Global Constants & Global Variables

The ***namedConstants*** class is used to hold all named constants for CNS. HEXAGON_NETWORK, RECTANGULAR_NETWORK, MOBILE_MOVE_STATE, and MOBILE_HANDOFF_STATE are examples of named constants. The ***GlobalVar*** class is used to hold all global variables SIM_DURATION, CELL_SIZE, ALGORITHM_TYPE, MOBILE_SPEED are examples of global variables.

3.7 Sequence Diagrams

The following sequence diagrams will describe the typical use of the simulator. Figure 3.28 describes the basic use of the CNS. The ***Interface*** object inserts the start event into the priority queue. The ***Scheduler*** object takes the event from the queue and assigns it to the ***Sim*** object. The ***Sim*** object inserts the mobile event into the heap. The ***Scheduler*** takes the event out of the queue and invokes the mobile station constructor. The ***Mobile Station*** adds all of the events needed during its lifetime. This means that the ***Mobile Station*** will find out all the necessary events in advance and add them into the queue. The ***Scheduler*** takes the events one by one out of the queue and invokes the appropriate methods. Those events include requests for frequency channels, requests for movement, requests for hand off and end of call. The same scenario happens repeatedly.

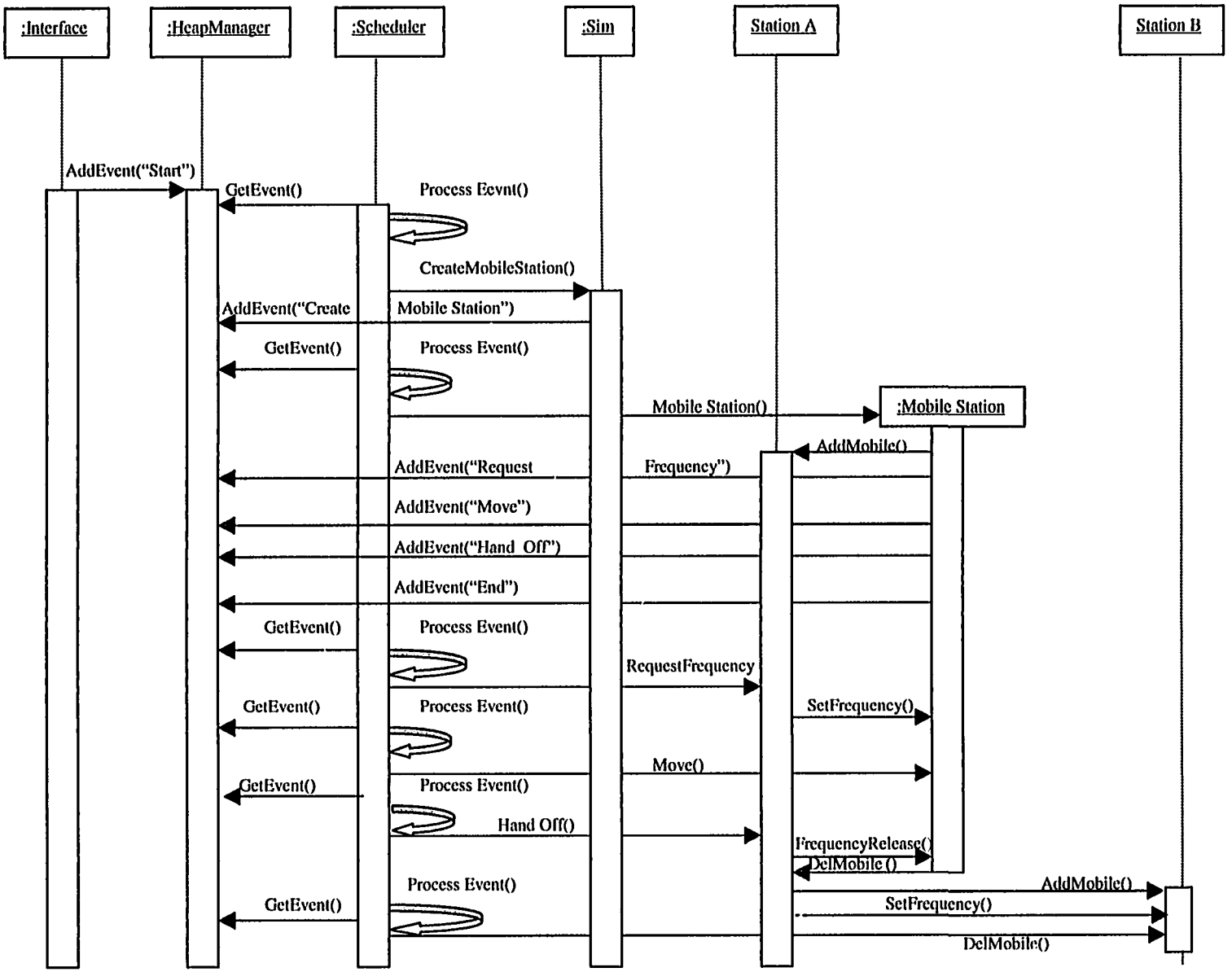

:Interface
:HeapManager
:Scheduler
:Sim
Station A
Station B
AddEvent("Start")
GetEvent()
Process Eevnt()
CreateMobileStation()
AddEvent("Create
Mobile Station")
GetEvent()
Process Event()
Mobile Station()
:Mobile Station
AddMobile()
AddEvent("Request
Frequency")
AddEvent("Move")
AddEvent("Hand Off")
AddEvent("End")
GetEvent()
Process Event()
RequestFrequency
GetEvent()
Process Event()
SetFrequency()
Move()
GetEvent()
Process Event()
Hand Off()
FrequencyRelease()
DelMobile ()
Process Event()
AddMobile()
GetEvent()
SetFrequency()
DelMobile()

Figure 3.31

The next sequence diagram (Figure 3.32) describes the Playback mechanism in CNS. CNS events are created randomly using a random generator algorithm. The value of the seed completely determines the sequence of random numbers that will be generated. Thus, instead of recording the states of all objects during the whole simulation, CNS only records the seed of the previous simulation and feeds the random algorithm with the same seed to generate the exact same simulation. The user pushes the playback button, the ***Interface*** sets the simulation seed with the saved seed value of the last simulation and then calls the *kickoff* method of the ***Sim*** object and the simulation starts.

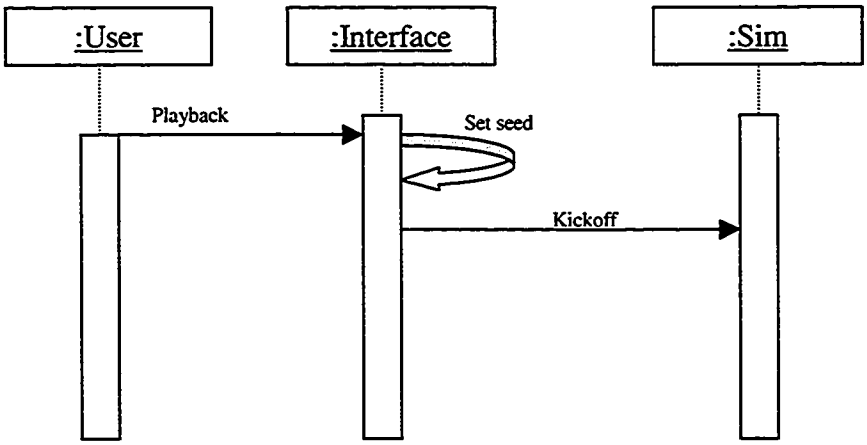


Figure 3.32

The last two sequence diagrams, Figure 3.33 and 3.34, describe the use of CNS from its command line interface. Figure 3.33 demonstrates the sequence of events when the user passes all simulation parameter as command line arguments, while Figure 3.34 shows the sequence of events when a user only passes a file name that contains the simulation

parameters. This case is very useful because with a simple script, the user can run several simulations at the same time. The user also does not need to interact with the system.

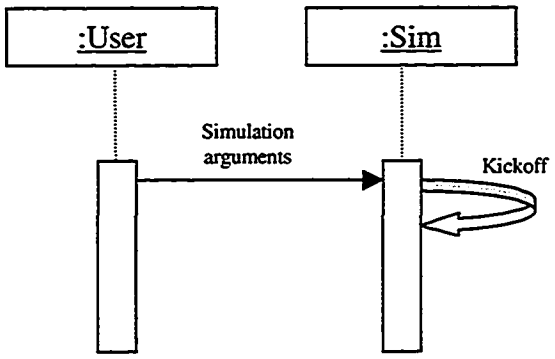


Figure 3.33

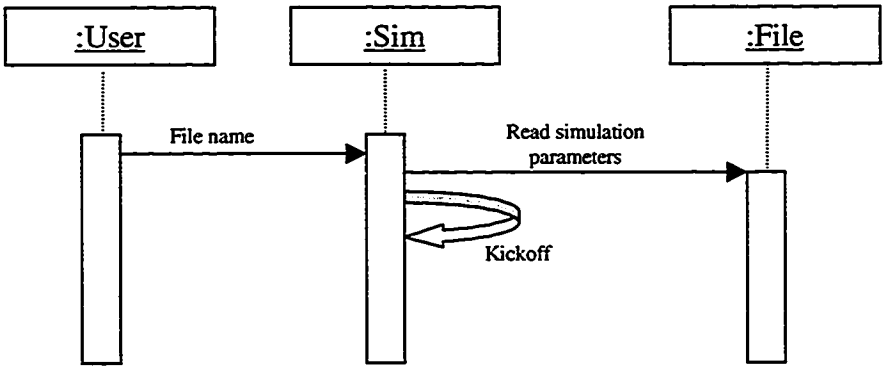


Figure 3.34

4 Interface Design

CNS has two interfaces, a graphical user interface and a command line interface. The graphical user interface should be used when the user wants to see and observe the simulation, but on the other hand, the command line user interface is useful when the user wants to run many simulations and collect statistics. The GUI interface can be run both by appletviewer or any internet browser.

4.1 Graphical User Interface

4.1.1 Main Window

The main interface of CNS is shown in Figure 4.1. In order to start the simulation the **Play** button should be pushed. The **Step** button can be used to run a simulation step by step. This is useful when the user wants to follow the simulation step by step. In this mode, the user has to push the **Step** button each time to proceed to the next step, which means to the next event. The **Stop** button will stop the simulation. The user can set the simulation parameters by pushing the **Parameters** button. This activates the parameter window (see Figure 4.2). After each simulation, the user needs to refresh the screen. This can be done by pushing the **Refresh** button. The user can change simulator parameters by using the **Reset** button. Simulation statistics can be viewed at any time by pushing the **Statistics** button. The user can play back the previous simulation by pushing the **Playback** button.

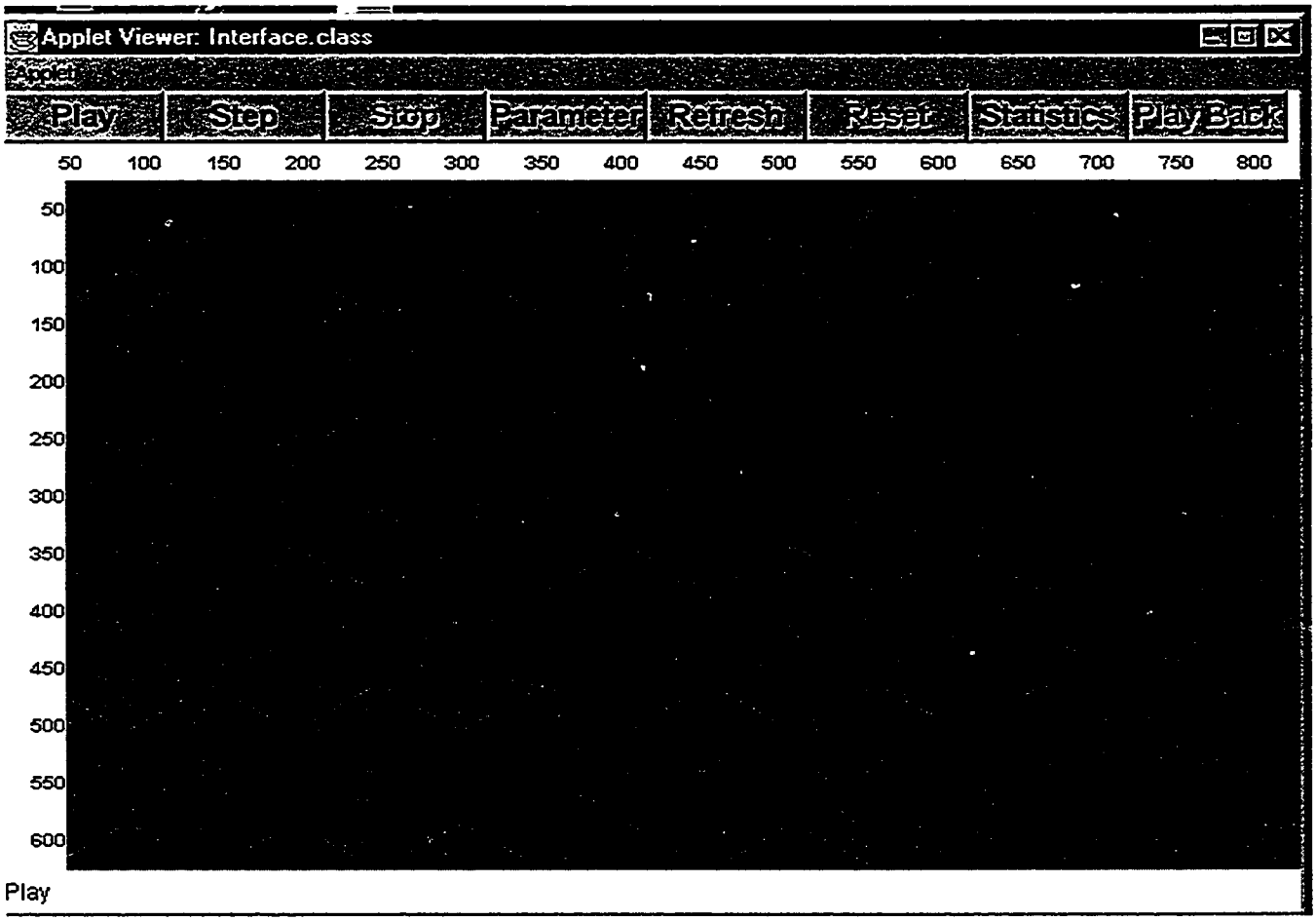


Figure 4.1

4.1.2 Parameters window

The values of **Start Col, Start Row, Stop Col, and Stop Row** will determine the simulation network boundaries. The value of **Call Duration** represents the average duration of a call. The value of **Call Inter-Arrival** is the average call inter-arrival time. The value of **Simulation Duration** is the total simulation duration. The value of **Cell Size** is the radius of a cell. **Mobile Speed** is the speed of a mobile based on the cell size. For instance, a value of 1 means that in each movement, a mobile moves half of the cell

size. **Step Move** is specific to Step Movement and identifies after how many steps a mobile will change direction. It is used when a mobile needs to change direction. **Max Spectrum** is the **maximum** number of available frequency channels. **Rectangular Size** is the width of a rectangular cell and is specific to the rectangular network. The value of **Network Type** represents the kind of network that will be simulated. The value of **Algorithm Type** represents the kind of frequency channel assignment algorithm that will be used. The value of **Cell Type** is of no use for this version but has not been omitted because it might be needed in the next version. The value of **Movement Method** represents the kind of movement algorithm that will be used. The value of **Seed** will be used for generating the first random number and will be used only once at the start of the simulation.

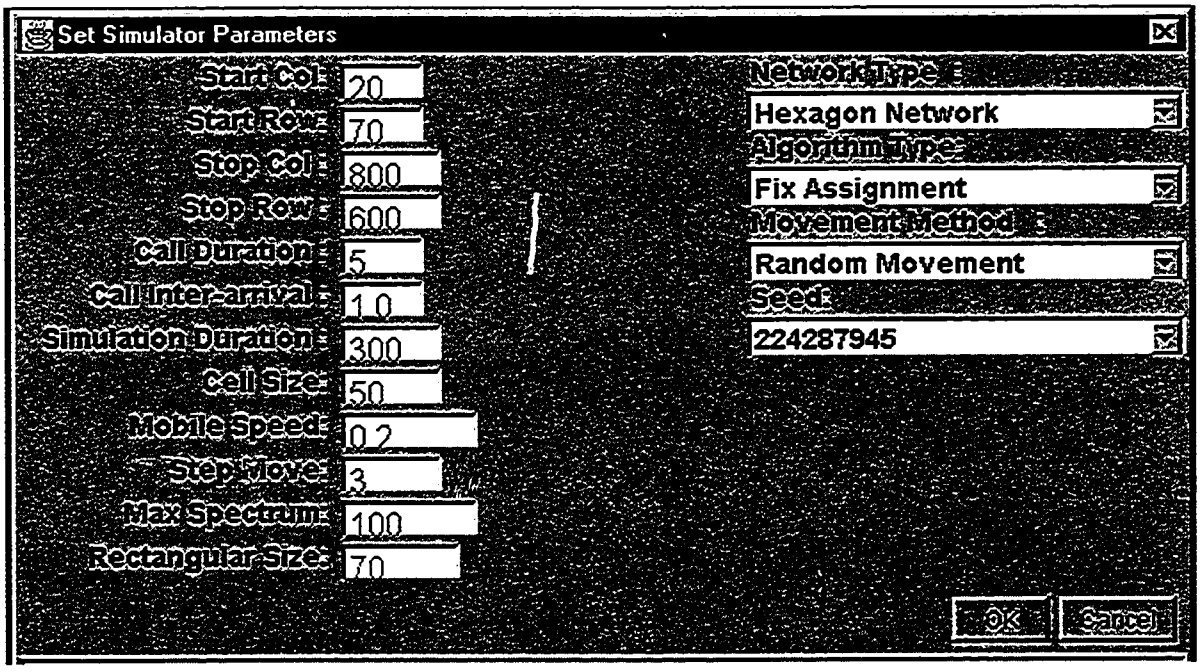


Figure 4.2

4.1.3 Statistics window

The parameters are self–explanatory. Other parameters can easily be added as needed.

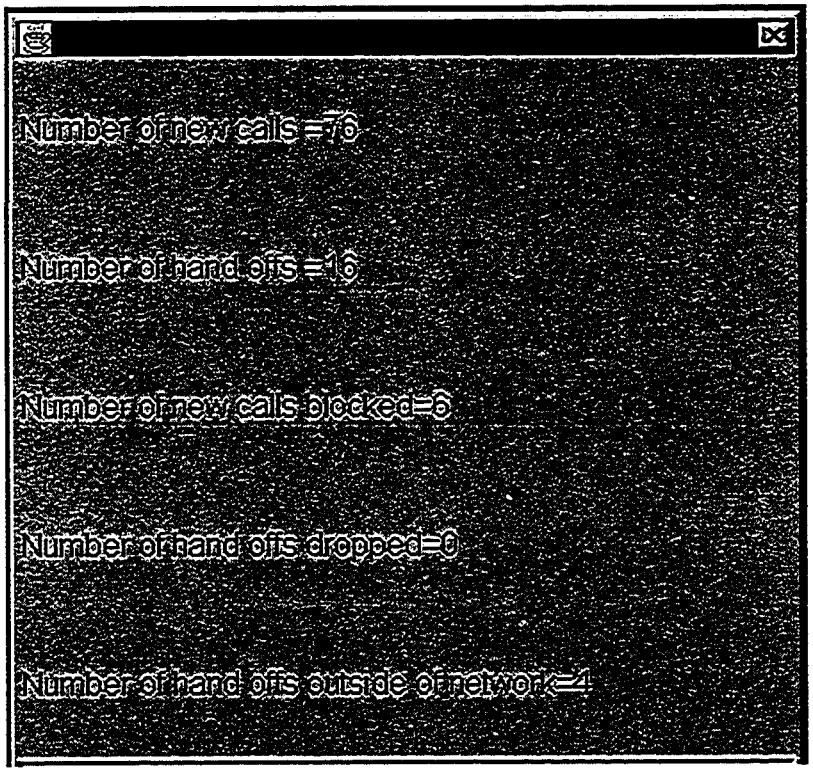


Figure 4.3

4.2 Command Line Interface

The command line user interface should be used when it is desired to run a simulation without need of any interaction. This is a very useful option to run long simulations or several simulations at the same time. All simulation parameters can be set by the users; otherwise the default parameters will be taken. For more information, please see Appendix B.

5 Conclusions

5.1 Design Lessons

Designing and implementing a robust object-oriented application is a demanding and challenging job to accomplish, and almost impossible to get it 'right' the first time.

Making mistakes is unavoidable and there is a need for constant review and correction of the design. However, the most important thing is to learn a lesson from each design project and try to use it in the future.

The design and implementation of CNS was not an exception. After several months of design and even implementing part of the application we had to stop and redesign almost every thing from scratch. It is important to identify those mistakes in order to avoid them in the future. Here are the two of the most important factors that led us to redesigning the system.

1- **Unclear Requirements:**

The system requirements were unclear in the beginning of the project, partly because of miscommunication leading to different understanding of the same problem, and partly because of undecided implementation decisions that led the programmer to make a decision by his own and assume that this is what is required but ended up to be wrong. It is important to spend some time and make sure that the problem has been understood by everybody in the project and prepare a detailed written list of all requirements.

2- Lack of experience applying Design Patterns:

For solving the same problem sometimes it is possible to apply different patterns but choosing the right pattern that suits the problem the best needs experience and deep

understanding of the patterns. Designers should try to understand a pattern and its use in depth before applying them in a design model.

5.2 Future Enhancements

The following areas could be considered for the future enhancement of CNS.

Network Type: Currently two network types have been implemented, Hexagon Network and Rectangular Network, other types of Networks should be implemented in the future such as macro/micro and sectorized networks.

Movement Type: Random Movement, Rectangular Movement and Step Movement have been implemented, and more types of movements could be added to the system.

Channel Assignment: There is definitely a need to add more frequency channel assignment algorithms to the system.

GUI Enhancement: There are a few areas regarding the graphical user interface that, could be improved such as: adding diagrams to the Statistics window, and making the Parameters window more user friendly. In addition, some cosmetic changes could be considered to make the interface more elegant.

5.3 Extendability

The design of CNS was intended to be extendable. In particular, one of the requirements was that it should be easy to add new types of networks, channel assignment algorithms and mobile movements. While complete extendability is difficult or impossible to achieve in any design, we believe the more limited extendability mentioned above has been achieved by our design. In general, to add a new network (or channel assignment algorithm or movement type), a new class for the new network type

should be created which is derived from the ***Network*** class, some methods should be overwritten, and the Parameters window in the ***DataEntry*** class should be modified. The other classes remain unaffected. In this section we describe in detail how CNS can be extended as well as limitations to the extendability.

Network Type

In order to add a new network the following steps should be taken:

1- The ***Network*** class which is the base class for all network classes should be modified to be able to identify the new network type and invoke the constructor of the new network class.

2- A new class should be created. This class should inherit from the ***Network*** base class. The new class should inherit all attributes and methods of its parent ***Network*** class. The only method that must be overwritten is *BuildNetwork*, which should construct the specific implementation of the new network. The construction algorithm is specific to the new network type. However the construction algorithm for different networks may be similar. For example, the algorithm for overlapped hexagonal network will be similar to hexagonal network, except that the boundaries of cells should be calculated in such a way that the neighboring cells overlap.

3- If the new network type includes a cell type that has not defined yet (cell type beside Hexagonal and Rectangular) then the new cell type should be created. The steps are as follows: (a) Modify the GUI to add new cell type (b) Create a new cell type inherited from the ***Cell*** base class. The only methods needed to be implemented are the class constructor and the *paint* method.

4- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new network type. The ***DataEntry*** class includes the implementation of this window.

5- The new network's name should be added to the ***namedConstants*** class.

It is important to understand that any new implementation of network type might have a direct impact on other components of the system such as frequency channel assignment algorithm. This is because the distributions of spectrum and hand-off algorithms in some of the frequency channel assignments are closely related to the way that cells are organized in the network. For instance with the current implementation of CNS, a mobile queries the network in order to find its cell number. The first cell that contains the mobile will reply. Although with the network types already implemented, this is an acceptable mechanism, it may not be sufficient for a macro/micro cellular network, or a network where cells overlap, as a mobile can potentially belong to more than one cell in such networks. What mechanism to adopt may then depend on the channel assignment algorithm used.

Movement type

In order to add a new movement type the following steps should be taken:

1- The ***Mobile Movement*** class, which is the base class for all movement classes should be modified to be able to identify the new movement type and invoke the constructor of the new movement class.

2- A new class must be created and derived from the ***Mobile Movement*** base class. The *setDirection* and *MoveLocation* methods must be implemented.

3- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new movement type. The ***DataEntry*** class includes the implementation of this window.

4- The new movement type's name should be added to the ***namedConstants*** class.

Movement types are closely tied to the network type. Thus any new type of movement should be implemented in a manner suitable for the specific type of network. For instance in the rectangular network mobiles are not allowed to randomly change their direction so we can not have random movement type in a rectangular network.

Frequency Channel Assignment

In order to add a new frequency channel assignment algorithm the following steps should be taken:

1- The ***Channel Assignment*** class, which is the base class for all frequency channel assignment classes should be modified to be able to identify the new frequency channel assignment type and invoke the constructor of the new frequency channel assignment class.

2- A new class must be created and derived from the ***Channel Assignment*** base class. The *getFrequency, Handoff and FrequencyRelease* methods should be implemented.

3- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new frequency channel assignment type. The ***DataEntry*** class includes the implementation of this window.

4- The new channel assignment algorithm's name should be added to the ***namedConstants*** class.

As mentioned earlier, it is important to understand that not all frequency channel assignments can be used for all network types. For a specific network, an appropriate frequency channel assignment algorithm should be selected. This is because the distributions of spectrum and hand-off algorithms are based on the specific behavior and construction of the network.

Validation

In this section, we have described how CNS can be extended in order to add new network types, movement types and channel assignment types. In fact, the extendibility of the design was tested in practice in two ways. First, in the course of our implementation, we started by implementing one network type (Hexagonal Network), one movement type (Random Movement), and a simple frequency channel assignment and hand-off algorithm (Fixed Assignment). We then added a new network type (Rectangular Network) and new movement types (Step Movement and Rectangular Movement) by using the procedures described above. Secondly, CNS has already been used as a simulation tool in [1]. Al-Sumait added a new channel assignment and handoff algorithm as part of her work in [1]. This provides some direct evidence towards the extendibility of our simulator.

Appendix A: Design Patterns

The following has been extracted from the Design Patterns book, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [3], with minor amount of editing and rewording.

A.1 Builder

Intent

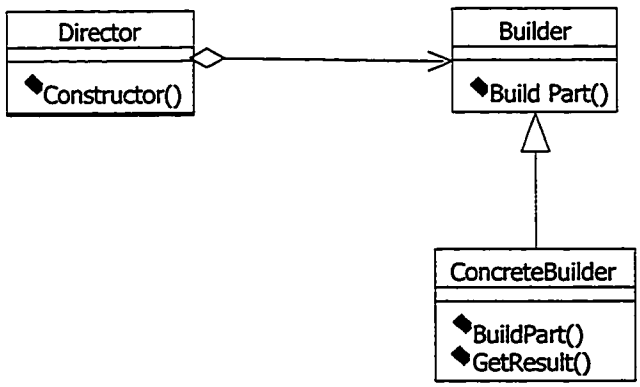
Separate the construction of a complex object from its representation so that the same construction can create different representations.

Applicability

The Builder pattern should be used when:

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Structure



Participants

- **Builder**
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product.
- **Director**
  - constructs an object using the Builder interface.

Description:

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

A.2 Singleton

Intent

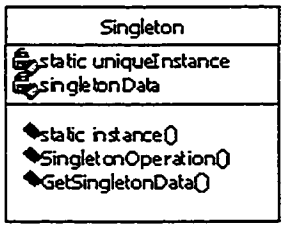
Ensure a class only has one instance, and provide a global point of access to it.

Applicability

The Singleton pattern should be used when:

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance.
  - may be responsible for creating its own unique instance.

Description:

- Clients access a singleton instance solely through Singleton's Instance operation.

A.3 Observer

Intent

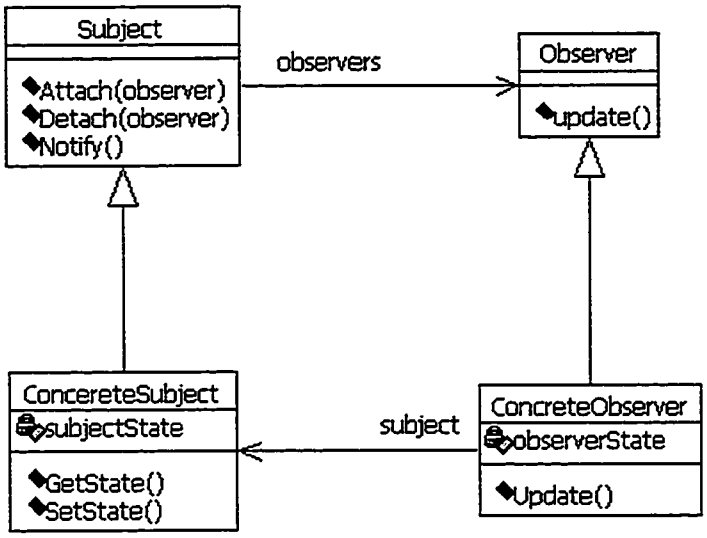
Defines a one-to-many dependency between objects so that when one object changes states, all its dependents are notified and updated automatically.

Applicability

The Observer pattern should be used when:

- an abstraction has two aspects, one dependent on the other. Encapsulating these aspects makes it possible to reuse them independently.
- a change to one object requires changing others, do not know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who those objects are.

Structure



Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

Description

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

B.4 Strategy

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

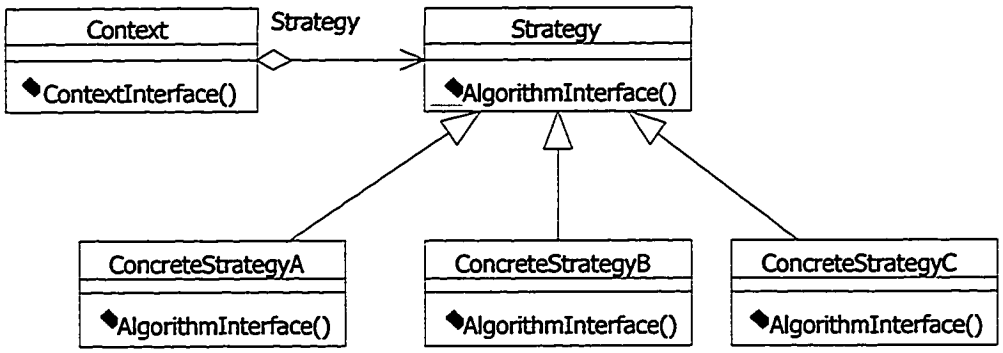
Applicability

The Strategy pattern should be used when:

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- different variants of an algorithm are needed.

- an algorithm uses data that clients shouldn't know about. Use of the Strategy pattern will avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**
  - implements the algorithm using the Strategy interface.
- **Context**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Description:

- Strategy and context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; therefore, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Appendix B: Command Line Interface

java sim [-a] [-b] [-c] [-d] [-t] [-s] [-g] [-n] [-f] [-p] [-i] [-v] [-m] [-u] [-w]

-a: START ROW

-b: START COL

-c: END ROW

-d :END COL

-t: SIM DURATION

-s: CELL SIZE

-g:ALGORITHM TYPE

-n:NETWORK TYPE

-f: FILE NAME, name of playback file

-p:MOBILE DURATION

-i :CALL ARRIVAL

-e:MOBILE SPEED

-v: MOVE METHOD

-m :STEP MOVE

-u: MAX SPECTRUM

-w: File Name, name of the record file.

References

[1] A. Alsumait, A new scheme for prioritizing handoffs in cellular networks, MComp Science thesis, Concordia University, 2000.

[2] G. Calhoun, *Digital Cellular Radio*, Artech House, 1988.

[3] E. Gamma, R. Helm, R. Johnson, and J.Vlissides, *Design Patterns*, Addision-Wesley 1995.

[4] W. Hale, Frequency assignment: theory and applications, *Proceedings of the IEEE*, vol. 68, pp. 1497-1514, 1980.

[5] I. Katzela and S. Naghshineh, Channel assignment schémes for cellular mobile telecommunication systems: a comprehensive survey, *IEEE Personal Communications*, pp. 10-31, June 1996.

[6] L. Narayanan and S. Shende, Static frequency assignment in cellular networks, *Algorithmica*, vol. 29, pp. 369-409, 2001.

[7] M. Oliver and J. Borras, Performance evaluation of variable reservation policies for handoff prioritization in mobile networks, *IEEE INFOCOM* '99, vol. 9, pp. 1187-1194, 1999.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W.Lorensen, *Object–Oriented Modeling and Design*, Prentice Hall, 1991.

[9] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[10] http://www.dcs.ed.ac.uk/home/hase/simjava/

[11] http://www.it.kth.se/labs/sim/projects/

[12] http://www.dbce.csiro.au/ind-serv/brochures/cellsim/cellsim.htm

[13] http://www.cmpe.boun.edu.tr/~emre/research/msthesis/node41.html