2 Theory

This section introduces some of the libraries and techniques used to implement NTS. A quick overview of the communication library on which the project is based, an overview of the **GUI**[3] toolkit used for the Master user interface, and some general programming techniques in the form of object oriented design patterns are presented. In addition, a description of the first incarnation of the Network Topology Simulator: AOM, and the upgrades that were included in NTS compared with this previous incarnation, will be presented.

2.1 PVM ([1])

Unlike on a MPP, a network used for distributed computing usually connects many types of machines, each maybe with its own architecture, data format, and computational speed. When a programmer wishes to take advantage of the collection of networked computers, he has to deal with these differences plus each individual machine load and the full network load. Once communication problems are taken care of, the difference in computational speed is usually the main constraint: one must make sure that if a supercomputer and a workstation are connected together, the supercomputer will not stay idle waiting for a message from the workstation.

Parallel Virtual Machine (PVM) software "provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware ([1])". PVM enables a set of heterogeneous computer systems to be viewed as a single parallel virtual machine, using the message-passing model to allow programmers to exploit the distributed computing potential. It transparently handles all message routing, data conversion, and task scheduling across a network of compatible stations.

Simple and yet general, PVM can accommodate a wide variety of program applications. The structure of use is deliberately simple. The user organizes his application as a collection of cooperating tasks, which access PVM through a library of standard interface routines. These routines allow initiation and termination of tasks, synchronization, and message passing. PVM's message-passing primitives allow for

[3] GUI: Graphical User Interface

heterogeneous operation, involving constructs of buffering and transmission. Simple send and receive is supported as well as high level structures such as broadcast, barrier synchronization, and global sum.

"Owing to its ubiquitous nature (specifically, the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance scientific computing community ([1])".

2.2 Design Patterns ([4])

Design patterns are an attempt to collect and catalog the smallest recurring architectures in object oriented software systems. A design pattern typically captures the solution to a problem that has been observed in many systems. Design patterns are abstract, they have no immediate implementation; they encapsulate the essence of the classes used to solve a problem. Design patterns are design level counterparts of programming idioms, and most object oriented applications will use several patterns to solve their design problems.

Design patterns fall into three categories: creational patterns, structural patterns, and behavioral patterns:

2.2.1 Creational Patterns

A creational design pattern abstracts the instantiation process, encapsulates the knowledge about which concrete class will be used, and hides how the instances of these classes are created and composed together. Creational patterns give a lot of flexibility to what objects get created, who creates them, and how they get created, as well as when they get created. Two types of creational patterns exist: a class creational and an object creational. The former pattern uses inheritance to vary the class that is instantiated, while the latter pattern delegates the instantiation to another object.

2.2.2 Structural Patterns

Structural design patterns are concerned with how the classes and objects are combined to form larger structures. A class structural pattern uses inheritance to order interfaces or implementations. These compositions are fixed at design time. An object structural pattern describes constructs to associate objects in a manner to realize new

functionality. There is added flexibility in the object composition coming in the form of the ability to change the composition at run-time.

2.2.3 Behavioral Patterns

Behavioral design patterns are involved with the rules of behavior and the assignment of responsibilities between objects. Behavioral class patterns use inheritance to distribute behavior between classes while behavioral object patterns use object composition rather than inheritance. Some object patterns describe how a group of peer objects cooperate to perform a task no single object could carry out by itself, others are concerned with encapsulating a specific behavior in an object and delegating the request to this object.

2.2.4 Elements of a Design Pattern

Design patterns usually have a set of elements that describe the pattern and make the pattern a unique identity. These elements of a design pattern are:

- A name,
- The problem that the pattern solves, including conditions for the pattern to be applicable,
- The solution to the problem brought by the pattern, the elements (class-objects) involved, their roles, responsibilities, relationships, and collaborations,
- The consequences of applying the pattern, time and space trade-off, language and implementation issues, effects on flexibility, extensibility, portability,
- To visualize the code a UML diagram is usually present.

2.2.5 Observer Pattern

The observer pattern assumes a one-to-many relationship, where when the one is updated the dependent observers must be updated as well. Some typical examples of this problem are: different GUI elements depicting the same application data (Figure 9) or different objects showing different views of the same application model.

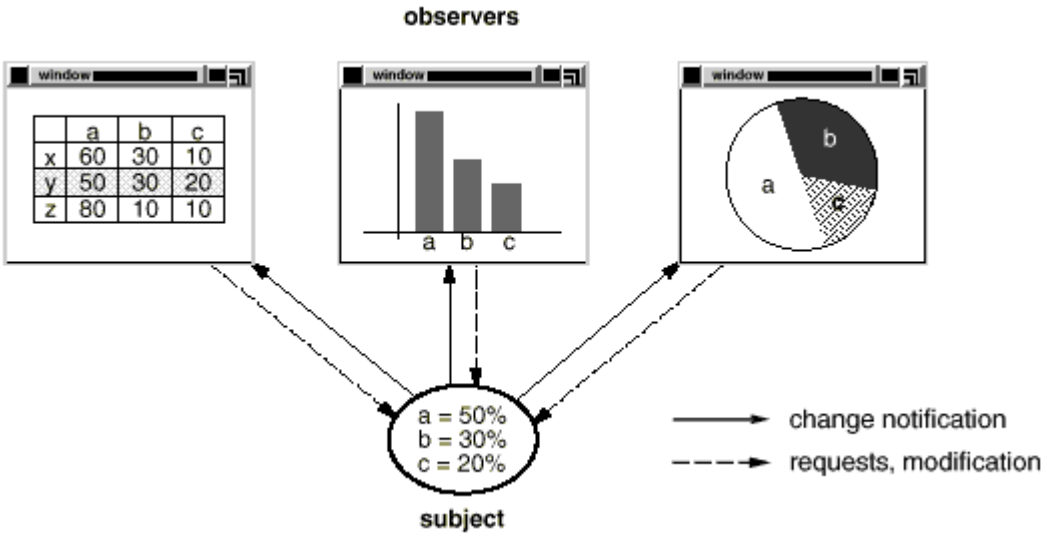

Figure 9: Example of many windows showing different views of the same data ([4])

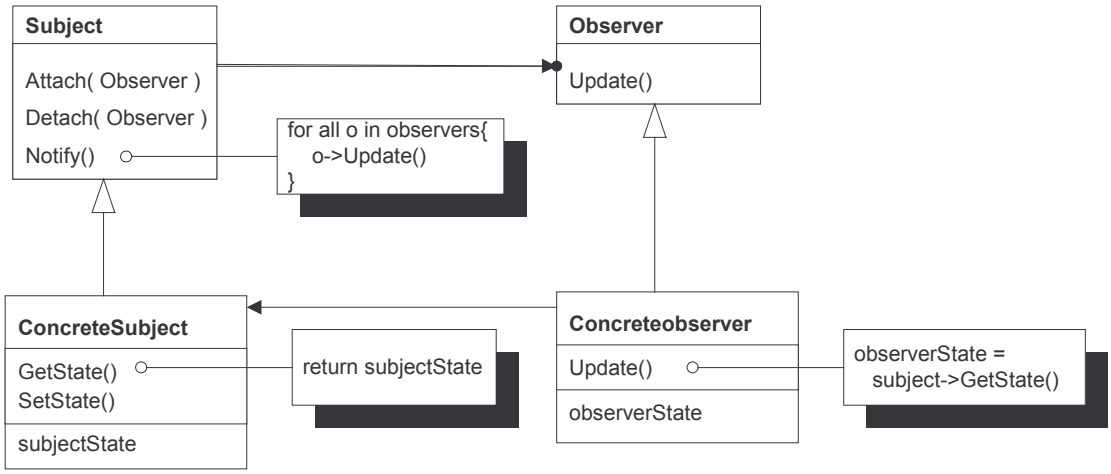

Figure 10: The Observer Pattern Structure

Participants (Figure 10)**:**

- **Subject**: The subject knows its observers and provides them with an interface to attach (subscribe) and detach (unsubscribe) themselves. It also provides a notify method that calls the update function on all of its subscribers,
- **Observer**: Provides an update interface for observers of the Subject,
- **Concrete Subject**: The instantiated class of the Subject, it maintains its state for the relevance of the application, it provides the necessary methods for getting this state and changing it. When a change does occur, calls the notify function to initiate the update of all its subscribers,

- **Concrete Observer**: The instantiated Observer class, it maintains a reference to the Concrete Subject to be able to get its relevant information when an update call is issued.

Collaborations:

Whenever the Concrete Subject is deemed to have changed state, it calls its notify function. The notify will call the update function of each Concrete Listener. Each Concrete Listener will query the Concrete Subject for the information that is relevant to it, thus updating only the needed parts. One of the Observers or any other class can initiate the change. If an Observer initiates the change, this one should not change its own state to reflect the issued change immediately but wait for the update call, which will then put it up to date.

Consequences:

Abstract and minimal coupling between the Subject and its Observers takes place. The Concrete Subject class is not aware of any Concrete Observers; it just uses the interface provided by the Subject. Each class can be reused independently, and the Subject and Observer could even belong to different abstraction layers in the system.

This is a form of broadcasting of messages, the Subject does not need to specify a specific receiver, and it will send it to all subscribed Observers. Though because of the fact the Subject does not know the Observers, sending out an update may sometimes trigger a cascade of heavy updates while a small operation was just executed. Either implementing a Change Manager in between the Concrete Subject and the Concrete Observers or making smart updates can solve this problem.

2.2.6 Singleton Pattern

The singleton pattern is used when one and only one instance of a class is required throughout an application. The pattern also provides a single point of access to the instance of the class.

Participant (Figure 11)**:**

There is only a single participant, the singleton class itself. The class is given the responsibility to create and store its own unique instance. The class defines a static "instance" operation, which lets clients access its unique instance.

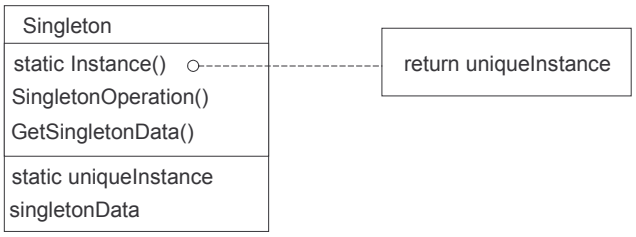

Figure 11: The singleton pattern structure

Collaboration:

The class level instance function will either return or create and return the sole instance permitted of the class. A class level attribute will hold a reference to the unique instance or a default indicating it has not been created yet.

Consequences:

This pattern controls the access to the sole instance of the class and because the class in itself encapsulates the whole pattern, it can have strict control. You can easily modify this pattern to allow a certain number of instances throughout the application instead of just the one. As a side effect the pattern reduces the name space instead of "polluting" it with global variables.

2.2.7 Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Extensively used in the Standard Template Library with its Container/Iterator model, this is a widely used pattern. With it you can support multiple types of transversals of the same data while providing a uniform interface to do it with.

Participants (Figure 12)**:**

- **Iterator**: It defines an interface for accessing and traversing elements,
- **Concrete Iterator**: Instantiated with a reference to the List it is transversing, it keeps the current location with the transversal, and can compute the succeeding jump,
- **Aggregate**: Defines an interface for defining an Iterator object,
- **Concrete Aggregate**: Implements the interface to return the proper Iterator to transverse itself.

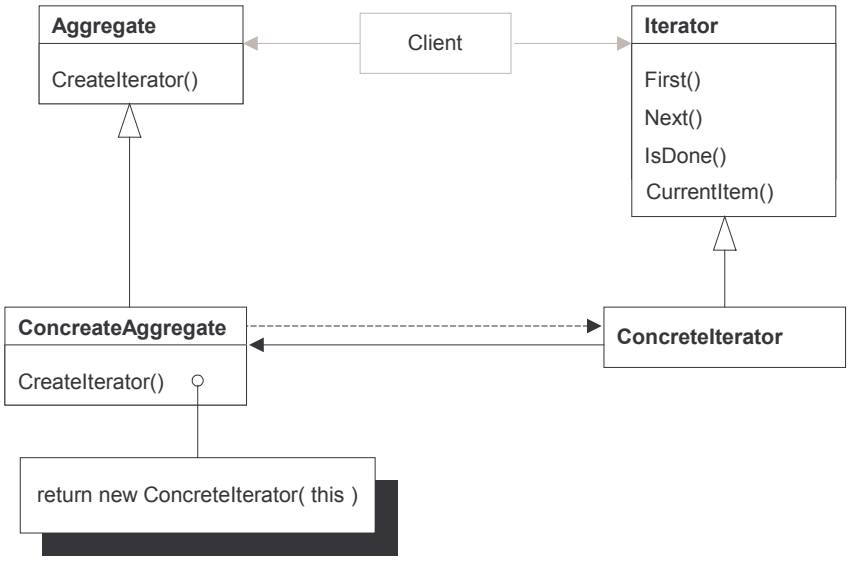

Figure 12: Iterator Pattern structure

Consequences:

With this pattern aggregates can be transversed in many ways by just using a different iterator. Multiple transversals can be done at the same time, all in a simple interface.

2.2.8 Façade Pattern

The Façade pattern provides a unified interface to subsystem to make it easier to use. The Façade is used to provide a simple interface to a complex subsystem, decouple a subsystem from clients and other subsystems, and create layered subsystems by providing an interface to each subsystem level.

Participants (Figure 13)**:**

- **Façade**: knows which subsystem classes are responsible for a request and delegates a client request to the appropriate class,
- **Subsystem Classes**: implement the subsystem functionality, but have no knowledge of the Façade, i.e. they keep no reference to the Façade object,
- **Clients**: clients send their requests to the Façade, which will in turn forward them to the appropriate subsystem object.

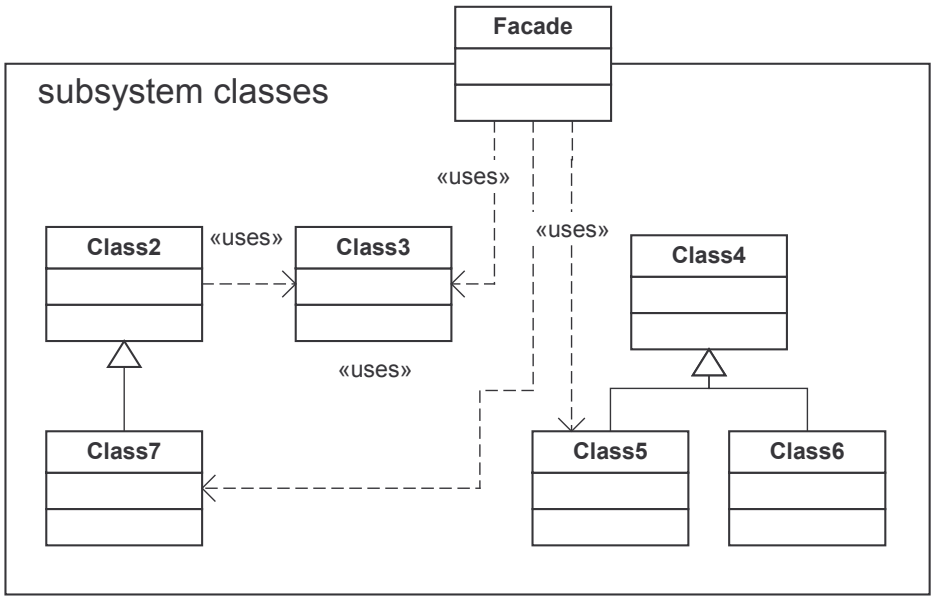

Figure 13: Façade Pattern Structure

Consequences:

This pattern shields clients from subsystem components thereby reducing the number of objects that clients deal with, thus making the subsystem easier to use. It promotes weak coupling between the subsystem and its clients allowing for subsystem modifications without affecting the clients. It reduces compilation dependencies in large compilations. Though it does not prevent the direct access to the subsystem classes if needed.

2.3 Qt ([11])

Qt is a multi-platform C++ GUI toolkit created and maintained by Trolltech[4]. Qt provides application developers with the classes and functionality that is needed to build applications with graphical user interfaces. Qt's features, such as being fully object-oriented, easily extensible, and truly component programming compatible, make it easy to use and extremely powerful and flexible. Being supported on the following platforms:

- MS/Windows -- 95, 98, NT 4.0, ME, 2000, and XP
- Unix/X11 -- Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX and a wide range of others
- Macintosh -- Mac OS X

[4] www.trolltech.com

- Embedded -- Linux platforms with frame buffer support.

and with its "program once, recompile" approach, Qt is truly a multi-platform toolkit.

In GUI programming, often the programmer wants a **widget** or graphical component to be notified of a change in another widget. More generally, objects of any kind want to be able to communicate with one another. Qt provides the **signal and slot** method for this communication. A signal can be emitted from an object anywhere in its code, for example when it changes state. The signal is emitted application wide, and any slot (which is just an object function) in any other object that was "linked" to this specific signal will automatically be executed. Signals can even emit arguments for the slots to use as parameters. The emitter of the signal will not be aware of any of the slots executed because of its signal, but does wait for all slots to return before continuing on with its own execution.

An alternative way of viewing this system: the Qt signal and slot mechanism implements the Observer Object-Oriented Pattern, but on such a scale (any object that has inherited from the Qt QObject class can have signals and slots) that the whole application is part of the pattern. The Pattern is also modeled to include specific calls for Observers to subscribe to so that they are updated only when a certain type of update is performed. This is an extremely useful, easy to use and powerful tool.

For more information on Qt, please consult the relevant documentation in the References section ([6]).

2.4 Graphs

Graphs are a common model for nets and networks. Graphs also have a natural graphical representation. Graphs are based on nodes and edges. A node can be seen as a joint where edges can connect into a pattern. Edges in a graph can have direction, or be bi-directional. Many algorithms can be calculated from a graph, such as exhaustive search, shortest path, minimum spanning tree, matching, and network flow.

A package for using graphs in C++ exists: GTL: the graph template library[5].

[5] GTL: http://infosun.fmi.uni-passau.de/GTL/index.html

2.5 *AOM* ([6])

The project made by Olivier THONNARD, Ir, Antoine HUPIN, and Min ZHANG for the course of Parallel Systems at the VUB, Simulation of an Interconnection Network, and affectionately called AOM, is in fact the first iteration of NTS. NTS is the continuation of the work initiated by this project, and the revision of the code for robust wide user release.

NTS maintains all the design constraints from AOM, addressing some of the issues and recommendations made from the project. NTS keeps some of the old code of AOM, updating it where needed, while sporting some complete rewrites. Mainly the library code has been extended in NTS to overload even more PVM functionality to NTS simulated network. The router routing table package coded by Olivier Thonnard was incorporated into NTS with only minor modification. Updates to the router program for added features of NTS were made.

NTS, though based on AOM, introduces new ideas such as directional links, and message forwarding logging which permit more complex networks to be simulated and more information to be extracted from the simulation. The original Master program was completely recoded, introducing a graphical user interface and many more options for the NTS users. Qt was the toolkit used for the GUI.

For more information on the original release, please consult the original AOM project report.