UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Envisioning Group Policy

for a heterogeneous Linux environment

David O Neill - 0813001

Supervised by J.J Collins

May 10, 2012

Abstract

The objective of this Final Year Project was to develop a Configuration Management framework that supported specification of policies for a Linux heterogeneous environment. The framework had to be extensible, provide a rich user interface, and allow administrators to specify rules using a policy oriented language in the form of a Domain Specific Language. Extensibility was largely achieved through the development of a composite-based architecture, extensive specification of interfaces, and the use of reflection. The end result is an administrative platform that fulfils the functional and non-functional requirements of the project, and an implementation that has been evaluated using stress testing and software quality metrics. The ensuing critique is favourable but balanced, and the software artefact itself was well received by a knowledgeable audience.

Acknowledgements

Everyone and clichès!

Table Of Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview Of The Final Year Project

"Group policy" a term more commonly associated with the "Microsoft Windows Active Directory" is defined as "a set of rules that govern an environments user and computer accounts". Group policy provides a means of centralising the management and therein the control of the configuration, of client operating systems and their features.

Centralised management in a Linux environment is seen as a far more difficult problem. The environment, given the multitude of different distributions, all subscribing to their own implementation philosophies has brought about almost infinite diversity. This requires companies to employ highly skilled technicians to manage these ever changing platforms. These distributions or "flavours", seen to be highly different, however employ the same characteristics in their fundamental implementation; only differing in the tools provided to control these characteristics.

This presents a problem to administrators who wish to apply common rules to all of these operating systems from a central location. A way to overcome this problem would be to provide an abstraction from these differing vendor tools.

Windows Active Directory Group Policy Object (GPO) editor has applied this concept. Since Microsoft is the most successful company in enterprise management, it is deemed that this approach provides the best, well understood solution to this common problem. As each new Microsoft operating system is released it provides the ability to transform rules into operating system specific settings, even though many of the tools may have changed dramatically.

The scope of this document therefore is to design an extensible composite framework that provides centralised management in a Linux environment. A further objective is to realise an abatement of highly skilled technicians, that are knowledgeable in all the varying flavours of Linux, by providing an abstraction in the form of a Domain Specific Language (DSL).

1.2 Objectives

The following section depicts some high level artefacts for the framework under development:

Client Server Model

Centralised management being the theme of this report establishes the inception of the concept of a central authority and therein a client server model. One of the fundamental requirements of such an implementation is to provide a client server model that can operate on a multitude of different systems with varying revisions of supporting packages.

For highest availability possible the client server should be implemented in language that runs on the majority of Linux systems. The client should be able to support updating as new functionality becomes available. This implies it must be extensible, and as such a compiled language would be least preferable as there would be hardware architecture (Alpha, ARM, PPC,

X86 x86_64 & SPARC) concerns in a large environment. An interpreted language such as Perl supports these two constraints.

Administrator Front End

The server or central authority must be controllable in some fashion. As seen in Windows Active Directory, a user interface is provided to accomplish this. As the administrator interface provides the ability of managing the configuration of this evolving environment, it must be extensible. Therefore good software architecture concerns must be considered when designing this interface.

In a large environment it is unlikely that the same policies will be applied to all the machines. The management of a large environment must provide a logical grouping of computers, allowing for the application of different polices to different groups. As an example, a University would have logical distinction between student and staff computers, with different concerns and the resulting policies. This grouping of computers is generally referred to as organisational units.

Directory Services Schema

The logical environmental representations or data must be stored somewhere for easy retrieval and modification. A database typically would be used for such a purpose and the schema should be able to represent computer accounts and be logically divided into organisational units. Policies which will be applied to these organisational units must also be stored in some fashion.

Domain Specific Language

The domain specific language has to represent an abstraction of distribution specific tools and characteristics. Therefore as the operating systems change and support new functionality, the domain specific language must also support change.

1.3 Scope

Given the brief introduction and overview of software objectives, the following scope defines the requirements elicited:

1.3.1 Scope Disclaimer

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119.

1.3.2 Scope Statement

The intent of this project is to provide a Linux port of the Windows Group Policy Object (GPO) editor and a means to deploy these policies to a multitude of Gnus not Unix (GNU) systems through a common non-ambiguous domain specific language (DSL).

1.3.3 Scoped Objectives

Client Server Model

Centralised management being the theme of this project establishes the inception of the concept of a central authority and therein a client server model. The implementation is REQUIRED to provide a client-server model that MUST exhibit the following.

1. The implementation MUST support at least two variants of Linux to prove applicability.
2. The design MUST support the ability to be extended to support a multitude of different systems with varying revisions of supporting packages.
3. The implementation SHOULD use the Practical Extraction and Reporting Language (PERL) for highest availability possible.
4. The server is REQUIRED to scale from two test clients to hundreds of clients.
5. This SHOULD be achieved via the Master Slave Design Pattern or similar pattern.
6. A master with multiple slave servers implementation is RECOMMENDED.
7. A compiled language MUST NOT be used as updates to the client SHALL be sent via the server which will deployed on varying architectures.
8. The Server SHOULD support both Push and Pull; at least MUST implement support for pull requests.

Domain Specific Language

The domain specific language should exhibit the following characteristics.

1. The domain specific language MUST be extensible.
2. As new modules SHALL be deployed to the client, these Modules MAY provides hooks to the interpreter which it MUST accept.
3. The Domain Specific Language(DSL) SHOULD be a implemented as a hybrid language.
    - The interpreter SHALL parse the Domain Specific Language(DSL) and process it.
    - The interpreter SHALL execute embedded general purpose language instructions defined in the Client server model.

Directory Services Schema

The directory services schema MUST provide a means of representing and storing the following elements.

1. Files, Permissions, Lvalue and Rvalue of elements within configuration files.
2. Services and with common names, which MUST be translated by the client into the distribution specific names.
3. A representation organisational groups in hierarchy fashion compared to active directory MUST be implemented.
4. Computer objects SHALL be recognised by a globally unique identifier (GUID) represented as a 32-character hexadecimal string.

Administrator Front End

The Administrator Group Policy editor should exhibit the following characteristics.

1. MUST provide the ability of creating organisational units.
2. MUST provide the ability to move computers between organisational units.
3. MUST Implement a hierarchy of organisational units where by policies MAY be inherited and overrides where applicable.
4. MUST provide a means to modify policies.
5. On application or change of a policy the server MUST generate new domain specific language script.
6. The front end MUST provide a means of importing existing policies in the form of the Domain Specific Language(DSL).
7. The User Interface SHOULD implement the ability to push updates to the client.

1.4 Methodology

The following describes the methodology used for this project:

1. **Review of Existing Systems**
   To determine state of the art with respect to user interface design, tool support, and architectural concerns.
2. **Scoping**
   Specifying project objectives and mapping these to requirements
3. **Prototyping**
   Selecting an architecture and testing it
4. **Implementation**
   Choice of design patterns and development technologies
5. **Evaluation** To discover future improvements

1.5 Overview Of This Report

As a keen software developer the product produced and encompassing report is entirely software centric. The following chapters presents a high level overview of the development artefacts and an evaluation of these software artefacts. Due to the size and nature of the project many areas of software engineering are touched on including networking, software architecture & programming, facets of operating systems and enterprise management. As a result of this, I have tried to keep the report as succinct as possible.

Some discussion provides low-level detailed insights into facets of the implementation that may be of interest to the reader. An overview of the technologies concerned is beyond the scope of this report, and discretion is advised.

2 Configuration Management Concepts

This chapter gives insights into some of the complexities in the problem domain. Given the plethora of topics in configuration management and software engineering a "short history of nearly everything" is unattainable and the choice was to present the following.

2.1 Directory service

A Directory service is a software system that stores, organises and provides access to a hierarchical database known as a Directory Information Base (DIB). This Directory Information Base (DIB) is stored as object classes with each named attribute mapped to one or more values.

These object classes consisting of Organisational Units (OU), Common Names (CN) etc. are identified by a Distinguished Name (DN). A distinguished name is a concatenation of these attributes from a series of entries. A Relative Distinguished Name (RDN) is the tree along the path from the root to the named entry. In the below example we have introduced more of the grammar associated with a Directory Information Base (DIB) look up as well as the syntax.

The Distinguished Name (DN) at the beginning followed by the look up in right to left order. As that of a Fully Qualified Domain Name (FQDN) www.csis.ul.ie, the top of the tree is IE followed by UL, CSIS and so on. In Fig. 2.1 for illustrative purposes I have ordered it from left to right, as this is how most people visualise it.

DN: cn=J.J Collins + uid=j.jcollins, ou=Lecturers, dc=ul, dc=ie

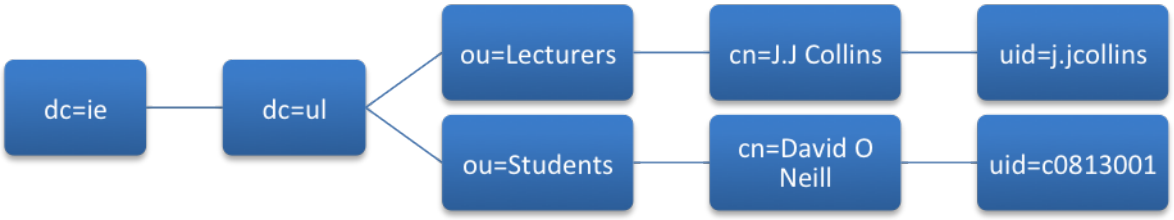

Fig. 2.1: Directory Information Base Lookup

In this diagram we can identify five separate Relative Distinguished Names (RDN).

RDN : dc=ie
RDN : dc=ul,dc=ie
RDN : ou=Lecturers, dc=ul,dc=ie
RDN : cn=J.JCollins,ou=Lecturers, dc=ul,dc=ie.

We can also identify an attribute signified by the plus symbol in the first example.
+ uid which is also an RDN.

RDN : cn=J.JCollins + uid=j.jcollins,ou=Lecturers, dc=ul,dc=ie

2.2 Domain Name System

The Domain Name System (DNS) is a hierarchical naming system for computers, services and any other resource identified by an Internet Protocol (IP) address. It associates names to nodes on a network known as uniform resource locators (URL).

Before the advent of the Domain Name System (DNS), nodes relied on stored name/IP address pairs in a file called hosts. When name lookups were requested, this text file was queried which resolved a name to an IP address. As the networks grow in size so did this hosts file and this became unsustainable.

Two name spaces make up the domain hierarchy, the name address space and the Internet Protocol (IP) address. The domain name server maintains the mappings between the names and the Internet Protocol (IP) addresses. Henceforth the domain name server stores records such as address (A) records, name server (NS) records and mail exchanger (MX) records, as well some 30 other types. When request from a client is not found on a name server it requests it from its name servers thereby creating the physical hierarchy.

Clients make requests to the name server using its Internet Protocol (IP) address. These requests are sent to the server on port 53 in a similar fashion to the way a host would query the hosts file. The server receives the name request, performs the look up and sends the Internet Protocol (IP) address pair mapping back to the client.

This process is further facilitated by the Dynamic Host Configuration Protocol (DHCP) which provides dynamic Internet Protocol (IP) assignments to requesting clients. A Dynamic Host Configuration Protocol (DHCP) client running on a client machine broadcast a request to 255.255.255.255 that is reserved for broadcast traffic.

The Dynamic Host Configuration Protocol (DHCP) server listening for this broadcast traffic sends a response using the same method back to the client consisting of an Internet Protocol (IP) address, a gateway address which is the host that joins to two networks together, the broadcast address for the network and the domain name servers that the client can query for name resolving.

The domain name space is split up logically in zones beginning at the root zone. The sub zones are name spaces which Internet Corporation for Assigned Names and Numbers (ICANN) has delegated administrative responsibility.

Domain names are split up labels concatenated by full stops. For example the right most label in the Fully Qualified Domain Name (FQDN) `www.csis.ul.ie` being IE represents the top-level of the hierarchy to which authority has been delegated to Ireland's Domain Registry (IEDR).

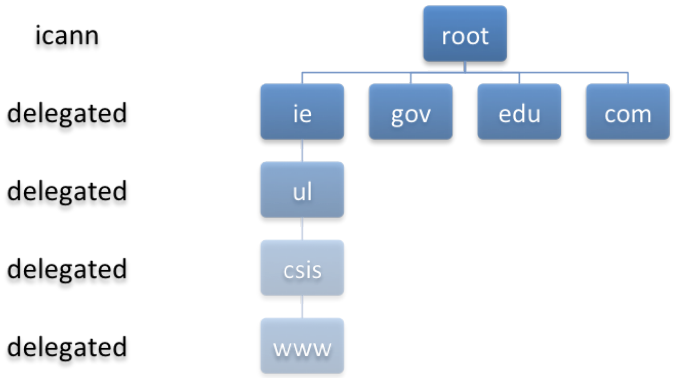

Fig. 2.2: Domain name system hierarchy

2.3 X.500 Specification

The X.500 specification is an amalgamation of computer networking standards for electronic directory services. These standards built upon Open Systems Interconnect (OSI) networking stack were the first approved implementation of the directory services. The X.500 originally consisted of four main protocols, Directory Access Protocol (DAP), the Directory System Protocol (DSP), the Directory Information Shadowing Protocol (DISP) and the Directory Operational Bindings Management Protocol (DOP).

The Directory Access Protocol (DAP) or X.511 standard promulgated in 1998 by Open Systems Interconnect (OSI) is used by client computers as a means of querying the directory services. The query operations including bind, read, list, search, add, compare modify and delete. This protocol has long since been abandoned in favor of use of the Lightweight Directory Access Protocol (LDAP) which was implemented using the transmission control protocol/internet protocol (TCP/IP), which supports extensibility as well as introducing extensive error handling.

The Lightweight Directory Access Protocol (LDAP) introduced amongst the most relevant advancements, Transport Later Security (TLS), the abandon operation allowing the client server to abandon a request and the Extended Operation used to define other operations. Standalone servers soon followed supporting both the Lightweight Directory Access Protocol (LDAP) and Directory Access Protocol (DAP). The next most prominent component of the X.500 specification was the Directory Services Protocol (DSP), which outlines the communication specification between the Directory Services Agent (DSA) and the Directory User Agent (DUA). The directory services protocol controls the interaction between the user agents (clients) and the Services agent (server).

The Directory Information Shadowing Protocol (DISP) described in the X.525 and X.519 specifications outlines the protocols and procedures for the replication of directory information. Replication being the act of copying from one Directory Services Agent (DSA) to another. This operation is generally achieved with the man in the middle approach (chaining and referrals), this ensures synchronisation of data and reduces mismatch between Directory Services Agent"s (DSA) shown below.

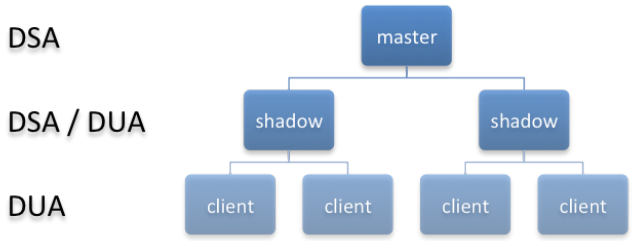

Fig. 2.3: X.500 master and shadow servers

The Directory Operational Bindings Management Protocol (DOP) was proposed to deal with the problem of multiple Directory Services Agent"s (DSA); where no clear defined master could be identified. Unlike the previous example of chaining and referrals the relationship is clear.

A shadow holds a subset of the master directory. If the shadow does not have some information is refers it to the master. But what if we have more than one master? There is no clear chain of responsibility. Therefore a relationship of responsibility between the co-operating Directory Services Agent"s (DSA) must be established.

Two different types of operational binding were standardised, the Hierarchical Operational Binding (HOB) and the Shadow Operational Binding (SOB). These binding states allowed for the disjoint parts of a directory services to conglomerate seamlessly, aware of what each other"s responsibilities were. Consider the below figure.

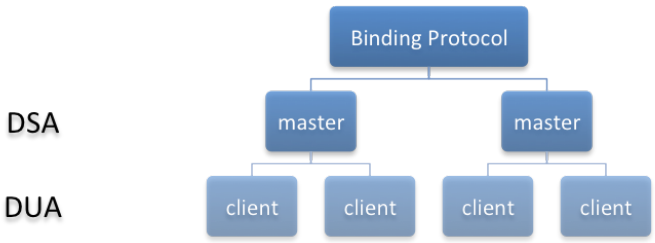

Fig. 2.4: X.500 binding protocol

2.4 Network Information Services

Network Information Services (NIS) previously known as Yellow Pages, is a client server directory services technology used for network logins. The Network Information Services (NIS) server (ypserv) distributes maps of resources, typically Network File System (NFS) shares on a network as well as the contents of "/etc/passwd", "/etc/shadow" and "/etc/group".

A client bound to the server using the (ypbind) client automatically binds at boot. When I user attempts a login on a client a machine the Pluggable Authentication Module (PAM) loads the authentication modules in the order defined in the file "/etc/nsswtich.conf", For eg. "passwd: files NIS LDAP".

If the login information is not found in the local "/etc/passwd" file, the Pluggable Authentication Module (PAM) loads the next login module in this case defined as "NIS".

The maps distributed by (ypserv) are then checked and if the user is found authenticated, if not it then fails onto the next authentication module until eventually failing to login and subsequently notifying the user.

As previously mentioned one of the maps provided by Network Information Services (NIS) being Network File System (NFS) shares are typically also used during the login process. One of the secondary objectives of Network Information Services (NIS) is to make available user resources on any networked machine they login to.

This is done using these maps and another application called "autofs" or "automount". This map of shares defines the location of a user"s home directory in the form of the local mount point and the network resource location where is home files are kept. For example "/home/dave 10.1.1.1:/home/dave".

Active Directory is a directory service technology created by Microsoft and is included with Windows Server. Active Directory serves primarily as a network administration and security backbone, but also provides advanced configuration of client machines in the form of group policy, which is highly sought after in the Linux community and the motivation of this project.

The structure of Active Directory is similar to that of the X.500 structure when multiple master domains are present. These levels are logically divided into Forests, Tree and domains. A tree being the most common implementation of organisations Directory Services architecture is a collection of domains in contiguous space with an implicit transitive trust hierarchy.

For example, Intel Corporation"s tree is split into five domains. Each domain represents a region in the world. If I user from the greater Europe region attempts to access a machine in the American region, the target machine asks the domain controller for that region for authorisation of the users account. As the account does not exist in the American domain the domain controller transitions the login request to the European domain controller which authorises the login, envisioning the trust. The authorisation is then cached on the American domain controller for later login requests.

The domains themselves are further sub divided into Organisational Units. Typically computer objects are grouped together and user accounts are grouped together. These organisational units can be further sub divided into smaller units creating a hierarchical structure. Group policies as previously discussed in 1.1 are attached to these organisational units.

As default each organisational unit has a default policy associated with it. This default policy attributes are not configured as default, meaning nothing will be modified on the target machine or adjusted on a users account. An administrator then modifies this policy, which is applied to the hierarchy top down in an inherited fashion. A top-level policy can be over ridden by a sub-level policy allowing for granular control of the sub units without having to have repeated policies within the hierarchy.

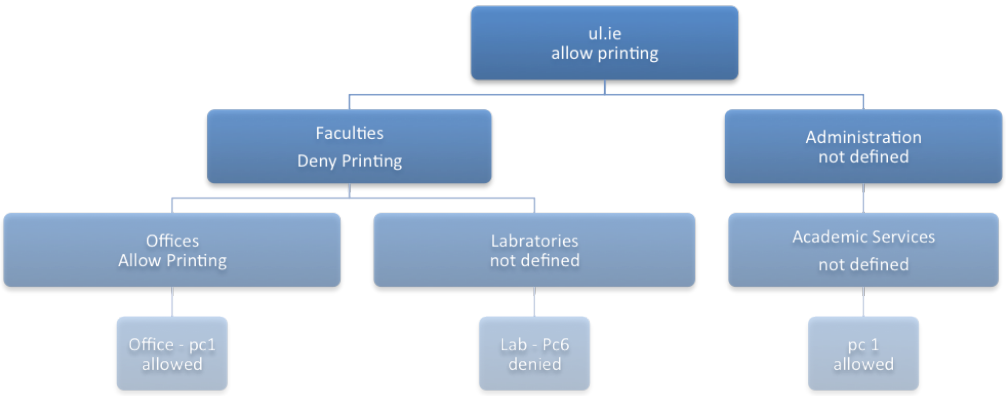

Fig. 2.5: Active directory domain hierarchy

2.5 Linux Configuration Management

Many Linux configuration tools exist for both local and remote administration. There has been varying degrees of success in these solutions most notably with Cfengine.

2.5.1 Cfengine

One of the core issues with Cfengine is the requirement of the administrator to define policies via its Domain Specific Language (DSL) similar to that of Javascript Object Notation (JSON). This notation while catering for a wide variety of configuration possibilities falls short of being truly easy to use as a result of the implementation"s huge vocabulary. Furthermore it still requires expert knowledge in all the varying Linux systems, as each policy has to be coded by handed requiring the administrator to aware of all the varying configurations and implementations of the client machines. There is no front end or point and click interface that comparable in any way of the of Windows Group Policy Editor.

2.5.2 BCFG2

The second most notable configuration management application being BCFG2 provides software provisioning and configuration. The policies are written in Extensible Mark-up Language (XML). Each individual system and its configuration have to be addressed once again requiring a domain expert. The system is pluggable however unlike Cfengine allowing administrators to extend its functionality. No front end for the configuration of the policies exists other than industry standard tools for specifying and validating Extensible Mark-up Language (XML).

2.5.3 Chef

Chef the third most notable Linux configuration manager distributes policies in the form of an internal Domain Specific Language (DSL) (DSL). Cookbooks, which define the policies, are short snippets of ruby code evaluated and executed by the client. This solution although very powerful and does not require the extensive learning of that of Cfengine but does however require the Administrator to be familiar with the ruby general-purpose language. The solution does not abstract the domain specific knowledge set and still requires expert knowledge.

3 Development

In this chapter I will give a brief overview of the key concepts of the software development process. Firstly an overview of the life cycle of the product will be presented, followed by requirements and an introduction to architecture. Finally each major artifact of the product will be analysed separately in terms of (if applicable) :

1. Recapitulation of requirements
2. Technologies
3. Design
4. Architecture & Design Patterns
5. Coding
6. Analysis
7. Improvements

3.1 Life cycle

There are many software development models to choose from when deciding to enter into the software development process. The most basic of these software development models include the Waterfall model, Spiral model, V-shaped model, Iterative model and numerous hybrid models including the Rational Unified Process(RUP) or Model Driven Software Development(MDSD).

The choice depends largely on one key factor, the clarity of the requirements Lenz G (2003), Nurmuliani (2004). From immersive reflection in the Linux management field, as a Microsoft Certified and Network Certified Professional, I had a good understanding of the requirements and the challenges in this project.

The formalisation of these requirements as described in Sec.1.3 (Scope), indicated that many of these software development models were partially applicable to the project and its goals. As hybrid models are based around teams and parallel processes such as in the Rational Unified Process, these models were not applicable to this project.

The choice for a static model was indicated as these offered the best match for the software development process to be carried out. The waterfall model development process includes 6 stages, requirements, design, implementation, testing and validation, deployment and maintenance. This was an appropriate candidate as these processes and their interactions are not typically parallel and can be carried out by an individual.

If the requirements are not well understood or are subject to change, then the waterfall process should be avoided as other models such as the Rational Unified process allow for a degree of change within all stages of the software process being undertaken.

The Waterfall model however has attracted criticisms. One of the main criticisms is the idea of carrying out a stage of a software product's life cycle perfectly and moving onto each incremental stage. However a simple adaptation leading to an iterative waterfall process allowed

me to implement changes to the software product proposed. In Fig. 3.1 the waterfall model presented shows this iterative adaptation. Here we can see that the process was carried out via iterations in design, implementation, testing and deployment.

As the requirements were clearly established before entering into the design, development and testing stages, this mitigated the risks. The risk being the cost of change, that would normally be associated with poorly scoped projects in conjunction with the waterfall model.

The cost of change through each successive stage is an exponential curve that can result in a high monetary cost; if a flaw is discovered in the latter stages of the waterfall model. A change in requirements when in the deployment phase could require extensive modifications to the design, implementation and resulting testing and validation stages.

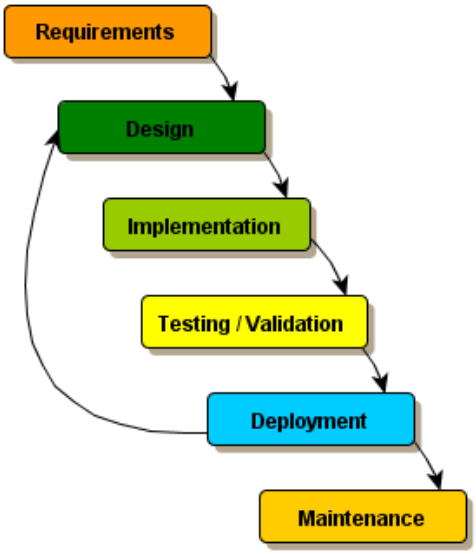

Fig. 3.1: Adapted Waterfall Model

In the next section I will identify these requirements solidified, negating the requirement for change in the resulting development process.

3.2 Requirements

The elicitation of requirements can be obtained from many different techniques. The categorisation of requirements capture methods according to Goguen and Linde (1993) can be divided into 4 main categories, traditional, cognitive, collaborative & contextual techniques.

- **Traditional**
  including introspection, reading existing documents, analysing data, interviews, surveys & questionnaires and meetings.
- **Cognitive**
  including task analysis, protocol analysis & knowledge acquisition techniques.
- **Collaborative**
  including group techniques, workshops, prototyping & participatory design.
- **Contextual**
  including ethnographic analysis, discourse analysis & sociotechnical methods.

Those who have completed human computer interaction studies will be familiar with these techniques. These techniques are outside the scope of this report and I ask the reader to refer to Jakob Neilson and his usability books series, which describe these techniques in detail.

Although no one categorisation was used in the elicitation of the requirements, there was a particular affinity to the cognitive category. I, having previous experience in the role of Unix Systems Administrator in multi nationals corporations, identified with the repetitive processes in

place in these companies. The tasks, protocol compliance and knowledge acquisition incepted the concept and the key requirements of this project. The following marchitecture (or marketecture) presents the key requirement artefacts.

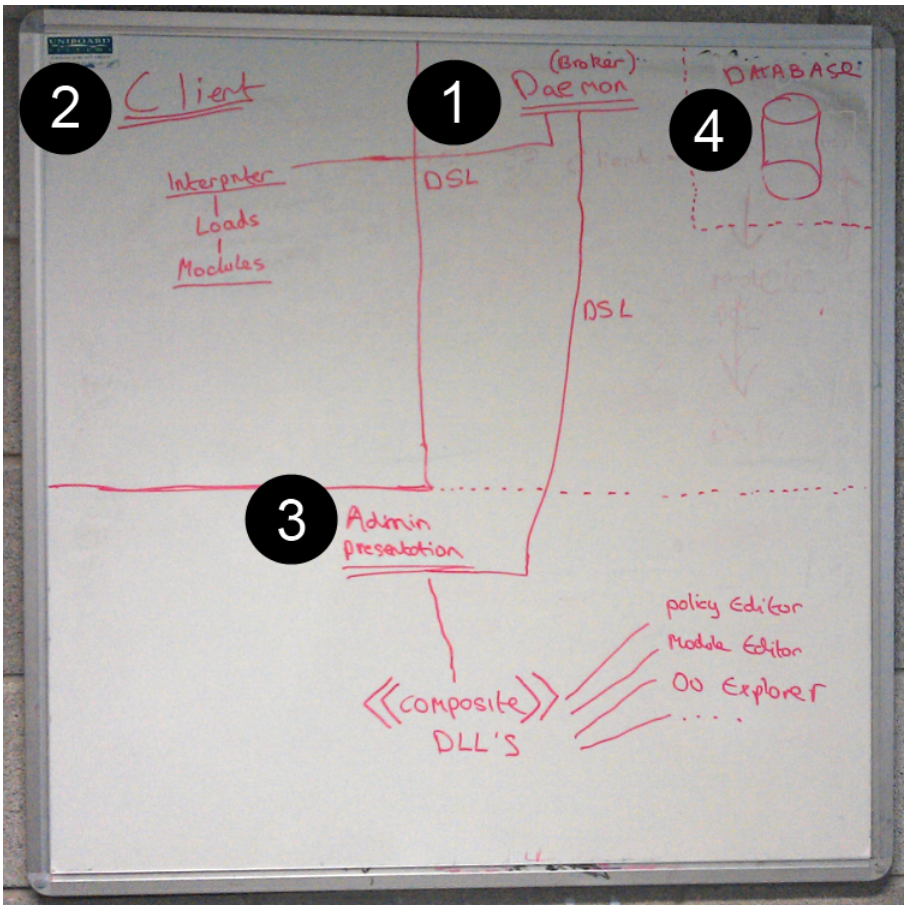

Fig. 3.2: Marchitecture

To begin to understand the requirements and the resulting implementation, Fig. 3.2 presents a marchitecture of a high level architectural design. The marchitecture indicates the four main artefacts that support the requirements.

1. **Daemon (Broker - Server)**
   - Broker between the admin presentation and the clients
2. **Client**
   - Interpreter which parses the domain specific language (DSL)
   - Interpreter which loads modules to extend the parsing (rules) capabilities of the interpreter.
3. **Admin Presentation**
   - Linux Group Policy Administrator interface
   - Composite architecture
   - Creates rules in the form of a domain specific language (DSL)
4. **Database**
   - Ultimately stores all domain specific data
   - Policies (domain specific language)

A recurring artefact here is the domain specific language (DSL). So I will provide a fifth section

5. **Domain Specific Language (DSL)**
   - How rules are specified through a domain specific language

3.3 Architecture

In the words of Martin Fowler, "The software industry delights in taking words and stretching them into a myriad of subtly contradictory meanings, one of the biggest suffers is architecture" - Fowler (2010b). This is the perfect quote to describe the problem of defining what software architecture is. The reader, you, has just read that line, interpret it and took your own profound meaning from it. This a similar to the scenario of identifying what is deemed to be structurally relevant in a software product. From a high level perspective most people can identify the main structural components of a software product, however as the analysis continues, disagreement lurks, and the resulting definition gets blurred.

As architecture is an inherited word from the construction industry, I'm going to go back to the earliest written works on the definition of architecture. De Architectura by the roman architect Vitruvius, said architecture should satisfy three main principles, firmitas, utilitas, venustas, or durability, utility and beauty.

Dan known as "the father of the spreadsheet" wrote a paper, "software that last 200 years", says that we need to start thinking about software in a way that mimics the construction of bridges, dams and sewers. Since software has only been around for a fraction of this time it's hard to determine whether the practices currently employed in software construction are going to stand the test of time. There is still Fortran code(1956) running today as well as Lisp(1958); however, the only product still running and constantly being evolved is Unix(1959). Dan Bricklin makes the argument, a software's durability is a product of its marketing strategy and that bridges would not be very successful if we tore it down every 10 years. Unix and its derivatives are successful, along with the GNU (Gnus Not Unix) utils, e.g Emacs 1978.

This brings us to utility, if Unix were not useful, it and its derivatives would have died out long ago. Its usefulness, in all manner of products from embedded systems to desktop personal computers and mainframes has been proven. Vitruvius says that it should be useful and provide function to the people using it. Beauty, the final principle of architecture according to Vitruvius is concept that changes with culture and time. The austerity of time proven techniques compared to new paradigms, new design patterns, and the concept of beautiful software architecture seems to change like the weather.

To satisfy the reader I will finish up with a definition of software architecture according to the most successfully marketed software developers Microsoft, *"Software application architecture is the process of defining a structured solution that meets all the technical requirements, while optimising common quality attributes such as performance, security and manageability"*

3.3.1 Software Architecture

There are many influencing software solutions that have influenced my thinking on software architectures and how they are deemed successful, but primarily useful to me and the end users. Software that exhibits modular or pluggable characteristics allows for a richer user experience. Therefore the development of a software product that exhibited these characteristics was a primary concern. Such frameworks as JQuery for Javascript, .Net for rapid application development and

CodeIgnitor for PHP have significantly influenced me. So much that I no longer look to reinvent the wheel but try in most scenarios to avail of mature already established frameworks to create software.

3.4 Daemon - (Broker - Server)

The preferred name for a "server" in the Linux domain is "daemon". The concept of a "daemon" is not unlike the concept of a "server", however using the term "daemon" negates the ambiguity associated with a "server process" running on a "server computer". Throughout this section the term "daemon" will be used in regards to the process and the term 'server' will be used in reference to a "daemon" running on a server computer.

Furthermore the term "broker" creates yet more ambiguity, in that it is associated with an architectural pattern for the message validation, transformation and routing. For the sake of simplicity the term "broker" will be used to convey the facilitated communication amongst peer processes, where a "daemon" in the middle between two computers facilitates this, effectively decoupling them from one another.

3.4.1 Requirements

1. Support for multiple clients
2. Use of PERL
3. Scalable to handle 100's of clients
4. Master with multiple slave servers.
5. Push & Pull technologies

3.4.2 Technologies

- **PERL threads**
  PERL threads were used extensively in the application for the client server model to gain maximum performance.

- **PERL threads::shared**
  To shared variables amongst threads.

- **PERL Thread::queue**
  Queues were used extensively in both the client and the server. The server has an incoming queue, processing queue and an outgoing queue. The queues are thread safe and are first come first served basis ensuring ordered processing.

- **PERL Thread::Semaphore**
  Certain critical sections in the client and server where identified and errors avoided by using semaphores.

- **PERL IO::Socket**
  A low level socket implementation was preferred for maximum performance.

- **PERL POSIX**
  Portable Operating System Interface (POSIX) was used in the application for the signalling of threads and user interrupts which is a small subset of the POSIX API.

- **PERL JSON**
  For message passing in the implementation between the nodes this was the preferably choice for message passing.

- **PERL Sys::Syslog**
  Syslog the de-facto means of logging in Linux and as such the choice to use Sys::Syslog to log client - server interactions was required.

- **PERL DBI**
  This database abstraction layer decouples the programmer from specific database API. As it is a wrapper a common database connection string is used to specify the database type and the abstraction layer takes care of the specific API calls. As such this allows for the use of most common database technologies, such as hierarchical, relation and object oriented databases.

3.4.3 Design

Although not subscribing to any particular design when designing the client server architecture, there were multiple influencing sources that lead to the architecture shown in Fig. 3.3. The book *Pattern Oriented Software Architecture - Patterns for concurrent and networking objects* was a key influence on this simplified design, specifically with reference to the key aspects of the "interceptor" and "leader / followers" design patterns.

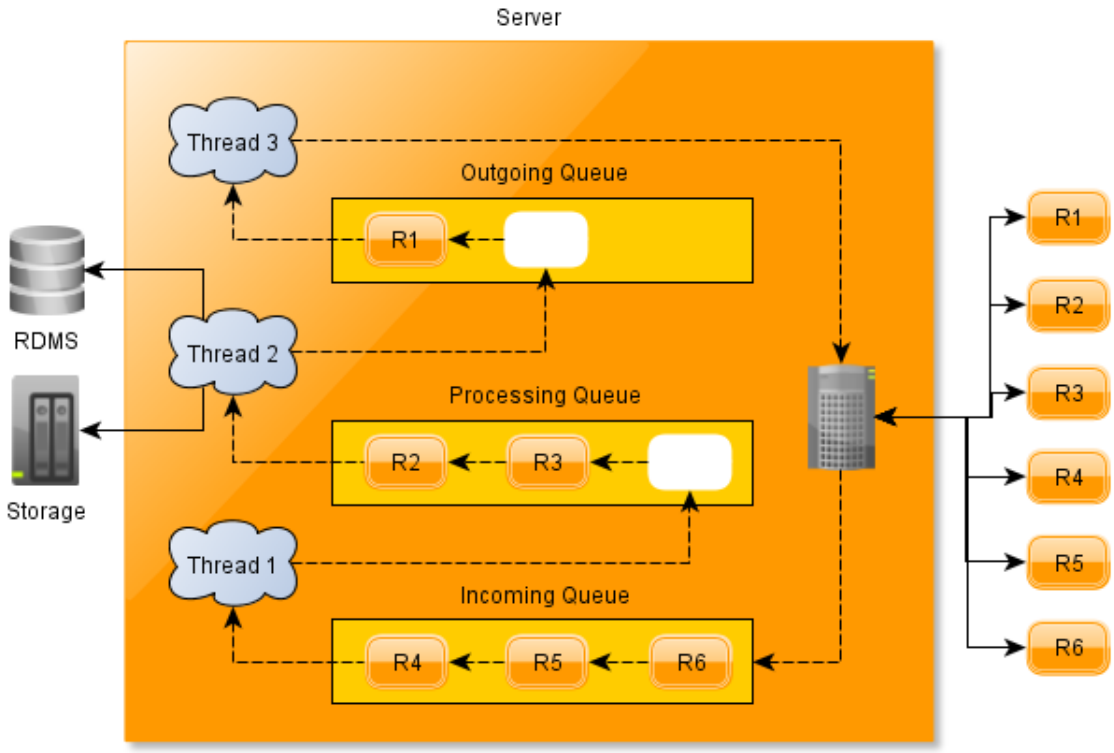

Fig. 3.3: Server Queues

The architecture presented shows (R)equests moving through the internal data lanes of the implementation. Three queues are the main transition points within the architecture, namely the incoming, processing and outgoing queues and the associated threads 1, 2 & 3. These requests internal messages can be divided into two major categories, Administrator and Client requests.

- **Administrator Requests**
  - Module List
  - Save / Update Module
  - Synchronise Policies
  - Push Policies to clients
  - Push Modules to clients
- **Client Requests**
  - Module Update
  - Policy Update

Although not immediately indicated the requests are asynchronous, meaning that the connection to the client is dropped as soon as the request is accepted. By analysing the diagram we can also see that the queues enforce the ordering of the events and subsequent resource interaction, such as database access or disk access.

The incoming queue is important as requests must be handled as fast as possible when implementing a server. If too many connections to a socket are open (IP address bound to a port eg. 127.0.0.1:80) there is the possibility of the server reaching the maximum allowed by the operating system. This in turn this would give a busy signal to the subsequent clients trying to connect. Connections that are accepted are immediately inspected for its request and this request is put into the incoming queue. Finally thread 1 is responsible for determining whether the messages is well formed before being placed into the processing queue.

The processing queue is the next main transition for a request. In the processing queue the requests internal message is analysed by thread 2 and subsequent processing is done depending on the request. Once the processing is complete the request is put into the out going queue.

Finally Thread 3 and the associated outgoing queue is responsible for sending a response back to the client/administrator where applicable. It is worth point out at this stage that the envelope in which the request is encapsulated, holds information such as the peer IP address, peer port, peer request etc.

With this information and the daemon response inserted into the request envelope by thread 2, the outgoing queue is then processed by thread 3. Thread 3 opens a connection to the destination and sends the response to the requester.

3.4.4 Architecture

An introduction to software architecture by Garlaw D (1993) makes reference to state transitions as an important aspect of the architectural design. Fig. 3.3 on the previous page offers three primary state transitions.

- **Incoming**
- **Processing**
- **Outgoing**

As a request passes through these data lanes or queues, the action of moving from one state to another offers pre marshalling and post marshalling capabilities. We can extend the transitions as follows :

- Incoming
  - **Pre incoming**
  - **Post incoming**
- Processing
  - **Pre processing**
  - **Post processing**
- Outgoing
  - **Pre outgoing**
  - **Post outgoing**

Each of these states offers transition points and the ability to include extensions to the framework to target specific requirements. As an example of this, the implementation of the architecture allows for the addition of different security extensions for the both the pre incoming and pre outgoing transitions. In an example scenario an encrypted request that has been accepted into the incoming

queue must first be decrypted before processing of the encrypted data can be accomplished. Similarly the encryption of the outgoing response can be handled at the pre outgoing transition. The following eight steps presented can be are identified in Fig. 3.4.

1. Encrypted peer request is received
2. Request is added to the incoming queue
3. Thread 1 processes next request on the incoming queue
4. Encryption extension decrypts message **(pre marshalling)**
5. Thread 1 places decrypted request into processing queue **(post marshalling)**
6. Thread 3 processes next item on out going queue
7. Encryption takes place on the out going message **(pre marshalling)**
8. Thread 3 sends the message to the peer **(post marshalling)**

This concept of pre and post marshalling effectively allows for the integration of different extensions to cover a multitude of future requirements and upgrades. These extensions could provide concepts such as load balancing, priority queues, multi-threaded queues and integration with other directory service systems.

Due to the time constraints of the project the decision to implement the daemon in this fashion provides the ability to satisfy the requirement of a master slave paradigm. In an example scenario, the master server could provide load balancing by re routing incoming requests to slave daemons running on other servers. The potential resource heavy and time sensitive processing operation is then passed and balanced among these slave servers.

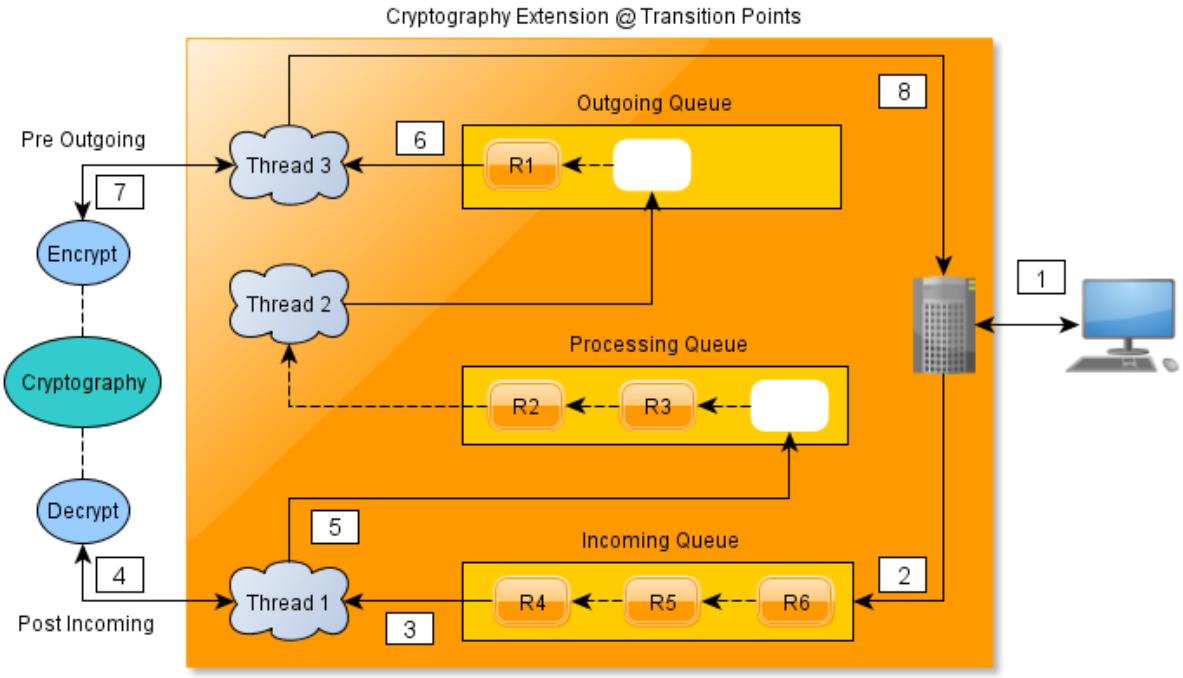

Fig. 3.4: Transition states marshalling

3.4.5 Coding

The discussion of the design and architecture covers the following five technologies. These cover the basic concepts of threading and critical sections, first in first out (FIFO) queueing data structures and key concepts of socket programming.

- **PERL threads**
- **PERL threads::shared**
- **PERL Thread::queue**
- **PERL Thread::Semaphore**
- **PERL IO::Socket**

All of these topics are huge topics in themselves and are well understood fundamentals in computer science. Therefore in this section I'm going to cover the more opaque technologies and provide coding snippets to support the reasoning for the choice of these technologies.

- **PERL POSIX**
- **PERL JSON**
- **PERL Sys::Syslog**
- **PERL DBI**

POSIX

The Portable Operating System Interface (POSIX) is an application programmers interface for maintainability compatibility between operating systems. The use of POSIX in the implementation was primarily used for thread signalling and ensuring the clean shut down of the daemon.

In order to communicate with the daemon from an administration point of view where an administrator is logged onto the server some sort of inter-process communication is required. For example, rather than abruptly killing the server daemon with a "kill -9 daemon" command, named pipes were used to send this messages to the daemon.

Named pipes are a basic form of inter process communication in Unix, Linux, Windows and Mac operating systems, however the semantics and resulting implementations differ. In Unix the implementation of this concept is achieved via use of the file system.

The mkfifo command on Unix and derivative platforms, makes use of the POSIX API to signal the kernel to prepare a file on the file system that acts as a first in first out queue, for inter process communication. Piping information into this file is placed in shared memory. The receiving process can then read this file (effectively the memory) achieving inter process communication. An example of this is given in Fig 3.5.

```
# where $self->{ _pipe } is /directory/filename

if( -e $self->{ _pipe } )
{
  system( "rm -rf " . $self->{ _pipe } );
}

system( "mkfifo " . $self->{ _pipe } );

if( sysopen( FIFO , $self->{ _pipe } , O_RDWR ) )
{
  while( ${ $self->{ _continue } } != -1 )
  {
    my $input = <FIFO>;
    chomp( $input );

    if( $input eq 'stop' )
    {
      ${ $self->{ _continue } } = -1;
      last;
    }
  }

  close(FIFO);
}
```

Fig. 3.5: Creating and reading a named pipe

In this example a named pipe is created and then opened for reading and writing. As soon as data is piped into the file, this line is read using the chomp Perl function. The line is inspected for keywords, in this case "stop" and the internal variable "continue" is set to minus one. All other threads within the application implement looping constructs based upon this "continue" variable. When "continue" it is set to minus one, this loop constructs discontinue, illustrated by the coding snippet in Fig. 3.6.

```
while( ${ $self->{ _continue } } != -1 )
{
  while( $queue->pending )
  {
    #work on queue
  }
}
```

Fig. 3.6: Threads discontinue processing

The second use of signalling is for the scenario where interrupts such as a kill signal is sent to the daemon process. As an example I will provide a simple scenario of running the daemon in the shell console context. Typically when a daemon is launched from within the context of a shell it detaches itself from the console. However if it does not do this, when the shell is finished or is closed it kills any processes which it has launched. This concept is familiar to programmers who write managed code where the garbage collector removes unused or unreferenced artefacts in memory.

Running a daemon in the shell context present a problem. Cleanly shutting down the process is

important when the cancel or kill signal is fired. This is typically done on systems using "CTRL ^C". To ensure clean shut down and the termination of threads, this signal must be "caught". Threads must be told to shut down and finally terminate the program.

This is accomplished via POSIX, terminals typically implement POSIX and encompassing signalling. So when "CTRL ^C" is pressed the terminal catches the signal and terminates the application. Therefore we must intercept this signal and handle this terminal event. PERL provides POSIX support and with this we can intercept the kill signal; Fig. 3.7 presents this.

```
$SIG{ HUP } = \&signal_interrupt; # hang up signal
$SIG{ INT } = \&signal_interrupt; # interrupt signal eg. ctrl ^ c
$SIG{ KILL } = \&signal_interrupt; # terminate immediately
$SIG{ TERM } = \&signal_interrupt; # Termination (request to terminate)

sub signal_interrupt
{
  use vars qw( %options $pid $server );

  $SIG{ INT } = \&signal_interrupt;
  $server->stop();
  $pid->destroy();

  sleep 1;
  exit(0);
}
```

Fig. 3.7: PERL Signals

In this example "SIG" which is a global variable provided by the POSIX module in PERL provides a means to map a signal to a closure or method stub. When the "INT" signal is received the "signal_interrupt" is called which tells the server to shut down. In order to override repeated "CTRL ^C" events we re-map the INT signal once again to the "signal_interrupt" closure. Finally the client sleeps for 1 second, this typically will allow threads to shut down before the application terminates.

Context switches in the kernel are guaranteed to be no more than 500ms. This duration allows all attached threads to shut down or terminate before the application exits. Typically in a modern processor, context switches can occur up 20,000 times per second. This means that this amount of threads could be terminated in this time allotment. In the implementation the stop method in the daemon class has more than enough time to achieve this goal. An excerpt of this stop method is provided in Fig. 3.8.

```
sub stop
{
  if( defined( $self->{ _receive_thread } ) )
  {
    $self->{ _receive_thread }->kill( 'KILL' )->detach();
  }

  exit(0);
}
```

Fig. 3.8: Stop Server Threads

JSON

Javascript Object Notation (JSON) is a lightweight data-interchange format for the transfer of platform independent communication. The name may suggest that it is for Javascript only, however most of the 3rd and 4th generation programming languages implement JSON parsers. The choice of JSON over the typically used Extensible Markup Language(XML) was the syntactic verbosity factor. There are examples on json.org that indicate XML can be 300% more verbose than JSON. In a system that is dependent on speed and efficiency, the smaller and hastily parsing of messages is important. Therefore JSON was a good candidate for encapsulation of messages due to its' lightweight envelope. Initially I had devised my own short form syntax for messages however as the implementation grew and the message became more complex, the need for a better envelope and formatting of messages was required.

```
$request = encode_base64( "{

  \"peerPort\" :   \"" . $self->{ _clientport } . "\" ,
  \"request\"  :   \"update\" ,
  \"guid\"  :   \"" . $computer->getGuid() . "\" ,
  \"os\"  :   \"" . $computer->getOs() . "\" ,
  \"version\"  :   \"" . $computer->getVersion() . "\" ,
  \"architecture\"  :   \"" . $computer->getArchitecture() . "\" ,
  \"kernel\"  :   \"" . $computer->getKernel() . "\" ,
  \"psuedoName\"  :   \"" . $computer->getPsuedoName() . "\" ,
  \"distribution\"  :   \"" . $computer->getDistribution() . "\" ,
  \"distroVersion\"  :   \"" . $computer->getDistroVersion() . "\" ,
  \"hostname\"  :   \"" . $computer->getHostname() . "\" ,
  \"clientVersion\" :    \"" . $self->{ _clientversion } . "\"

}" );
```

Fig. 3.9: JSON Client Request

In section 3.4.4 identified the messages that the server dealt with, one of them being a client update. As an example Fig 3.9 presents a request from a client for a policy update. The key "request" holds the value or the intended request to be sent to the server.

Syslog

Syslog is a data logging standard used by Unix and derivatives as a means to decouple processes from operating system logging mechanism. These logs can then be checked by administrators of the environment and reports made from them. In order to debug daemons where the standard output is not visible as the daemon is not attached to a console, a means of logging messages is required. The integration with the system implementation logging features is expected and as such the choice for Syslog was mandatory.

An entry is made into the Syslog configuration. The entry is then used to create a pipe by Syslog to which process can redirect messages to; resulting in the corresponding file.

local6.* /var/log/lgp

The following PERL excerpt in Fig. 3.10 shows the use of this logging mechanism and the corresponding log file excerpt below it.

```perl
use Sys::Syslog qw( :DEFAULT setlogsock);

sub log
{
  my( $self , $level , $message ) = @_;
  openlog( 'cmain.pl' , 'ndelay,nofatal,nowait,perror,pid' ,  'local6' );
  syslog( $level , $message );
  closelog();
}

# $level =
# LOG_ALERT - action must be taken immediately
# LOG_CRIT - critical conditions
# LOG_ERR - error conditions
# LOG_WARNING - warning conditions
# LOG_NOTICE - normal, but significant, condition
# LOG_INFO - informational message
```

Fig. 3.10: PERL Syslog

Mar 29 17:16:25 testmachine2 cmain.pl[3657]: Client created tcp connection on 192.168.1.10:50000

DBI

As the back end of the architecture depends on database for storage, a major concern is the subscription to a vendor specific database. This vendor database may in the future be deprecated and the need to migrate to a new database vendor may occur. The PERL DBI (PERL Database Interface) offers an abstraction for programmers using the PERL programming language from vendor specific database implement ion application programmer interfaces. Programmers who are familiar with connection strings such as that used in .NET of ODBC will be familiar with this concept. Other languages such as PHP provide the PHP Data Objects (PDO) as a comparable abstraction. This abstraction layer allows for the choice of database based upon a connection string. In Fig. 3.11 the connect stub of the database PERL module in the implementation provides the ability to change the database type at rune time.

```perl
sub Connect
{
  my( $self , $dbtype , $host , $database , $username , $password ) = @_;

  $self->{ connectionString } = 'DBI:DBTYPE:DBNAME;host=DBHOST';
  $self->{ connectionString } =~ s/DBNAME/$database/m;
  $self->{ connectionString } =~ s/DBHOST/$host/m;
  $self->{ connectionString } =~ s/DBTYPE/$dbtype/m;
  $self->{ connection } = DBI->connect( $self->{ connectionString } , $username, $password );

  if( defined( $self->{ connection } ) ) {
    $self->{ _log }->log( 'info' , "LPGServer $dbtype connection made successfully" );
    return 1;
  }
  $self->{ _log }->log( 'err' , "LPGServer $dbtype connection failed" );
  return 0;
}
```

Fig. 3.11: PERL DBI

3.4.6 Analysis

Many concerns arise when attempting to do an analysis of the daemon; namely speed, scalability, extensibility, maintainability, portability and security. There are many topics. In this analysis I'm going to tackle the speed concerns, and the resulting scalability.

Stress testing was done on the implementation as a means to elicit empirical evidence to validate the requirements, namely "Scalable to handle 100's of clients". A stress test harness was setup to address this concern. The stress test consisted of six virtual machines, one server and five clients running on a single computer. Table 3.1 presents the test machine characteristics.

Test Machine	**Cores**	**Threads**	
Intel Core I7 965 @ 3.2 GHZ	4	8	
	Ram		
DDR3 PC3-10666 Triple Channel	16 Gigabytes		
	IOPS	**Form Factor**	**RPM**
OCZ RevoDrive	70000	PCI-Express RamDrive	N/A
Vmware Virtual Machines	**Ram**	**Threads**	**Function**
Redhat 32	2 Gigabytes	2	Daemon
Fedora 32	2 Gigabytes	2	Client
Fedora 64	2 Gigabytes	2	Client
Gentoo	2 Gigabytes	2	Client
Ubuntu	2 Gigabytes	2	Client
Open Suse	2 Gigabytes	2	Client

Table 3.1: Test Machine

It was important to negate all possible bottle necks in the test equipment. To achieve characteristics of enterprise level server performance. I negated the hard drive bottle neck that would be associated with personal computers. As in an enterprise environment, "filers" would be used as a data centres backbone for storage; due to their high performance.

Making the comparison between OCZ RevoDrive in table 3.1 and that of consumer grade hard drive is presented in table 3.2. The form factor presented indicates approximately 14 times faster hard drive access speed.

Comparison	**IOPS**	**Form Factor**	**RPM**
Sata 3 Hard Drive	150	Sata 3 HDD	7200
Sata 3 Solid State Drive	5000	Sata 3 SSD	N/A

Table 3.2: Hard Drives Comparison

The test composed of 500 requests to the daemon to push out updates to the clients, each with a delay of 30ms between each request. In effect this meant that there were 500 + (5 * 500) equalling 3000 asynchronous requests.

A sampling interval of 1 second was used to capture information, such as processor, queueing and network loads. The resulting data is shown in Fig. 3.12, 3.13 and 3.14 respectively.

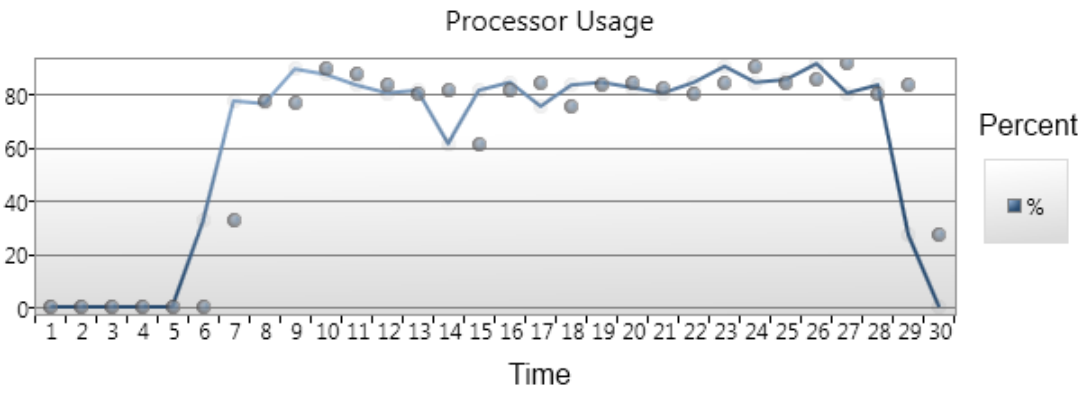

Fig. 3.12: Server CPU Usage

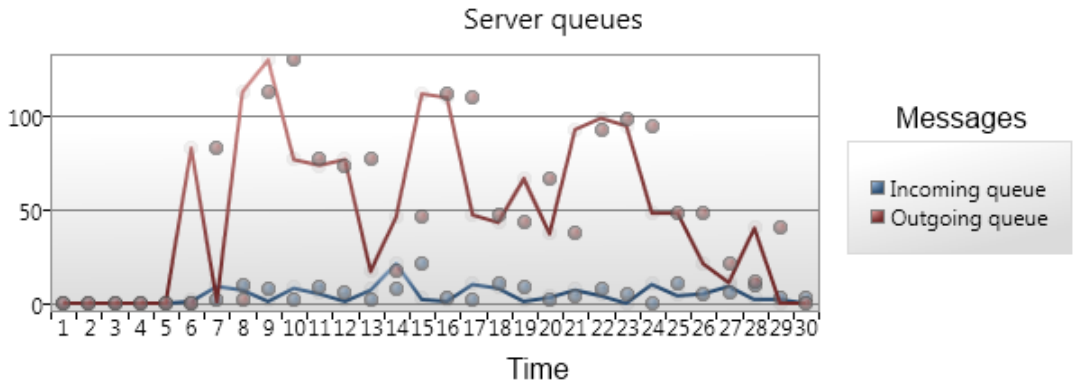

Fig. 3.13: Server Queues Usage

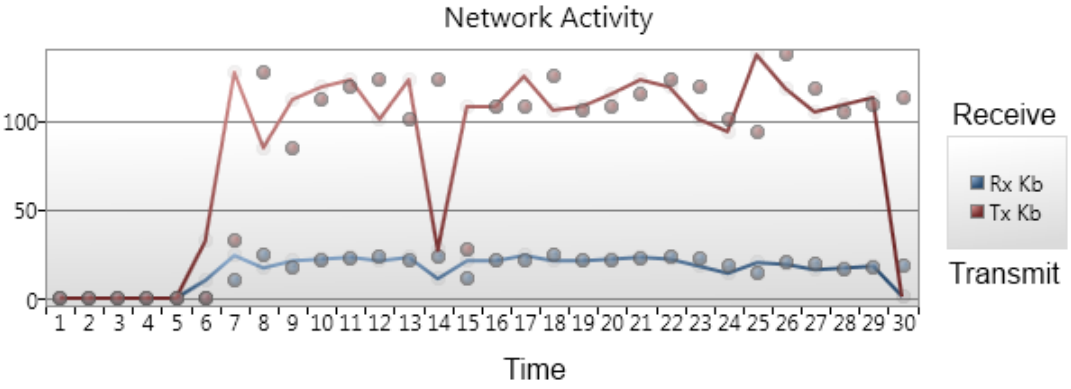

Fig. 3.14: Server Network Usage

The test took approximately twenty five seconds, equaling one hundred client updates per second. In the implementation however, the client checks for updates every twenty minutes. As there is 1200 seconds in this time, potentially under optimal conditions the server could handle

120,000 clients. This hypothetical scenario is based upon a policy that equates to approximately one kilobyte and optimal testing conditions. Under real conditions this scenario may not be feasible, due to larger policies and possible network degradation. The achievement of 5% of this performance this still equates to approximately six thousand client updates, in this twenty minute interval.

3.4.7 Improvements

In *Patterns of Enterprise Application Architecture* by Martin Fowler, he makes numerous references to case studies were performance testing presented bottle necks in string processing. The resulting degraded application performance was in some cases diabolical and light weight changes resulted in 100% increased efficiency. In the server architecture the daemon makes use of numerous text conditionals and JSON. Ultimately these text conditionals need to be lexicographically parsed, resulting in high processor usage as indicated by figure 3.12.

To begin to target this contention in future improvements, the use of JSON as an envelope for message passing might be dropped. Initially during development I did not use JSON, however as the complexity grew a need for a structured message was required, as sequenced events became difficult to maintain.

Furthermore as the client sends requests as text, these text conditionals need to be lexicographically inspected and compared by the daemon to create the correct response. These text conditionals could be replaced by integer values that require very little computation in terms of comparison by a processors arithmetic logic unit (ALU).

3.5 Client

In this section the discussion of the client; it's operations and supporting technologies will be consummated. along with a brief introduction to the transliteration of the domain specific language.

The client implementation is responsible for downloading a policy from the daemon and executing it. The execution of the policy, a domain specific language (DSL) is the application of the rules defined by the administrator. The client determines the operating system distribution it is executing on and through the use of an interpreter, it applies the policy via distribution specific commands.

3.5.1 Requirements

1. Support for multiple distributions
2. Client must be extensible
3. Use of PERL
4. Updates to varying architectures
5. Push & Pull technologies

3.5.2 Technologies

- **PERL threads**
  PERL threads were used extensively in the application for the client server model to gain maximum performance.
- **PERL threads::shared**
  To shared variables amongst threads.
- **PERL Thread::queue**
  Queues were used extensively in both the client and the server. The server has an incoming queue, processing queue and an outgoing queue. The queues are thread safe and are first come first served basis ensuring ordered processing.
- **PERL Thread::Semaphore**
  Certain critical sections in the client and server were identified and errors avoided by using semaphores.
- **PERL IO::Socket**
  A low level socket implementation was preferred for maximum performance.
- **PERL POSIX**
  Portable Operating System Interface (POSIX) was used in the application for the signalling of threads and user interrupts which is a small subset of the POSIX API.
- **PERL JSON**
  For message passing in the implementation between the nodes this was the preferably choice for message passing.
- **PERL Sys::Syslog**
  Syslog the de facto means of logging in Linux is assumed by all major daemon installations and as such the choice to use Sys::Syslog to log client server interactions was required.
- **PERL Filter::Simple**
  Source filtering or pre processing is an immensely powerful feature of PERL. Effectively, it allows or provides the ability to create micro languages such as a DSL by preprocessing the input script translating it as necessary.

3.5.3 Design & Architecture

The client operates in the same way as the server architecture shown in Fig. 3.3 with the minor exceptions. The two main differences of the client are the messages and the operations done within the in the processing queue. The design analysis in this section will be therefore focused upon these operations and the encompassing interpreter. As the interpreter facilitates the parsing of the domain specific language (DSL) resulting in distribution specific commands, the resulting architectural design evolved to support the extensibility of this framework and the interpreter. Fig. 3.15 presents a high level design overview.

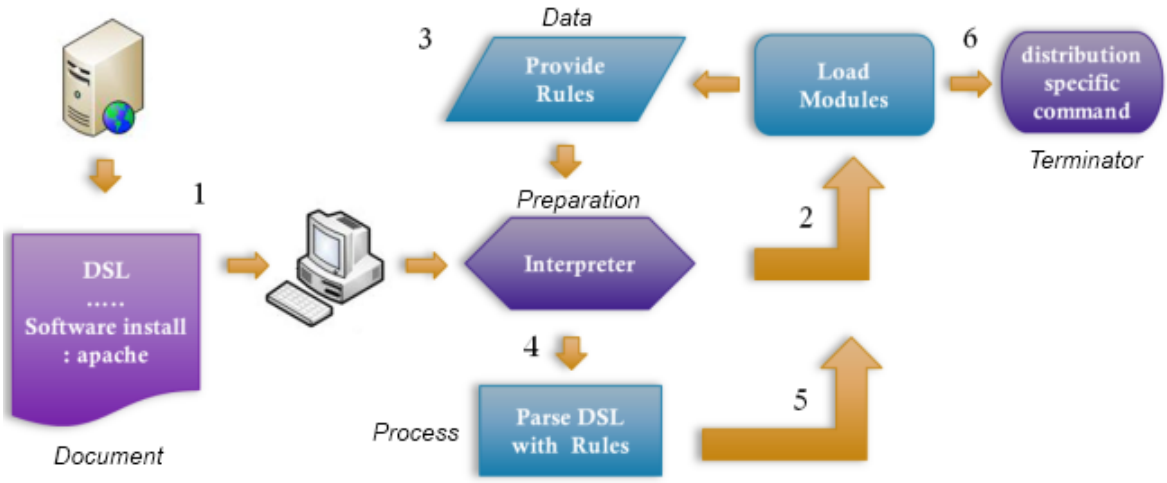

Fig. 3.15: Client Interpreter

The marchitecture or control flow diagram (simplified) here presents the major sequences of events leading to a policy (set of rules) application. A Unified Modelling Language (UML) Diagram is presented in 6.1 in Appendix 3.

1. **Receive policy update**
   Server sends a policy update in the form of a domain specific language
2. **Interpreter initialises**
   The interpreter is executed and loads the interpreter extensions (PERL modules)
3. **Modules provide parsing rules**
   The module(s) provide lexicographical parsing rules for the interpreter
4. **Policy is parsed**
   The interpreter parses the domain specific language (policy) using the parsing rules
5. **lexicographically accepted lines are sent back to the modules**
   For each successful line parsed, this line is sent back to the PERL module that provided the rule
6. **Module performs distribution specific command**
   Further inspection is done on the line, eventually resulting in a distribution specific command

Unlike the server architecture, the messages can only come from the server to which the client is bound. There is no direct communication between the administrator interface and the clients, hence the concept of the broker. As an example, if the administrator sends *Push Policies* to the server, then the server iterates the known or bound clients and pushes out the policy in the form of the DSL to the clients.

The client messages are as follows :

1. **Request Policy Update**
2. **Receive Policy Update**
3. **Request Modules Update**
4. **Receive Modules Update**

These messages provide a second point of interest from an administrator perspective. How does an administrator easily update the client and the interpreter modules? In a large enterprise environment updating the client with new modules via typical software installation means, ie.

sitting at the computer and updating it, which is not feasible. Any software extensibility provided by the client would be potentially negated by this time consuming operation. Therefore the design to support extensibility has been bolstered by the ability to push out new client modules and updates.

These modules are kept on the same server as the daemon. The clients check for modules updates at set intervals. To reduce network traffic a checksum of each module is sent to the client. The client compares this checksum against it's local copy. If there is a difference, an update has occurred and this modified or new module is requested by the client from the daemon.

This allows the administrators to create and modify modules on a continual basis, thereby meeting the enterprise specific regulations and requirements, that can subject to change on a continual basis.

3.5.4 Patterns

As previously discussed the modules have two primary functions :

- **To extend the interpreter and the language that it understands**
- **The Interpretation of rules, resulting in distribution specific commands**

Administrators who intend to implement new modules are implementing observers as seen in the Observer Design Pattern, Fig. 3.16.

These observers provide new grammar to the interpreter, which in turn sends updates, using the Observer Design Pattern update mechanism. Each individual rule(line) that is accepted by the interpreter using the provided grammar rules, is sent to all modules. Each module knows if this update is detonated for it, as it has provided the rule to the interpreter.

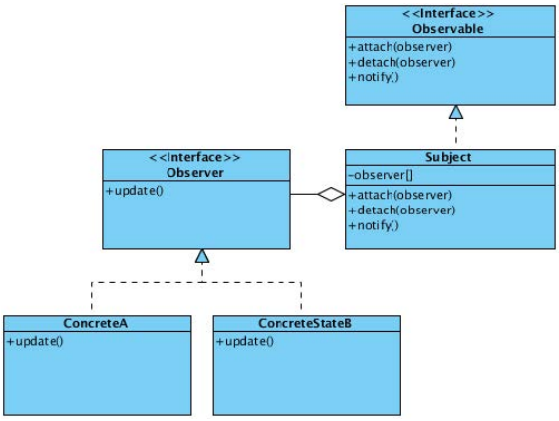

Fig. 3.16: Observer Design Pattern

This offers the ability to provide modules outside of this preordained behaviour. Modules could be designed to monitor, log or validate these messages and their outcomes. This one to many relationship (policy line to modules) does create a little extra processing, however, it provides more flexibility for administrators creating modules. A module that has too many responsibilities may become difficult to maintain, therefore the separation of logging, monitoring & validating into different modules are handled by this design decision.

3.5.5 Coding

Dissecting the domain specific language (DSL) in Fig. 3.17, we can see at the beginning the interpreter is included. The interpreter which will eventually go on to read the DSL, firstly constructs the framework which in turns loads the modules.

```
#!/usr/bin/perl -w
use lib "/opt/lgp/client/includes"
use Interpretor;

#ACTION:
  SERVICE_CONTROL : httpd off
#END:
```

Fig. 3.17: DSL

During the loading of the modules, the "registerGrammer" method is called on each module (Fig. 3.18). There is pure assumption from module loader that module being loaded provides a new method and a "registerGrammer" method. This reflects the concept of late binding, unlike the early binding method of using reflection to firstly inspect an object for a method and binding appropriately.

```
foreach my $module ( @modules )
{
  if( $module =~ /(.*)\.pm/s )
  {
    my $moduleName = $1;

    eval
    {
        require "$module";
    };

    my $reference = $moduleName->new();
    $reference->registerGrammer();
    $self->attach( \$reference );
  }
}
```

Fig. 3.18: Load modules - Simplified

There are of course positive and negative effects of this. As PERL has run time flexibility as seen in most interpreted languages. Up front validation of the module implementation is difficult requiring a try and test approach of module development. This would not be advisable in a live environment and should be done in a test harness, a group of machines separate from the live environment.
The positive effect however, is that no compiling is required, and gives greater flexibility and interoperability in deployment of a client onto an unknown system.

Filter::Simple

Syntax Directed Translation (SDT) of the domain specific language in Fig. 3.17 is achieved via source filtering. Syntax Directed Translation (SDT) is a method of translating sentential forms to machine code or some other intermediary language, such as Java byte code. There are many

different techniques to achieve this outcome and multiple operations may be done to the input before the translation is completed. This is the fundamental process used by compilers and interpreters.

One of the initial steps in this process is the processing of preprocessor directives. Fig. 3.19 presents an example of the "use" keyword. This tells the PERL interpreter that the module "File2" will needed to be included, before any subsequent processing takes place. The next stage of the interpretation, is in this example, source filtering.

```
1 use File2;
2
3 # something to be filtered
```

Fig. 3.19: File 1

File 1 (Fig. 3.19) provides the preprocessor directive "use File2". File 1 (Fig. 3.19) is then subject to source filtering. File 2 (Fig. 3.20) provides a filter block, which takes the input of the File 1 (Fig. 3.19) and places it input the PERL variable $_. Grammar rules in the form of regular expressions, provided by the modules in Fig. 3.18, are then used to parse the data in the $_ variable.

Delving further into this subject will take place in section 3.8 (DSL).

```
1 use Filter::Simple;
2
3 FILTER
4 {
5   $_ = processText( $_ );
6 };
7
8 1;
```

Fig. 3.20: File 2

3.5.6 Analysis & Improvements

The client solution developed meets the requirements imposed. In the case for **support for multiple distributions** and **extensibility**, Unix and derivatives provide PERL as standard. The support for multiple distributions is then a matter of extensibility. This extensibility is achieved via the creation and modification of modules. As new distributions are created and the distribution specific commands evolve or change, this is ultimately handled by the extensibility nature of the interpreter via these modules.

As PERL is not a compiled language, updates can be pushed out to the clients regardless of the architecture. In the future a test harness for the testing of modules should be provided. There is a risk of an administrator unintentionally distributing a fallible module, which could result in costly outcomes.

Although the architecture presented meets the requirements, some improvements could be done to the interpreter. As the interpreter's grammar rules are provided in the form of regular expressions, the complexity of these rules could become rather difficult to understand. However

immensely powerful regular expressions are, for future improvement a relaxing of this validation technique may be catered for. Stubs or blocks that have opening and closing elements as seen in the extensible markup language, may provide the ability to break up parsing rules into more manageable discrete statements.

3.6 Admin Presentation

3.6.1 Requirements

1. Creating organisational units.
2. Move objects in the organisational hierarchy.
3. Hierarchy of organisational units.
4. Modify policies.
5. Rules creation results in DSL script.
6. Provide means of importing rules.
7. Push updates to the client.

3.6.2 Technologies

- **C#**
  C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines. [cit]

- **AvalonDock**
  Avalondock is a windows presentation foundation(WPF) controls library that provides the functionality of a dockable layout system for UserControls within a user interface. It supports fly-out panes dockable pinning expanders and floating windows.

- **AvalonEdit**
  Avalonedit is the WPF rich text user control for editing source code in SharpDevelop the open source alternative to Visual Studio. The editor features many of the post Intellij integrated development environment features such as code completion, code folding and syntax highlighting.

- **Infralution localization**
  Infralution localization is an open source class library that integrates localisation features easily into windows presentation foundation(WPF) projects by taking control of resources files associated with a visual studio project. This middle tier layer between the compiled code and the translations provides a means to instantly change the language in an application at run time.

- **JSON.net**
  Json.Net is an open source Javascript Object Notation framework for .Net. For message passing in the implementation between the nodes this was the preferably choice for message passing.

- **Microsoft Prism Event Aggregator**
  The Microsoft implementation of Marting Fowler's Event Aggregator was chosen as a means for internal message passing within the application framework as it allows for events to be decoupled for the publishers and subscribers.

- **Visual Studio 2010**
  The Microsoft Visual Studio Integrated Development Environment is by far the best in the field for rapid application development in my opinion. The deployment options, such as Web/Desktop/Phone etc coupled with the Common Language Runtime (CLR) run time environment allows for the development of applications that bring interoperability and manageability to new heights.

- **Windows Presentation Foundation**
  The Windows Presentation Foundation graphical subsystem introduced with .Net 3.0 and at the core of Windows Vista / 7 renders applications directly into DirectX and is graphically accelerated on industry standard graphics cards. The idea of desktop compositioning driven by earlier projects in the open source community such as Beryl and Compiz made the push for this. Apple and Microsoft quickly followed suit. Furthermore WPF provides a model view-view model design approach for the separation of the code and interface design somewhat comparable to Model View Controller introduced by smalltalk.

- **Xaml**
  The Extensible Application Markup Language (Xaml) is a declarative XML based language used for the creation of user interfaces. Xaml directly maps for CLR run time object instances and the attributes to the associated properties allowing for most of the user interface functionally to be kept out of the business model or controller.

3.6.3 Design

The design of the Linux Group Policy presentation or administrator interface was extensive. With up wards of fifty thousand lines of code, the need for a high quality design was exigent. The support for the 'ilities' or non functional requirements motivated the design and eventual architectural framework. Many influences lead to the design of the composite framework, such authors as Martin Fowler, Joshua Kerievsky and Gamma et al.

Fig. 3.21 presents a high level architectural design model. The design could be described as a three tier architecture. Where the separation of presentation, logic and the data is shown. This distinction however in reality a little clouded, but the three tier architecture seems the best method to articulate the real nature of the design.

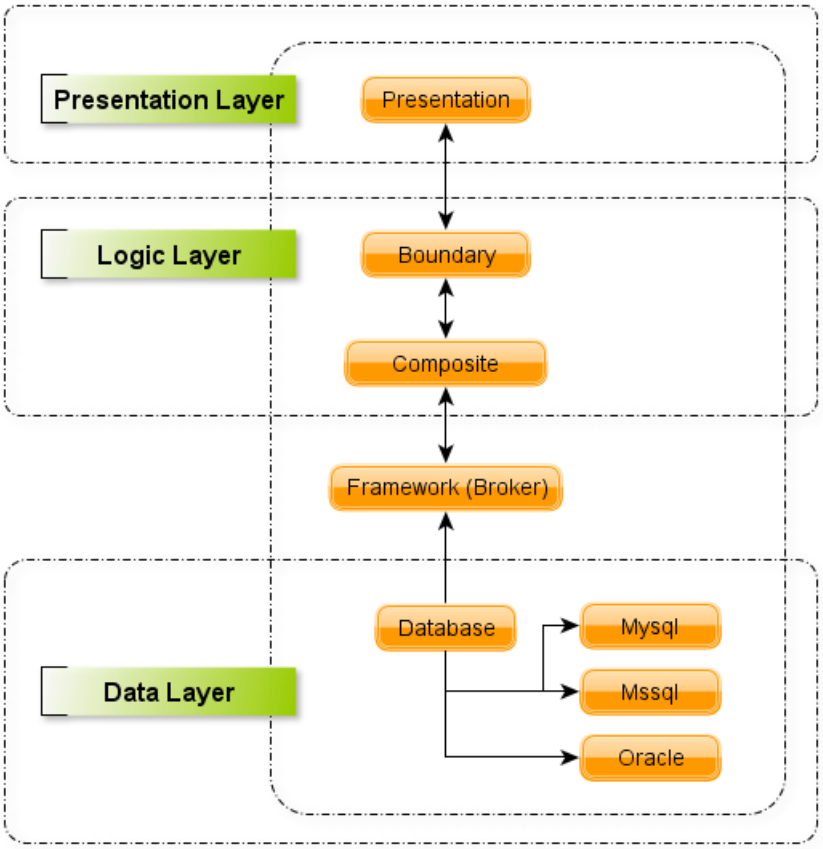

Fig. 3.21: Design Overview

Each composite has it's own responsibilities and encapsulates all the presentation and logic layers for which it is concerned. There are two composites that deviate from this assertion.

It was decided upon to separate the data tier from the end composite developers. This separation of the data layer negates or reduces unfavourable database operations. The obfuscation of the structured query language(SQL) and the encapsulating of it into representing domain types provides traceability, monitoring and maintainability.

The database composite is primarily concerned with the data layer, however it exhibits presentation (settings panes) and logic layer functionality also. The database composite therefore abstracts all the domain concerns and the framework or the brokering layer facilitates the access to this data layer or the database composite.

The factory composite provides all the domain and architectural interfaces for the entire architecture. This framework layer decouples all composites and implements patterns to facilitate communication between these decoupled composites. It does not provide a presentation or data layer, however it provides extensive logic layer functionality for the loading of the composites and subsequent brokered interactions.

3.6.4 Architecture

The Architecture of the Linux Group Policy Studio Administration interface is comprised of separate interchangeable dynamic link libraries as can be seen in the composite architectural overview in Fig. 3.22. This type of software technique is commonly known as composite or modular programming.

Each individual DLL (dynamic link library) is designed for a specific scenario and encapsulates all the necessary logic for the scenario and its use cases. This separation of concerns greatly improves the maintainability, reliability and reduces the fallibility of the software product as a whole.

These dynamic link library components are loaded at run time and inspected via reflection for predefined architectural interfaces. Based on the interface the incorporation into the system is decided upon. For example, if a class realises the interface IDatabaseStrategy, we know this component is to be used by the Database connector and therefore it is loaded and referenced in the appropriate location, making it accessible to the database connector.

The main core of the modular framework consists of the lgp.components.factory dynamic link library which exposes the public static class 'Framework'. A static class is a class where all properties and methods are said to be static. C# allows the use of the static keyword in the class definition thereby enforcing this rule.

Framework is a boundary class and with the use of the architectural and domain interfaces, it exposes the dynamic link libraries concrete types that match, support, realise these interfaces. For example, rather that exposing the concrete Type 'Utilities' which is part of the framework; rather the interface for this type is exposed.

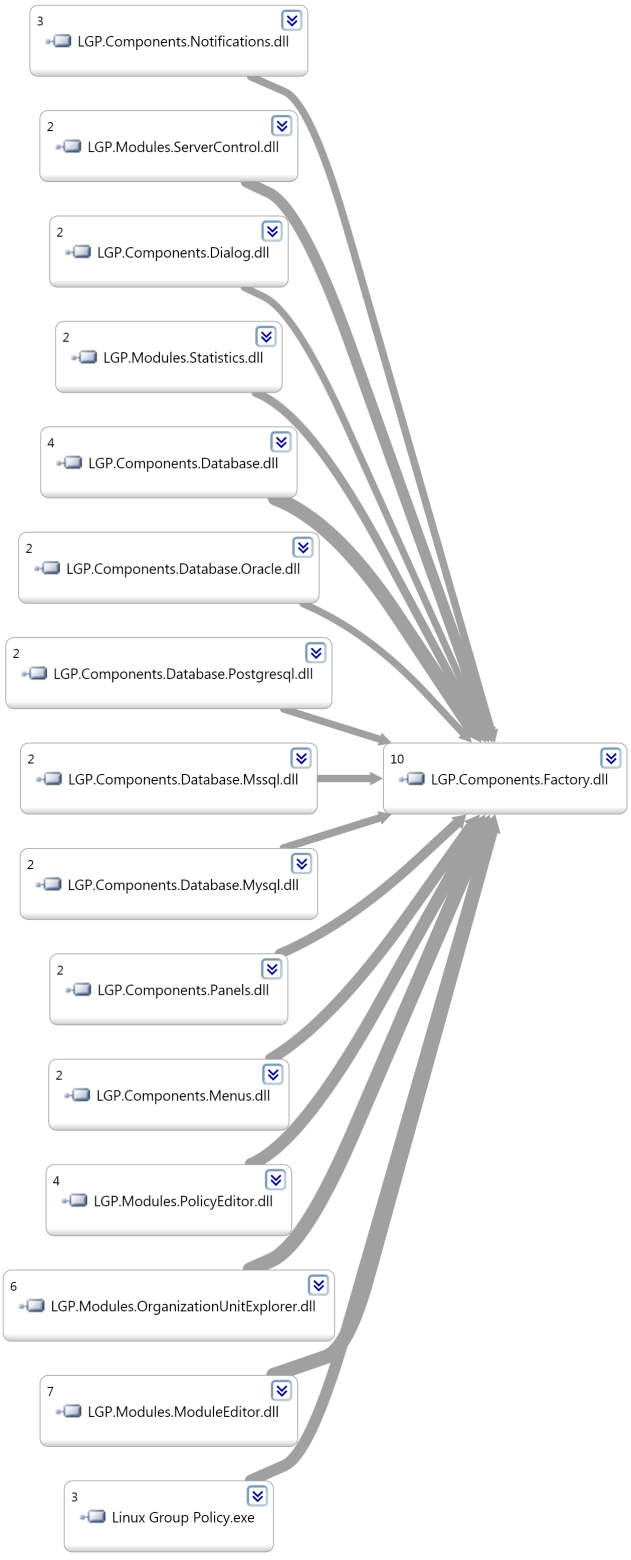

Fig. 3.22: Composite Architecture Overview

This reduces the coupling and in effect the modules are not dependent on any one particular concrete type other than that of the Framework boundary class. The framework class exposes either directly or indirectly 38 separate interfaces consisting of domain specific and non domain specific interfaces. An excerpt of the framework class's auto properties (gets/sets) can be seen in Fig. 3.23

```
namespace LGP.Components.Factory
{
  // Simplified factory for the loading of Modules and application components
  public static class Framework
  {
    public delegate void ShuttingDown( object sender , CancelEventArgs e );
    public static ShuttingDown ShutDown;
    private static readonly LibraryHandler ClassLibraryHandler;
    private static Window _applicationWindow;
    private static IMenu _applicationMenu;
    private static IPanel _applicationPanel;

    static Framework()
    {
      // loads up all the libraries
      ClassLibraryHandler = Internal.LibraryHandler.GetInstance();
    }

    public static IPanel Panels{...}
    public static IPanel Panel{...}
    public static INotification Notification{...}
    public static IDialog Dialog{...}
    public static IMenu Menu{...}
    public static IContextMenus ContextMenus{...}
    public static IRegistryHandler Registry{...}
    public static IImageHandler Images{...}
    public static IDatabase Database{...}
    public static IEventSystem EventBus{...}
    public static INetwork Network{...}
    public static Window ApplicationWindow{...}
    public static List< IModule > Modules{...}
    public static string[ ] Libraries{...}
    public static List< Type > DatabaseTypes{...}
    public static List< Type > PreferencesPanes{...}
    public static IUtilities Utils{...}
    public static bool HasError{...}
    public static string Error{...}
    public static IClassLibraryHandler LibraryHandler{...}
    public static IDragDrop DragDrop{...}
    private static void Shutdown( object sender , CancelEventArgs e ){...}
  }
}
```

Fig. 3.23: Framework Static Class

Where certain modular interaction is required the framework also provides a global uniform support for message passing. This implementation is based upon Martin Fowler's Event Aggregator design pattern. This pattern allows publishers and subscribers to evolve independently of one another, furthermore this modular approach to message passing fits well with the overall architectural design. The published events of the event aggregator in the implementation are

somewhat generic and therefore limited in application.

For specific interaction, an alternative approach was needed and the identification certain state transitions were identified for exposure within the dynamic link library entity classes. For example, right click context menus in modular systems are populated not only with menu options from the implementing dynamic link library control class, but also the allowance or ability for other dynamic link library classes to inject options into that menu as required.

This menu is generally specific to a type, ie. menu options for inode control (file, directories) would not be applicable to a text editors right click menu. Therefore differentiation of the specific contexts are needed and in the framework implementation; these contexts are identified by the 38 separate interfaces Fig. 3.24, specifically the domain specific interfaces Fig. 3.25. For third party developers, it is their choice whether or not to expose these transition states.

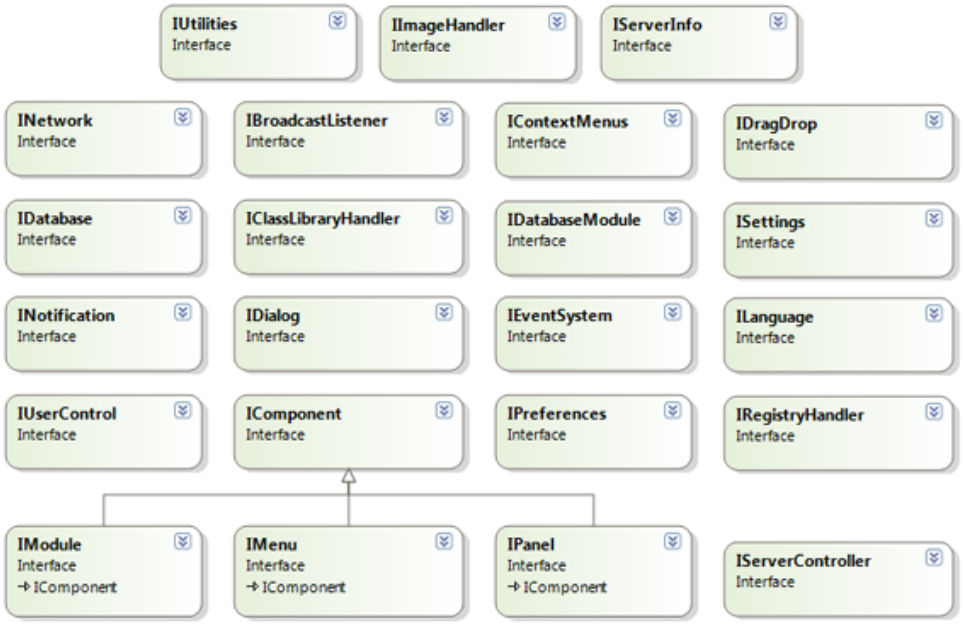

Fig. 3.24: Architectural Interfaces

An example of this within the application can be observed within the Organisational Unit explorer. The right click menus on an Organisational Unit are exposed and the policy editor which is a separate independent module, injects its' "edit policy" menu. The Event Aggregator pattern and Context Menu Injection Pattern can be referenced in section 3.6.6.

Interfaces

The architectural interfaces in Fig. 3.24 & 3.25 are used to decouple the dynamic link libraries from one another. These architectural interfaces are focused primarily with the inner workings of the application, the identification of non domains specific classes, such as boundary or control classes concerned with user interface components, database access and networking.

The following, IOu, IClient, IGrammer, IPolicy and IModule are the domain specific interfaces. The gateway interfaces are oriented around the Martin Fowlers' Table Gateway Pattern discussed later in this section, the subsequent observers and the separation of these interfaces.

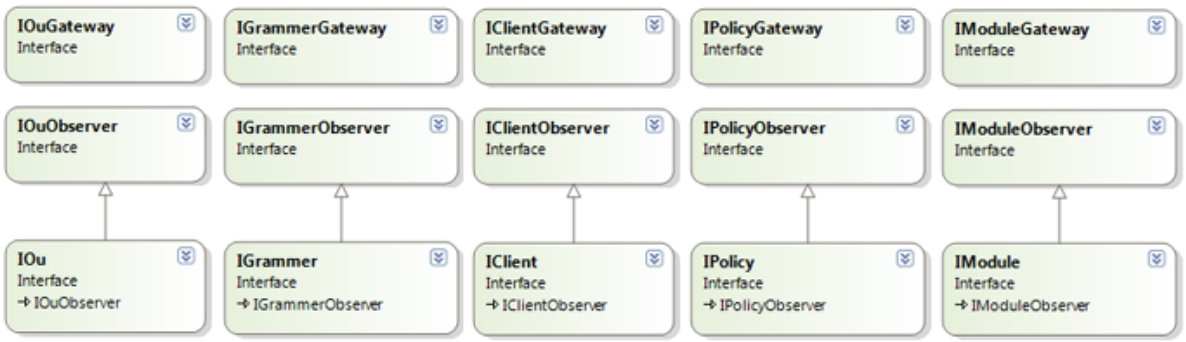

Fig. 3.25: Domain Specific Interfaces

3.6.5 Patterns

A key concern of any database centric product is the validity of the data. Dirty data in a database can lead to problems, for migrations and upgrades. Research into the area revealed that many approaches to controlling database access have been maturely established. Many patterns were considered before implementation occurred. Some of the design patterns considered where Query Object, Record Set, Row Data Gateway, Single Table Inheritance, Table Data Gateway, Table Module, Gateway, Active Record, Association Table mapping, Class Table Inheritance, Concrete Table Inheritance, Data Mapper and Data Session State to name a few. After careful consideration of uses cases, a modification of Table Data Gateway and Row Data Gateway was chosen.

Table Data Gateway

Table Data Gateway was an obvious candidate for manipulating a database table as a whole, from Patterns of Enterprise Application Architecture, Martin Fowler describes it as *"An object that acts as a gateway to a database table. One instance handles all rows in the table"* - Fowler (2010b). Simply put the Table Data Gateway is more like a collection data structure (list, arraylist, linkedlist), smarts bits can be implemented to control or buffer the data too and from the database.

This gateway implements methods similar to that found in the aforementioned collection types. For example an Employee gateway would implement methods such as, get, add, remove and search. These methods encapsulate the SQL (structured query language) statements that are used to query the database. This abstracts database interaction from the plugin developers and reduces / negates the possibility of poorly constructed queries that may cause inefficient queries or in some cases harmful queries. The following sample is used in the implementation as a control class for accessing the organisational unit table.

```
1  public interface IOuGateway
2  {
3    IOu CreateOu( string ouName , int parentId );
4    void DeleteOu( int ouid );
5    IOu GetOuById( int ouid );
6    IOu GetOuParentById( int ouid );
7    List< IOu > GetOuListing();
8    List< IOu > GetChildren( int ouid );
9    List< IOu > GetRoots();
10   void Refresh();
11 }
```

Fig. 3.26: Organisational Unit Table Gateway

Row Data Gateway

On inspection of the interface in Fig. 3.26 another type is recognised, “Iou”. This brings us to the second pattern Row Data Gateway and from Patterns of Enterprise Application Architecture, Martin Fowler describes it as *”An object that acts as a gateway to a single record in a data source. There is one instance per row”* - Fowler (2010b). As previously stated the Table Gateway can be described as a collection class and as a result Row Data Gateway complements this pattern.

Each row in the table is to be represented as an object in the system, to be encapsulated by the collection class. Each column field within that row is a property of this object which needs to be encapsulated and appropriate rules or business logic embedded into accessors and mutators of this object. This adds a second layer of data checking and enforcement of constraints to the data. This second layer of perplexity ensures the reduction of dirty data in the tables. A database generally achieves this via foreign key constraints within the database. However for fields in the database where constraints cannot be determined or accurately implemented via the internal data restriction mechanisms, the mutators of the object representation offer an area to ensure the validity of the data before updating the database.

```
1  public interface IOu : IOuObserver
2  {
3    void SetOuId( int val );
4    int GetOuId();
5    void SetName( string val );
6    string GetName();
7    void SetParentOuId( int val );
8    int GetParentOuId();
9    void SetPolicyGroup( string val );
10   string GetPolicyGroup();
11   Image GetOuImage( int size );
12 }
```

Fig. 3.27: Organisational Unit Row Gateway

These two patterns in conjunction have many implementation concerns, such as the where and how updates are done. In this proof of concept implementation i decided to reduce the complexity and separate the concerns. As such the Table Gateway is concerned with managing the collection, and the row gateway is concerned with modifying its represented row.

Developers may choose to use generics; also know as templates in C++, to implement generic gateways to be instantiated with specific types at run time. This is an implementation concern that serves promote the re-usability of these gateways, however at the possible cost of security and control of the data, it was not decided to go this direction.

Event Aggregator

The Event Aggregator design pattern by Martin Fowler is a publish subscribe design pattern. Events are published and subscribed to by means of an event handler control class. This control class provides access to concrete implementations of composite presentation events which acts a broker for type based messages.

Typical implementations of this pattern approach it in this manner, for the separation of event types. The reason for this approach is that in large systems with many publishers and subscribers, the iteration of all the subscribers for a published message, can be expensive. Therefore the separation of the events and their composition event control classes is preferred. Fig. 3.28 shows a high level overview of this approach.

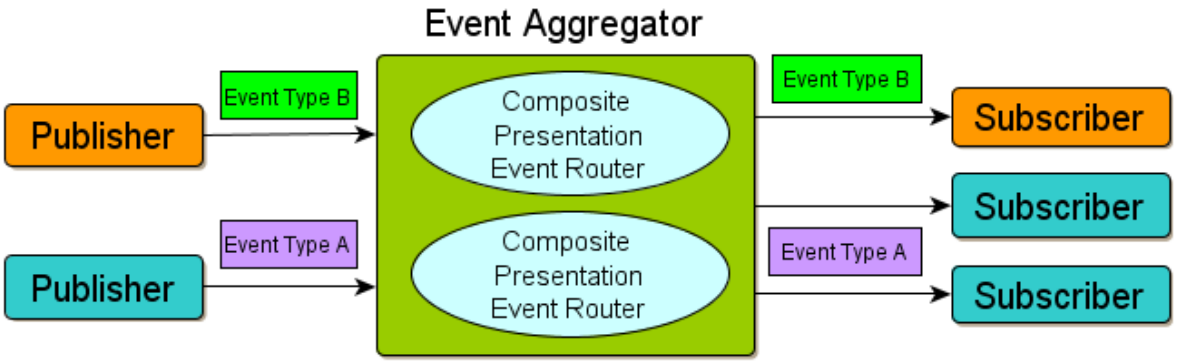

Fig. 3.28: Event Aggregator - type based

For a large architecture, this is the ideal approach for the best performance, security and decoupling concerns. However in the project these concerns did not out way the overhead of the separation of events, their handlers, their interfaces and the overall increase of types and resulting coupling within the system. Therefore I decided to use a generics approach. By creating a generic composite presentation event router and with the use of domain interfaces this explosion of types was negated. I left it up to the subscriber to determine whether the message that was received was of the type it was concerned with. This moved the responsibility of the type interpretation from the event router to the event receiver. Fig. presents this generics approach.

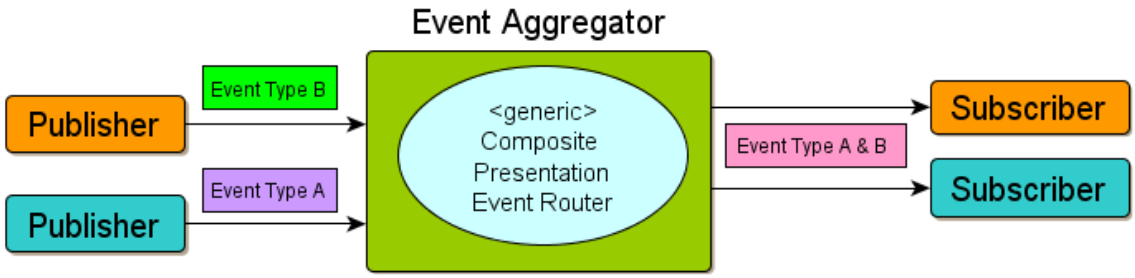

Fig. 3.29: Event Aggregator - generic based

3.6.6 Coding

Unlike the relatively simplistic flexibility of interpreted languages, the implementation of the presentation user interface required greater knowledge. A deep understanding of the concepts of binding, dynamic dispatch and reflection had to be well understood in order to create the composite architecture. The knowledge base required was not limited to these fundamental or computer science software characteristics. An understanding of user interface design and concepts such as routed events and event bubbling was mandatory.

In this section i will attempt to articulate these concepts, as they were used in the software architecture developed.

Reflection and the loading of composites

As previously discussed in the architectural overview, the composite approach is achieved via the use of reflection. Reflection is a process by which a computer program can inspect a type at run time. It is not limited to just inspection, as it can be used to modify an objects structure and behavior at run time. In the architectural design however, reflection is primarily used for inspection of the of composites and their internal types. These types that match, support, realise architectural interfaces are loaded and referenced in the appropriate location. These objects are then exposed via the Framework static class as previously discussed.

Fig. 3.30 presents an example of the use of reflection. In this example a composite or assembly is loaded and through this assembly reference, the inspection of it's internal types is done. The goal in this example, is the search for a type that realises the IContextMenus interface. When it is found, it is assigned to an internal member property, which can then be later accessed via an accessor (GET method).

```
1  // where dll path = ./someFile.dll
2  var assembly = Assembly.LoadFile( dllpath );
3
4  // get the types
5  foreach( var type in assembly.GetTypes() )
6  {
7    // ignore if not a class or not public
8    if( !type.IsClass || type.IsNotPublic )
9    {
10     continue;
11   }
12
13   // get its interfaces
14   var interfaces = type.GetInterfaces();
15
16   if( interfaces.Contains( typeof( IContextMenus ) ) )
17   {
18     var obj = Activator.CreateInstance( type );
19     this._contextMenusHandler = ( IContextMenus ) obj;
20   }
21 }
```

Fig. 3.30: Using reflection to load modules

Exposing of these loaded types

As seen here in Fig. 3.30 which is an excerpt of the ClassLibraryHandler class, a class that realises this IContextMenus is loaded and exposed. To put this in context, Fig. 3.31 shows the exposure of this type via the Framework static class. The type exposed is the interface and not the concrete type, therefore this effectively decouples other composites from the concrete implementation of said type. In order to make use of this functionality, other composites include a reference to the Framework static class.

```
public static class Framework
{
  public static IContextMenus ContextMenus
  {
    get { return ClassLibraryHandler.GetConextMenusHandler(); }
  }
}
```

Fig. 3.31: Framework exposing plug-ins via interfaces

IContextMenus

Following on from the loading and exposing of this type, Fig. 3.32 depicts the method signatures of this interface. This interface provides access to the architectural support, for registering of menu items, in context menus dynamically at run time.

```
public interface IContextMenus
{
  void AttachMenuItemRegistrations( ContextMenu menuparent , Type type , Action< Object , RoutedEventArgs > action );
  void CallBack( MenuItem sender , object theobject );
  void RegisterContextMenuItem( Type type , MenuItem item , Action< Object > action );
}
```

Fig. 3.32: IContextMenus interface

The RegisterContextMenuItem allows plug-ins to register menu items to be dynamically embedded into a context menu. The three parameters specify the type that will be passed back to the registering handler, the menu item which will physically be displayed in the context menu and finally the Action, which is a delegate for an encapsulated callback method.

In Figure 3.33 plugin A registers menu items with the context menus handler (IContextMenus). The callback method, the final destination of the menu item click is where the ITargetItem is sent.

```
// for simplicity purposes the concrete type of
// IContextMenus is referred to as IContextMenus

IContextMenus.RegisterContextMenuItem( typeof( ITargetType ) , new MenuItem() , this.CallBackMethod );

private void CallBackMethod( object passBackItem )
{
  var targetItem = ( ITargetType ) passBackItem;
  // do something with it
}
```

Fig. 3.33: Plugin A : Register menu item with the framework

In Figure 3.34 plugin B accepts menu registrations for ItargetType and requests the context menus handler (IContextMenus) to append them to the context menu. The menu item click handles the actual click event. This allows "smart bits" to occur before asking the context menus handler (IContextMenus) to invoke the CallBackMethod in plugin A. This decouples the plug-ins from one another, coupling the two with IContextMenus and any "Type" in this instance ITargetType.

```
1  // for simplicity purposes
2  // the concrete type of
3  // IContextMenus is left out
4  // and referred to as IContextMenus
5
6  private void MenuContextMenuOpening( object sender , ContextMenuEventArgs e )
7  {
8    var menu = this;
9    IContextMenus.AttachMenuItemRegistrations( ref menu , typeof( ITargetType ) , this.MenuItemClick );
10 }
11
12
13 private void MenuItemClick( object sender , RoutedEventArgs e )
14 {
15   var menuitem = sender as MenuItem;
16   IContextMenus.CallBack( menuitem , TargetType );
17 }
```

Fig. 3.34: Plugin B : Handle menu item click and invoke callback

Dynamic Context Menus Sequence Diagram

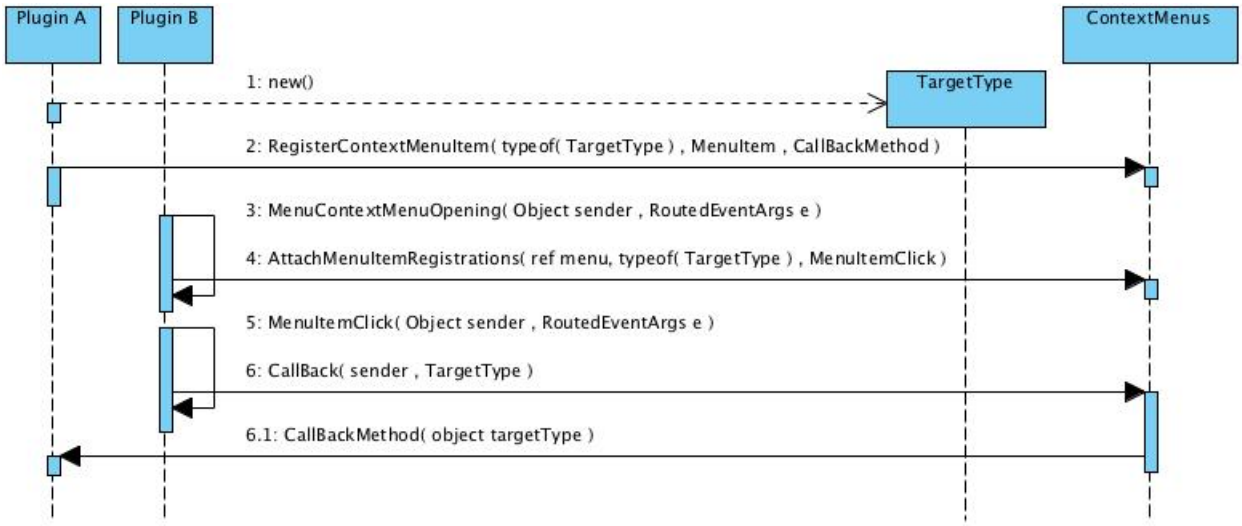

Fig. 3.35: Context Menus hooks sequence diagram

3.6.7 Analysis

A widely accepted concept in software design is the decoupling of components(composites) or the minimisation of inter dependencies, to promote re-usability. There are many papers that target interface definition as the means to accomplish this, such as, "A comprehensive interface definition framework for software components" & "An Approach to Software Component Specification" by Han (1999). With this software paradigm preposition, I am going take a look at the more fundamental concern of component (composite) based design.

David Lloyd of JGroup Expert defines the distinction between libraries and frameworks :

- ***Library***
  - *Set of classes instantiated by client*
  - *Client calls functions*
  - *No predefined flow of control*
  - *No predefined interaction*
  - *No default behaviour*
- ***Framework***
  - *Customisation by sub-classing*
  - *Calls client functions*
  - *Controls flow of execution*
  - *Defines object interaction*
  - *Provides default behaviour*

Given the aforementioned premises of architectural design goals and the distinction between libraries and frameworks; it may seem that libraries do not exhibit characteristics prevalent to achieve decoupled components (composites). The instantiation of classes and calling of functions directly, couples components (composites).

Composite design therefore even with good interfaces and supporting factory methods still presents coupling. This coupling as stated comes in the form of factory methods that require access to one or more concrete types in a composite. If there are multiple interdependencies between composites changing out a composite may present difficultly in the form of refactoring.

The fundamental concern is then composite interdependencies. The characteristics of a framework provide greater composite decoupling. The presentation framework for the admin interface provides only one concrete dependency relationship, targeting this concern. The admin presentation interface evolved to the characteristics of a framework, which I believe is a bigger concern, than good interface definition. This concept comparable to networking is the difference between a mesh network and a star network, as seen in Fig. 3.36.

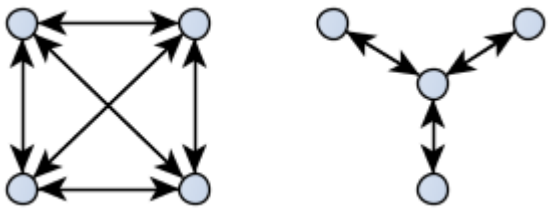

Fig. 3.36: Interdependence - Mesh(L) - Star(R)

3.6.8 Improvements

There are many advantages and disadvantages to the framework developed. The architectural framework devised is rigid in many respects. As all classes in the factory composite are defined as "internal" classes and exposed via their interfaces, this does not offer any sub-classing and changing of behaviour to composite developers. This effectively locks developers into the behaviour of the architectural framework.

To address this concern, firstly the reasoning for this approach must first be addressed. During coding it is common practice to exercise encapsulation when implementing classes and through the use of accessors and mutators, this basic object oriented principle is adhered to. Similarly when designing composites this practice can be achieved via "internal" classes. This effectively means that peer composites cannot instantiate these classes marked as internal.

For future improvements to the architectural framework, a relaxing of this principle may be applied. Classes that are not deemed critical to core functionality or do not present a security concern, may be marked as public. Alternatively patterns to support other functionality or out of band services would be the preferred approach. An analysis of composite developer requirements should be completed and patterns implemented in future revisions to support these requirements.

3.7 Database

3.7.1 Requirements

1. Storage of rules
2. Common non ambiguous names for rules
3. Organisational Units and Clients
4. Globally unique identifiers (GUID) for computer objects.

3.7.2 Design

The design of the database was minimalist given the requirements. Fig. 3.37 presents the relational design and abstraction of the domain concepts. As seen in presentation admin interface these entities are represented by design patterns. The main requirements are satisfied, "policy" table for the storage of rules, and the representation of clients and organisational units.

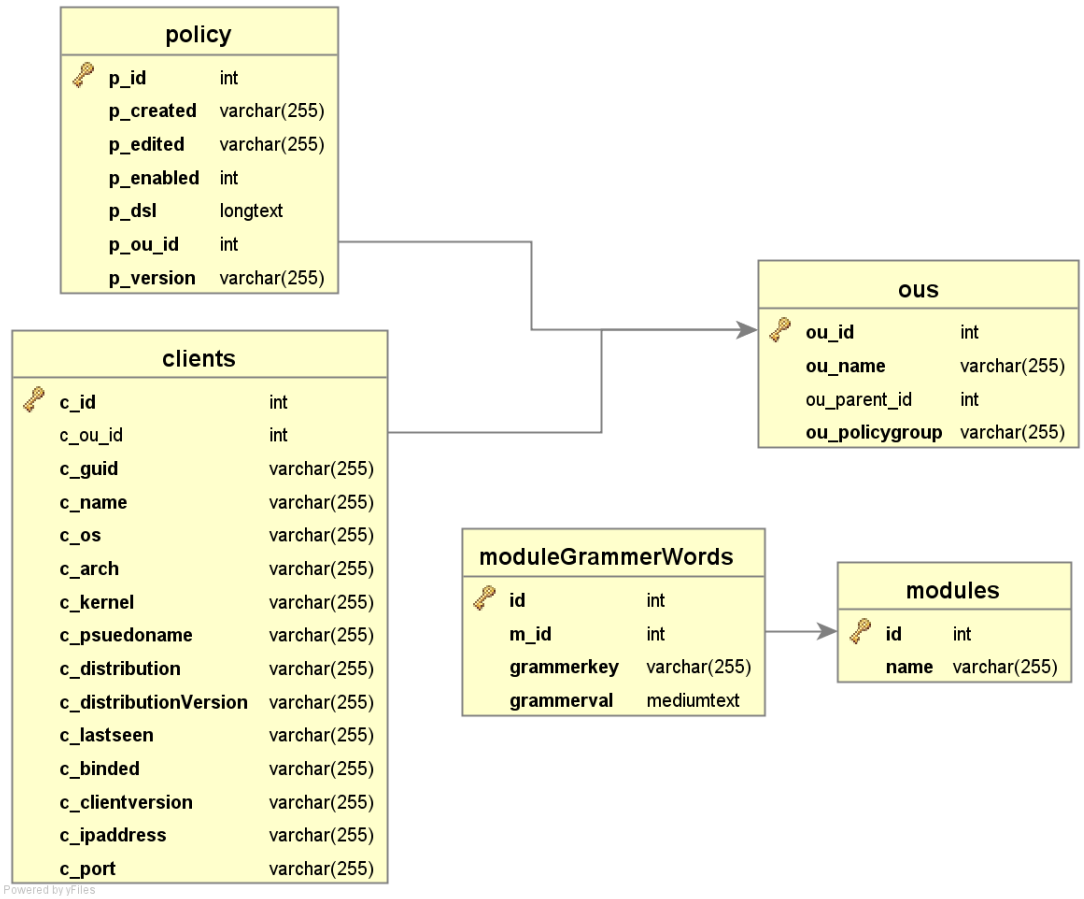

Fig. 3.37: Database Design Overview

3.7.3 Analysis & Improvements

Without an analysis of component developer requirements and future development, it's hard to say what improvement would be made to the database design. An obvious choice we be the adjustments of the types. There is extensive used of varchar, however as the data is used primarily as strings within the application for display purposes, this may not be necessary.

3.8 Domain Specific Language

A Domain Specific Language in many circles is generally described as a computer programming language with limited expressiveness, focused on a particular domain. The best example of this description is the hypertext markup language (HTML) and of course everyone's favourite the structured query language(SQL).

Domain specific languages are not a new concept. The structured query language(SQL) was conceived 1986 & the hypertext markup language (HTML) 1989. The choice for a Domain Specific Language (DSL) depends largely on what you are trying to achieve and whether this succinct language offers more efficient, less verbose and more flexibility; but even more so, is it better suited for the domain. Would it be efficient for people to do database queries in the Extensible Markup Language (XML)? For those who are familiar with oracle's structured query language(SQL) command line interface, imagine using it with the Extensible Markup Language (XML).

Moving quickly on from this eye brow raising statement, the choice for a Domain Specific Language in the solution was evident. Having experience in other solutions, such as INI files, Javascript Object Notation (JSON) and Extensible Markup Language (XML); these seemed rather verbose and restrictive to administrators who wished to define rules quickly and cleanly. Language cacophony is a common objection to taking on a Domain Specific Language (DSL). However this argument is quickly avoided by the purpose of the Domain Specific Language (DSL). The purpose of the Domain Specific Language (DSL) is to reduce the complexity of the statements used to achieve a domain specific goal. Writing a web site in Java would be quiet a difficult and time consuming task; this is why hypertext markup language (HTML) was conceived.

As PERL was used as the client side implementation, this offered the perfect solution for text processing and the foundation for a Domain Specific Language (DSL). PERL, the practical extraction and reporting language was developed for text processing. PERL has many tools for this activity, one of them being Filters. Filters or pre-processing of an input file, allowed for the formulation of this conceptual requirement. As pre-processing can analyse the domain specific language, post-processing can also execute internal or embedded PERL statements, giving rise to the concept of a hybrid Domain Specific Language (DSL).

Stepping back from this assertion for the moment, what is the difference between an "internal" and an "external" Domain Specific Language (DSL)? To be short and succinct in the definition of these two terms, an internal Domain Specific Language (DSL) is code that can be evaluated and executed without syntax directed translation taking place, to achieve the native language understood by the parser; while an external Domain Specific Language (DSL) is code that requires some sort of translation before it is understood by the parser.

The following sections will identify these concepts in the implementation.

3.8.1 Requirements

1. An extensible Domain Specific Language.
2. Domain Specific Language grammar hooks.
3. Hybrid Domain Specific Language.
    - Internal characteristics, rules provided by modules.
    - External characteristics, embedded PERL.

3.8.2 Design

The design of the Domain Specific Language (DSL) is based primarily on the extensibility of the client interpreter. In previous sections we looked at how new parsing rules were injected into the interpreter. These rules essentially define the Domain Specific Language (DSL). Therefore the language is not restricted by a document type definition (syntax and ordering rules). This means that the Domain Specific Language (DSL) constantly changes as interpreter modules are created and edited. Therefore the design of the Domain Specific Language (DSL) is the sole responsibility of the PERL module developers.

Fig. 3.38 shows both examples of the internal & external Domain Specific Language (DSL) statements. The internal Domain Specific Language (DSL) statements are identified between the % symbols. An example of an external Domain Specific Language (DSL) statement, namely "SERVICE_CONTROL" will be parsed by the rules provided by an interpreter module.

```
#!/usr/bin/perl -w
use lib "/opt/lgp/client/includes"
use Interpretor;

#ACTION:
  SERVICE_CONTROL : httpd off
  <% print "hello world" %>
 # END:
```

Fig. 3.38: DSL - Hybrid

Parsing of the internal statements is quiet simple and articulated in Fig. 3.40, while the parsing of the external statements is shown in Fig. 3.39

```
sub processDSLExternal
{
  my $_ = shift;

  my $regex_blocks = '#(\w+)(\s+:|:)(.*?):(\w+)#';
    my $interchange = "";

    while( $_ =~ /$regex_blocks/gis )
    {
    $interchange = InterpreterHelper::getInstance()->interpretDslActions( $3 );
    }
    s/$regex_blocks/$interchange/sm;

  return $_;
}
```

Fig. 3.39: DSL - External Parsing

```
sub processDSLInternal
{
  my $_ = shift;
  my $result = '';

    my $regex_code = "<%|%>";
    my $i = 0;

    for( split( /$regex_code/ , $_ ) )
    {
        if( $i++ % 2 == 0 )
        {
            s/[{}]/\\$&/g;
            $result .= "print q{". $_ ."};";
        }
        else
        {
            $result .= $_;
        }
    }

  return $result;
}
```

Fig. 3.40: DSL - Internal Parsing

3.8.3 Analysis & Improvements

The hybrid characteristics of the Domain Specific Language (DSL) offers the greatest possible flexibility for administrators. By allowing the embedment of PERL into the Domain Specific Language (DSL), administrators who are waiting for new additional functionality to be provided by me the developer, can work around any deficiencies discovered within the implementation.

3.9 Supporting Technologies

- **Gentoo Linux**
  Gentoo a distribution of Linux is built on top of the portage package management system. The system is compiled directly from source and allows the user to choose the features or compile options when building their system.

- **Ubuntu Linux**
  The Ubuntu distribution is a fork of the Debian Linux operating system. The design philosophy of this distribution is geared towards usability and the desktop experience and therefore for home personal use.

- **Opensuse Linux**
  Open use is the free version of Novell's Suse Linux Enterprise Server. The design of Suse is about constant refinement as opposed to bleeding edge software. As such its considered one of the mature Linux operating systems.

- **Redhat Linux**
  Red Hat the largest corporate contributor to the Linux kernel, provides Enterprise Linux and open-source enterprise middle ware solutions such as JBoss. Red hat provides operating-system platforms along with middle ware, applications, and management products, as well as support, training, and consulting services. Red Hat creates, maintains, and contributes to many free software projects.

- **Fedora Linux**
  Fedora sponsored by Red Hat is mainly driven towards the promotion and advancement leading edge of open source technologies

- **CentOs Linux**
  CentOs is the licence free downstream version of the Red Hat Enterprise Linux operating system.

- **VmWare Workstation**
  Vmware Workstation, Vmware flagship consumer product is the desktop solution for Virtual Operating Systems. This product allows for multiple virtual machines to be hosted on a single computer.

- **VmWare Workstation**
  Vmware Workstation, Vmware flagship consumer product is the desktop solution for Virtual Operating Systems. This product allows for multiple virtual machines to be hosted on a single computer.

- **Visual Paradigm**
  Visual Paradigm was used for some of the unified modelling language diagrams

- **Subversion**
  Subversion was used as the primary source version control.

4 Testing & Evaluation

4.1 Overview

An example of a high level use case is detailed in table 4.1. Use cases were used to contrive test cases in line with the requirements in section 1.3. By designing tests cases around actors and their actions; scenarios (instances of use cases) provided the best coverage for exploratory testing or black box testing and resulting quality assurement process.

Use cases were further broken down into discrete use cases indicated by the description steps and extensions points in the use case sample 4.1.

Use Case	**Edit Organisational Unit Policy**	
Preconditions	Administrator interface running Database connectivity Organisational unit created Edit policy option selected	
Success End Conditions	Rule(s) successfully created Policy Saved	
Failed End Conditions	Unable to created rules Unable to save	
Primary Actor	Administrator	
Trigger	Administrator wants to apply policy	
Description	Step	Action
	1	Open Administrator Interface
	2	Open Organisational Unit Explorer
	3	Select Organisational Unit
	4	Initiate right click context menu
	5	Select Policy Editor
	6	Type or use visual editor to create rule(s)
	7	Activate save option
Extensions	Step	Action
	1a	Administrator interface fails to load
	2a	O.U. Explorer Unavailable
	3a	No Organisational units defined
	4a	Unable to initiate menu
	5a	Unable to select Policy Editor
	6a	Unable to modify/add rule(s)
	7a	Save option fails
Performance	Thousands of rules. Saved in 2 seconds	
Frequency	1000's/day	
Channel	Policy Database Gateway	
Open issues	Policy Data validation	

Table 4.1: Use Case sample

As this project was software-centric, testing played a huge role in the completion, successful delivery and resulting quality. One of the software fallacies (Perfect Software: And Other Illusions about Testing) is the concept of testing for everything, otherwise known as exhaustive testing Weinberg (2008). Therefore the application of use case realisation; the identification of collaborating objects in the application domain; offered the perfect means to derive test cases.

The JetBrains Resharper utility was used as a white-box testing enhancement tool, which actively interrogated the software under development. Resharper looks for common problems such as possible null value exceptions, code coverage and unreachable code, boundary value assessment etc. Other utilities such as CodeRush by Devexpress have similar functionality.

4.2 Metrics

The Metrics provided is a static analysis of the code. The gathering of statistics such as inheritance depth, lines of code, number of classes etc. The estimation of complexity and maintainability is computed from this information with the goal of determining the quality of the software.

The study of code metrics can help identify potential issues in large software projects. The resulting identification of trouble spots and the re-engineering can help to support the software evolution and the improvement of software process improvement (SPI) techniques.

4.2.1 Static Analysis Tools

There are multiple tools and plug-ins available to achieve a static code analysis. Three tools were chosen as each provided different types of measurements for the analysis.

- **Visual Studio**
  - provides minimal metrics including, lines of code, maintainability index, cyclomatic complexity and class coupling
- **Campwood Software LLC SourceMonitor**
  - focuses on method analysis such as documentation, minimum/maximum average complexity and nesting levels
- **NDepend**
  - focuses more on quality attributes such as coupling and dependency management
- **JetBrains DotCover**
  - was used for code coverage analysis.

As each tool implements their analysis slightly differently resulting in different metrics, the choice for three tools was obvious in gaining a more comprehensive overview. Some vendors choose either Mccabe metrics or Halstead Metrics and may also add additional vendor massaging eg. Logarithmic scales. The tool used will be identified for each metric presented.

4.2.2 Maintainability Index

- ***Maintainability Index***
  - *Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code*
  - *A high value means better maintainability*
  - *A green rating is between 20 and 100 and indicates that the code has good maintainability*
  - *A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable*
  - *A red rating is between 0 and 9 and indicates low maintainability*

– MicrosoftCodeMetricsValues

Interpretation

Microsoft here use qualitative terms to define the maintainability index, in more scientific or quantitative terms, the formula for this is shown in Fig. 4.1.

Maurice Howard Halstead introduced the 'Halstead complexity measures' in 1977 as a means of establishing an empirical science for software development. A further analysis of this theory is outside the scope of this paper and i will ask the reader to refer to the reference. (Halstead)

```
MAX
( 0,(  (
        171
        -
        ( 5.2  * ln( Halstead Volume ) )
        -
        ( 0.23 * Cyclomatic Complexity )
        -
        ( 16.2 * ln( Lines of Code ) )
      ) * 100
   ) / 171
)
```

Fig. 4.1: Maintainability Index Formula

Analysis & Improvement

There is not much we can tell from this information other than the size may indicate maintainability and understandability problems.

During the calculation of these metrics the blank lines of code were left out by source monitor, contrary to the definition. For the C# analysis the larger of the two indicators 44,996, includes generated designer files generated by visual studio, but modified during development.

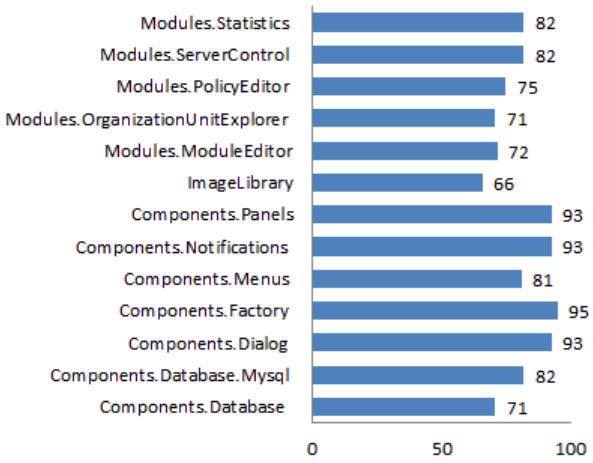

Fig. 4.2: Maintainability index

4.2.3 Cyclomatic Complexity

- ***Cyclomatic Complexity***
  - *Measures the structural complexity of the code*
  - *It is created by calculating the number of different code paths in the flow of the program*
  - *A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable*

– MicrosoftCodeMetricsValues

Interpretation

This measures the total number of individual paths or block levels through the code. This is calculated by counting the number of decision points such as if, switch, while, foreach and for loops statements. This measurement is a good indication on the number of unit tests and resulting code coverage. Lower is better. It can also indicate the complexity of the test cases which would be as complex as the code itself.

cyclomatic complexity = edges - nodes + 1

In this formula the nodes represent logic branches and an edge represents the path between nodes. The rule calculation reports a possible problem when the cyclomatic complexity is greater than 25.

Analysis & Improvement

From face value the cyclomatic index seems rather high for some of the dynamic link libraries. These values however are based upon all the files that are encapsulated within the dynamic link libraries, including structs and classes.

The factory dynamic link library which is the most complex encapsulates 18 classes and has an average complexity of 30 (545 complexity/18 classes).

Although slightly over the recommended limit of 25, i don't believe that this is cause for concern as there is a degree of complexity associated with composite, pluggable or modular architectures as indicated by the analysis of Microsoft Prism composite library in Fig. 4.4.

In section 4.3 (line spread) we can see the average complexity for the entire product is approximately 15. This figure is well within the recommended 25 limit.

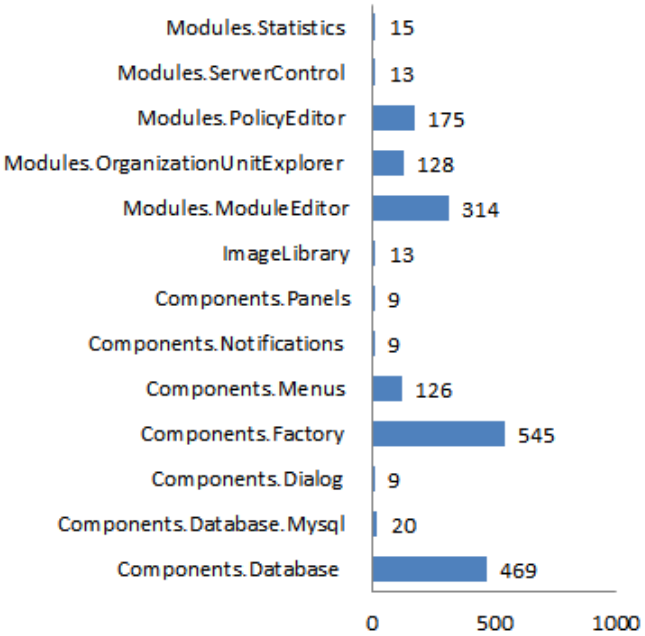

Fig. 4.3: Cyclomatic Complexity

Hierarchy	Cyclomatic ...
MefModulesForTesting (Debug)	2
Prism.Desktop (Debug)	1,749
Prism.Desktop.Tests (Debug)	1,636
Prism.MefExtensions.Desktop (Debug)	149
Prism.MefExtensions.Desktop.Tests (Debug	257
Prism.TestSupport (Debug)	31
Prism.UnityExtensions.Desktop (Debug)	35
Prism.UnityExtensions.Desktop.Tests (Debu	155

Fig. 4.4: Microsoft Prism - Best Practices Metric

4.2.4 Depth Of Inheritance

- ***Depth of Inheritance***
  - *Indicates the number of class definitions that extend to the root of the class hierarchy*
  - *The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined*

– MicrosoftCodeMetricsValues

Interpretation

Types that derive directly from Object are said to have an inheritance of 1. This number does not take into consideration the depth of any implemented interfaces. Deep inheritance is problematic in terms of modifiability and maintainability and as such the rule "favour delegation over inheritance" should be used as indicated by the book Design Patterns - Gamma et al.

Analysis & Improvement

There is always the temptation to extend from another class and as such we have violated the rule set forth by gamma et al, "favour delegation over inheritance".

In the metrics provided we can see two possible instances of this rule violation.

From the two class definitions in Fig. 4.6 we can see the result of this. TreeViewOuElement extending from TreeViewItem introduced nine inherited dependencies and the DragAdorner extending from Adorner introduced 6 dependencies inherited as part of the inheritance chain the Dot Net Framework.

An argument for this and the reason for the intentional rule violation in terms of TreeViewOuElement is that, to create all the functionality of a TreeViewItem so that it conforms or encompasses all the functionality that is required to realise it as a TreeViewItem, is simply too much work. As indicated by the Dot Net Frameworks TreeViewItem the class is quiet complex, therefore we have to acknowledge the trade offs when deciding to re-implement the wheel as opposed to inheriting.

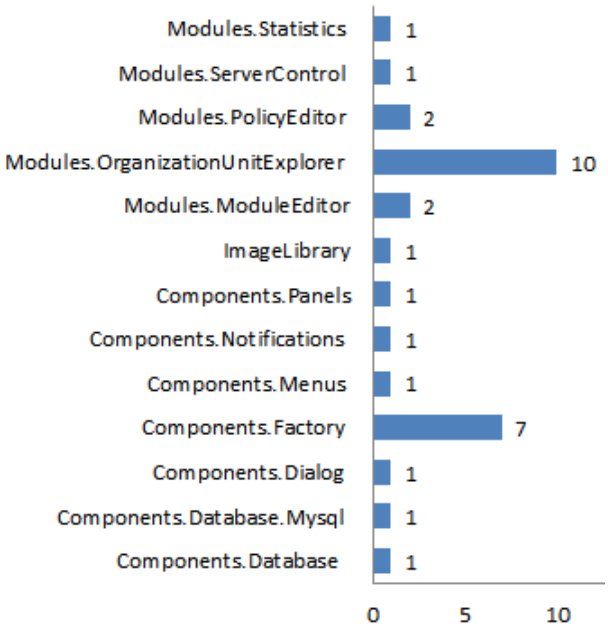

Fig. 4.5: Depth of Inheritance

```
/* Threading.DispatcherObject
    ^ DependencyObject
      ^ Media.Visual
        ^ UIElement
          ^ FrameworkElement
            ^ Controls.Control
              ^ Controls.ItemsControl
                ^ Controls.HeaderedItemsControl
                  ^ Controls.TreeViewItem
*/
class TreeViewOuElement : TreeViewItem , IOuObserver

class DragAdorner : Adorner
```

Fig. 4.6: Inheritance Coupling

From the two class definitions in Fig. 4.6 we can see the result of this. TreeViewOuElement extending from TreeViewItem introduced nine inherited dependencies and the DragAdorner extending from Adorner introduced 6 dependencies inherited as part of the inheritance chain the Dot Net Framework.

An argument for this and the reason for the intentional rule violation in terms of TreeViewOuElement is that, to create all the functionality of a TreeViewItem at so that it conforms or encompasses all the functionality that is required to realise it as a TreeViewItem, is simply too much work. As indicated by the Dot Net Frameworks TreeViewItem the class is quiet complex, therefore we have to acknowledge the trade offs when deciding to re-implement the wheel as opposed to inheriting.

A second argument, and probably the best approach would have been to create a collection class with pair mappings of IOu and TreeViewItem. As indicated by the interface realisation in 4.5, the reason for the inheritance was to add a member IOu and the appropriate accessor and mutator. By creating a collection class with a hash table for instance, the separation of TreeViewItem & IOu could have been achieved. The appropriate accessor & mutator could have been encapsulated by the collection class and indirection used to access the appropriate pair IOu. Unfortunately, this was over looked during development, but is easily fixed in future releases.

NDepend gives a more in depth analysis of the application class coupling, in section. 7.1.1. In fact according to the NDepend qualitative definitions only the TreeViewOuElement is in violation of this rule.

4.2.5 Class Coupling

- ***Class Coupling***
  - *Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration.*
  - *Good software design dictates that types and methods should have high cohesion and low coupling.*
  - *High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types*

– MicrosoftCodeMetricsValues

Interpretation

This indicates the total number of dependencies that an item has on other types. This number excludes primitive and built-in types such as Int32, String and Object, However does not exclude compound inbuilt types such as Button.

Analysis & Improvement

It is important for the reader to be aware that inbound or afferent coupling is to be observed on the Factory, while efferent or outbound coupling from the components.

Unfortunately visual studio does not distinguish between developer types and inbuilt .Net framework types. Furthermore the disbursement between Afferent Coupling (inbound) & Efferent Coupling (outbound) and other concerns such as derived coupling, is not immediately indicated. A lower value indicates the reusability of a class.

From the metrics provided by NDepend in section. 7.1.3 a better analysis can be observed. There are areas from improvement and some refactoring to resolve these areas should be done in successive releases.

Areas of high coupling are indicated in red. Once again due to the composite nature of the application; coupling seems rather high due to afferent and efferent coupling. In the future releases a possible reduction of architectural interfaces could help to reduce the coupling.

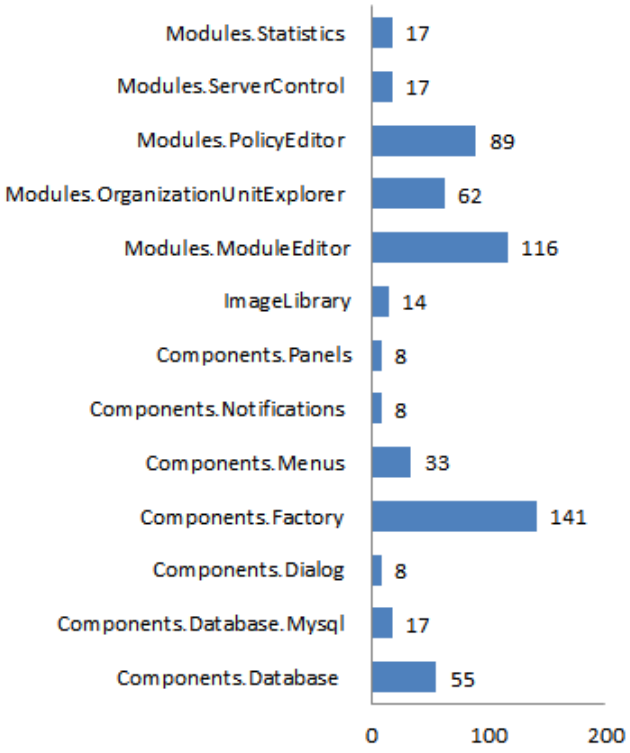

Fig. 4.7: Class Coupling

There are many methodologies for interface reductions such as described in A Simple Process for Specifying Component-Based Software. John Cheesman (2000)

4.2.6 Source Lines Of Code

There is are two types of SLOC measurements, physical SLOC (LOC) and logical SLOC (LLOC). Specific definitions of these two measurements vary, however physical SLOC is a count of lines in the source code including comment lines.

Interpretation

In SLOC blank lines are also included, if blank lines exceed 25% they are not counted toward lines of code. This is the simplest form of metric, this quantitative metric usually indicates the scale of the project, in terms of size.

Improvement Points

During the calculation of these metrics the blank lines of code were left out by source monitor, contrary to the definition.

Language	Lines
C#	25968(44996)
Perl	7076
Xaml	2353
Resx Language Translations	133251

Table 4.2: SourceMonitor - Source Lines of code - SLOC

For the C# analysis the larger of the two indicators 44,996, includes generated designer files generated by visual studio, but modified during development. There is not much we can tell from this information other than the size may indicate maintainability and understandability problems.

4.2.7 Line Spread

This section shows the spread of the source code lines amongst files, the file types, classes, interfaces etc. Documentation eg. (JavaDoc), comments and an introduction into the complexity of each file also.

Interpretation

From this information we can now begin to see a better overview of the source code analysis. The number of classes, interfaces and structs together with the number of lines per file may indicate the complexity as well as the maintainability of the source code. If the commenting and documentation is at a low level understanding the code may prove difficult.

Files	294
lines	44996
lines / file	153.05
Classes, Interfaces, Structs	243
Methods per Class	6.01
Comments / file	7.20
Documentation Lines / file	20.26
Maximum Cyclomatic Complexity	92
Average Cyclomatic Complexity / file	15.68

Table 4.3: SourceMonitor - Line Spread

Analysis

From these statistics we can ascertain that approximately 20%(153/27) is made of documentation and commenting. Typically a comment percentage less than 10 percent is considered insufficient. - (Resource Standard Metrics). Steve McConnell in his book *Code Complete* recommends about 7 methods per class due to people being able to remember 7 items. McConnell (2004). From these statistics we comply with this recommendation.

Steve McConnell also recommends that a method should be no larger than what can be displayed on a screen - McConnell (2004). This makes perfect sense to me, in that a programmer does not want to be scrolling up and down as it impairs concentration. Taking 153 lines per document and dividing it by 6 methods, we get an average of 25 lines. This is very comfortably displayed and leaves plenty of space to work with on a screen.

Improvement

Even though complying with the recommendation (20%) the spread of documentation and commenting percentage results, I would have to disagree with. The balance and quality of the commenting is more important than the amount. Too much commenting can make the code unreadable and as such should be clear and to the point. That said more commenting(not documentation) is needed, but a focus on quality comments, not metrics.

4.2.8 Methods & Statements

In computer science statements are one of two types, simple statements such as return, method calls and assignments. Examples of compound statements are if, switch and while. In this section we are primarily concerned with compound statements and the resulting branching or Nested Block Depth (NBD). An example of which can be seen in Fig. 4.8.

```
namespace blocks
{
  class blocks
  {
    void block()
    {
      // block 0
      for(;;)
      {
        // block 1
        if(true)
        {
          // block 2
          if(true)
          {
            // block 3
            if(true)
            {
              // block 4
              if(true)
              {
                // block 5
              }
            }
          }
        }
      }
} } }
```

Fig. 4.8: Statement nesting

Interpretation

Analysing the code and discovering the Nested Block Depth (NBD) or block level depth as seen in Fig. 4.8 can be used as an indicator for the perceived test difficulty.

A method with too much branching with many outcomes or directions can create problems when designing tests cases to test all the branches. The complexity of the resulting test can be in order as complex as the method. This outcome generally indicates with high probability that refactoring is needed.

Analysis

With the recommended Nested Block Depth being 5 Jurgen (2007), the analysis indicates that the code is within this recommendation. Roughly 1% of the code here is at block level 6. If a block of code has over 5 nested blocks refactoring is needed. Jurgen (2007).

Lines of Code at the method level is recommended to have a maximum of 50 statements - If a method has over 50 statements it is suggested that the method be broken up for readability and maintainability. Jurgen (2007). Given this metric we can see that the analysis has on average 8.74 statements per method well within this cut off.

Statements	18053
Statements / file	74.23
Calls per method	1.26
Statements per Method	8.74
Statements at block level 0	22.99%
Statements at block level 1	19.01%
Statements at block level 2	13.42%
Statements at block level 3	15.90%
Statements at block level 4	12.69%
Statements at block level 5	15.1%
Statements at block level 6	0.89%
Average Maximum Block Depth	3.78
Average Block Depth	1.53

Table 4.4: Methods & Statements

Improvement

A method with too many responsibilities may be difficult to understand and in general indicates refactoring is needed. In the book *Refactoring - Improving the design of existing code* techniques such as "inline class" or "replace nested condition with guard clauses" Fowler (1999).

As illustrated by Fig. 4.9 28% of the code is at block level 4, 5 & 6, this could be further improved by replacing blocks 4, 5 & 6 with a delegated object method call or internal method call, Fig. 4.9.

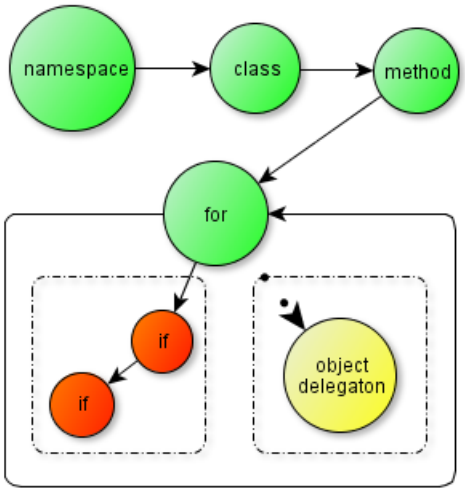

Fig. 4.9: Block level flow chart

Even though the metrics are in line with Jurgen (2007)'s research on static analysis and metrics; by analysing the final two metrics Average Maximum Block Depth (3.78) and Average Block Depth (1.53) it may be possible to refactor the implementation further. By negating blocks 4,5,6 as much as possible, the average maximum block depth and resulting overall average could be reduced making it easier to manage, debug, test etc.

4.2.9 Metrics Overview

An overall summary of the maintainability, cyclomatic complexity, depth of inheritance and class coupling as provided by visual studio.

Components	Cyclomatic Complexity	Class Coupling	Depth of Inheritance	Maintainability Index
Components.Database	469	55	1	71
Components.Database.Mysql	20	17	1	82
Components.Dialog	9	8	1	93
Components.Factory	545	141	7	95
Components.Menus	126	33	1	81
Components.Notifications	9	8	1	93
Components.Panels	9	8	1	93
ImageLibrary	13	14	1	66
Modules.ModuleEditor	314	116	2	72
Modules.OrganizationUnitExplorer	128	62	10	71
Modules.PolicyEditor	175	89	2	75
Modules.ServerControl	13	17	1	82
Modules.Statistics	15	17	1	82

Fig. 4.10: Visual Studio Metrics Overview

4.2.10 Code Coverage

Considering the quantity of code involved and the time frame available, the exploratory testing achieved a reasonable coverage analysis as illustrated in Figure 4.12. It's important to point out that during the tests, the majority of exploratory paths were attempted and the large percentage of the untested code is exception handling or non used interface methods that are realised by classes.

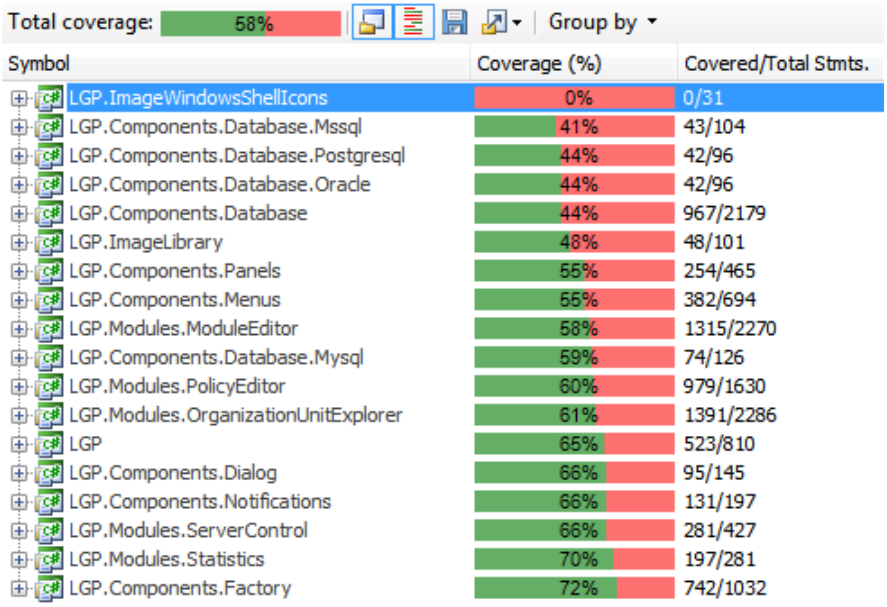

Fig. 4.11: Code Coverage Overview

Empirical studies pointed out the Bullseye Software Testing Group of real projects found that increasing code coverage above 70-80% is time consuming and not cost effective - Cornett (2011). The Risk assessment of a project should be ascertained, for instance aviation requires 100% coverage. As this is not a high risk product and people are not going to die (my supervisor suggested I personalise this a little), 100% code coverage is not required.

Bullseye also go onto say 100% code coverage is estimated to expose 50% of the bugs, low code coverage indicates inadequate testing and high code coverage guarantees nothing - Cornett (2011). Bullseye and other static code analysis investors generally point towards the 90, 80, 70 rule of, 90% unit testing, 80% integration testing, and 70% system testing. Cornett (2011)

While it's not indicated in the post coverage analysis, a large proportion of the indicated untested code was tested during unit testing, in the development phase. During development better defensive programming was introduced on the discovery of bugs. As each successive bug was negated via better inspection (if-else) the quality improved. The testing of else blocks or catch blocks became increasingly more difficult to test and as indicated by the analysis.

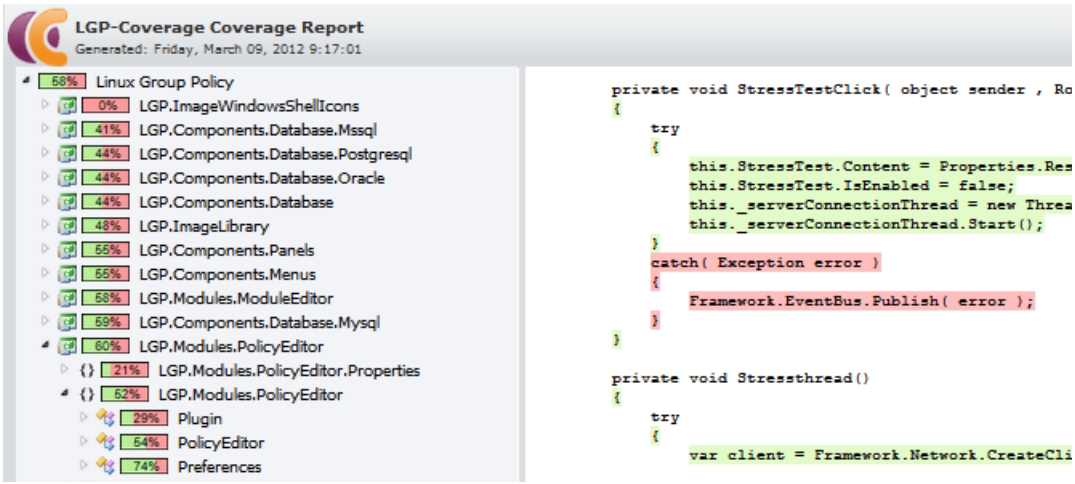

Fig. 4.12: Code Coverage Analysis

4.3 Qualitative Analysis

There are many software quality models that have been formulated for the determination of quality improvement and analysis. It's important to understand that when doing a qualitative analysis that "all models are flawed, but some are useful". In my undergraduate learning a software quality model was undertaken were I learned about Software Process Improvement(SPI) as well as some of the models available for this process.

A Taxonomy to Compare software process improvement(SPI) Frameworks by Christian Printzell Halvorsen and Reidar Conradi can be used in the determination of the suitable software process improvement(SPI) framework for an organisation, however this from an under graduate perspective, qualitative aspects required in the field are realistically unachievable such as Geographic origin/spread, scientific origin, development/stability or Popularity.

Similarly this issue arose in choosing a qualitative analysis model and as such the choice for the ISO/IEC 9126 for the evaluation of software quality was accepted due to the high level nature of the model.

ISO/IEC 9126 outlines the following characteristics and from a quick glance the reader should be able to identify that this quality model has a lot of the "ilities" or non functional requirements.

4.3.1 ISO/IEC 9126

- **Functionality**
  - Suitability
  - Accuracy
  - Interoperability
  - Security
  - Compliance
- **Reliability**
  - Maturity
  - Fault Tolerance
  - Recoverability
  - Compliance
- **Usability**
  - Understandability
  - Learnability
  - Operability
  - Attractiveness
  - Compliance
- **Efficiency**
  - Time Behaviour
  - Resource Utilisation
  - Compliance
- **Maintainability**
  - Analysability
  - Changeability
  - Stability
  - Testability
  - Compliance
- **Portability**
  - Adaptability
  - Installability
  - Co-Existence
  - Replaceability
  - Compliance

Functionality

The implementation met all the requirements set out in the objectives and as such is suitability functional. As security was not taken into account when under development, some changes have to be made in order to comply with the ISO/IEC 9126 model. Integration with site authentication mechanisms such as Kerberos (Public Key Cryptography) or some other public key technology would be the next logical step to ensure security compliance.

Reliability

Due the extensive testing during development most critical and bugs were identified and resolved and as a result the client server implementation as run for days consistently without any internal errors. Administrators who create modules will need to test their modules in a non live environment to ensure the stability and fault tolerance of these modules.

Usability

The choice to implement the administrator interface using C# and Windows Presentation Foundation (WPF) give greater accessibility to the non savvy administrators looking for an a bridge from the windows administration environment to the Linux environment. As such the administrator interface offers an attractable assessable means to accomplish greater learnability and understandability from a non Linux administrator.

Efficiency

The client server implementation has the ability to serve 1000 requests and replies in an 8 second period. This is more than sufficient to handle a large network environment. A trade off had to be made for the implementation in terms of portability and extensibility, in that Perl was chosen instead of per say C++. As PERL is an interpreted language is does not have the same efficiency as a compiled language such as C++. In a large scale network, administrators would not rely on any one server to service all these requests and in this case a round robin in the domain name service is set up to help load balance requests on servers. In this case the scalability of the implementation supersedes and negative efficiency factors.

Maintainability

As per a modular design you would expect to see no coupling of the components, this is illustrated by the dependency graph in Fig. 4.13 & 4.14. This offers developers to ship updates to individual components.

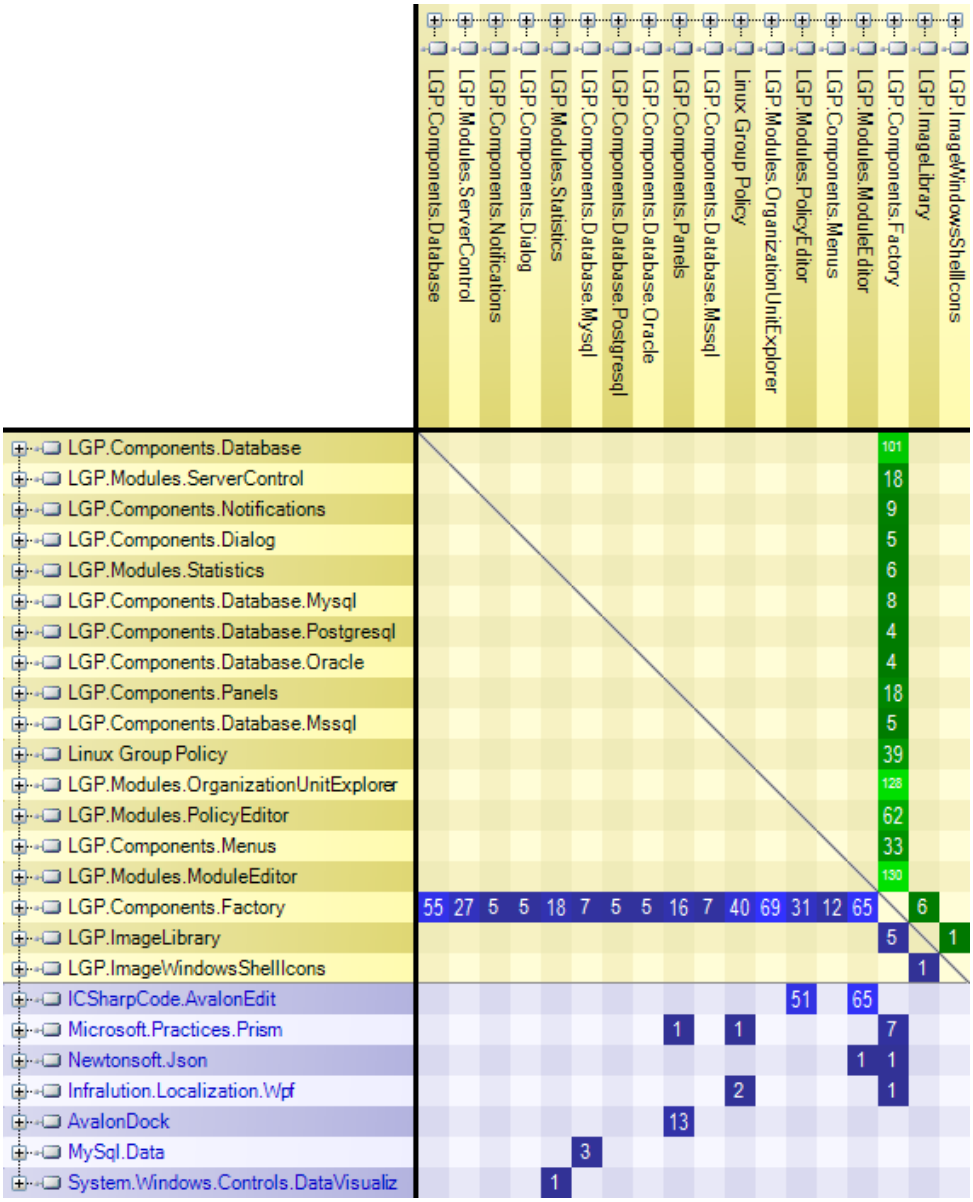

Fig. 4.13: Dependency Methods Relationship Diagram

Fig. 4.13 shows the method relationship or method call coupling between each of the composite components, while Fig. 4.14 shows the type relationship or coupling between each of the composite components.

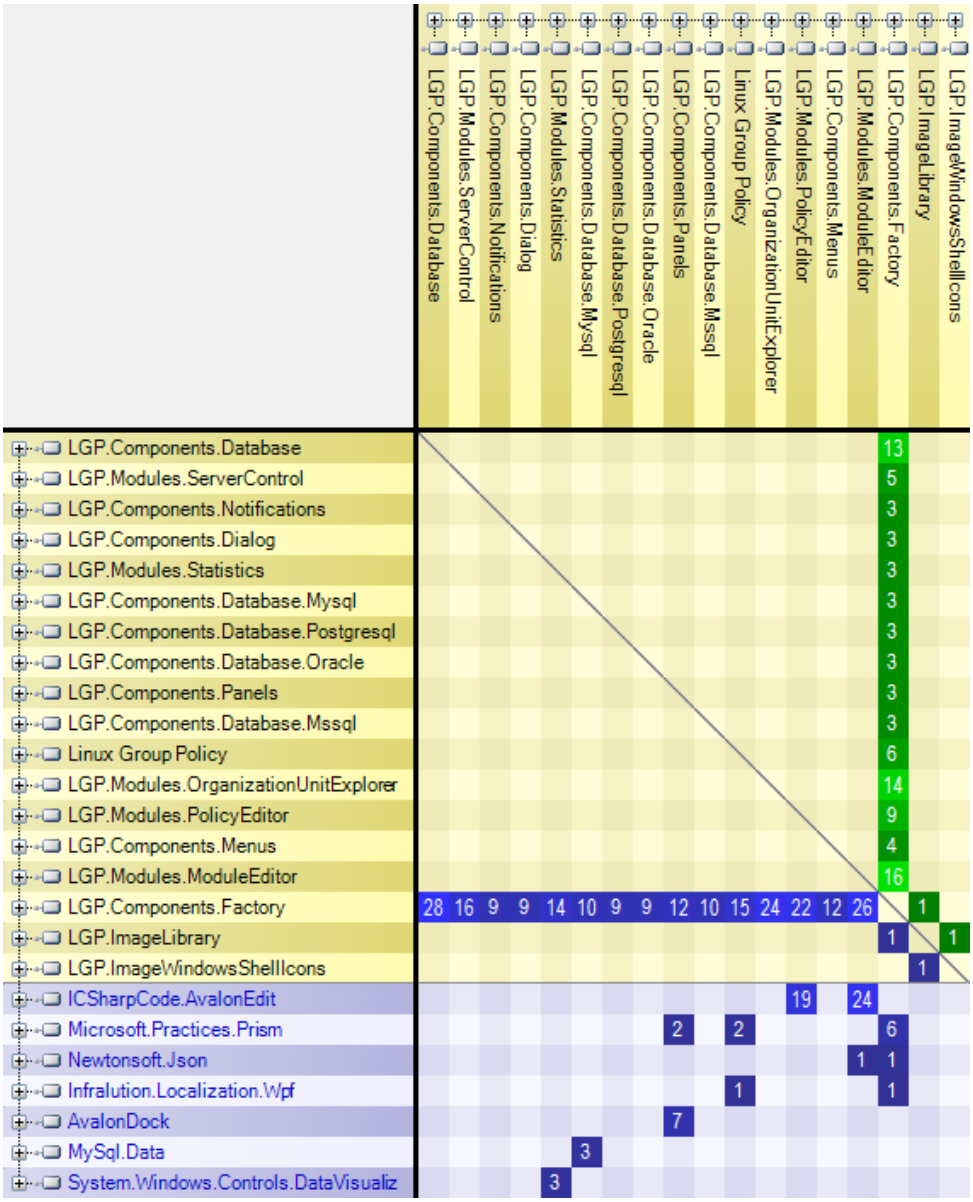

Fig. 4.14: Dependency Types Relationship Diagram

From these diagrams it is clearly indicated that there is no coupling of the implementation's composite components and as previously stated this allows me to develop a composite component further and ship this update separately if necessary.

Portability

The client server product is based upon Perl for maximum portability. Since all major distributions ship with Perl as default this ensure maximum portability. Further more the Comprehensive Perl Archive Network (CPAN) is accessible by Perl as default and allows administrators to update and install Perl modules relatively easily.

Vendor specific PERL implementations such as ActiveState also provide this functionality of updating modules on the fly. As there is no compilation needed the installability of the client is purely dependant on the existence of Perl on the particular distribution.

Due to the modular nature of the client and the associated interpreter, the replaceability and adaptability of the client server implementation is a factor of the administrators ability to create modular tailored for the organisation's environment.

In the long run, modules would be kept in a Comprehensive Perl Archive Network (CPAN) fashion on an external server. Administrators mere only search this online repository for modules that can be integrated into their environment giving maximum replaceability and adaptability.

Project Management

PMBoK describes Project management as the application of knowledge, skills, tools, and techniques to project activities in order to meet or exceed stakeholder needs and expectations from a project.

Meeting or exceeding stakeholder needs and expectations invariably involves balancing competing demands among Scope, time, cost & quality. - (PMBoK)

In this section we will look at the two excerpts from project management relevant to the successful completion of this product & the report.

- **Scope, time, cost, and quality**
- **Change control**

Scope, Time, Cost & Quality

Scope :

- *Initiation - committing the organisation to begin the next phase of the project*
- *Scope Planning - developing a written scope statement as the basis for future project decisions*
- *Scope Definition - subdividing the major project deliverables into smaller, more manageable components*

Fig. 4.15: Scope

- *Scope Verification - formalizing acceptance of the project scope*
- *Scope Change Control - controlling changes to project scope*

- (PMBoK)

The decision to commit to a project is highly dependant on the scope and associated skills. As the project was adequately defined, planned and verified this negated any possible side effect on time and cost.

Time :

- *Activity Definition - identifying the specific activities that must be per-formed to produce the various project deliverables*
- *Activity Sequencing - identifying and documenting interactivity dependencies*
- *Activity Duration Estimating - estimating the number of work periods which will be needed to complete individual activities*

Fig. 4.16: Time

- *Schedule Development - analyzing activity sequences, activity durations, and resource requirements to create the project schedule*
- *Schedule Control - controlling changes to the project schedule*

- (PMBoK)

Time is a major concern in final year projects and as such the decision to undertake the project was carefully considered. Of the 5 lecturers I spoke too, one of them said outright it would not be possible, another said it would be a PHD project, while three of the lecturers exhibited mixed reactions towards to project.

The Gannt chart in Fig. 4.19 however help me to realize the project was feasible in the allotted time frame. I was confident in my skills and believed through previous experience that the project was very feasible; Although a little more than I wanted to do in order to achieve a successful result.

Cost :

- *Resource Planning - determining what resources (people, equipment, mate-rials) and what quantities of each should be used to perform project activities*
- *Cost Estimating - developing an approximation (estimate) of the costs of the resources needed to complete project activities*
- *Cost Budgeting - allocating the overall cost estimate to individual work items*
- *Cost Control - controlling changes to the project budget*

Fig. 4.17: Cost

- (PMBoK)

The cost factor for final year projects I believe has two aspects. Unlike the monetary aspects of commercial software, final year projects have to be balanced in-line with other work in other subjects. For the first semester of 4th year I achieved a semester grade of 3.6. So the cost was

relatively low in this regard. The second cost for me was hair loss, i gave it about 25%, I think I lost around 10%, so it has been a "win win" in terms of cost so far!

Quality :

- *Quality Planning - identifying which quality standards are relevant to the project and determining how to satisfy them*
- *Quality Assurance - evaluating overall project performance on a regular basis to provide confidence that the project will satisfy the relevant quality standards*

Fig. 4.18: Quality

- *Quality Planning - identifying which quality standards are relevant to the project and determining how to satisfy them*
- *Quality Assurance - evaluating overall project performance on a regular basis to provide confidence that the project will satisfy the relevant quality standards*

- (PMBoK)

The result when balancing from the outcomes of scope achievement, cost and time management determine the quality of the project in project management. As all the requirements of the scope were met, it was achieved ahead of time and the cost was low, the quality of the project and its outcomes are deemed to be high quality!

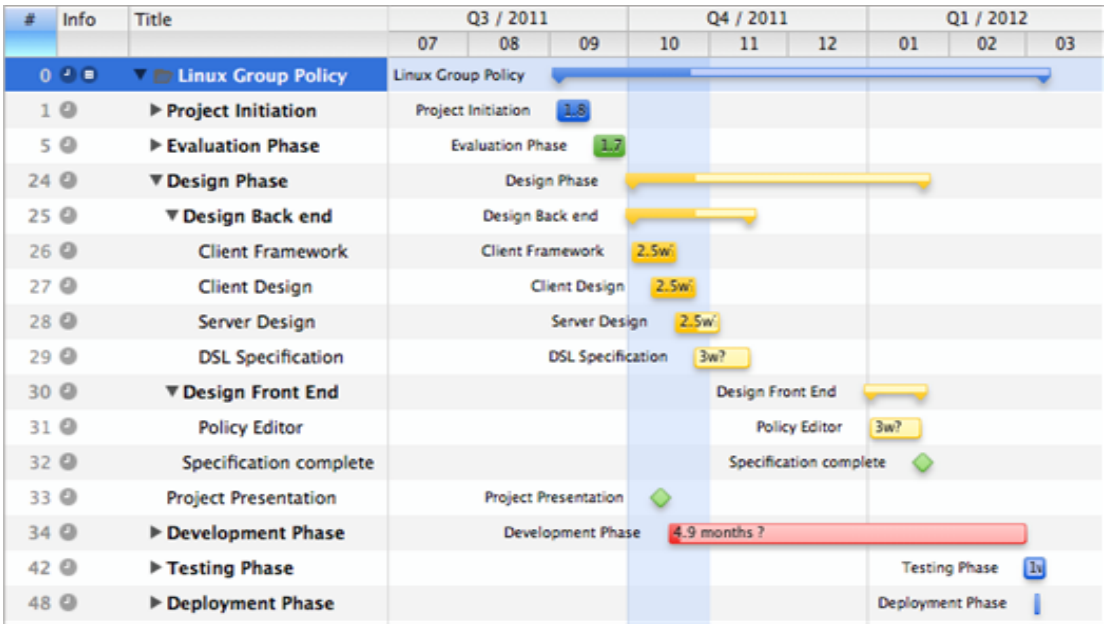

Fig. 4.19: Gannt Chart

4.3.2 Change Control

As the project manager is was my job to actively manage change control. My supervisor tried a few times to change some of the requirements for which I was in favour and in some cases against. I knew the time, cost and scope of what had to be done and in most cases I rejected the changes put forth.

Some projects have difficulties due to project managers accepting any and all change requests resulting in disillusioned, lack of direction development. I was aware of this from previous experience and from the learning outcomes of the Project Management module I had done previously.

The choice for me when accepting a change, is generally quiet simple and although may seem like the "lazy" approach, it generally results in a favourable outcome. Only minor changes would be accepted and they would result in less scope. The reason for this is based upon, how much development needs to change, and always to take less work, not more work. Therefore if a change control results in less work in the future with an initial increase in work this is deemed acceptable. This approach generally increases the overall quality of the software.

Version Control

We can see from the commit history in Fig. 4.20 that the project had finished in early February, approximately 2 weeks ahead of schedule.

We can see the time-line of the development and that most of that work had been done in the winter holiday period. Fig. 4.21 shows the lines of code and churn level also indicating the increase of change during the lead up to the end of development.

By using a version control software it allows me, the developer to provide the solution to multiple customers at different stages of development. By creating forks at specific versions allows me to fix bugs while still supporting the version that the customer has deployed.

Eventually a subsequent merge of these forks would be done in order to reintegrate the software as a whole. Too many forks are difficult to manage and merge.

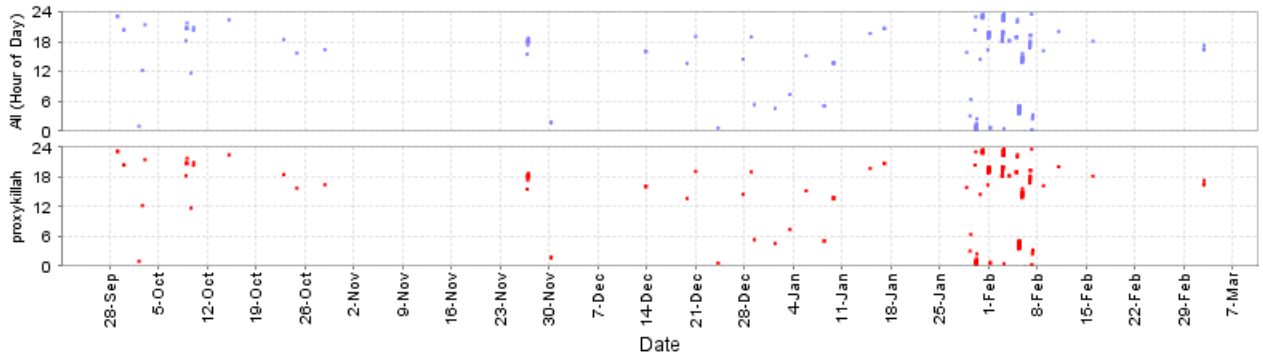

Fig. 4.20: Commit activity time line

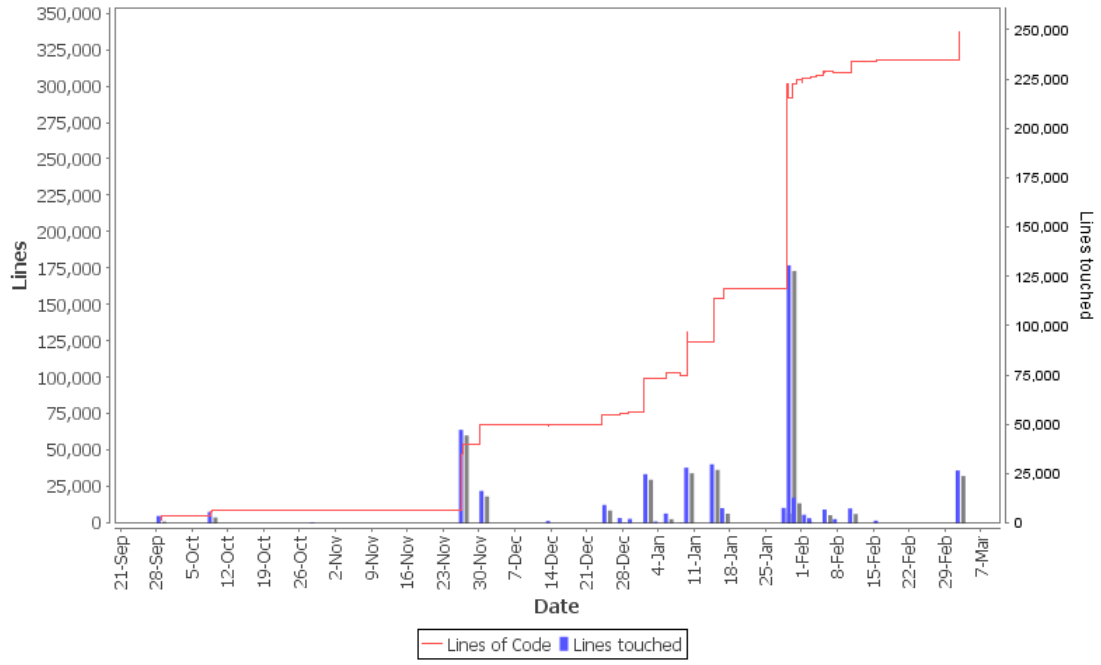

Fig. 4.21: Lines of code & churn level

I have also provided Fig. 4.22 an analysis of the daily work & Fig. 4.23 the hourly rate of work to get a more discrete breakdown of the software development life cycle.

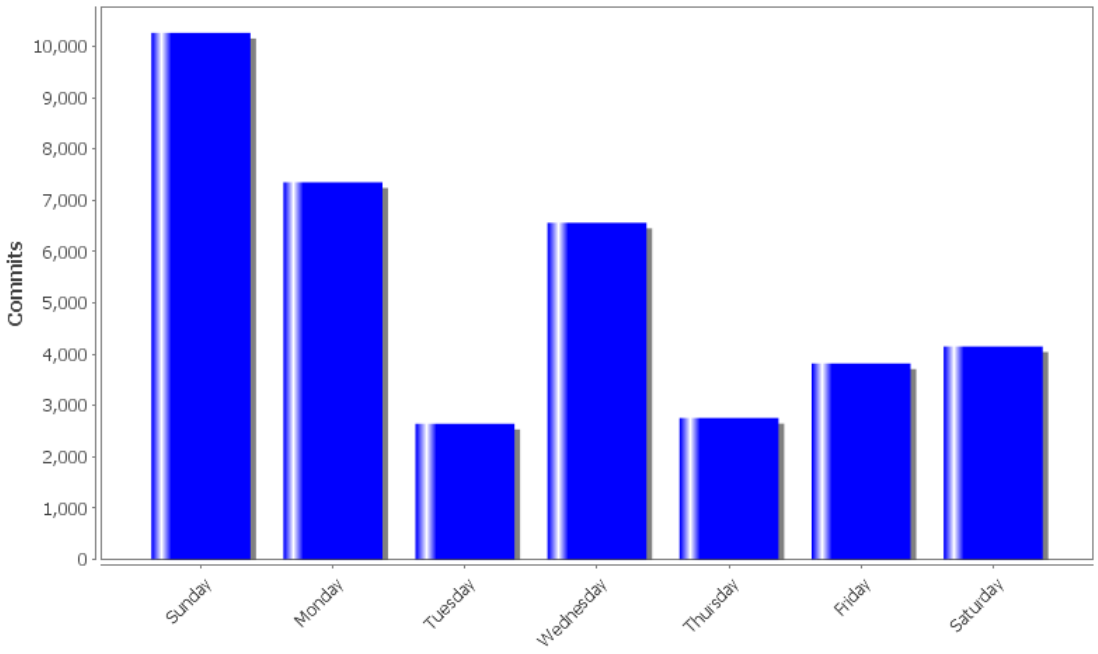

Fig. 4.22: Commit activity by day of the week

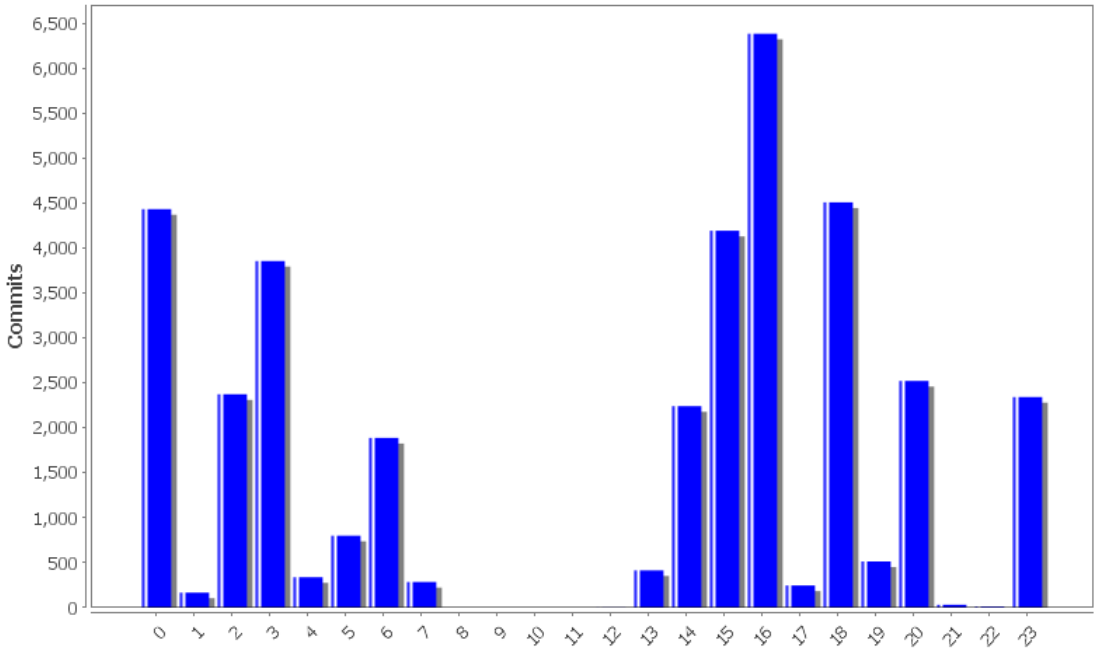

Fig. 4.23: Commit activity by hour of the day

5 Discussion and Conclusion

Providing a conclusion at this point is difficult considering the amount of material covered. Therefore in this section as part of the achievements, I will provide a simplified requirements mapping. Finally a critique and improvements points will be covered.

5.1 Achievements

The following tables directly maps to the requirements in section 1.3 (scope).

Client Server Model

#	Achieved	Notes	Supporting Figures
1	Yes	5 distributions tested for demo day	6.9
2	Yes	Modules can be pushed to clients	6.15
3	Yes	Yes PERL was used	6.6
4	Yes	Server tested to handle 6,000 clients	3.12, 3.13 & 3.14
5	Partially	Extension points in the architectural design provides for the master slave paradigm	**??**
6	Yes	Support was catered for in state transitions points	**??**
7	Yes	Interpreted language used, deployable to multiple architectures	
8	Yes	Push and pull implemented	6.15

Table 5.1: Requirements Mapping - Client server model

The client server model as detailed in the development chapter is highly successfully. During the requirements capture it was deemed the implementation would need to be able to handle 100's of clients. In the testing phase it was discovered that the architecture far exceeded this goal and is able to handle thousands of clients.

Domain Specific Language

#	Achieved	Notes	Supporting Figures
1	Yes	DSL is extensible	3.16 , 3.15
2	Yes	Client accepts new modules	3.16 , 3.15
3	Yes	DSL is an PERL filter script and allows embedded PERL	3.17

Table 5.2: Requirements Mapping - Domain specific language

With the use of PERL filters the domain specific language exhibited both characteristics of an internal and external domain specific language. As the domain specific language is interpreted by the PERL interpreter, this allowed for the embedding of PERL, resulting in maximum flexibility.

Directory Services Schema

#	Achieved	Notes	3.37
1	Yes	Configurations represented in DB	3.37
2	Yes	Yes module provides translation	3.37
3	Yes	Yes organisational units represented	3.37
4	Yes	Client represented by GUID	3.37

Table 5.3: Requirements Mapping - Directory services schema

The database schema although ending up be quiet simple, provided support for all the necessary requirements.

Administrator Front End

#	Achieved	Notes	Supporting Figures
1	Yes	Ability to create organisational units	6.11
2	Yes	Move organisational units	6.13
3	Yes	Hierarchy of organisational units	6.8, 6.9 & 6.13
4	Yes	MUST provide a means to modify policies	6.14
5	Yes	Generate new domain specific language	6.14
6	Yes	Import existing policies in the form of the DSL	6.14
7	Yes	Push updates to the client	6.15

Table 5.4: Requirements Mapping - Administrator front end

All the requirements for the Administrator front end were achieved. However the true achievement is the architecture. This architecture provides maximum flexibility and support for non functional requirements and resulting future improvements.

5.2 Critique & Improvements

There are numerous critiques of the software implementation. To begin with the implementation a primary critique point of the client, server architecture is the message passing. The messages as previously discussed are encapsulated in a Javascript Object Notation (JSON) envelope. This envelope and the construction of its contents (key/value pairs) is hard coded in the implementation. This does not offer out of band services to append to the (key/value) pairs. New services that require this functionality are therefore not catered for. Future improvements to the framework should provide the ability to append to this envelope.

Patterns and supporting technologies such as the Common Object Request Broker Architecture (CORBA) should be considered to support object passing.

A possible problem associated with the interpreter is the use of regular expressions. Regular expressions although offering the best validation method for strings can become difficult to

understand and maintain. Other languages such as Prolog provide the ability to define definite clause grammars more easily. For future improvements more research should be done to determine the best solution to defining rules in a more readable manner.

Finally the last and most overlooked software concern of this project was the database requirement. Although the implementation catered for the requirements, further research needs to be done in order to better improve the representation schema for rules. To provide the most flexibility during development the rules associated with an organisational unit were stored in a single field in the database. Although offering the best flexibility future improvement could be done to represent each line and its individual parts. Database internal constraints could then be used to further validate the data.

5.3 Conclusion

From the outset, the scope of this project was to design an extensible software framework and to provide a Domain Specific Language(DSL) as a means to abstract distribution specific knowledge. These two points were successfully achieved in both the administrator front end and the client interpreter, leading to a uniform acknowledgement of quality, of the demonstrated product.

Bibliography

C# delegates and events in depth, a. URL http://www.ikriv.com/en/prog/info/dotnet/Delegates.html.

Wpf - eventaggregator in prism 4.0, b. URL http://shujaatsiddiqi.blogspot.com/2010/12/wpf-eventaggregator-in-prism-40-cal.html.

G. Morrisett A. Basu, M. Hayden and T. von Eicken. A language-based approach to protocol construction.

Ed A. Sciberras. Lightweight directory access protocol (ldap): Schema for user applications. *Network Working Group*, 2006.

P. Klint A. van Deursen and C. Verhoef. Research issues in software renovation. *Fundamental Approaches to Software Engineering*, 1999.

Leonard M. Andany, J. and C Palisser. Management of schema evolution in databases. *17th International Conference on Very Large Databases*, 1991.

Chou H.-T. Garza J.F. Kim W. Woelk D. anerjee, J. and N Ballou. Data model issues for object-oriented applications. *ACM Trans*, 1987.

G. Arango. Domain analysis: From art form to engineering discipline. *ACM SIGSOFT*, 1989.

Dot Net Architecture. The publish subscribe pattern in c#. URL http://www.dotnetarchitecthouston.com/post/The-Publish-Subscribe-Pattern-in-C-and-some-gotchas.aspx.

G. Ariav. Temporally oriented data definitions: managing schema evolution in temporally oriented databases. 1991.

Ausiello C.-Batini C. Atzeni, P. and M Moscarini. Inclusion and equivalence between relational database schemata. *Theoretical Computer Science*, 1982.

P. Atzeni and V De Antonellis. Relational database theory. 1993.

R. Sethi A.V. Aho and J.D. Ullman. Compiler: Principles, techniques and tools. *Addison-Wesley*, 1986.

Chou H.-T. Kim H.J. Banerjee, J. and H.F Korth. Schema evolution in object-oriented persistent databases. *Advanced Database Symposium*, 1986.

Chou H.-T. Kim H.J. Banerjee, J. and H.F Korth. Semantics and implementation of schema evolution in object-oriented databases. *ACM SIGMOD conference*, 1987.

J. Banerjee. Data model issues for object-oriented applications. *ACM Computing Surveys*, 1987.

P. Barker. Dsa metrics (osi-ds 34 (v3)). *Network Working Group*, 1994.

D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, 1985.

D. Beech and B Mahbod. Generalised version control in an object-oriented database. *IEEE Computer Society*, 1988.

J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 1986.

J.A. Bergstra and P. Klint. The discrete time toolbus–a software coordination architecture. *Science of Computer Programming*, 1998.

F. Bertrand and M. Augeraud. Bdl: A specialized language for per-object reactive control. *DSL-IEEE*.

D. Bruce. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation. *Karmin*.

Mark BURGESS. Micro promises, 2011. URL `http://www.iu.hio.no/~mark/BookOfPromises.pdf`.

C. Cowan J. Walpole C. Pu, A. Black and C. Consel. Microlanguages for operating system specialization. *Kamin*.

L. Cardelli and R. Davies. Service combinators for web computing. *DSL-IEEE*.

E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 1990.

J. C. Cleaveland. Building application generators. *IEEE Software*, 1998.

Steve Cornett. Minimum acceptable code coverage, 2011. URL `http://www.bullseye.com/minimum.html`.

R. F. Crew. Astlog: A language for examining abstract syntax trees. *Ramming*.

K. Czarnecki and U. Eisenecker. Generative programming: Methods, techniques and applications. *Addison-Wesley*, 1999.

G. Bruns D. Atkins, T. Ball and K. Cox. A domain-specific language for form-based services. *DSL-IEEE*.

A. Rogers D. Bonachea, K. Fisher and F. Smith. Hancock: A language for processing very large-scale data. *DSL-99*.

J.-M. DeBaud and K. Schmid. A systematic approach to derive the scope of software product lines. *21st International Conference on Software Engineering*, 1999.

C. Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *DSL-IEEE*.

D. R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *DSL-IEEE*.

Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns : Elements of reusable object orientated software*. Addison Wesley, 1995.

R. Faith. A dictionary server protocol. *Network Working Group*, 1997.

M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 1997.

Martin Fowler. Event aggregator. URL http://martinfowler.com/eaaDev/EventAggregator.html.

Martin Fowler. *Rafactoring : Improving the design of existing code*. Addison Wesley, 1999.

Martin Fowler. *UML Distilled*. Addison Wesley, 2009.

Martin Fowler. *Domain Specific Languages*. Addison Wesley, 2010a.

Martin Fowler. *Patterns of enterprise application architecture*. Addison Wesley, 2010b.

Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. *Association for Computing Machinery*, 1998.

Greg Franks and Murray Woodside. Effectiveness of early replies in client-server systems. *Performance Evaluation*, 1999.

Greg Franks and Murray Woodside. Multiclass multiservers with deferred operations in layered queueing networks, with software system applications. *Twelfth IEEE/ACM International Symposium on Modeling*, 2004.

Freebsd. Network information system. URL http://www.freebsd.org/doc/handbook/network-nis.html.

M. Fuchs. Domain specific languages for ad hoc distributed applications. *Ramming*.

J. Lamping J.-M. Loingtier C. Lopes C. Maeda G. Kiczales, J. Irwin and A. Mendhekar. Aspect oriented programming. *Kamin*.

Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly, 1995.

M Garlaw D, Shaw. An introduction to software architecture. 1993.

Joseph A. Goguen and Charlotte Linde. Techniques for requirements elicitation. 1993.

Murray Woodside Jing Xu Greg Franks, Dorina Petriu and Peter Tregunno. Layered bottlenecks and their mitigation. *Quantative Evaluation of Systems*, 2006.

Maurice Howard Halstead. Halstead metrics. URL http://www.verifysoft.de/en_halstead_metrics.html.

Christian Printzell Halvorsen and Reidar Conradi. A taxonomy to compare spi frameworks. *Proceedings of the 8th European Workshop on Software Process Technology*, pages 217–235, 2001.

Jun Han. A comprehensive interface definition framework for software components. *Software Engineering Conference*, 1998.

Jun Han. An approach to software component specification. *International Workshop*, 1999.

Nis Handbook. Freebsd nis handbook. URL http://www.freebsd.org/doc/handbook/network-nis.html.

R. Hedberg. A tagged index object for use in the common indexing protocol. *Network Working Group*, 1999.

R. Housley. Internet x.509 public key infrastructure certificate and crl profile. *Network Working Group*, 1999.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 1996.

D. Hoffman J. Coplien and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 1998.

John Daniels John Cheesman. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Professional, 2000.

R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1988.

Gintenreiter Jurgen. Considerations regarding metrics in software development. *Bachelor-Semina*, 2007.

Ed K. Zeilenga. Lightweight directory access protocol (ldap): Additional matching rules. *Network Working Group*, 2004.

Joshua Kerievsky. *Refactoring to patterns*. Addison Wesley, 2004.

N. Klarlund and M. I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *DSL-IEEE*.

C. W. Krueger. Software reuse. *ACM Computing Surveys*, 1992.

D. A. Ladd and J. C. Ramming. Two application languages in software production. *USENIX Very High Level Languages Symposium Proceedings*, 1994.

D. Leijen and E. Meijer. Domain specific embedded compilers. *DSL-99*.

Moeller Th Lenz G. . *NET-A Complete Development Cycle*. Addison-Wesley Professional, 2003.

V. Gupta M. Fromherz and V. Saraswat. Cc - a generic framework for domain-specific languages. *Kamin*.

Steve McConnell. *Code Complete, Second Edition*. Microsoft, 2004.

Microsoft. Generics introduction, a. URL `http://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx`.

Microsoft. Activator.createinstance method, b. URL `http://msdn.microsoft.com/en-us/library/aa310404(v=vs.71).aspx`.

Microsoft. Routedeventhandler delegate, c. URL `http://msdn.microsoft.com/en-us/library/system.windows.routedeventhandler.aspx`.

Microsoft. Csharp volitile fields, d. URL `http://msdn.microsoft.com/en-us/library/aa645755(v=vs.71).aspx`.

MicrosoftCodeMetricsValues. Code metrics values. URL `http://msdn.microsoft.com/en-us/library/bb385914.aspx`.

L. Nakatani and M. Jones. Jargons and infocentrism. *Kamin*.

D; Powell S. Nurmuliani, N; Zowghi. Analysis of requirements volatility during software development life cycle. *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, 2004.

Stack Overflow. C# dynamic event subscription. URL http://stackoverflow.com/questions/45779/c-sharp-dynamic-event-subscription.

Code Project. Codeproject event aggregator, a. URL http://www.codeproject.com/KB/architecture/event_aggregator.aspx.

Code Project. C# event implementation fundamentals, b. URL http://www.codeproject.com/KB/cs/event_fundamentals.aspx.

Code Project. Reflection, c. URL http://www.codeproject.com/KB/cs/introreflection.aspx.

J. M. Bell J. Hook A. Kotov J. Lewis D. P. Oliva T. Sheard I. Smith R. B. Kieburtz, L. McKinney and L. Walton. A software engineering experiment in software component generation. *Software Engineering ICSE*, 1996.

L. S. Nyland R. E. Faith and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. *Ramming*.

W. Tracz R. N. Taylor and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 1995.

D. Roberts and R. Johnson. Evolve frameworks into domain-specific languages. *3rd International Conference on Pattern Languages*, 1996.

J.F Roddick. Dynamically changing schemas within database modules. *Australian Computer Journal*, 1991.

J.F Roddick. Sql/se - a query language extension for databases supporting schema evolution. *SIGMOD Record*, 1992.

B. Richards S. Chandra and J. R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. *DSL-IEEE*.

M. Smith. Definition of an x.500 attribute type and an object class to hold uniform resource identifiers (uris). *Network Working Group*, 1997.

M. Haveraaen T. B. Dinesh and J. Heering. An algebraic programming style for numerical software and its optimization. *Technical Report SEN-R9844*, 1998.

J. Takahashi. Hybrid relations for database schema evolution. *IEEE Computer Society*, 1990.

MICROSOFT TECHNET. Domain name system, a. URL http://technet.microsoft.com/en-us/network/bb629410.aspx.

MICROSOFT TECHNET. Server 2008 gloassry of terms. b. URL http://technet.microsoft.com/en-us/library/cc788089(WS.10).aspx.

A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. *Smalltalk and Java in Industry and Academia*, 1997.

A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 1998.

V. Ventrone and S Heiler. Semantic heterogeneity as a result of domain evolution. *SIGMOD Record*, 1991.

M. Wahl. A summary of the x.500(96) user schema for use with ldapv3. *Network Working Group*, 1997.

Dr. Gerald M. Weinberg. *Perfect Software: And Other Illusions about Testing*. Dorset House, 2008.

K. Zeilenga. Subentries in the lightweight directory access protocol (ldap). *Network Working Group*, 2003.

K. Zeilenga. Lightweight directory access protocol (ldap) cancel operation. *Network Working Group*, 2004.

K. Zeilenga. Lightweight directory access protocol (ldap) schema definitions for x.509 certificates. *Network Working Group*, 2006.

6 Appendix - Chapter 3

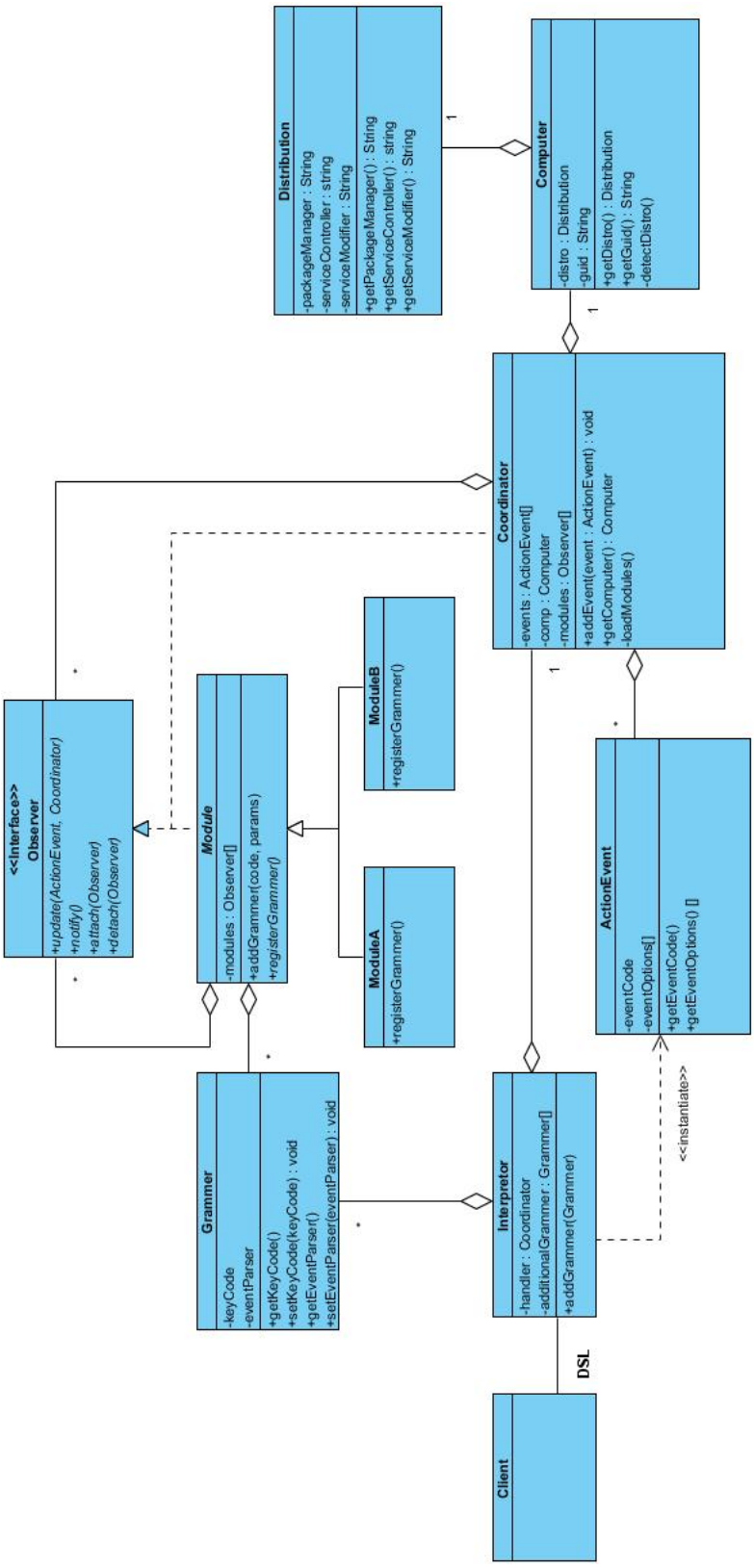

Fig. 6.1: Interpreter Framework

6.1 Screen Captures

In Fig. 6.2 we can see the application loading. The splash screen shows the dynamic link libraries being loaded. During this process types are inspected via reflection for architectural interfaces.

Fig. 6.2: Loading screen

In Fig. 6.3 the interface can be seen. At this stage the panels are not visible however are present and waiting for menu events to load the appropriate panes and place the UserControl into the panes tab control.

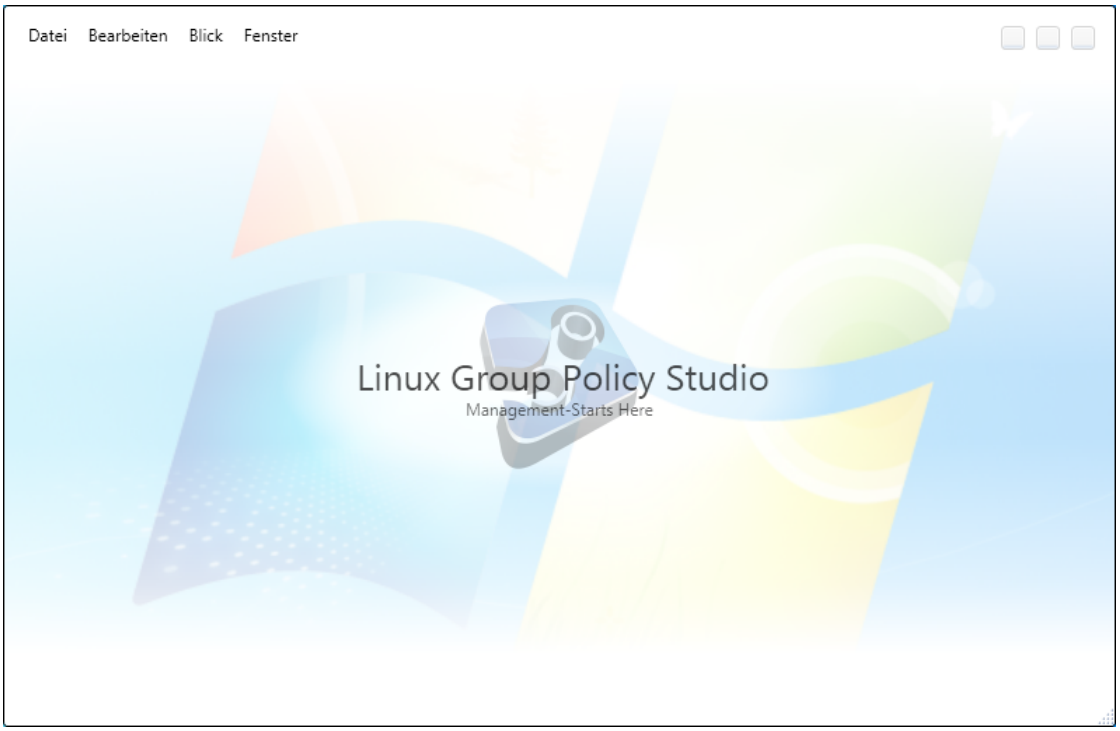

Fig. 6.3: Main window

Here in Fig. 6.4, the main preferences, the user is able to select the preferred language and choose a server to connect to.

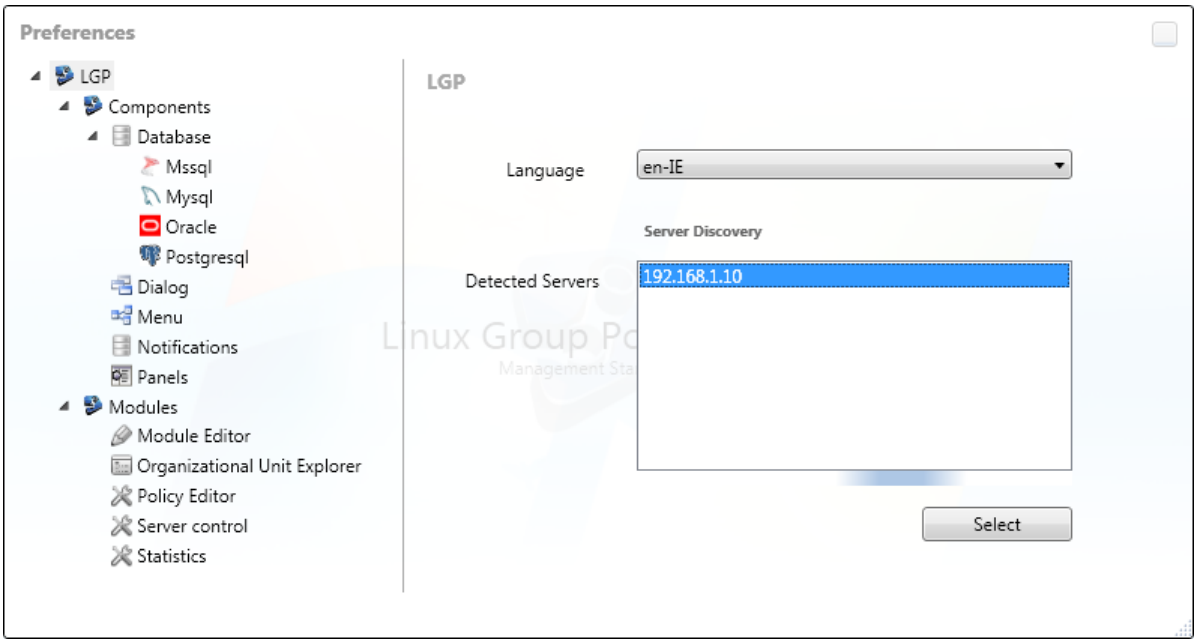

Fig. 6.4: Preferences Main

In Fig. 6.5, the detected database strategy contexts have been detected and selection of the preferred database type is chosen along with the typical database settings. On the left we can also see the modules themselves which may provide context specific configuration elements.

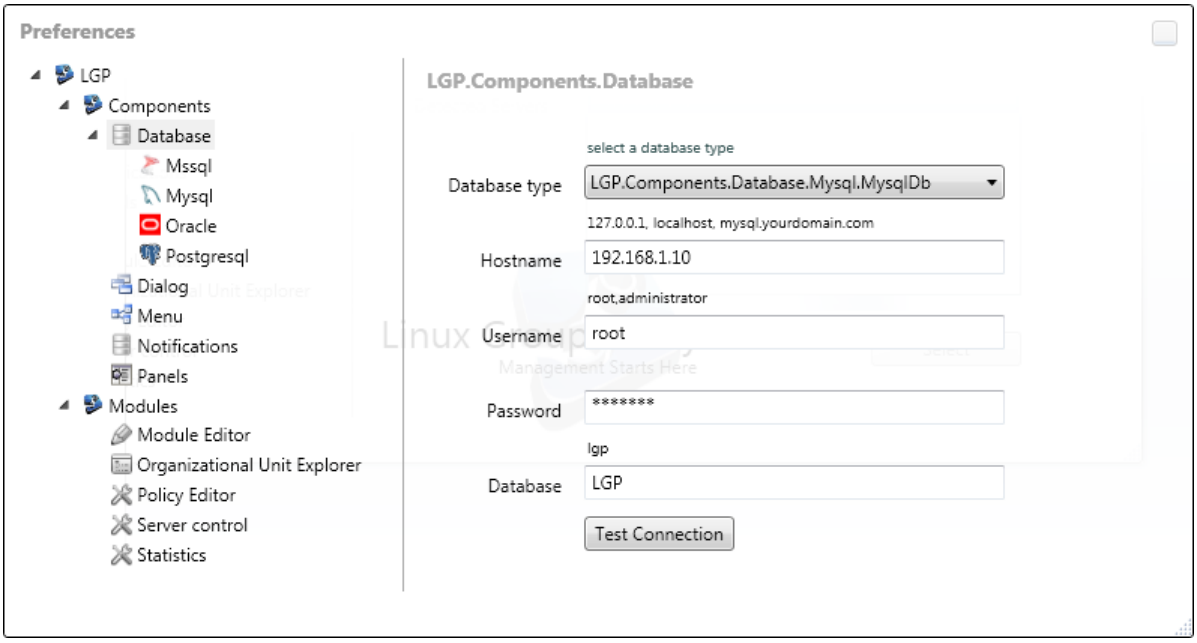

Fig. 6.5: Preferences Database

In Fig. 6.6 / 6.7, the module editor is visible, on the right hand side we have the list of modules on the server. On the right hand side is the detected grammar in the module, and available in the middle is the editing area.

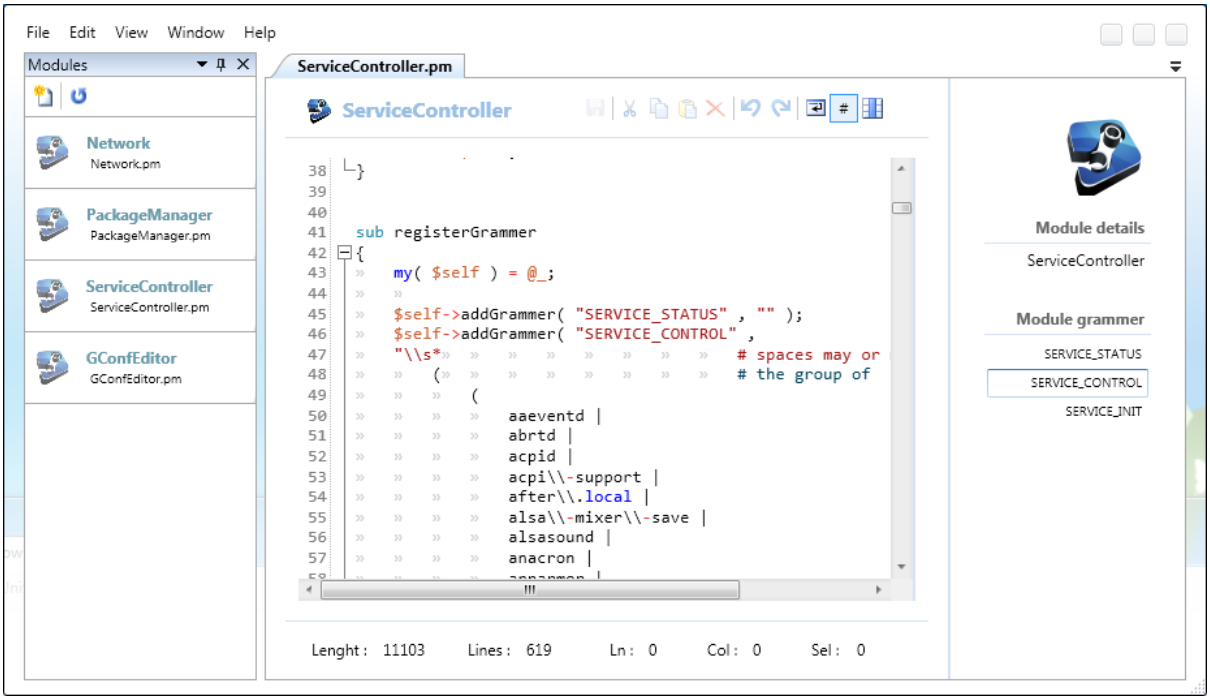

Fig. 6.6: Module editor

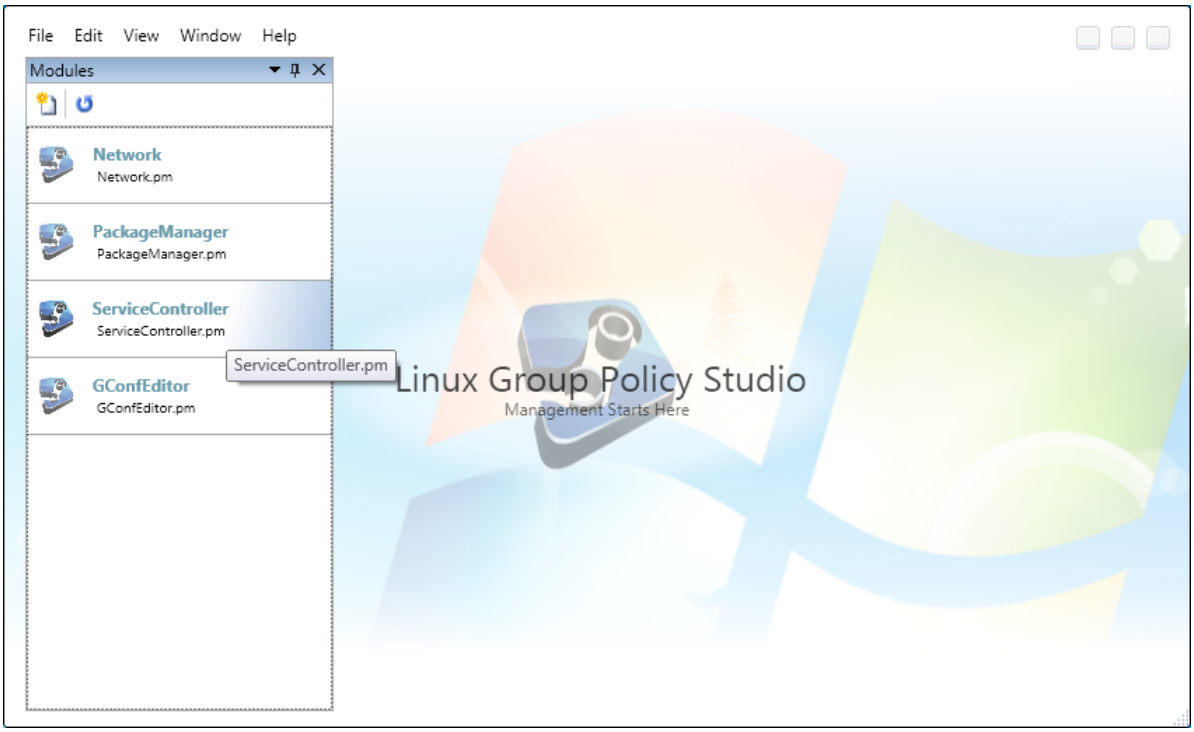

Fig. 6.7: Module list

In Fig. 6.8 / 6.9 , the admin is showing off the drag and drop adorner. A client machine is being drag into another ou, also the client information on the side has been activated. The unique machine identifier along with the distribution and Perl client info is available.

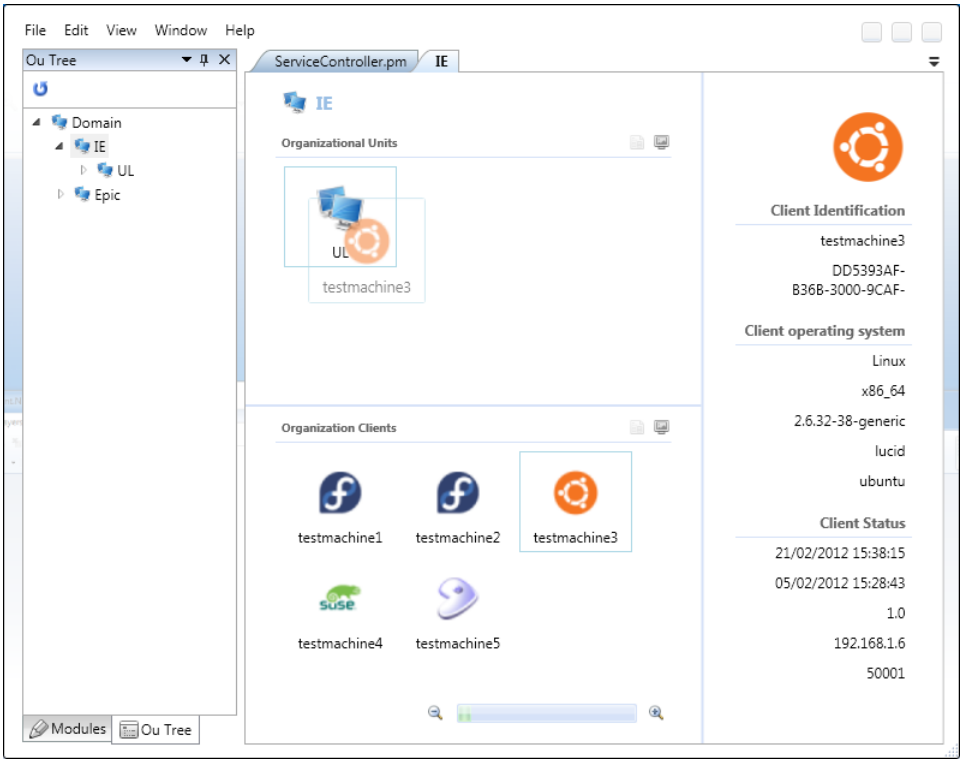

Fig. 6.8: Ou - Drag / drop

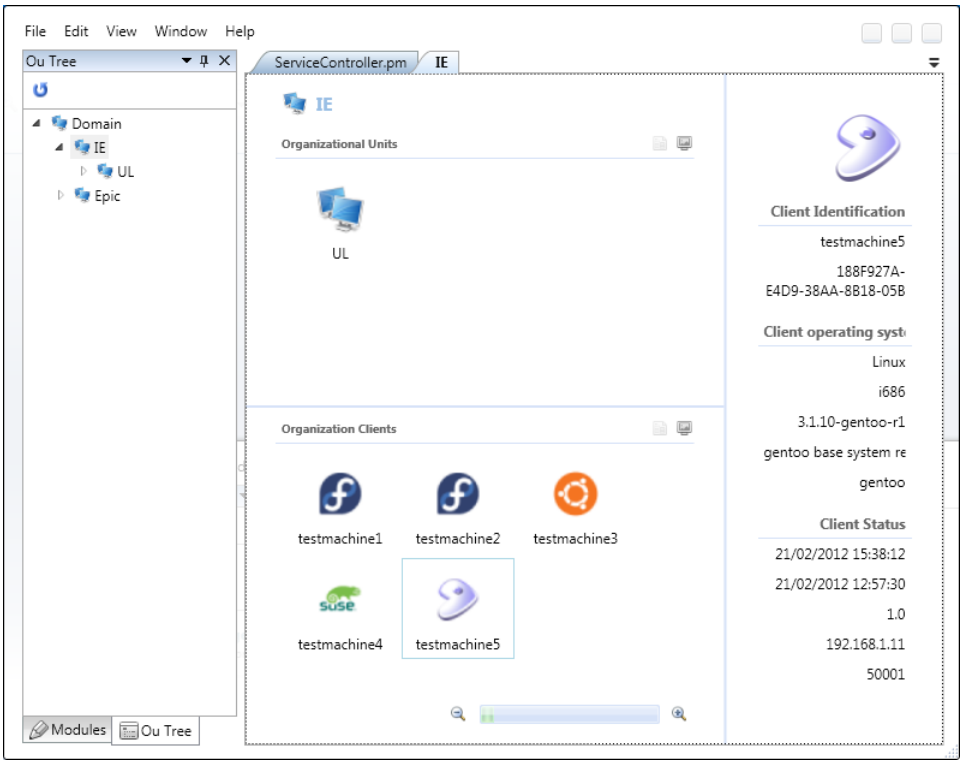

Fig. 6.9: Ou - Client

In Fig.6.10, The limited organizational unit information is available. In later revisions it will show clients and sub OUs etc. The organizational menu options are visible. The options provided by the Organizational Unit Explorer Plug-in are seen as well as menu options provided by the policy editor

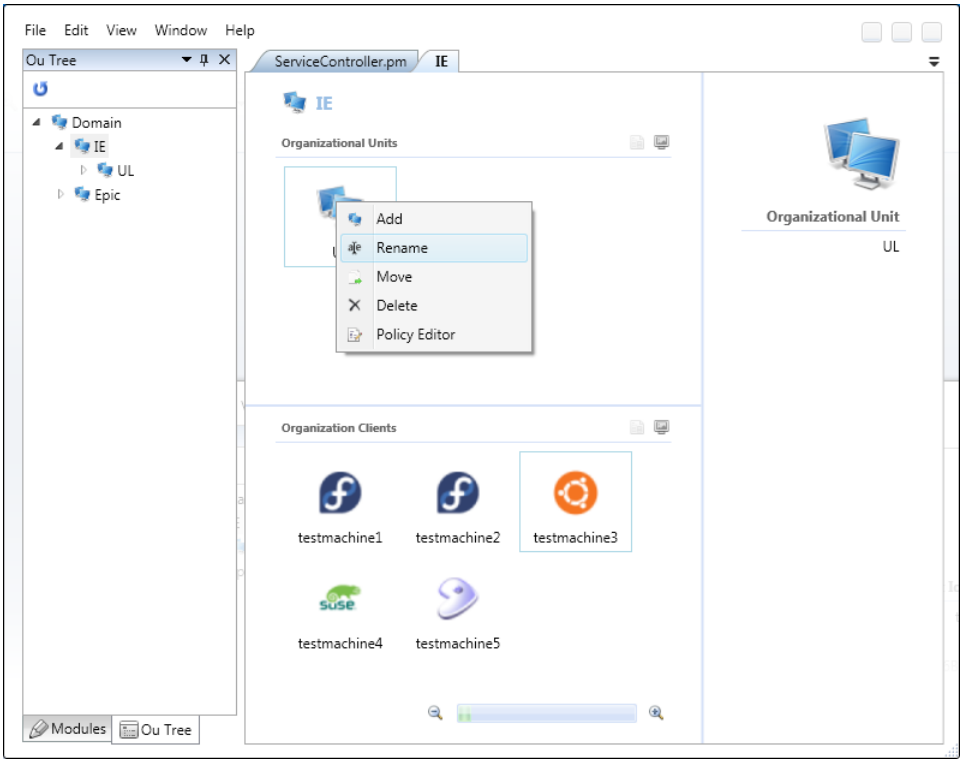

Fig. 6.10: Ou - options

The following three figures 6.11 / 6.12 / 6.13 , depict the organizational pop up windows for the adding, moving and deleting of OUs respectively.

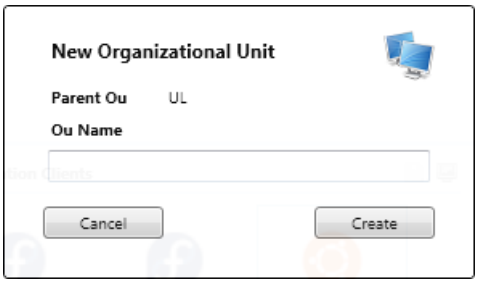

Fig. 6.11: Ou - add

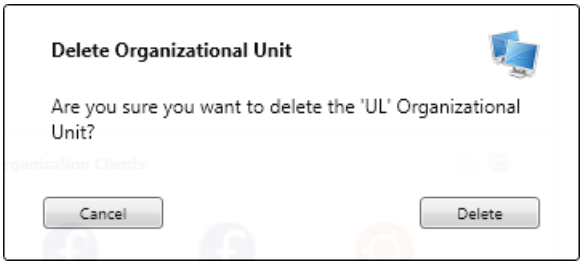

Fig. 6.12: Ou - delete

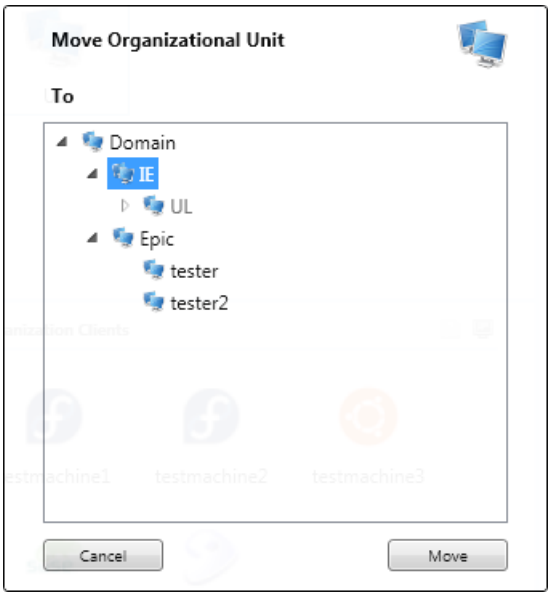

Fig. 6.13: Ou - move

Here in Fig. 6.14 the administrator has created an invalid rule as indicated by the red high light. The rules are visible on the right hand side as well as the code editor.

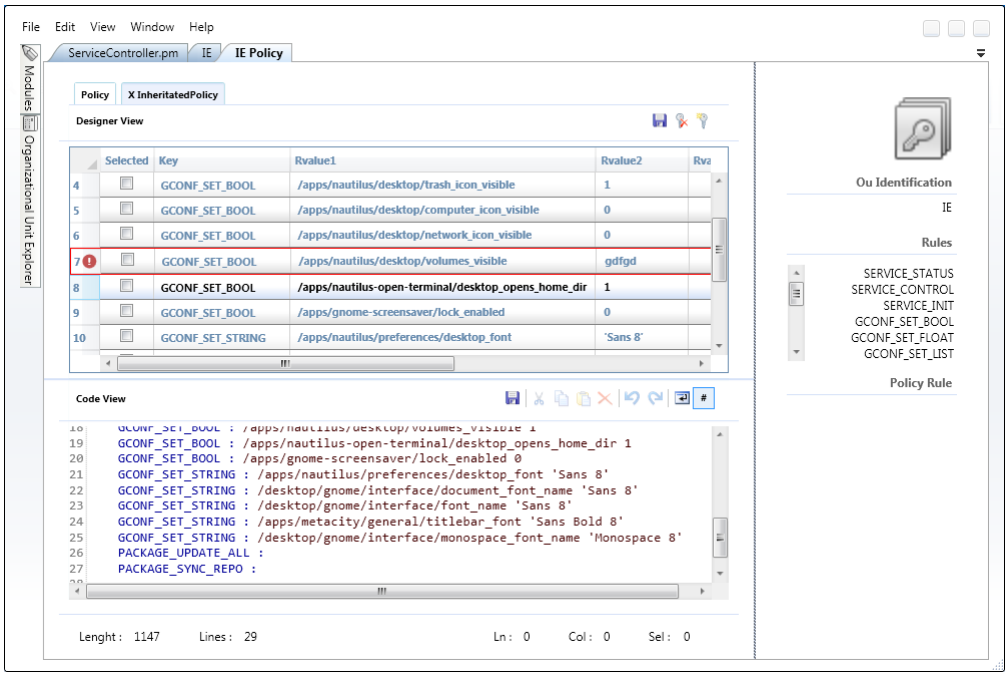

Fig. 6.14: Ou - policy editor

Fig. 6.15 depicts the push features provided by the Server Control Module Stress testing features are available also.

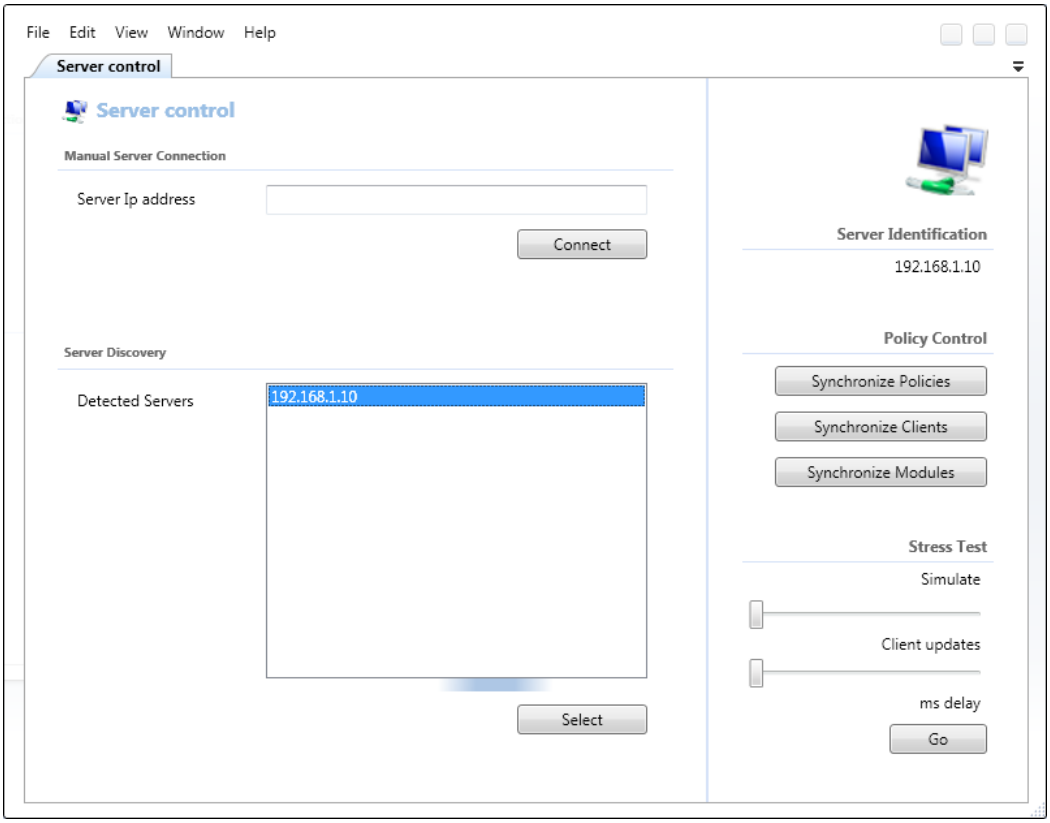

Fig. 6.15: Ou - Server Control

Fig. 6.16 shows the monitoring of the server, three simple statistics are provided, the queue size on the server, network usage and the processor usage.

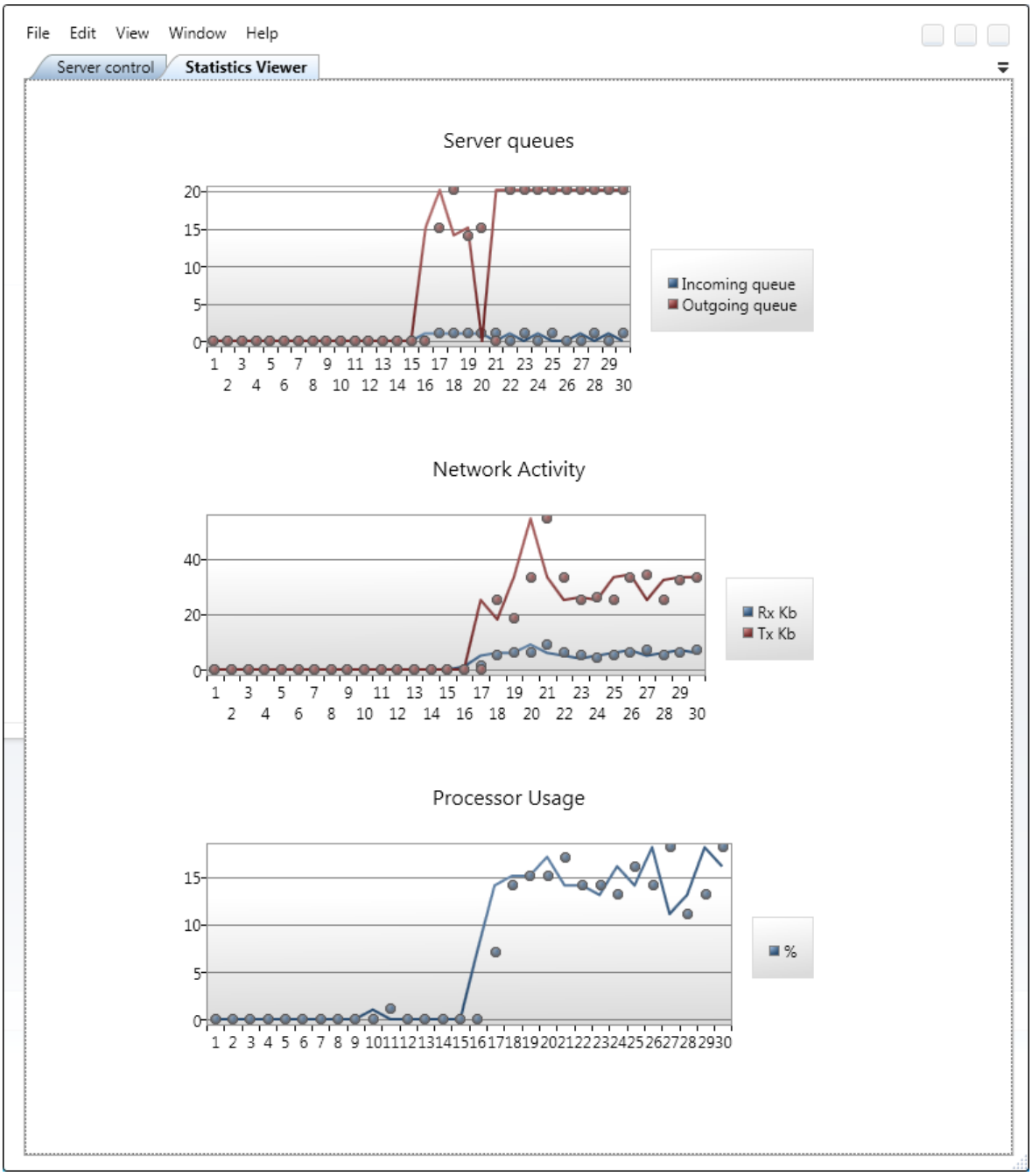

Fig. 6.16: Ou - Statistics

6.2 Composites

LGP

The LGP (Linux group policy) name-space represents the top logical level of the software product. It is also the name-space that encapsulates to entry point for the application or executable.

The two primary classes in the following diagram namely LGPWindow and Splash. Splash is the window used to display the loading information to the user. This loading information is provided by the Framework and presented to the user to indicate which modules are being loaded.

Secondly the LGPWindow is the main window of the application to which UserControls (System.Controls.UserControl) (a derived class of System.Controls.Window in Windows Presentation Foundation) are shown.

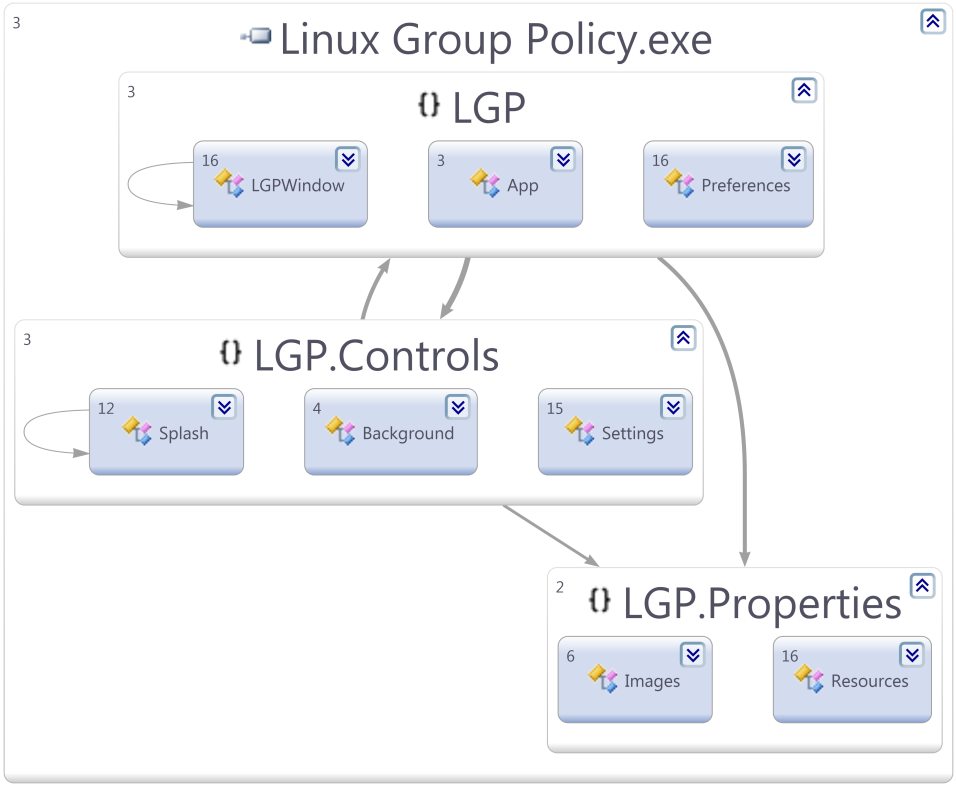

Fig. 6.17: LGP

LGP.Components.Factory

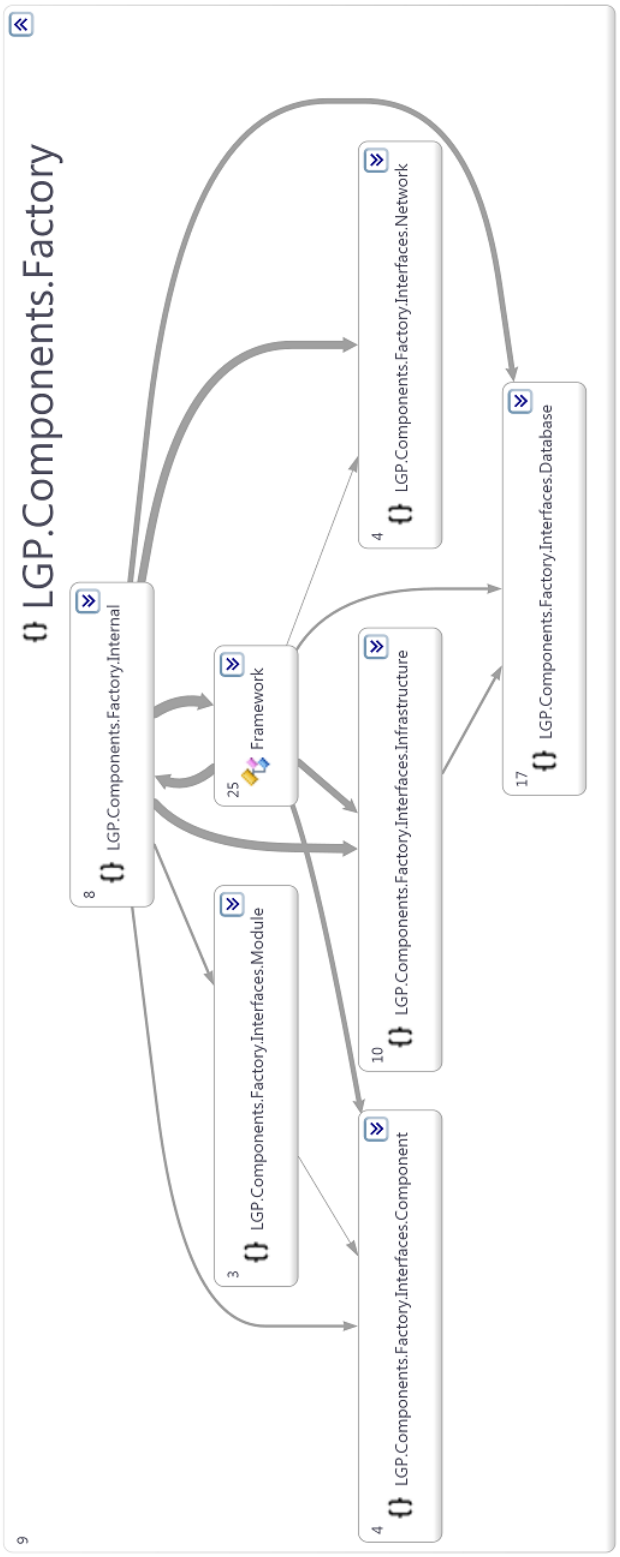

Fig. 6.18: LGP.Components.Factory

LGP.Components.Database

Fig. 6.19: LGP.Components.Database

LGP.Components.Database.Mysql

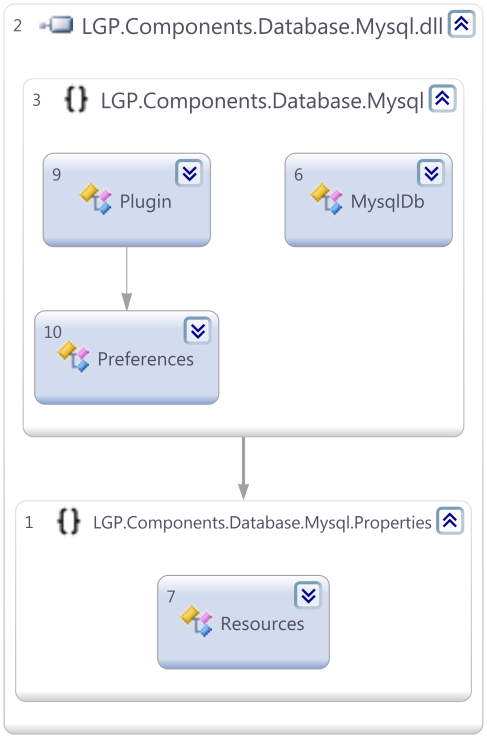

Fig. 6.20: LGP.Components.Database.Mysql

LGP.Components.Database.Mssql

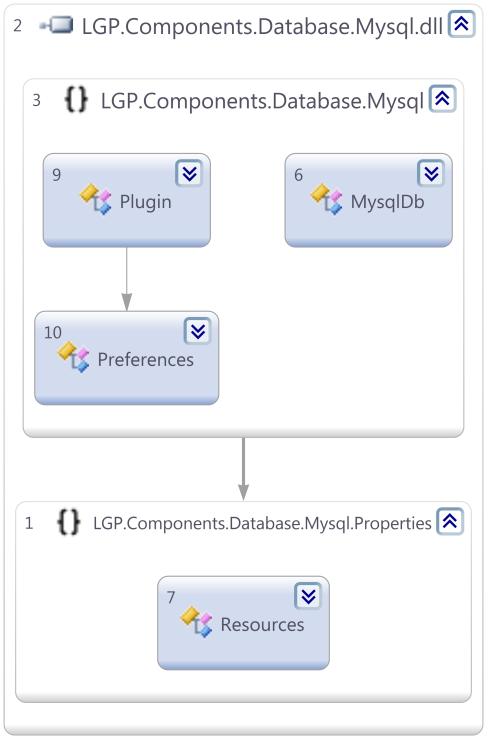

Fig. 6.21: LGP.Components.Database.Mssql

LGP.Components.Database.Oracle

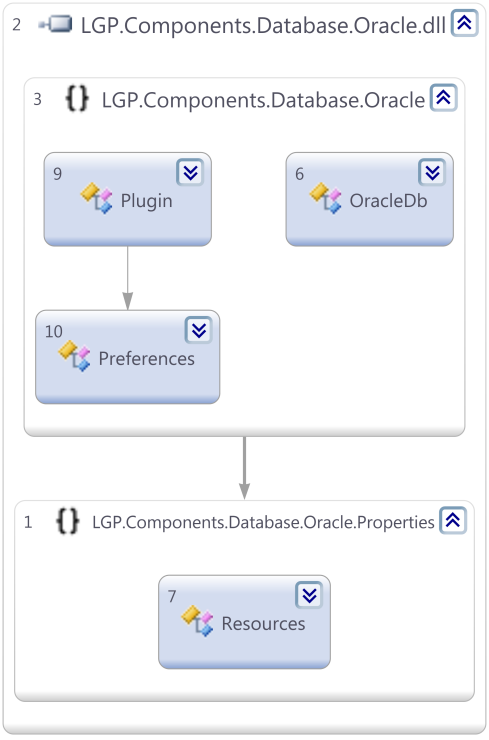

Fig. 6.22: LGP.Components.Database.Oracle

LGP.Components.Database.Postgresql

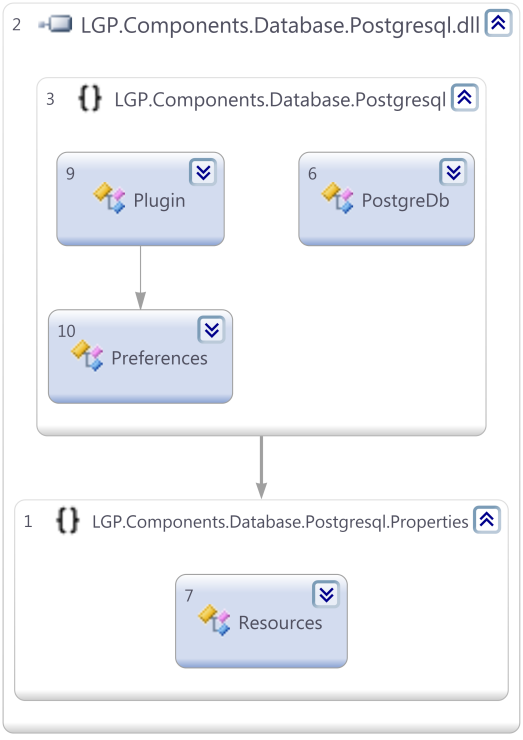

Fig. 6.23: LGP.Components.Database.Postgresql

LGP.Components.Notifications

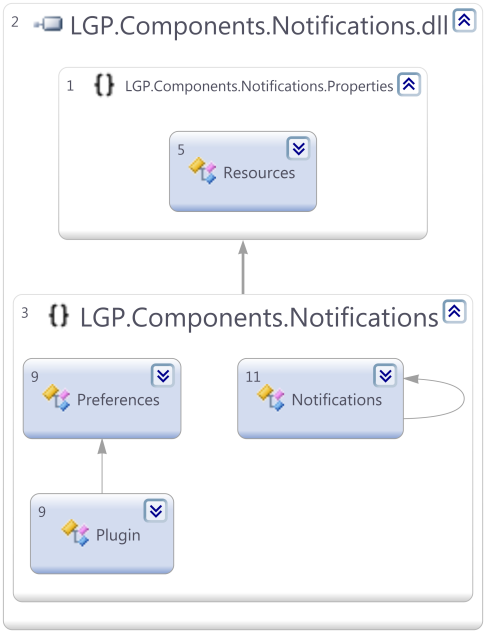

Fig. 6.24: LGP.Components.Notifications

LGP.Components.Panels

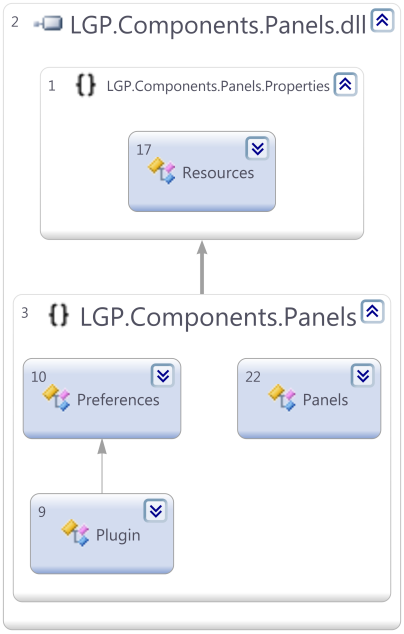

Fig. 6.26: LGP.Components.Panels

LGP.Components.Dialog

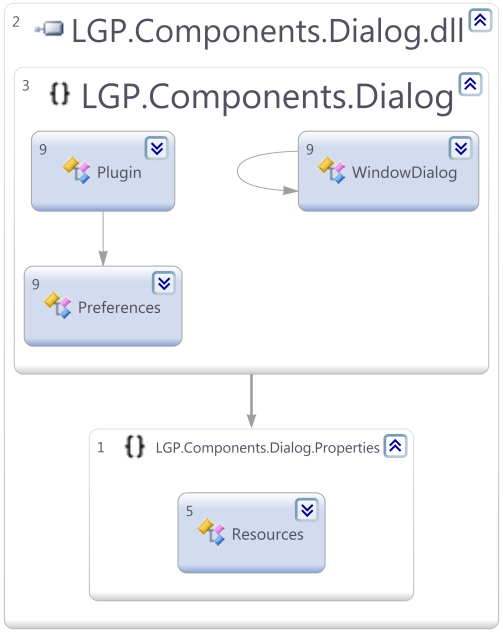

Fig. 6.25: LGP.Components.Dialog

LGP.Components.ServerControl

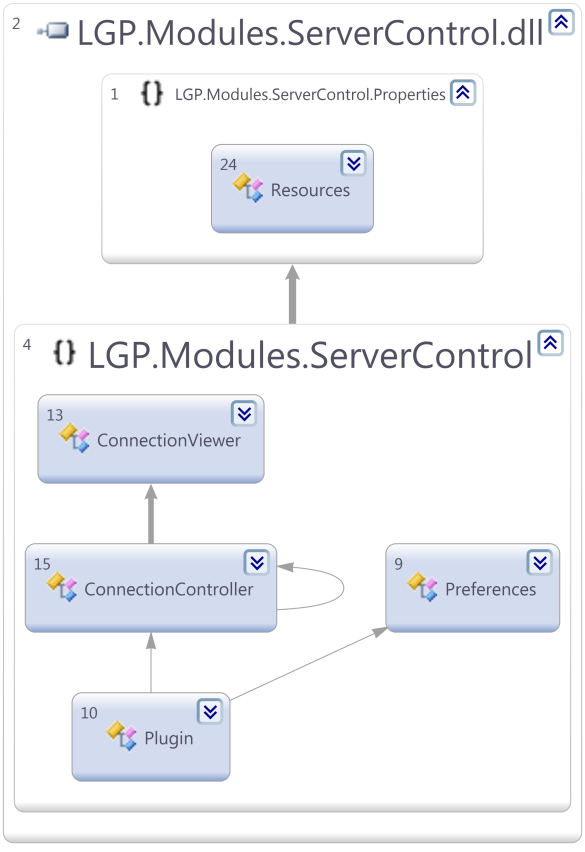

Fig. 6.27: LGP.Modules.ServerControl

LGP.Components.ImageLibrary

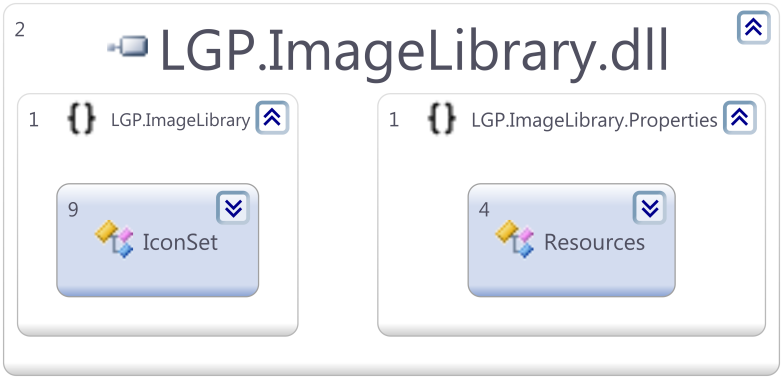

Fig. 6.28: LGP.ImageLibrary

LGP.Components.Menus

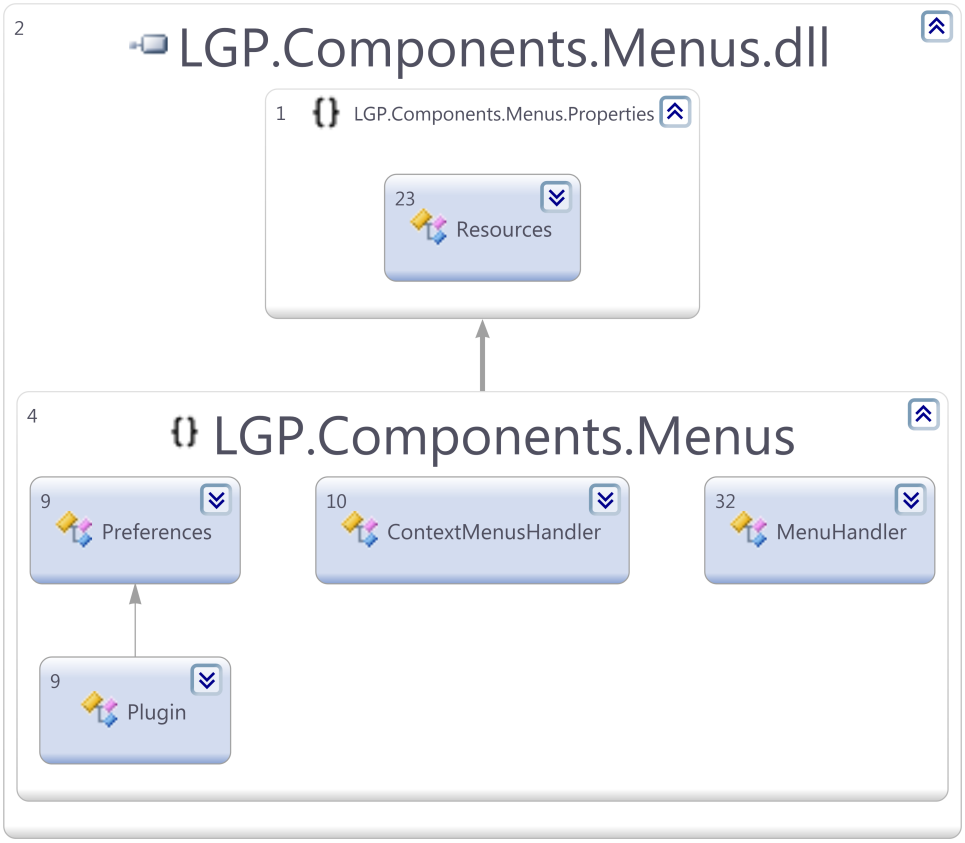

Fig. 6.29: LGP.Components.Menus

LGP.Components.Statistics

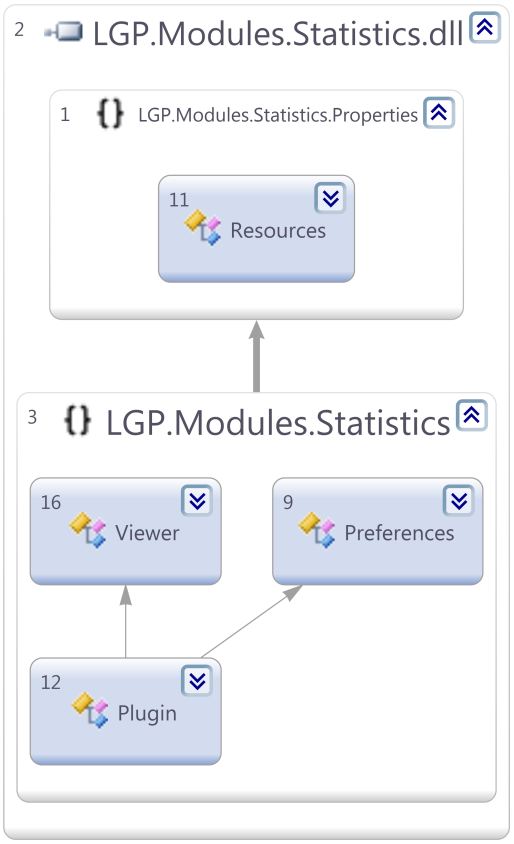

Fig. 6.30: LGP.Modules.Statistics

LGP.Components.PolicyEditor

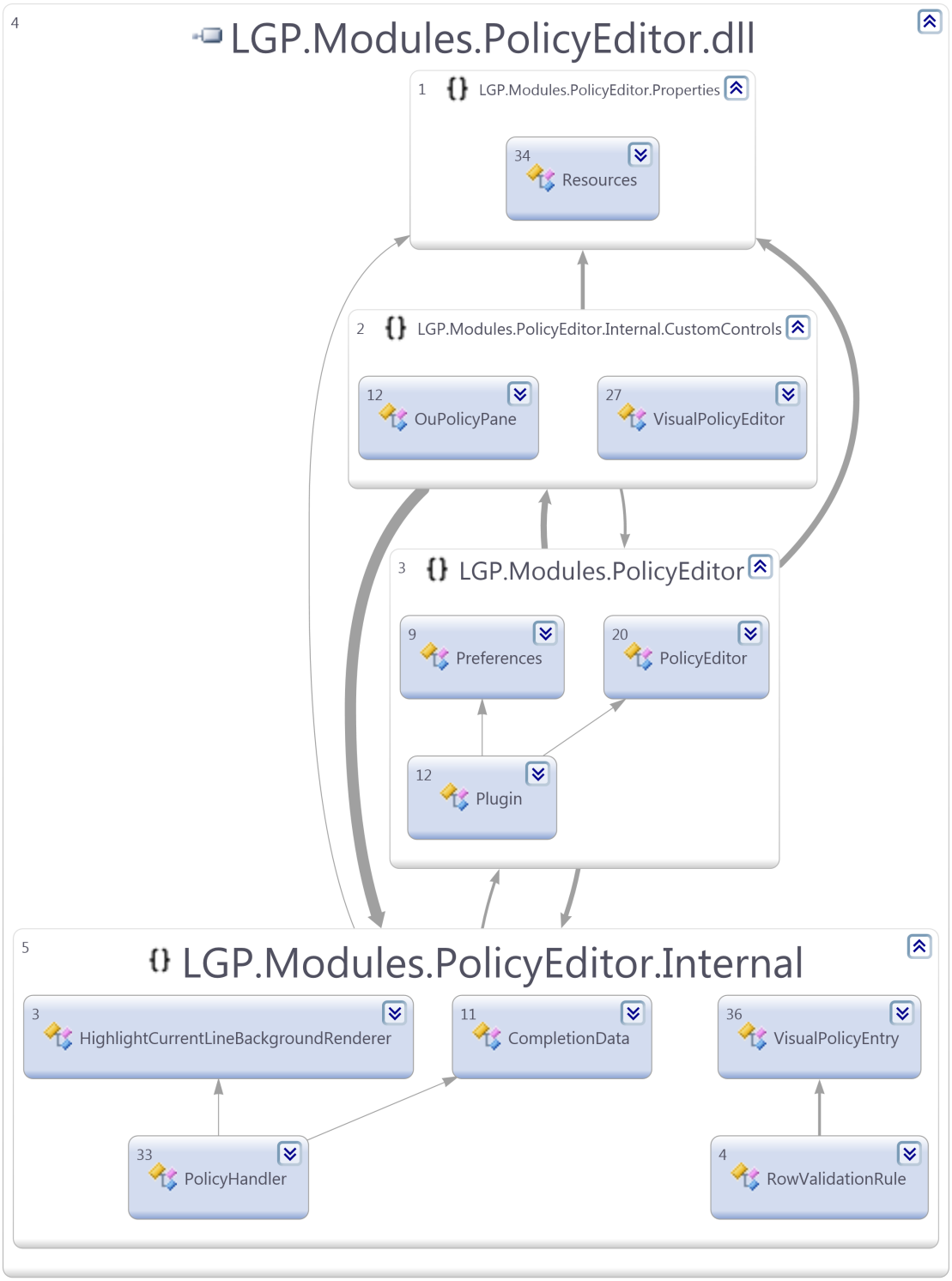

Fig. 6.31: LGP.Modules.PolicyEditor

LGP.Components.OrganizationalUnitExplorer

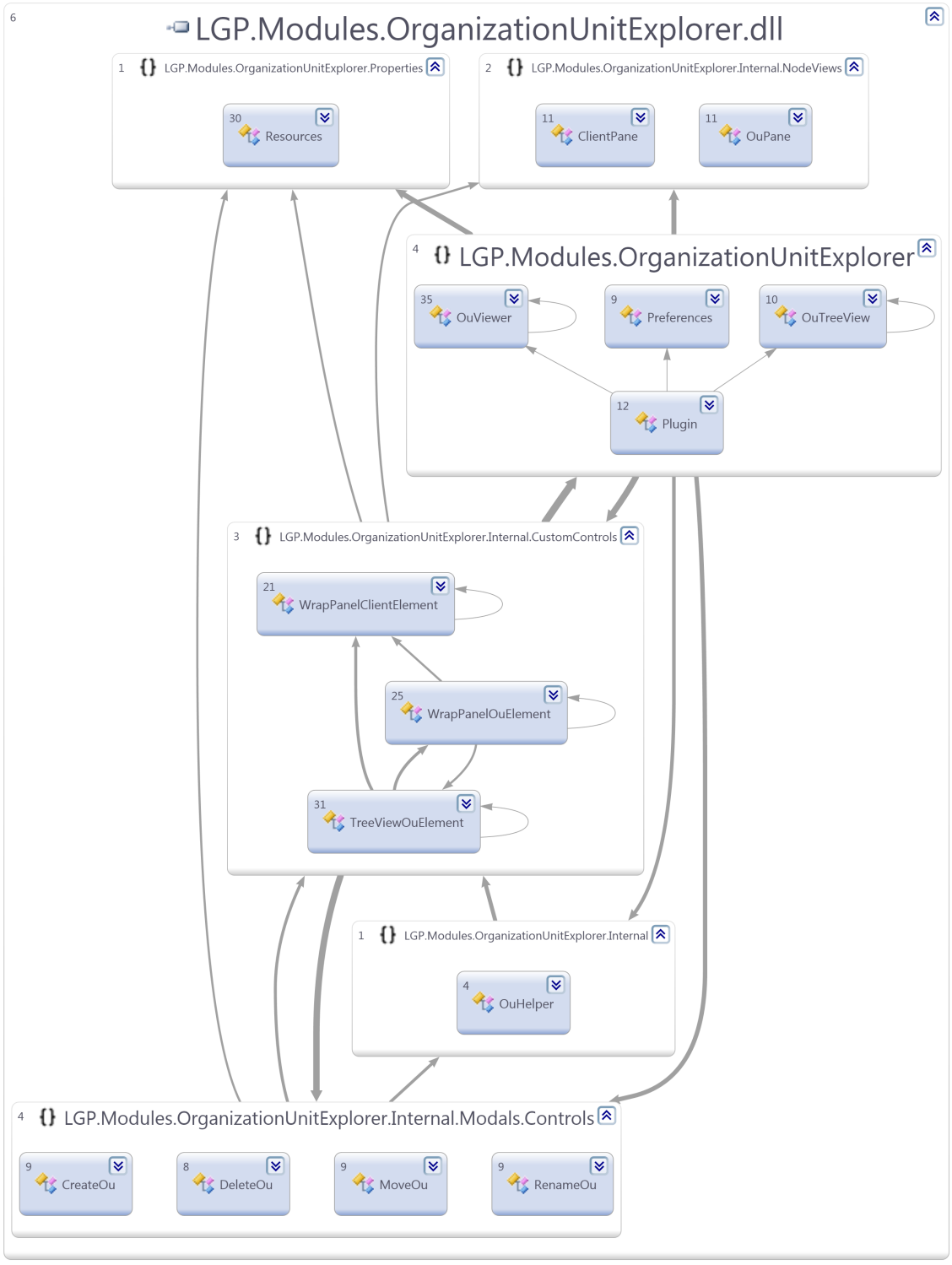

Fig. 6.32: LGP.Modules.OrganizationalUnitExplorer

LGP.Components.ModuleEditor

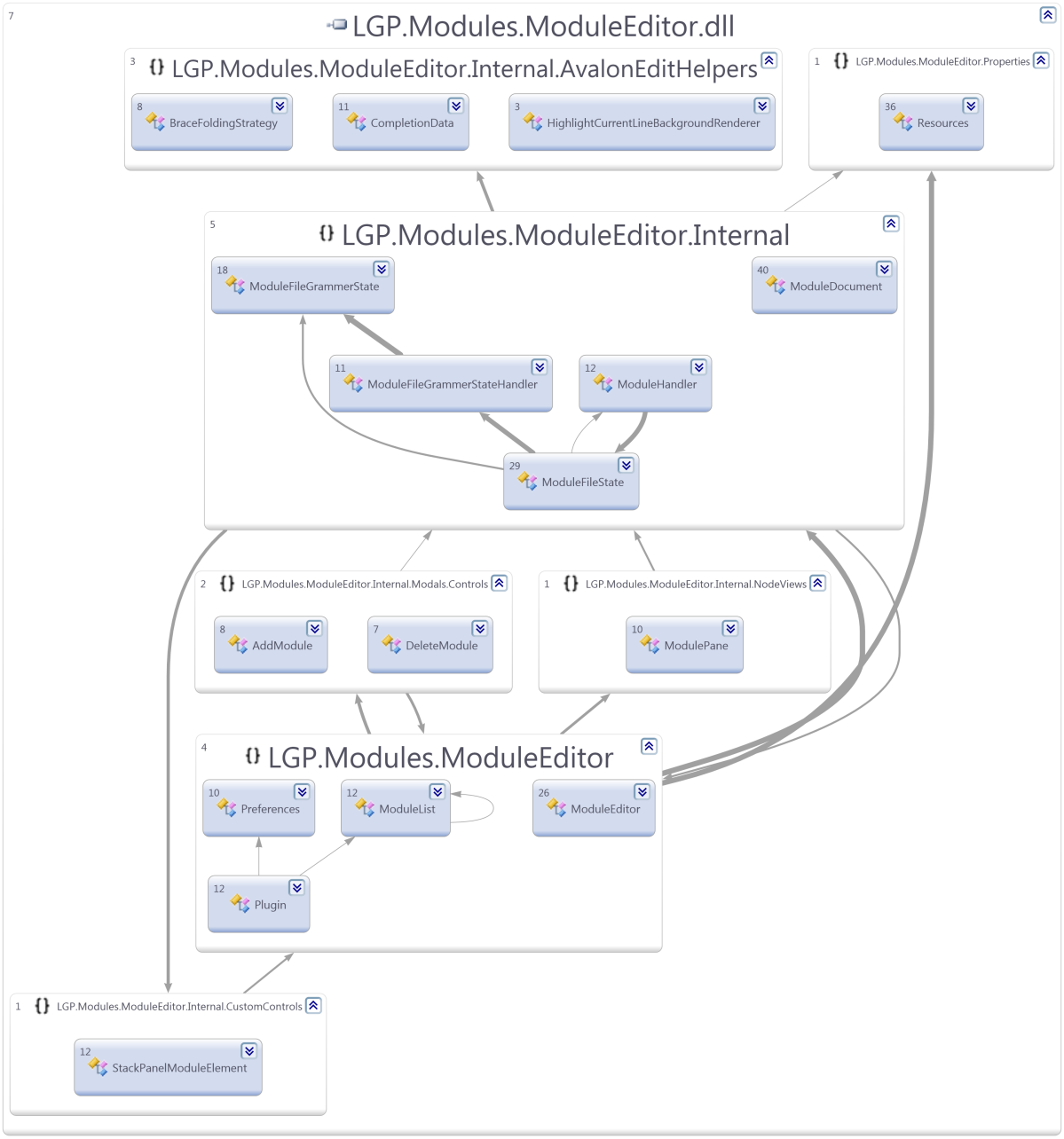

Fig. 6.33: LGP.Modules.ModuleEditor

7 Appendix - Chapter 4

7.1 Extended Metrics

7.1.1 Sub Classes & Inheritance

Namespace	Type	Instance Methods	Sub classes	Depth of inheritance
LGP	LGPWindow	16	0	9
LGP	App	2	0	3
LGP.Components.Database	Preferences	14	0	9
LGP.Components.Database	Plugin	9	0	1
LGP.Components.Database	DatabaseHandler	15	0	1
LGP.Components.Database.Entities	Grammer	15	0	1
LGP.Components.Database.Entities	Module	20	0	1
LGP.Components.Database.Entities	Client	40	0	1
LGP.Components.Database.Entities	Ou	16	0	1
LGP.Components.Database.Entities	Policy	20	0	1
LGP.Components.Database.Gateways	GrammerGateway	9	0	1
LGP.Components.Database.Gateways	ClientGateway	4	0	1
LGP.Components.Database.Gateways	ModuleGateway	6	0	1
LGP.Components.Database.Gateways	OuGateway	9	0	1
LGP.Components.Database.Gateways	PolicyGateway	14	0	1
LGP.Components.Database.Mssql	Preferences	10	0	9
LGP.Components.Database.Mssql	Plugin	9	0	1
LGP.Components.Database.Mssql	MssqlDb	6	0	1
LGP.Components.Database.Mssql.Properties	Resources	1	0	1
LGP.Components.Database.Mysql	Preferences	10	0	9
LGP.Components.Database.Mysql	MysqlDb	6	0	1
LGP.Components.Database.Mysql	Plugin	9	0	1
LGP.Components.Database.Mysql.Properties	Resources	1	0	1
LGP.Components.Database.Oracle	Preferences	10	0	9
LGP.Components.Database.Oracle	OracleDb	6	0	1
LGP.Components.Database.Oracle	Plugin	9	0	1
LGP.Components.Database.Oracle.Properties	Resources	1	0	1
LGP.Components.Database.Postgresql	Preferences	10	0	9
LGP.Components.Database.Postgresql	PostgreDb	6	0	1
LGP.Components.Database.Postgresql	Plugin	9	0	1
LGP.Components.Database.Postgresql.Properties	Resources	1	0	1
LGP.Components.Database.Properties	Resources	1	0	1
LGP.Components.Dialog	Preferences	9	0	9
LGP.Components.Dialog	Plugin	9	0	1
LGP.Components.Dialog	WindowDialog	9	0	9
LGP.Components.Dialog.Properties	Resources	1	0	1
LGP.Components.Factory	Framework	0	0	1
LGP.Components.Factory.Interfaces.Component	IComponent	3	-	-

Table 7.1: Sub classes & Inheritance 1

Namespace	Type	Instance Methods	Sub classes	Depth of inheritance
LGP.Components.Factory.Interfaces.Component	IMenu	9	-	-
LGP.Components.Factory.Interfaces.Component	ISettings	1	-	-
LGP.Components.Factory.Interfaces.Component	IPanel	7	-	-
LGP.Components.Factory.Interfaces.Database	IGrammer	7	-	-
LGP.Components.Factory.Interfaces.Database	IOu	9	-	-
LGP.Components.Factory.Interfaces.Database	IModule	13	-	-
LGP.Components.Factory.Interfaces.Database	IClient	33	-	-
LGP.Components.Factory.Interfaces.Database	IDatabase	13	-	-
LGP.Components.Factory.Interfaces.Database	IPolicy	13	-	-
LGP.Components.Factory.Interfaces.Database	IGrammerObserver	5	-	-
LGP.Components.Factory.Interfaces.Database	IOuObserver	5	-	-
LGP.Components.Factory.Interfaces.Database	IModuleObserver	5	-	-
LGP.Components.Factory.Interfaces.Database	IDatabaseModule	5	-	-
LGP.Components.Factory.Interfaces.Database	IClientObserver	5	-	-
LGP.Components.Factory.Interfaces.Database	IPolicyObserver	5	-	-
LGP.Components.Factory.Interfaces.Database	IOuGateway	8	-	-
LGP.Components.Factory.Interfaces.Database	IGrammerGateway	8	-	-
LGP.Components.Factory.Interfaces.Database	IClientGateway	3	-	-
LGP.Components.Factory.Interfaces.Database	IModuleGateway	5	-	-
LGP.Components.Factory.Interfaces.Database	IPolicyGateway	13	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IEventSystem	6	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IImageHandler	6	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IUtilities	6	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IDialog	5	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IRegistryHandler	4	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IContextMenus	5	-	-
LGP.Components.Factory.Interfaces.Infrastructure	INotification	1	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IClassLibraryHandler	19	-	-
LGP.Components.Factory.Interfaces.Infrastructure	IDragDrop	1	-	-
LGP.Components.Factory.Interfaces.Infrastructure	ILanguage	0	-	-
LGP.Components.Factory.Interfaces.Module	IModule	5	-	-
LGP.Components.Factory.Interfaces.Module	IPreferences	6	-	-
LGP.Components.Factory.Interfaces.Module	IUserControl	4	-	-
LGP.Components.Factory.Interfaces.Network	IServerInfo	15	-	-
LGP.Components.Factory.Interfaces.Network	IServerController	14	-	-
LGP.Components.Factory.Interfaces.Network	INetwork	6	-	-
LGP.Components.Factory.Interfaces.Network	IBroadcastListener	3	-	-
LGP.Components.Factory.Internal	Utilities	7	0	1
LGP.Components.Factory.Internal	RegistryHandler	5	0	1
LGP.Components.Factory.Internal	DragDropAdorner	5	0	1
LGP.Components.Factory.Internal	LibraryHandler	20	0	1
LGP.Components.Factory.Internal	EventSystem	8	0	1
LGP.Components.Factory.Internal	ImageHandler	7	0	1
LGP.Components.Factory.Internal	DragAdorner	18	0	7

Table 7.2: Sub classes & Inheritance 2

Namespace	Type	Instance Methods	Sub classes	Depth of inheritance
LGP.Components.Factory.Internal.ServerControl	DataReceived	4	0	3
LGP.Components.Factory.Internal.ServerControl	ServerInfo	16	0	1
LGP.Components.Factory.Internal.ServerControl	NetworkController	7	0	1
LGP.Components.Factory.Internal.ServerControl	BroadcastListener	6	0	1
LGP.Components.Factory.Internal.ServerControl	ServerController	15	0	1
LGP.Components.Factory.Properties	Resources	1	0	1
LGP.Components.Factory.Publishers.Events	Event	3	3	1
LGP.Components.Factory.Publishers.Events	MenuEvent	3	0	2
LGP.Components.Factory.Publishers.Events	PreferencesEvent	3	0	2
LGP.Components.Factory.Publishers.Events	PanelEvent	1	0	2
LGP.Components.Menus	Preferences	9	0	9
LGP.Components.Menus	MenuHandler	32	0	1
LGP.Components.Menus	Plugin	9	0	1
LGP.Components.Menus	ContextMenusHandler	9	0	1
LGP.Components.Menus.Properties	Resources	1	0	1
LGP.Components.Notifications	Notifications	8	0	9
LGP.Components.Notifications	Preferences	9	0	9
LGP.Components.Notifications	Plugin	9	0	1
LGP.Components.Notifications.Properties	Resources	1	0	1
LGP.Components.Panels	Preferences	10	0	9
LGP.Components.Panels	Plugin	9	0	1
LGP.Components.Panels	Panels	21	0	9
LGP.Components.Panels.Properties	Resources	1	0	1
LGP.Controls	Splash	9	0	9
LGP.Controls	Settings	15	0	9
LGP.Controls	Background	4	0	9
LGP.ImageLibrary	IconSet	1	0	1
LGP.ImageLibrary.Properties	Resources	1	0	1
LGP.ImageWindowsShellIcons	ShellIcon	1	0	1
LGP.ImageWindowsShellIcons.Properties	Resources	1	0	1
LGP.Modules.ModuleEditor	ModuleList	10	0	9
LGP.Modules.ModuleEditor	Preferences	10	0	9
LGP.Modules.ModuleEditor	ModuleEditor	26	0	9
LGP.Modules.ModuleEditor	Plugin	12	0	1
LGP.Modules.ModuleEditor.Internal	ModuleHandler	1	0	1
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammerState	17	0	1
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammer StateHandler	10	0	1
LGP.Modules.ModuleEditor.Internal	ModuleDocument	38	0	1
LGP.Modules.ModuleEditor.Internal	ModuleFileState	28	0	1
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	BraceFoldingStrategy	7	0	2
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	HighlightCurrentLine BackgroundRenderer	3	0	1
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	CompletionData	10	0	1

Table 7.3: Sub classes & Inheritance 3

Namespace	Type	Instance Methods	Sub classes	Depth of inheritance
LGP.Modules.ModuleEditor.Internal.CustomControls	StackPanelModuleElement	12	0	9
LGP.Modules.ModuleEditor.Internal.Modals.Controls	AddModule	8	0	9
LGP.Modules.ModuleEditor.Internal.Modals.Controls	DeleteModule	7	0	9
LGP.Modules.ModuleEditor.Internal.NodeViews	ModulePane	10	0	9
LGP.Modules.ModuleEditor.Properties	Resources	1	0	1
LGP.Modules.OrganizationUnitExplorer	OuViewer	35	0	9
LGP.Modules.OrganizationUnitExplorer	OuTreeView	9	0	9
LGP.Modules.OrganizationUnitExplorer	Preferences	9	0	9
LGP.Modules.OrganizationUnitExplorer	Plugin	12	0	1
LGP.Modules.OrganizationUnitExplorer.Internal	OuHelper	1	0	1
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	TreeViewOuElement	27	0	10
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelClientElement	21	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelOuElement	25	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	CreateOu	9	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	MoveOu	9	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	DeleteOu	8	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	RenameOu	9	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	ClientPane	11	0	9
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	OuPane	11	0	9
LGP.Modules.OrganizationUnitExplorer.Properties	Resources	1	0	1
LGP.Modules.PolicyEditor	PolicyEditor	20	0	9
LGP.Modules.PolicyEditor	Preferences	9	0	9
LGP.Modules.PolicyEditor	Plugin	12	0	1
LGP.Modules.PolicyEditor.Internal	VisualPolicyEntry	33	0	1
LGP.Modules.PolicyEditor.Internal	PolicyHandler	33	0	1
LGP.Modules.PolicyEditor.Internal	CompletionData	10	0	1
LGP.Modules.PolicyEditor.Internal	HighlightCurrentLine BackgroundRenderer	3	0	1
LGP.Modules.PolicyEditor.Internal	RowValidationRule	2	0	2
LGP.Modules.PolicyEditor.Internal.CustomControls	VisualPolicyEditor	26	0	9
LGP.Modules.PolicyEditor.Internal.CustomControls	OuPolicyPane	11	0	9
LGP.Modules.PolicyEditor.Properties	Resources	1	0	1
LGP.Modules.ServerControl	ConnectionViewer	12	0	9
LGP.Modules.ServerControl	ConnectionController	13	0	9
LGP.Modules.ServerControl	Preferences	9	0	9
LGP.Modules.ServerControl	Plugin	10	0	1
LGP.Modules.ServerControl.Properties	Resources	1	0	1
LGP.Modules.Statistics	Viewer	15	0	9
LGP.Modules.Statistics	Preferences	9	0	9
LGP.Modules.Statistics	Plugin	12	0	1
LGP.Modules.Statistics.Properties	Resources	1	0	1
LGP.Properties	Images	1	0	1
LGP.Properties	Resources	1	0	1

Table 7.4: Sub classes & Inheritance 4

7.1.2 Association & Cohesiveness

Namespace	Type	Method cohesiveness	Association between classes
LGP	LGPWindow	0.91	77
LGP	App	0	5
LGP.Components.Database	Preferences	0.83	59
LGP.Components.Database	Plugin	0	5
LGP.Components.Database	DatabaseHandler	0.62	19
LGP.Components.Database.Entities	Grammer	0.73	15
LGP.Components.Database.Entities	Module	0.71	23
LGP.Components.Database.Entities	Client	0.92	14
LGP.Components.Database.Entities	Ou	0.74	19
LGP.Components.Database.Entities	Policy	0.8	19
LGP.Components.Database.Gateways	GrammerGateway	0	31
LGP.Components.Database.Gateways	ClientGateway	0	27
LGP.Components.Database.Gateways	ModuleGateway	0	29
LGP.Components.Database.Gateways	OuGateway	0	33
LGP.Components.Database.Gateways	PolicyGateway	0	41
LGP.Components.Database.Mssql	Preferences	0.8	16
LGP.Components.Database.Mssql	Plugin	0	5
LGP.Components.Database.Mssql	MssqlDb	0.83	7
LGP.Components.Database.Mssql.Properties	Resources	0	6
LGP.Components.Database.Mysql	Preferences	0.82	16
LGP.Components.Database.Mysql	MysqlDb	0.17	13
LGP.Components.Database.Mysql	Plugin	0	5
LGP.Components.Database.Mysql.Properties	Resources	0	6
LGP.Components.Database.Oracle	Preferences	0.82	16
LGP.Components.Database.Oracle	OracleDb	0	1
LGP.Components.Database.Oracle	Plugin	0	5
LGP.Components.Database.Oracle.Properties	Resources	0	6
LGP.Components.Database.Postgresql	Preferences	0.82	16
LGP.Components.Database.Postgresql	PostgreDb	0	1
LGP.Components.Database.Postgresql	Plugin	0	5
LGP.Components.Database.Postgresql.Properties	Resources	0	6
LGP.Components.Database.Properties	Resources	0	6
LGP.Components.Dialog	Preferences	0.78	8
LGP.Components.Dialog	Plugin	0	5
LGP.Components.Dialog	WindowDialog	0.78	48
LGP.Components.Dialog.Properties	Resources	0	6
LGP.Components.Factory	Framework	0	26
LGP.Components.Factory.Interfaces.Component	IComponent	-	0
LGP.Components.Factory.Interfaces.Component	IMenu	-	0
LGP.Components.Factory.Interfaces.Component	ISettings	-	0
LGP.Components.Factory.Interfaces.Component	IPanel	-	0
LGP.Components.Factory.Interfaces.Database	IGrammer	-	0

Table 7.5: Association & Cohesiveness 1

Namespace	Type	Method cohesiveness	Association between classes
LGP.Components.Factory.Interfaces.Database	IOu	-	0
LGP.Components.Factory.Interfaces.Database	IModule	-	0
LGP.Components.Factory.Interfaces.Database	IClient	-	0
LGP.Components.Factory.Interfaces.Database	IDatabase	-	0
LGP.Components.Factory.Interfaces.Database	IPolicy	-	0
LGP.Components.Factory.Interfaces.Database	IGrammerObserver	-	0
LGP.Components.Factory.Interfaces.Database	IOuObserver	-	0
LGP.Components.Factory.Interfaces.Database	IModuleObserver	-	0
LGP.Components.Factory.Interfaces.Database	IDatabaseModule	-	0
LGP.Components.Factory.Interfaces.Database	IClientObserver	-	0
LGP.Components.Factory.Interfaces.Database	IPolicyObserver	-	0
LGP.Components.Factory.Interfaces.Database	IOuGateway	-	0
LGP.Components.Factory.Interfaces.Database	IGrammerGateway	-	0
LGP.Components.Factory.Interfaces.Database	IClientGateway	-	0
LGP.Components.Factory.Interfaces.Database	IModuleGateway	-	0
LGP.Components.Factory.Interfaces.Database	IPolicyGateway	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IEventSystem	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IImageHandler	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IUtilities	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IDialog	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IRegistryHandler	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IContextMenus	-	0
LGP.Components.Factory.Interfaces.Infrastructure	INotification	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IClassLibraryHandler	-	0
LGP.Components.Factory.Interfaces.Infrastructure	IDragDrop	-	0
LGP.Components.Factory.Interfaces.Infrastructure	ILanguage	-	0
LGP.Components.Factory.Interfaces.Module	IModule	-	0
LGP.Components.Factory.Interfaces.Module	IPreferences	-	0
LGP.Components.Factory.Interfaces.Module	IUserControl	-	0
LGP.Components.Factory.Interfaces.Network	IServerInfo	-	0
LGP.Components.Factory.Interfaces.Network	IServerController	-	0
LGP.Components.Factory.Interfaces.Network	INetwork	-	0
LGP.Components.Factory.Interfaces.Network	IBroadcastListener	-	0
LGP.Components.Factory.Internal	Utilities	0	27
LGP.Components.Factory.Internal	RegistryHandler	0.17	8
LGP.Components.Factory.Internal	DragDropAdorner	0	51
LGP.Components.Factory.Internal	LibraryHandler	0.89	24
LGP.Components.Factory.Internal	EventSystem	0	12
LGP.Components.Factory.Internal	ImageHandler	0	6
LGP.Components.Factory.Internal	DragAdorner	0.89	23
LGP.Components.Factory.Internal.ServerControl	DataReceived	-	0
LGP.Components.Factory.Internal.ServerControl	ServerInfo	0.81	5

Table 7.6: Association & Cohesiveness 2

Namespace	Type	Method cohesiveness	Association between classes
LGP.Components.Factory.Internal.ServerControl	NetworkController	0.75	7
LGP.Components.Factory.Internal.ServerControl	BroadcastListener	0.76	52
LGP.Components.Factory.Internal.ServerControl	ServerController	0.71	38
LGP.Components.Factory.Properties	Resources	0	5
LGP.Components.Factory.Publishers.Events	Event	0	1
LGP.Components.Factory.Publishers.Events	MenuEvent	0	1
LGP.Components.Factory.Publishers.Events	PreferencesEvent	0	1
LGP.Components.Factory.Publishers.Events	PanelEvent	0	1
LGP.Components.Menus	Preferences	0.78	8
LGP.Components.Menus	MenuHandler	0.73	30
LGP.Components.Menus	Plugin	0	5
LGP.Components.Menus	ContextMenusHandler	0.33	29
LGP.Components.Menus.Properties	Resources	0	6
LGP.Components.Notifications	Notifications	0.75	64
LGP.Components.Notifications	Preferences	0.78	8
LGP.Components.Notifications	Plugin	0	5
LGP.Components.Notifications.Properties	Resources	0	6
LGP.Components.Panels	Preferences	0.8	25
LGP.Components.Panels	Plugin	0	5
LGP.Components.Panels	Panels	0.84	84
LGP.Components.Panels.Properties	Resources	0	6
LGP.Controls	Splash	0.82	60
LGP.Controls	Settings	0.8	84
LGP.Controls	Background	0.62	12
LGP.ImageLibrary	IconSet	0	25
LGP.ImageLibrary.Properties	Resources	0	5
LGP.ImageWindowsShellIcons	ShellIcon	0	7
LGP.ImageWindowsShellIcons.Properties	Resources	0	5
LGP.Modules.ModuleEditor	ModuleList	0.86	38
LGP.Modules.ModuleEditor	Preferences	0.82	23
LGP.Modules.ModuleEditor	ModuleEditor	0.91	92
LGP.Modules.ModuleEditor	Plugin	0	19
LGP.Modules.ModuleEditor.Internal	ModuleHandler	0	58
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammerState	0.79	15
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammer StateHandler	0.52	38
LGP.Modules.ModuleEditor.Internal	ModuleDocument	0.89	142
LGP.Modules.ModuleEditor.Internal	ModuleFileState	0.83	49
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	BraceFoldingStrategy	0	15
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	HighlightCurrentLine BackgroundRenderer	0.33	21

Table 7.7: Association & Cohesiveness 3

Namespace	Type	Method cohesiveness	Association between classes
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	CompletionData	0.82	9
LGP.Modules.ModuleEditor.Internal.CustomControls	StackPanelModuleElement	0.79	29
LGP.Modules.ModuleEditor.Internal.Modals.Controls	AddModule	0.76	28
LGP.Modules.ModuleEditor.Internal.Modals.Controls	DeleteModule	0.75	20
LGP.Modules.ModuleEditor.Internal.NodeViews	ModulePane	0.8	39
LGP.Modules.ModuleEditor.Properties	Resources	0	6
LGP.Modules.OrganizationUnitExplorer	OuViewer	0.93	116
LGP.Modules.OrganizationUnitExplorer	OuTreeView	0.82	33
LGP.Modules.OrganizationUnitExplorer	Preferences	0.78	10
LGP.Modules.OrganizationUnitExplorer	Plugin	0	21
LGP.Modules.OrganizationUnitExplorer.Internal	OuHelper	0	17
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	TreeViewOuElement	0.76	89
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelClientElement	0.85	42
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelOuElement	0.83	69
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	CreateOu	0.8	31
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	MoveOu	0.8	24
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	DeleteOu	0.75	28
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	RenameOu	0.78	26
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	ClientPane	0.82	34
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	OuPane	0.8	16
LGP.Modules.OrganizationUnitExplorer.Properties	Resources	0	6
LGP.Modules.PolicyEditor	PolicyEditor	0.91	46
LGP.Modules.PolicyEditor	Preferences	0.78	10
LGP.Modules.PolicyEditor	Plugin	0	21
LGP.Modules.PolicyEditor.Internal	VisualPolicyEntry	0.92	32
LGP.Modules.PolicyEditor.Internal	PolicyHandler	0.88	114
LGP.Modules.PolicyEditor.Internal	CompletionData	0.82	9
LGP.Modules.PolicyEditor.Internal	HighlightCurrentLine BackgroundRenderer	0.33	21
LGP.Modules.PolicyEditor.Internal	RowValidationRule	0	7
LGP.Modules.PolicyEditor.Internal.CustomControls	VisualPolicyEditor	0.91	163
LGP.Modules.PolicyEditor.Internal.CustomControls	OuPolicyPane	0.84	33
LGP.Modules.PolicyEditor.Properties	Resources	0	6
LGP.Modules.ServerControl	ConnectionViewer	0.83	44
LGP.Modules.ServerControl	ConnectionController	0.88	53
LGP.Modules.ServerControl	Preferences	0.78	8
LGP.Modules.ServerControl	Plugin	0	17
LGP.Modules.ServerControl.Properties	Resources	0	6
LGP.Modules.Statistics	Viewer	0.82	40
LGP.Modules.Statistics	Preferences	0.78	8
LGP.Modules.Statistics	Plugin	0	17
LGP.Modules.Statistics.Properties	Resources	0	6
LGP.Properties	Images	0	6
LGP.Properties	Resources	0	6

Table 7.8: Association & Cohesiveness 4

7.1.3 Complexity & Coupling

Namespace	Type	Cyclomatic Complexity	Afferent Coupling	Efferent Coupling
LGP	LGPWindow	35	0	73
LGP	App	0	0	11
LGP.Components.Database	Preferences	27	2	66
LGP.Components.Database	Plugin	8	0	13
LGP.Components.Database	DatabaseHandler	20	0	28
LGP.Components.Database.Entities	Grammer	29	1	12
LGP.Components.Database.Entities	Module	33	1	16
LGP.Components.Database.Entities	Client	54	1	13
LGP.Components.Database.Entities	Ou	37	1	15
LGP.Components.Database.Entities	Policy	40	1	14
LGP.Components.Database.Gateways	GrammerGateway	55	2	22
LGP.Components.Database.Gateways	ClientGateway	22	1	23
LGP.Components.Database.Gateways	ModuleGateway	34	1	23
LGP.Components.Database.Gateways	OuGateway	49	1	24
LGP.Components.Database.Gateways	PolicyGateway	80	1	27
LGP.Components.Database.Mssql	Preferences	10	1	38
LGP.Components.Database.Mssql	Plugin	8	0	13
LGP.Components.Database.Mssql	MssqlDb	6	0	13
LGP.Components.Database.Mssql.Properties	Resources	8	2	13
LGP.Components.Database.Mysql	Preferences	10	1	38
LGP.Components.Database.Mysql	MysqlDb	10	0	17
LGP.Components.Database.Mysql	Plugin	8	0	13
LGP.Components.Database.Mysql.Properties	Resources	8	2	13
LGP.Components.Database.Oracle	Preferences	10	1	38
LGP.Components.Database.Oracle	OracleDb	5	0	7
LGP.Components.Database.Oracle	Plugin	8	0	13
LGP.Components.Database.Oracle.Properties	Resources	8	2	13
LGP.Components.Database.Postgresql	Preferences	10	1	38
LGP.Components.Database.Postgresql	PostgreDb	5	0	7
LGP.Components.Database.Postgresql	Plugin	8	0	13
LGP.Components.Database.Postgresql.Properties	Resources	8	2	13
LGP.Components.Database.Properties	Resources	15	2	13
LGP.Components.Dialog	Preferences	7	1	30
LGP.Components.Dialog	Plugin	8	0	13
LGP.Components.Dialog	WindowDialog	10	0	45
LGP.Components.Dialog.Properties	Resources	6	2	13
LGP.Components.Factory	Framework	28	92	36
LGP.Components.Factory.Interfaces.Component	IComponent	-	21	4
LGP.Components.Factory.Interfaces.Component	IMenu	-	21	7
LGP.Components.Factory.Interfaces.Component	ISettings	-	17	1
LGP.Components.Factory.Interfaces.Component	IPanel	-	17	5
LGP.Components.Factory.Interfaces.Database	IGrammer	-	14	4

Table 7.9: Complexity & Coupling 1

Namespace	Type	Cyclomatic Complexity	Afferent Coupling	Efferent Coupling
LGP.Components.Factory.Interfaces.Database	IOu	-	19	5
LGP.Components.Factory.Interfaces.Database	IModule	-	13	6
LGP.Components.Factory.Interfaces.Database	IClient	-	9	5
LGP.Components.Factory.Interfaces.Database	IDatabase	-	24	12
LGP.Components.Factory.Interfaces.Database	IPolicy	-	7	5
LGP.Components.Factory.Interfaces.Database	IGrammerObserver	-	3	2
LGP.Components.Factory.Interfaces.Database	IOuObserver	-	11	2
LGP.Components.Factory.Interfaces.Database	IModuleObserver	-	9	2
LGP.Components.Factory.Interfaces.Database	IDatabaseModule	-	10	4
LGP.Components.Factory.Interfaces.Database	IClientObserver	-	5	2
LGP.Components.Factory.Interfaces.Database	IPolicyObserver	-	5	2
LGP.Components.Factory.Interfaces.Database	IOuGateway	-	9	5
LGP.Components.Factory.Interfaces.Database	IGrammerGateway	-	8	5
LGP.Components.Factory.Interfaces.Database	IClientGateway	-	5	4
LGP.Components.Factory.Interfaces.Database	IModuleGateway	-	5	4
LGP.Components.Factory.Interfaces.Database	IPolicyGateway	-	4	6
LGP.Components.Factory.Interfaces.Infrastructure	IEventSystem	-	76	7
LGP.Components.Factory.Interfaces.Infrastructure	IImageHandler	-	56	5
LGP.Components.Factory.Interfaces.Infrastructure	IUtilities	-	26	5
LGP.Components.Factory.Interfaces.Infrastructure	IDialog	-	15	3
LGP.Components.Factory.Interfaces.Infrastructure	IRegistryHandler	-	9	3
LGP.Components.Factory.Interfaces.Infrastructure	IContextMenus	-	8	9
LGP.Components.Factory.Interfaces.Infrastructure	INotification	-	9	3
LGP.Components.Factory.Interfaces.Infrastructure	IClassLibraryHandler	-	5	12
LGP.Components.Factory.Interfaces.Infrastructure	IDragDrop	-	6	3
LGP.Components.Factory.Interfaces.Infrastructure	ILanguage	-	0	0
LGP.Components.Factory.Interfaces.Module	IModule	-	24	2
LGP.Components.Factory.Interfaces.Module	IPreferences	-	17	6
LGP.Components.Factory.Interfaces.Module	IUserControl	-	8	1
LGP.Components.Factory.Interfaces.Network	IServerInfo	-	10	5
LGP.Components.Factory.Interfaces.Network	IServerController	-	7	6
LGP.Components.Factory.Interfaces.Network	INetwork	-	8	5
LGP.Components.Factory.Interfaces.Network	IBroadcastListener	-	2	3
LGP.Components.Factory.Internal	Utilities	18	1	26
LGP.Components.Factory.Internal	RegistryHandler	7	1	8
LGP.Components.Factory.Internal	DragDropAdorner	19	1	36
LGP.Components.Factory.Internal	LibraryHandler	43	1	32
LGP.Components.Factory.Internal	EventSystem	10	1	16
LGP.Components.Factory.Internal	ImageHandler	9	1	9
LGP.Components.Factory.Internal	DragAdorner	30	1	26
LGP.Components.Factory.Internal.ServerControl	DataReceived	-	3	8
LGP.Components.Factory.Internal.ServerControl	ServerInfo	17	3	9

Table 7.10: Complexity & Coupling 2

Namespace	Type	Cyclomatic Complexity	Afferent Coupling	Efferent Coupling
LGP.Components.Factory.Internal.ServerControl	NetworkController	10	1	12
LGP.Components.Factory.Internal.ServerControl	BroadcastListener	19	1	30
LGP.Components.Factory.Internal.ServerControl	ServerController	28	1	26
LGP.Components.Factory.Properties	Resources	5	0	13
LGP.Components.Factory.Publishers.Events	Event	1	3	3
LGP.Components.Factory.Publishers.Events	MenuEvent	1	5	5
LGP.Components.Factory.Publishers.Events	PreferencesEvent	1	4	5
LGP.Components.Factory.Publishers.Events	PanelEvent	1	2	3
LGP.Components.Menus	Preferences	7	1	30
LGP.Components.Menus	MenuHandler	85	0	30
LGP.Components.Menus	Plugin	8	0	13
LGP.Components.Menus	ContextMenusHandler	31	0	29
LGP.Components.Menus.Properties	Resources	24	3	13
LGP.Components.Notifications	Notifications	17	2	59
LGP.Components.Notifications	Preferences	7	1	30
LGP.Components.Notifications	Plugin	8	0	13
LGP.Components.Notifications.Properties	Resources	6	2	13
LGP.Components.Panels	Preferences	17	1	44
LGP.Components.Panels	Plugin	8	0	13
LGP.Components.Panels	Panels	62	0	73
LGP.Components.Panels.Properties	Resources	18	3	13
LGP.Controls	Splash	19	3	57
LGP.Controls	Settings	49	1	72
LGP.Controls	Background	4	0	32
LGP.ImageLibrary	IconSet	11	1	28
LGP.ImageLibrary.Properties	Resources	5	0	13
LGP.ImageWindowsShellIcons	ShellIcon	2	1	10
LGP.ImageWindowsShellIcons.Properties	Resources	5	0	13
LGP.Modules.ModuleEditor	ModuleList	24	2	51
LGP.Modules.ModuleEditor	Preferences	11	1	43
LGP.Modules.ModuleEditor	ModuleEditor	58	2	78
LGP.Modules.ModuleEditor	Plugin	14	0	25
LGP.Modules.ModuleEditor.Internal	ModuleHandler	28	3	41
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammerState	29	2	12
LGP.Modules.ModuleEditor.Internal	ModuleFileGrammer StateHandler	45	1	23
LGP.Modules.ModuleEditor.Internal	ModuleDocument	87	2	81
LGP.Modules.ModuleEditor.Internal	ModuleFileState	53	1	28
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	BraceFoldingStrategy	12	1	18
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	HighlightCurrentLine BackgroundRenderer	8	1	30
LGP.Modules.ModuleEditor.Internal.AvalonEditHelpers	CompletionData	11	1	18
LGP.Modules.ModuleEditor.Internal.CustomControls	StackPanelModuleElement	22	1	48

Table 7.11: Complexity & Coupling 3

Namespace	Type	Cyclomatic Complexity	Afferent Coupling	Efferent Coupling
LGP.Modules.ModuleEditor.Internal.Modals.Controls	AddModule	13	1	43
LGP.Modules.ModuleEditor.Internal.Modals.Controls	DeleteModule	10	1	37
LGP.Modules.ModuleEditor.Internal.NodeViews	ModulePane	18	1	49
LGP.Modules.ModuleEditor.Properties	Resources	37	4	13
LGP.Modules.OrganizationUnitExplorer	OuViewer	96	4	93
LGP.Modules.OrganizationUnitExplorer	OuTreeView	18	2	50
LGP.Modules.OrganizationUnitExplorer	Preferences	8	1	32
LGP.Modules.OrganizationUnitExplorer	Plugin	14	0	26
LGP.Modules.OrganizationUnitExplorer.Internal	OuHelper	12	5	17
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	TreeViewOuElement	100	5	69
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelClientElement	48	3	55
LGP.Modules.OrganizationUnitExplorer.Internal.CustomControls	WrapPanelOuElement	66	2	70
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	CreateOu	15	3	44
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	MoveOu	14	3	39
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	DeleteOu	13	3	41
LGP.Modules.OrganizationUnitExplorer.Internal.Modals.Controls	RenameOu	15	2	40
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	ClientPane	20	2	37
LGP.Modules.OrganizationUnitExplorer.Internal.NodeViews	OuPane	18	2	33
LGP.Modules.OrganizationUnitExplorer.Properties	Resources	31	5	13
LGP.Modules.PolicyEditor	PolicyEditor	34	3	62
LGP.Modules.PolicyEditor	Preferences	8	1	32
LGP.Modules.PolicyEditor	Plugin	14	0	27
LGP.Modules.PolicyEditor.Internal	VisualPolicyEntry	54	2	25
LGP.Modules.PolicyEditor.Internal	PolicyHandler	62	2	73
LGP.Modules.PolicyEditor.Internal	CompletionData	11	1	18
LGP.Modules.PolicyEditor.Internal	HighlightCurrentLine BackgroundRenderer	8	1	30
LGP.Modules.PolicyEditor.Internal	RowValidationRule	8	1	15
LGP.Modules.PolicyEditor.Internal.CustomControls	VisualPolicyEditor	121	1	98
LGP.Modules.PolicyEditor.Internal.CustomControls	OuPolicyPane	22	1	47
LGP.Modules.PolicyEditor.Properties	Resources	35	5	13
LGP.Modules.ServerControl	ConnectionViewer	21	2	50
LGP.Modules.ServerControl	ConnectionController	32	2	63
LGP.Modules.ServerControl	Preferences	7	1	30
LGP.Modules.ServerControl	Plugin	12	0	24
LGP.Modules.ServerControl.Properties	Resources	25	5	13
LGP.Modules.Statistics	Viewer	32	1	52
LGP.Modules.Statistics	Preferences	7	1	30
LGP.Modules.Statistics	Plugin	14	0	24
LGP.Modules.Statistics.Properties	Resources	12	3	13
LGP.Properties	Images	7	3	15
LGP.Properties	Resources	17	1	13

Table 7.12: Complexity & Coupling 4