EMC² On-the-Fly Configurations

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Sponsoring Agency: EMC²
Submitted to:
      Project Advisor: Gary Pollice, WPI Professor
      On-Site Liaison: Jon Krasner, SPEA Manager

Submitted by:

_____
Andrew Ralich

_____
Gary Pollice

Date: 27 April 2006

Abstract

Due to the increasing costs of building, maintaining and upgrading on site labs, it has

become necessary for EMC^2 to develop tools which provide a means of laying out,

configuring and representing their various data storage products through software. Such

data configurations can then be used by other software tools as a representation of a given

lab environment. The paper describes the project to develop an on-the-fly configuration

utility for EMC^2. We developed a Java-based solution which allowed the user to build

hardware networks based on configuration options loaded at runtime and outputted the

representation as XML.

Acknowledgement and Thanks

Table of Contents

Table of Figures

Introduction

EMC^2 has developed multiple software tools which use a pre-compiled representation of a storage environment as an input. These text based representations, also called "seed files", contain specific configuration information for hosts, storage devices and the connections between them. In order to generate such a file, a debug log must be taken directly from one of the hosts. To do so requires that all the desired hardware is properly configured in a lab environment. As the number of available storage platforms has increased, representing all possible configurations in a lab environment has become exponentially difficult and expensive. EMC development groups need to expand existing tools to include more complex hardware configurations, and subsequently, it has become necessary to develop an application which can configure lab environments on the fly.

Because the basis of this problem is an ever evolving hardware model that must be represented as data, it is vital to the long term success of this project to have an application that is not only initially robust but also easily extensible. Standards must be developed not only for representing configurations, but also for defining the entities and constraints that make up such a configuration. Extensibility also relies on maintainability, or proper code formatting and documentation in order to accommodate future developers.

Other factors must be taken into account when developing an application that is designed to run in conjunction with existing tools. In order to ensure effective utilization over a broad customer base, the code must run on a variety of platforms. The application must accommodate various user input styles, including both mouse and keyboard based input.

Background

Organizational Background

EMC^2 is the industry leader in solutions for *information lifecycle management* (ILM). The hardware and software systems developed at EMC^2 are compatible with every mainstream computing platform and are deployed worldwide. EMC^2 has developed a wide array of hardware storage options, which vary in functionality and cost based on individual business needs. The Storage Platform Enablers and Applications (SPEA) group at EMC^2 develops software which provides a programmable interface to the Symmetrix line of storage arrays, as well as products used to manage these devices. The group consists of over 100 developers, with a wide range of preferred platforms and development styles.

Numerous software tools exist that aid the SPEA developers in the deployment, testing and debugging of Symmetrix management applications. Some of these tools require a representation of a given Symmetrix environment as an input. This representation is often derived from information found in a *debug log*, which can be generated by a user or developer if they have access to a host which is connected to an active Symmetrix setup.

The software developed by SPEA is compatible with the current line of Symmetrix arrays, and (in many cases) backwards compatible with previously released product lines. Many features of SPEA deployed software allow interaction between Symmetrix arrays and other storage devices (both EMC and third party vendors). Because debug logs must be generated by an active setup, EMC has built multiple lab environments which attempt to encapsulate as many configurations as possible. SPEA

finds it advantageous to develop tools which can generate data representations, similar to a debug log, through software. This would allow developers to generate configurations on-the-fly and could eventually lead to more robust testing mechanisms, which do not rely on an existing shared lab environment.

The Underlying Model

The tool will be designed to represent the following model. The EMC^2 Symmetrix and CLARiiON lines will be represented, as well as a set of hosts consistent with what is available in the EMC^2 labs. Each hardware class will be an abstraction of the base *entity* (see Figure 1). Common configuration options and control mechanisms are stored in the entity class and specific configuration details stored in the subsequent sub classes.
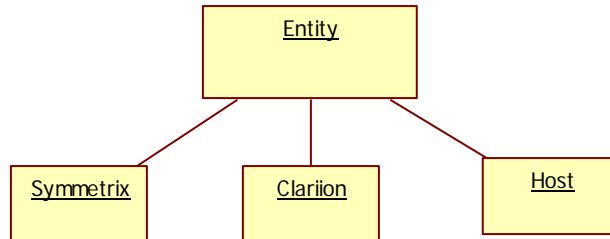
Figure 1: Initial Entity Hierarchy

As this abstract model evolves into a data model it becomes necessary to differentiate between the different Symmetrix, CLARiiON and Host entities and also define rules which define and govern specific configurable aspects of each type of device.

Methodology and Implementation

Software Engineering Principles

Developing an application from the ground up involves multiple development stages. These stages, though often approached in a linear fashion, are re-visited throughout the entire development cycle. A thorough analysis of the requirements for the application is conducted. Meta design plans, such as class or sequence diagrams, are produced using diagramming tools and used to guide the implementation process. During implementation careful consideration must be made of the trade-off between the time constraints, cost and scope of the project. There are multiple factors which will be considered throughout the entire development process, including input received directly from potential users. As previously mentioned, the extensibility of the application is a primary concern, because we are building on top of a constantly evolving model.

Object Oriented Design

Within the software engineering domain, extensibility is often associated with object-oriented concepts. Specifically, encapsulation, inheritance, and polymorphism. (Meyer, 21). Abstract types can be defined through the use of an object-oriented programming language. These abstract types provide a template for building new classes. Applying this idea to our underlying model, we can see that by abstracting entity representations we will expose a framework for future developers to seamlessly add specific entity types.

Common routines shared among entities, such as drawing routines, can be implemented within the abstract types and utilized over a set of sub-classed entities. Sharing a single definition (function, member variable, etc.) between multiple classes is

referred to as polymorphism. Classes which define any number of member variables or functions through a reference to an abstract parent class are said to inherit functions from that parent.

Design Patterns

As the software engineering field has evolved, generally accepted solutions for certain requirement sets have evolved. These standards for developing applications fitting a certain model are referred to as *design patterns*. Design patterns "solve specific design problems and make object-oriented designs more flexible, elegant and ultimately reusable."(Gamma, 2) They emerge over time and therefore provide well tested paradigms to base the overall design of a system.

Design patterns help ensure that the solution being developed will be complete for the problem at hand. Issues with major design decisions may not arise until late in a project implementation. Design patterns also provide extensibility. Design decisions are inherently justified to new developers through their recognition of an established design pattern implemented within the system.

Human Computer Interaction

When developing an application with a wide user base, one should consider principles of *human computer interaction* (HCI). It has been found that "bringing usability into the design process" leads to developing better interactive applications. (Preece, 2). Some in the field recommend that users themselves be brought into the design process whenever a question is raised involving a specific user experience. (Steinberg, 94)

Different users prefer various input types and methods, and in order for a tool to successfully integrate itself into the development process it must be embraced by those who use it. Window layouts, menus, and help facilities fall under the umbrella of HCI. The application interface should adapt to platform-based styles if it is going to be used in a cross platform environment. The design of dialogs and menus should follow standard conventions as to not confuse the user and minimize the learning curve for the application.

Though there are many metrics for evaluating the quality of a development project, in the end it is user acceptance and satisfaction which truly gauges success.

Iterative Development

Traditionally, development was approached as a linear process: analysis, design, implementation and testing. This method was referred to as the 'waterfall' approach, because each step is completed before the next and never reconsidered. (Royce) Developing using this approach assumes that the initial analysis and design is correct. Leaving testing until the very end is also dangerous.

Iterative development approaches analysis, design, implementation and testing as a sequence which is iterated over multiple times. Doing so allows the developer to see the implications of his design decisions early in the life of the project. This way, inaccurate analysis or bad design decisions are caught early. Also, testing throughout the development cycle provides a constant verification of requirements. IBM's Rational Unified Process® is an example of iterative development. (Figure 2)

Each iteration results in a sub-version of the final product. Because working software is produced, the customer can provide input throughout evolution of the product.

Customers can drive certain design decisions, and revaluate requirements based on time-constraints and changes to the business environment.
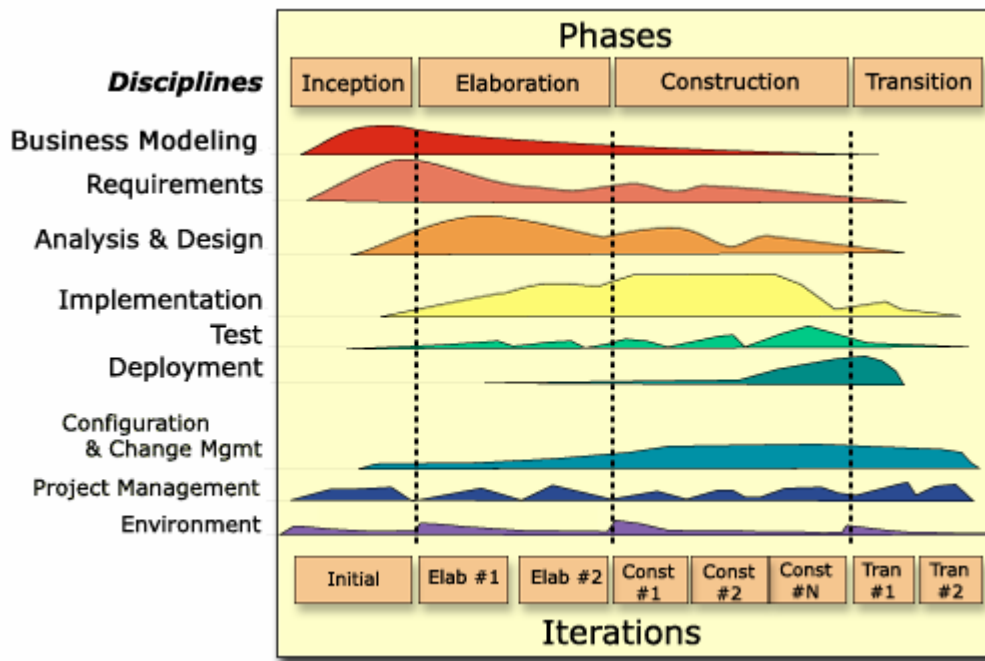
Figure 2: Iterative Development Phases

Implementation Details

This software project was approached using iterative development methods. First, major design and architecture decisions were made. After an overall system architecture had been developed, the underlying model was analyzed and a corresponding class structure was designed. Once the model had been integrated into the GUI system, more specific configuration details were fleshed out. Throughout the entire cycle, EMC employees provided input regarding design decisions and functionality.

Major Design Decisions

The first major design decision was to use the Java programming language. Java provides cross-platform compatibility, numerous GUI libraries and an object-oriented

11

development environment. Specifically, Java release 1.4.2 was chosen because existing

EMC^2 tools relied on this library and the choice of this version of Java insured that a

compatible runtime environment would be available.

The next choice was which Java GUI library to implement. After meeting with

developers at EMC^2 with previous Java GUI experience, Eclipse Foundation's *Standard

Widget Toolkit* (SWT) was chosen. Java's AWT library was generally considered to be

less platform independent and performance and aesthetic issues have been identified with

the Swing interface.

Applying Design Patterns

The primary requirement for this application is to provide on-the-fly layout

capabilities for a Symmetrix configuration. This calls for an architecture which maintains

a persistent representation of the configuration space, as well as an in-memory

representation of current configuration. The current configuration must also be visually

represented to the user, and user input must be processed in order to modify the model.

This input→processing→output architecture was identified as an applicable scenario for

the *Model-View-Controller* (MVC) design pattern.

The MVC paradigm separates the managing of data, user input and visual

representation into three distinct components (Figure 3). By decoupling the visual

representation from the model, functional interdependencies are minimized and a more

maintainable architecture is created.
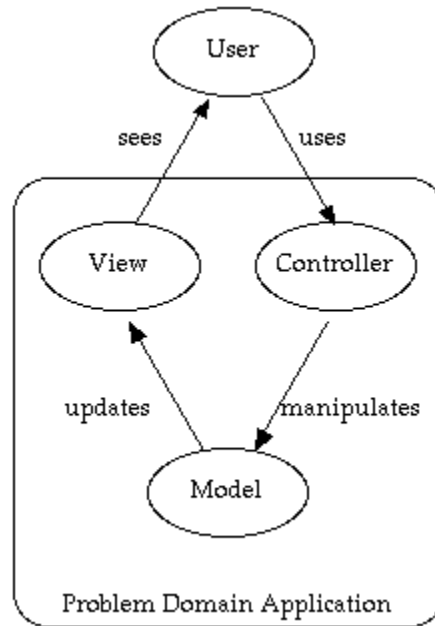
Figure 3: Model-View-Controller (Baray)

Commercially available layout packages (JGraph, Eclipse GDE) provide a template for developing MVC applications. They establish GUI components and input handling and connect them to a customizable model. Often times 3rd party packages include pre-defined layout routines. Due to the necessity for flexibility and the licensing costs associated with 3rd-party layout it was decided that the MVC architecture would be implemented from scratch.

Defining the Model

Once a base GUI was generated using the SWT library, it became necessary to begin developing the class model for representing the current configuration. Through an analysis of open-source MVC example implementations, an abstract model was developed (Figure 4). This abstract model involved an *EditPart* which was stored in a collection managed by the controller. An EditPart was associated with a separate model

13

element and view element which stored information corresponding to the data and visual representations respectively. This abstract model was eventually refined into a model specific to the configuration problem at hand (Figure 5).
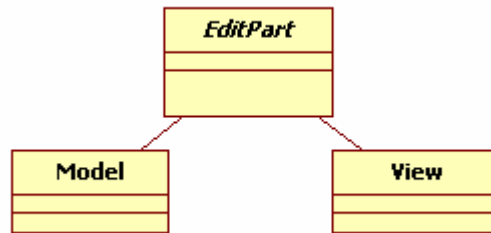
Figure 4: Base Abstraction
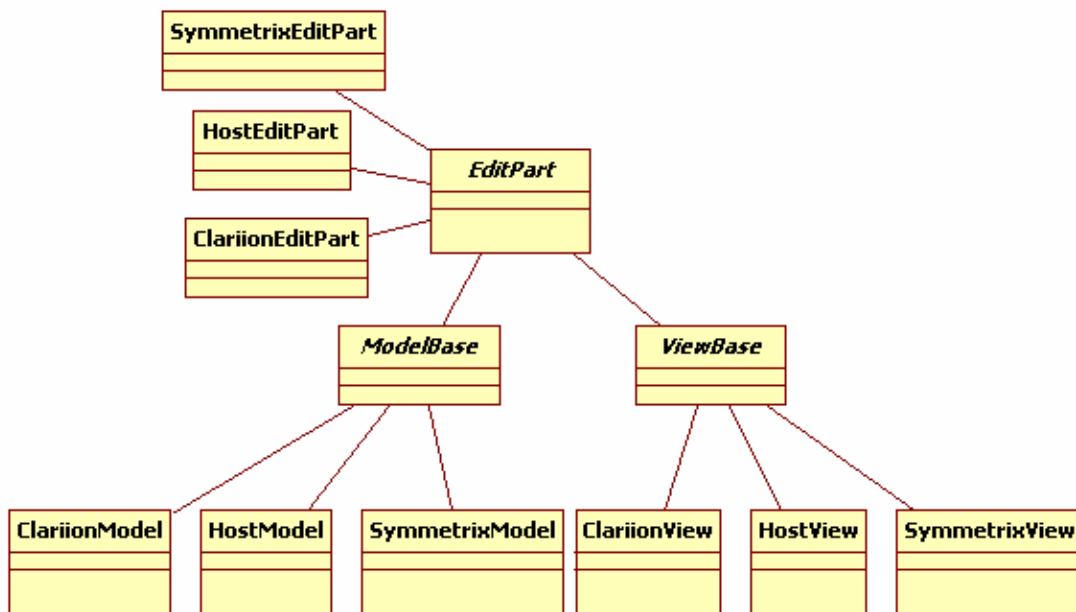
Figure 5: MVC Implementation

The ViewBase abstract class is associated with a SWT Label object, which displayed an image of the piece of hardware being represented. The ModelBase abstract class stores parameters specific to all entities, with subclasses which store specific information pertaining to a Host, Symmetrix or CLARiiON. When the controller

14

processes a modification request for a specific entity, as dictated by user input, the EditPart for that entity is updated. The EditPart inherited classes handle the relaying of events to both the ModelBase and the ViewBase.

Through use of this model, a system was developed in which the user could add, modify and delete basic entities. In the first pass, sub-entity types (Symmetrix generation or Host operating system) were managed by a hard-coded enumeration. All property values were stored as unconstrained member variables within the specific ModelBase instance. At this point it became necessary to modify the way in which specific properties were managed.

Refining Type Specifications

The complexity of the underlying model dictated a complicated means of defining and constraining the configurable aspects of each entity. Extensibility requirements, on the other hand, call for a user accessible means of defining and modifying these aspects. For these two reasons, it was decided that configuration files written in the Extensible Markup Language (XML) and parsed at run-time, would be used to define the specific parameters associated with each entity type. The loaded entity types would be exposed to the user through a palette, generated after loading the types from file.

XML is a structured data language which is quickly becoming a standard for data representation. XML parsers are widely available, and the structured nature of the language provides a level of readability which will accommodate user-level configuration. In addition, XML is universally compatible and includes language-inherent mechanisms for transforming and validating data.

Three parsers were developed which each handled a specific entity type

configuration file (Host, CLARiiON and Symmetrix). The configurable parameters of

each type were exposed as elements (see Figure 6 for an example type file). Parameter

constraints were grouped together with their dependent variable to provide constraint

information. The parsers translated the XML type representations into HostType,

ClariionType and SymmetrixType objects accordingly. These type objects could then be

associated with a ModelBase and exposed to the user through a configuration dialog.

In the cases where a property constrained the values of another property, sub-type

classes were defined (specifically HostArchType and SymmVersionType). These sub-

type classes were stored within their parent type class and used to govern the values

exposed to the user (through use of combo boxes) when editing properties through a

configuration dialog.

```
- <SymmetrixType>
        <name>DMX</name>
        <image>symm_dmx.jpg</image>
-        <version>
                <ucode>5670</ucode>
                 <max_disks>8000</max_disks>
                 <aliases>31</aliases>
         </version>
-        <version>
                <ucode>5671</ucode>
                <max_disks>16000</max_disks>
                <aliases>31</aliases>
        </version>
        <cache>64</cache>
        <num_connections>64</num_connections>
        <disk_size>73</disk_size>
         <disk_size>146</disk_size>
   </SymmetrixType>
```

Figure 6: Sample XML Type Definition

Generating and Verifying Output

  The next step was to develop the output mechanism used to generate the XML representation. It was important to ensure that all the data generated from the tool represented a valid configuration. XML provides various validation mechanisms, some which verify the *well-formedness* (formatting and syntax) of the XML itself, others which pertain to the actual validity of the data stored in the XML document. For this project, it was decided that XML Schema Definition (XSD) configuration files would be used to verify the output of the system.

  An XSD configuration file is a template for an XML document. It is written in XML, using conventions derived from the W3C, and is used in conjunction with an XML file. XSD files allow for typing of element values, as well as the constraining of element value ranges through enumeration. Tools and libraries exist which link XML and XSD files and provide automated verification.

  An XSD document was developed in conjunction with the XML output functionality encapsulated within the XMLGenerator class. The class iterated over the model to produce a series of entity elements, with specific properties stored as sub-elements (see Figure 7 for an output example). The full XSD type definition of the output of the system, as well as the XSD documents for type configuration files, can be found in Appendix B.

```
<Symmetrix>
        <id>0000000164</id>
        <ucode>5771</ucode>
        <disksize>400</disksize>
        <cache>64</cache>
        <type>DMX3</type>
        <num_devices> 400 </num_devices>
        <num_gk> 4 </num_gk>
        <device_set>
                <device_range>
                        <start_num>0</start_num>
                        <capacity>300</capacity>
                        <d_type>GateKeeper</d_type>
                        <d_emulation>4</d_emulation>
                </device_range>
        </device_set>
</Symmetrix>
```

Figure 7: Example Output

Enhancing the User Interface

One of the main criteria for the success or failure of this project was the usability

of the system itself. If users became frustrated or were aesthetically displeased with the

application, they would be less likely to use it to assist in their development. Towards the

end of the development cycle, three major changes were made to the user interface to

specifically accommodate suggestions encountered during testing.

The original dialogs used to expose entity properties to the user were described as

"busy" and "confusing". We decided that a generic dialog would be developed using

table objects. One column of the table would hold property name and the second column

would accommodate user input, either through text boxes or fixed-list combo boxes. The

property name and combo box values were filled using the type file associated with the

element. The dialogs for entering properties were now much more readable, and less

confusing as they adhered to the same standard throughout the entire application.

The second major modification involved the layout of the palette. Throughout the

development cycle many different approaches were taken to the palette. Originally, fixed

buttons were used. Due to the configurability added through run-time type parsing, it was impossible to use a fixed set of buttons, as the number of available types could change from run to run. Multiple designs were attempted to expand the use of buttons, in an attempt to emulate the cascading button menus seen in programs such as Microsoft Visio, all resulting in significantly time-consuming or aesthetically displeasing results. It was eventually decided that a 'TreeMenu' would be used for the palette. The expand and collapse capabilities of the tree menu, as well as configurable image icons for tree items provide a good solution, and time-efficient implementation.

The last enhancement made was the use of double-buffering to reduce flickering during entity dragging. It was found that, under certain hardware and software conditions, the layout would flicker when dragging and redrawing entities and connections. To fix this, on a re-paint of the canvas, an image was built separate from the canvas area and then painted onto the canvas once complete.

Documentation

Support is inherently provided to the future developer through XML schemas and inline code commenting. In addition to this, we decided that full system documentation would be written using the Javadoc facility. Javadoc provides commenting standards which provide specific information about methods, their intended functionality and parameters. Javadoc style comments can then be parsed out of Java code, using pre-developed utilities, to create HTML based documentation. This HTML documentation provides an entire system overview and can be used by future developers to both understand and debug the system.

Results

The result of this project was a fully functional application which met all the primary requirements specified by EMC. The final product consisted of 4179 lines of code, broken up into 43 separate classes, in 7 packages. Metrics run over the code, using tools in the Eclipse IDE, indicated minimal class interdependencies.

A presentation to the EMC group which will primarily use the tool received excellent feedback. The application is slated for integration into existing tools during the Summer of 2006. Future expansion is also planned. (see Figure 8 for a screenshot of the final product)
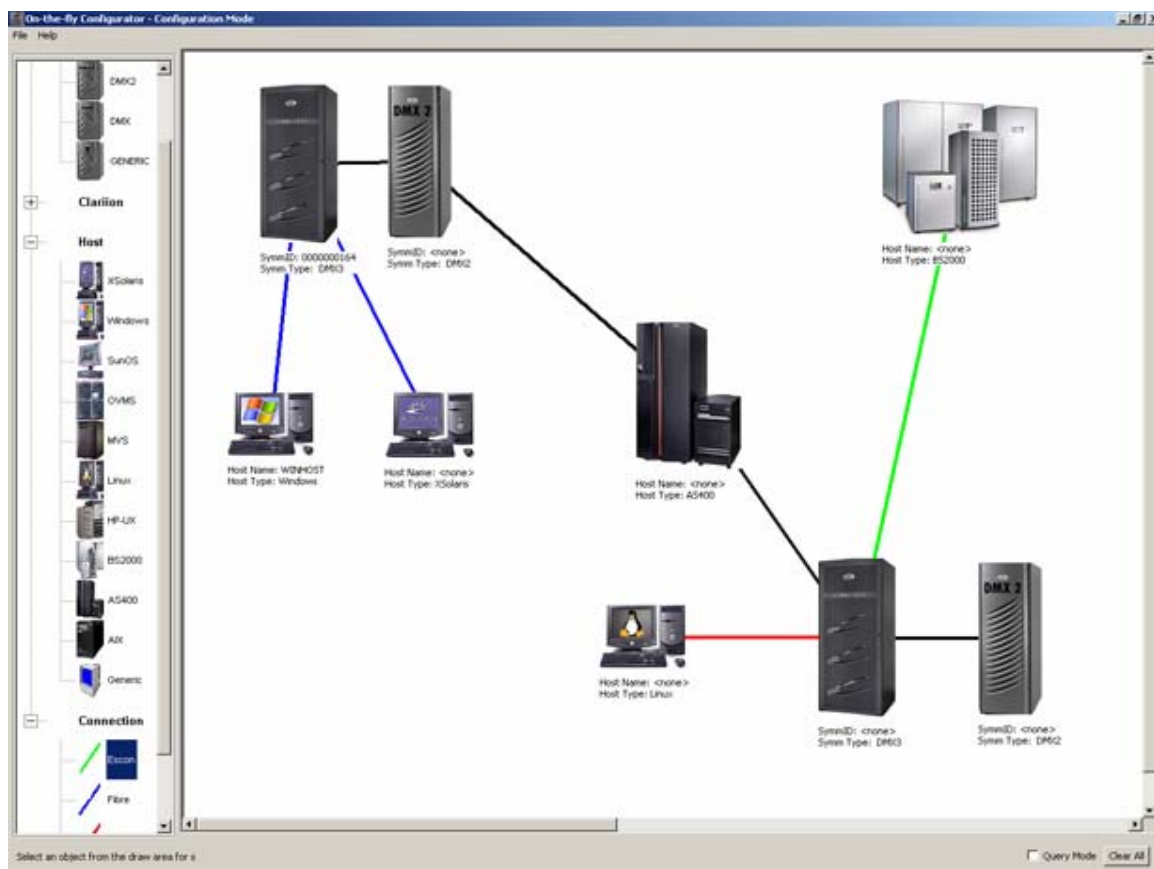

Figure 8: Final GUI

Conclusions

The iterative development process provides an excellent foundation for developing quality applications in a timely manner. The availability of the developer to receive customer feedback throughout the course of the project is critical in delivering a successful product.

When designing for extensibility it is important to keep the user and future developer in mind at all times throughout the development cycle. By paying careful attention to what decisions are made before committing to a design, future developers are supported by an architecture that should never require major modification.

Final Recommendations

There are a few features that, due to time constraints, were not able to be implemented within the first deployment of this tool, they will be individually outlined and explained here:

- **Auto-Layout Options**
  - Automatic layout of entities within the view area (including the minimization of connection crossing) was an ambitious venture for the first attack at this problem. In the future developers may want to consider implementing a MVC library in order to accomplish this task.
- **Load from XML Functionality**
  - Currently, once a layout is made and saved it can not be reloaded. The parsing of the output file would not be difficult, but has yet to be

implemented. This function is somewhat dependent on the previously

mentioned auto-layout functionality. Once load from XML is

implemented, users will be able to load previously generated seed files and

edit the configurations accordingly.

- **Exposure of Additional Parameters / Configuration Methods**
    - o One suggestion that came from EMC developers after the initial release is

        that Directors be configurable for Symmetrix entities. In addition to

        directors there are countless other configurable aspects that could be

        exposed in order to provide increased configurability. It has also been

        proposed that drag and drop placement of boards and drives be used to

        configure individual Symmetrix entities. This method would be much

        harder to implement, but much more intuitive.

- **Keyboard Driven Input**
    - o As previously mentioned, multiple input types were supported within this

        application. This included drag and drop, point and left-click and right-

        click. One input requirement that was not met was the use of the keyboard

        only to place and modify configurations. The framework for implementing

        new input types already exists within the system.

References

Baray, Cristobal. "The Model-View-Controller (MVC) Design Pattern." March 1999.
    <http://cristobal.baray.com/indiana/projects/mvc.html>

Gamma, Erich, et. al. <u>Design Patterns: Elements of Reusable Object-Oriented</u>
    <u>Software</u>. Addison-Wesley Professional, January 15, 1995.

Meyer, Bertrand. <u>Object-Oriented Software Construction</u>. Prentice Hall PTR,
    March 21, 2000.

Preece, Jenny, Rogers, Yvonne, Sharp, Helen. <u>Interaction Design</u>. Wiley,
    January 17, 2002.

Royce, Winston. "Managing the Development of Large Software Systems." Proceedings
    of IEEE WESCON. August 1970, 1-9

Steinberg, Daniel, Palmer, Daniel. <u>Extreme Software Engineering: A Hands-On</u>
    <u>Approach</u>. Prentice Hall, October 10, 2003.

Appendix A: Original Tech Specification (edited due to NDA)

This project is part of our plans to support new configurations on-the-fly. By using an XML file to describe the configuration, users can build up a non-existent lab environment that can then be fed into the [EXISTING TOOLS]. The onus is then on the [EXISTING TOOLS] developers to build up a suitable syscall cache to match the XML file. Once a "lab" is read into the [EXISTING TOOLS], it can be extended by reading in subsequent XML files describing changes to the lab.

As part of this effort, a GUI is envisioned that will allow the lab planner to drag and drop any number of hosts onto the lab designer window. Any number of Symms can also be dropped into the designer window. Connection can be made from any host to any Symm; these Symms will be the "Local Symms". Connections can be made from any local Symm to any other Symm; these are the "Remote Symms". Connections can be made from any Remote Symm to any other Symm; these are the "Remote Multi-hop Symms". Although the [EXISTING TOOLS] can handle any number of hops out from a local Symm; the SYMAPI software only can handle two hops out.

The user should be able to have control of the placement of the hosts and the Symms in the designer window, if developers have time; an on-the-fly placement algorithm is envisioned that will attempt to minimize connection crossings.

Once the lab is laid out (or at host or Symm placement time), the planner can dive into the hosts, specifying host names, OS types, architectures, and word sizes. Clicking on a

connection between a host and local symm, the planner can indicate the number of

PDEVs desired for that Symm; possibly even indicating the SCSI/Fibre controller, bus,

and LUN where applicable. By right clicking on a Symm, the planner can indicate the

Ucode revision, the Symm type, the Symm model, the number of disks in the Symm, the

number of devices (and the number of each type of device: BCV, R1, R2, R1BCV,

R2BCV, VDEV, RAID5, metas, etc…). If time allots, it may be possible for the GUI

developer to build in a drag and drop interface to load memory boards, disk adapters,

ESCON adapters, etc… into the Symms allowing almost any type of Symm

Configuration. Clicking on a connection between any two Symms, the planner can

specify the number of RDF groups (for both going and coming).

When the planner is satisfied with the lab layout, selecting the "Save" Button from the

"File" pull down menu will result in an XML description file being generated. This file

is suitable for importing into the [EXISTING TOOLS].

Additionally, if time permits, an XML description file may be imported into the

configuration designer tool and the hosts, Symms, and connections shall be automatically

placed in the lab designer window so as to minimize crosses of the connections. This

new automatically generated placement may then be altered by a planner with the altered

lab environment written out to a new XML file.

The following diagram depicts a possible implementation for the GUI:
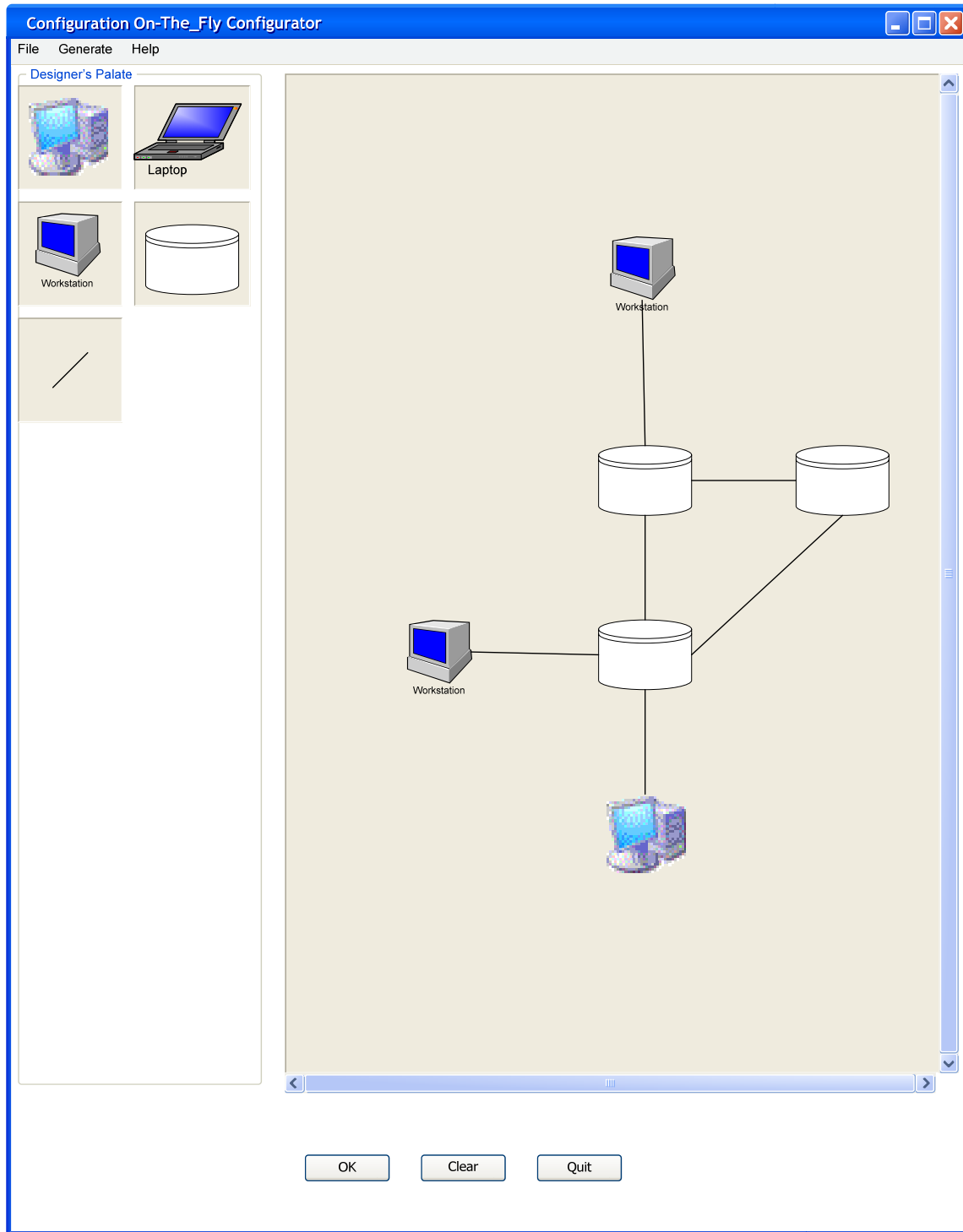
Appendix B: XSD Schema Files for Type and Output

Symmetrix Type Definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="SymmetrixTypeSet">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="SymmetrixType" type="SymmetrixType" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="SymmetrixType">
  <xsd:sequence>
   <xsd:element name="name" type="xsd:string"/>
   <xsd:element name="image" type="xsd:string"/>
   <xsd:element name="version" type="Version" minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element name="cache" type="xsd:integer" minOccurs="0" maxOccurs="1"/>
   <xsd:element name="num_connections" type="xsd:integer" minOccurs="0" maxOccurs="1"/>
   <xsd:element name="disk_size" type="xsd:integer" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Version">
  <xsd:sequence>
   <xsd:element name="ucode" type="xsd:integer"/>
   <xsd:element name="max_disks" type="xsd:integer"/>
   <xsd:element name="aliases" type="xsd:integer"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

Clariion Type Definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="ClariionTypeSet">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="ClariionType" type="HostType" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="HostType">
  <xsd:sequence>
   <xsd:element name="name" type="xsd:string"/>
   <xsd:element name="image" type="xsd:string"/>
  </xsd:sequence>
 </xsd:complexType>
```

```
</xsd:schema>
```

Host Type Definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="HostTypeSet">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="HostType" type="HostType" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="HostType">
  <xsd:sequence>
   <xsd:element name="name" type="xsd:string"/>
   <xsd:element name="image" type="xsd:string"/>
   <xsd:element name="os_value" type="xsd:string"/>
   <xsd:element name="os_string" type="xsd:string"/>
   <xsd:element name="sub_type" type="SubType" minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element name="arch_type" type="ArchType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="SubType">
  <xsd:sequence>
   <xsd:element name="sub_value" type="xsd:integer"/>
   <xsd:element name="sub_string" type="xsd:string"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="ArchType">
  <xsd:sequence>
   <xsd:element name="arch_value" type="xsd:string"/>
   <xsd:element name="arch_string" type="xsd:string"/>
   <xsd:element name="arch_os" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
   <xsd:element name="escon" type="xsd:boolean"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```