國立交通大學

資訊學院 資訊學程

碩士論文

以視覺化使用者介面建構方法論應用於儲存系統之人機介面設計及實作

Using Visual-Based User Interface Construction Methodology for the Man Machine Interface Design and Implementation of Storage Systems

研 究 生：黃承一

指導教授：陳登吉 博士

中華民國 九十六 年 七 月

以視覺化使用者介面建構方法論應用於儲存系統之人機介面設計及實作

Using Visual-Based User Interface Construction Methodology for the Man Machine Interface Design and Implementation of Storage Systems

研 究 生：黃承一　　Student：Cheng-Yi Huang

指導教授：陳登吉 博士　　Advisor：Deng-Jyi Chen

國 立 交 通 大 學

資 訊 學 院 資 訊 學 程

碩 士 論 文

A Thesis
Submitted to College of Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science
July 2007

Hsinchu, Taiwan, Republic of China

中華民國 九十六 年 七 月

以視覺化使用者介面建構方法論應用於儲存系統之人機介面設計及實作

學生：黃承一　　指導教授：陳登吉 博士

國立交通大學　資訊學院　資訊學程碩士班

摘　要

使用傳統的人機介面開發方法，當需求規格改變時，程式設計師必須更改人機介面相關的程式。因此在開發與維護人機介面的程式，需要花費大量的時間與人力。

本校資工系的軟體工程實驗室，已經發展出視覺化使用者介面建構方法論，可以提昇人機介面軟體的生產力、品質及可維護性。本研究將應用此方法論於儲存系統之人機介面設計及實作上，以克服傳統方法會遇到的難題。

儲存系統之人機介面有其獨特性與多樣性，我們將設計出適合於儲存系統之視覺化人機介面開發方法，其中包含視覺化編輯過程以及可用於儲存系統之通用型人機介面引擎。

為了驗證此開發流程的可行性，我們實作一個軟體框架，內含通用型人機介面引擎，可操作視覺化編輯工具產生的資料，並與儲存系統之管理程式庫互動，以控制一個實際的儲存系統裝置介面卡。藉由此系統的建立亦可驗證視覺化使用者建構方法論的可行性及應用性。

關鍵字：視覺化、使用者介面、人機介面、儲存系統

Using Visual-Based User Interface Construction Methodology for the Man Machine Interface Design and Implementation of Storage Systems

Student: Cheng-Yi Haung Advisor: Dr. Deng-Jyi Chen

Degree Program of Computer Science
National Chiao Tung University

Abstract

When the requirements of Man Machine Interface (MMI) are changed, programmers must change the related programs of MMI if they use the conventional MMI development. Therefore, they have to take long time to develop and spend much effort to maintain programs of MMI.

The Software Engineering Laboratory of NCTU had developed *Visual-Based User Interface Construction Methodology*. This methodology can improve the productivity, quality, and maintainability of MMI software. In this thesis, we use this methodology for the MMI design and implementation of storage systems, and try to conquer UI problems using conventional MMI development.

For the typical and various Man Machine Interfaces of storage systems, we design the *Visual MMI Development for Storage Systems*. It includes *Visual Authoring Process* and *Generic MMI Engine* for storage systems.

In order to demonstrate the feasibility of the *Visual MMI Development for Storage Systems*, we implement a software framework with *Generic MMI Engine* that can manipulate the output data of *Visual Authoring Tool*, interact with the Management API of storage systems, and use this system to control a functional storage adapter. A real application example using the proposed approach is applied to demonstrate the applicability of the methodology.

Keywords: Visual-based, UI, MMI, Storage System

誌　　謝

在累積工作經驗後，重回母校進修，不僅在研習過程中的想法與目標更為明確、對學術研究有更深入的理解與啟發，並自我期許能學以致用，在實務領域有更進一步的助益。

本論文的完成，首先要感謝　陳登吉老師不辭辛勞的指導與教誨，才得以順利付梓。老師一向認真堅毅的態度，是學生們最好的身教。同時亦感謝論文口試委員，蕭嘉宏教授、陳振炎教授、黃俊龍教授，提供許多寶貴的意見及精闢的建議，使得此論文未臻完備之處得以補正。

論文研究期間，承蒙蔡明志學長的勉勵與指教，以及翁浚恩、張正隆等同學們諸多的協助與相互扶持，還有工作上長官與同事們的分憂解勞，才能在學業與工作的雙重壓力之下，如期的完成此篇論文。

最後，我必須感謝最親愛的父親、母親與妻子婷婷，在我求學過程中無限的體諒與支持，是我能完成學業的依靠。有你們對我的期望與鼓勵，是我不斷向前進步的原動力。

黃承一　謹誌

交通大學　資訊學院　資訊學程碩士班

中華民國九十六年八月

Contents

List of Figures

List of Tables

Chapter 1
Introduction

In this chapter, we introduce and overview storage systems with their typical and various Man Machine Interfaces. Also, the motivation and goal of this thesis study are addressed.

1.1 Overview of Storage Systems

In current storage systems, we can classify storage services into 1) ***closed system*** and 2) ***opened system*** at first tier [1]. The *closed system* is one kind of mainframe that provides storage services for servers that use closed operating system, e.g. IBM AS400 servers. The *opened system* is a set of storage servers that use opened operating system, e.g. Microsoft Windows, Linux, or Unix, etc. Figure 1 illustrates the classification of storage systems.

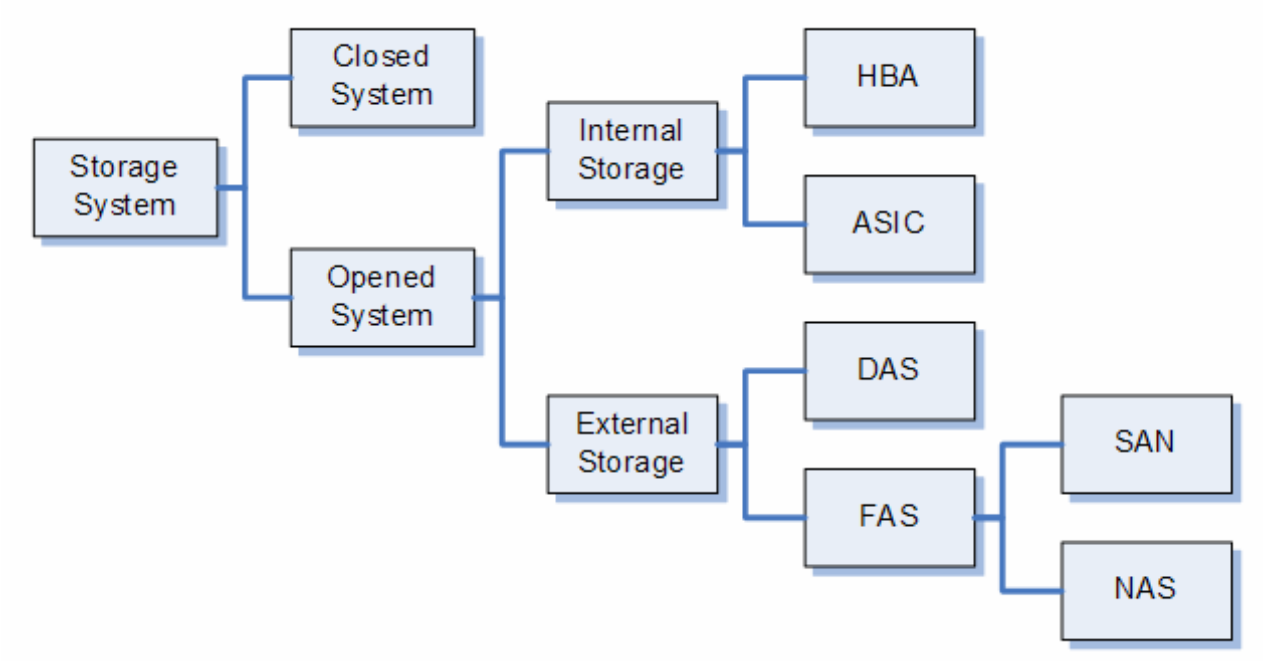


Figure 1. The Classification of Storage Systems

For *opened system*, we divide storage devices into 1) ***internal storage*** and 2) ***external storage*** at second tier by the connections between servers and storage devices.

There are two types of *internal storage systems*: ***HBA*** (Host Bus/Based Adapter) and ***ASIC*** (Application Specific Integrated Circuit). The administrator of servers can insert one or more standalone *HBA* adapters into the slots on server board to increase the capability to connect more hard disk drives. On the other hand, more and more chipsets are integrated with RAID (Redundant Array of Independent/Inexpensive Disks) controller or storage *ASIC* so that they also have the capability to connect many hard disk drives.

For *external storage systems*, there are also two types: ***DAS*** (Direct Attached Storage) and ***FAS*** (Fabric Attached Storage). The *DAS* devices usually use the SCSI bus as the external connection to the servers. Most of high-end *DAS* devices are the subsystems that run embedded operating system and can handle dozen of hard disk drives.

To describe *FAS* more clearly, we can separate it into ***SAN*** (Storage Area Network) and ***NAS*** (Network Attached Storage). Many *SAN* storage systems use fiber channel as the transfer media, and need a fiber channel switch to connect many devices in the same storage area network. On the contrary, the *NAS* storage systems use the popular Ethernet in the same local area network, and computers can share the *NAS* storage systems very easily.

About more details of *external storage systems*, please refer to Appendix A External Storage system.

1.2 Man Machine Interfaces of Storage Systems

No matter internal or external storage systems, they must have Man Machine Interfaces (MMI) so that users can configure and manage them. For example, users can get a list of physical hard disk drives in the storage system, and group up some of them to make up a disk array in one kind of RAID (Redundant Array of Independent/Inexpensive Disks) level.

After survey the products of storage systems in current market, we sum up them and generalize five kinds of Man Machine Interfaces for storage systems. They are 1) ***Booting Utility***, 2) ***Pre-OS Utility***, 3) ***Web-based Utility***, 4) ***Embedded Utility***, and 5) ***LCD Panel***.

Figure 2 illustrates these five kinds of MMI for storage systems and the dependences of internal and external storage systems on them.

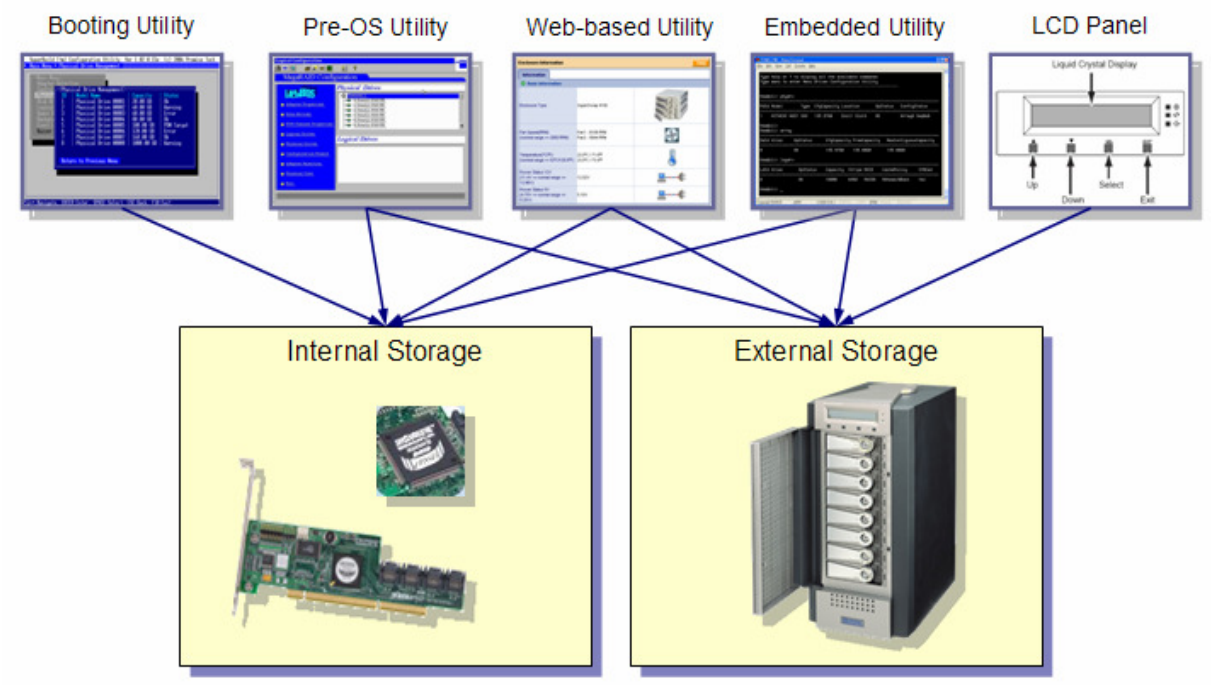


Figure 2. Man Machine Interfaces of Storage Systems

We describe these five kinds of MMI for storage systems as below.

1) ***Booting Utility***: Users can use the Booting Utility to configure hard disk drives and setup bootable drivers before OS is loaded. Usually, this utility is included in the expansion ROM of BIOS (Basic Input Output System) or EFI (Extensible Firmware Interface) [2] driver of storage HBA adapters. When system BIOS or EFI firmware boots up system, they will load the BIOS ROM or EFI driver of the storage HBA adapter into memory and run. The BIOS ROM or EFI driver will prompt some messages to tell users how to enter the Booting Utility. After users press a certain combination of hotkeys, they can enter this utility and manage storage devices. For example, users can select wanted hard disk drives to create a disk array in RAID (Redundant Array of Independent/Inexpensive Disks) level 1 or 5 with fault tolerance. The most particular functionality of the Booting Utility is it can tell system BIOS that which logical drives in the storage system can become bootable drives. However, this

kind of utility usually provides basic functions of storage systems because of the limitation of code and data size.

2) ***Pre-OS Utility***: This kind of utility is used in Pre-OS environments, e.g. DOS (Disk Operation System), EFI (Extensible Firmware Interface) shell, or Windows PE (Pre-installation Environment). Many large corporations use Pre-OS environments for the deployment of workstations and servers. Original Equipment Manufacturers (OEM) also use these environments to preinstall Windows client operating systems to personal computers or laptop and notebook computers during manufacturing. The Pre-OS Utility of storage systems has similar functionalities to the Booting Utility. The advantage of Pre-OS Utility is that users can run it at any time without system reboot.

3) ***Web-based Utility***: The Web-based Utility uses web pages as user interfaces. Since many years ago, almost every operating system provides at least one web browser for network surfing. The advantage of the Web-based Utility is that end users can connect to this kind of utility with web browser conveniently. They do not need to install any client application. Therefore, more and more storage system companies adopt the Web-based Utility for their products.

4) ***Embedded Utility***: Numerous storage systems apply the architecture of embedded system, and engineers usually use a serial cable to connect the console of embedded system. Engineers can use the Embedded Utility to setup advanced configuration or debug internal problems. This kind of utility is running in the embedded operating system. However, the serial console of embedded system and the Embedded Utility use the command-based or menu-based user interfaces. They are not user-friendly. Now, the Embedded Utility is not a popular MMI of storage systems for end users.

5) ***LCD Panel***: Some high-end storage systems have the LCD Panel at their faceplate or front-plane. The maintenance people can use the LCD Panel conveniently to verify new configuration, because these storage systems are usually in one isolated room for security reason. The LCD Panel has the related programs that run in the embedded operating system, too.

We describe the dependences of internal and external storage systems on these five kinds of MMI for storage systems in Table 1.

Table 1. The Dependences of Storage Systems and Their MMIs

MMI	Internal Storage	External Storage
Booting Utility	Supported	None
Pre-OS Utility	Supported	Supported
Web-based Utility	Supported	Supported
Embedded Utility	Supported	Supported
LCD Panel	None	Supported

Table 1 lists the dependences of storage systems and their Man Machine Interfaces. For *internal storage systems*, they have four supported man machine interfaces: 1) *Booting Utility*, 2) *Pre-OS Utility*, 3) *Web-based Utility*, and 4) *Embedded Utility*. For *external storage systems*, they also have four supported man machine interfaces: 1) *Pre-OS Utility*, 2) *Web-based Utility*, 3) *Embedded Utility*, and 4) *LCD Panel*.

Because *internal storage systems* are one part of host servers, they do not have the *LCD Panel*. On the other hand, *external storage systems* do not take the responsibility to boot up host servers, the *Booting Utility* is not necessary for them.

1.3 Motivation and Goal of This Thesis

The conventional approach to develop Man Machine Interfaces for storage system has limited flexibility while the MMI requirement is changed. During development phase, programmers have to change their handcrafted MMI application code again once the MMI requirement is changed. This change may occur repeatedly until the MMI requirement under consideration is satisfied.

For instance, we can study the cases in figure 3. The case on the left side is the MMI development for serial products, and the case on the right side is the MMI maintenance for different customers.

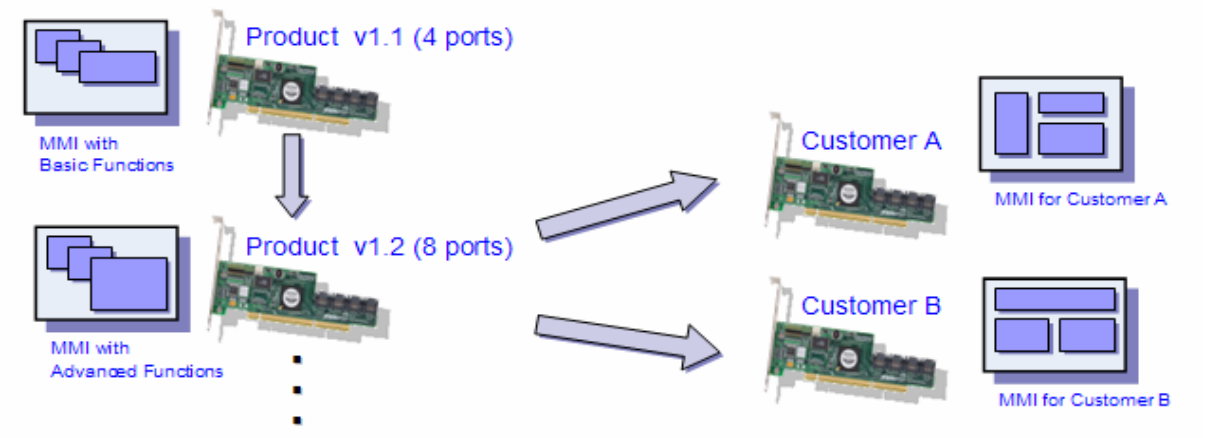


Figure 3 Various MMIs for Serial Products and Different Customers

1) For the case on the left side of figure 3: The MMI requirement for product version 1.1 (with 4 ports to connect hard disk drives) only requires basic functions of the storage system, while the MMI requirement for product version 1.2 (with 8 ports to connect hard disk drives) requires additional advanced functions. MMI designers have to change the menu size to support more ports for more hard disk drives, and more advanced functions for the new serial products. Consequently, MMI programmer must modify their MMI programs to meet these changes.

2) For the case on the right side of figure 3: Different customers bought the same kind of product. Customer A wants to change the layout of MMI into layout A, while customer B wants his layout of MMI in layout B. Since layout A and B are different, the MMI designers and programmers must maintain various MMI programs for these different customers eventually.

Unfortunately, the time to market is usually very short and the human resource to develop and maintain products is very tight, how to shorten the MMI development time and reduce maintenance effort become a major issue for the market competition.

In this thesis study, we attempt to conquer above problems by using ***Visual-Based UI Construction Methodology*** to design the *Visual MMI Development for Storage Systems*. After the construction of this system, we also demonstrate the feasibility and applicability of the *Visual-Based UI Construction Methodology*.

Chapter 2
Related Work

In this chapter, we describe the related work of conventional MMI development, the *Visual-Based UI Construction Methodology*, and the conventional software framework for MMIs of storage systems.

2.1 Conventional MMI Development

In a general software process, we follow the principles and models of *Software Engineering* [3], e.g. *Waterfall Model* in figure 4. During a general software life cycle [4], we compose requirement documents and specifications, plan and design architecture of system and software framework, implement modules on target platforms, do many tests on units and system for verification and validation, and maintain software for upgrade and optimization.

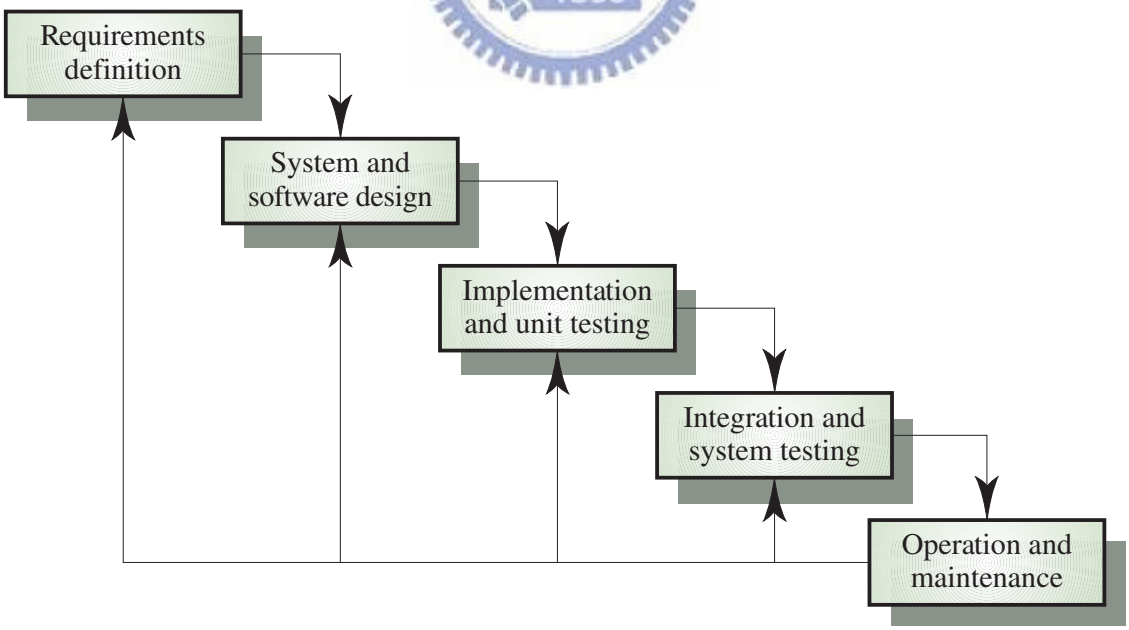


Figure 4. The Waterfall Model for Software Process

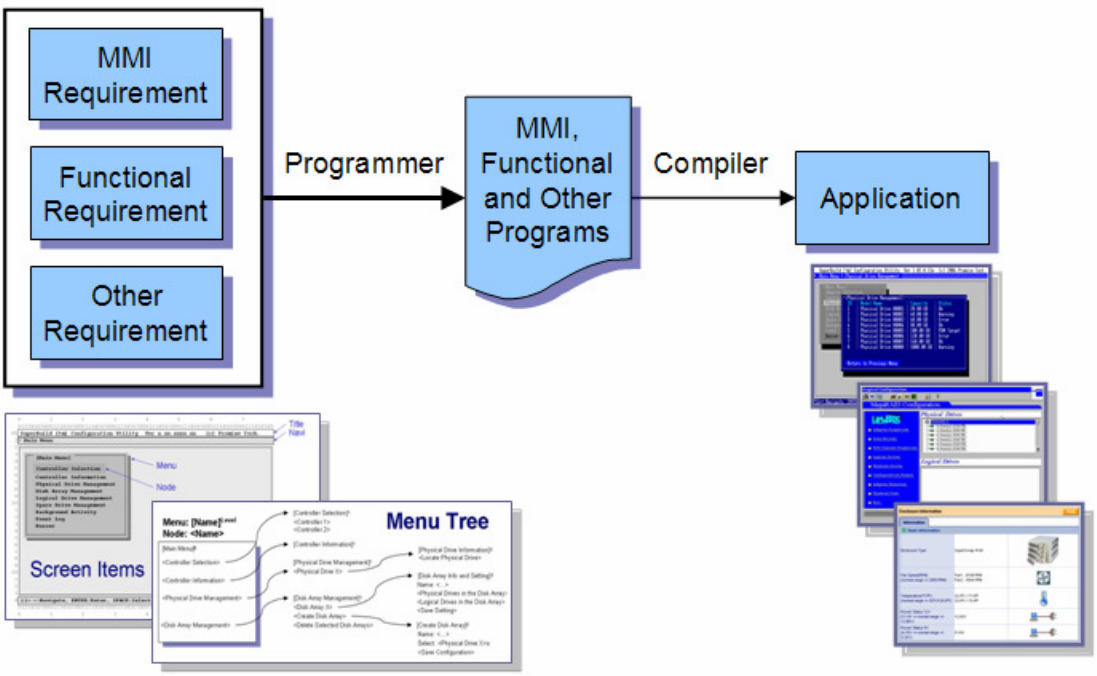


Figure 5. The Concept of Conventional MMI Development

In the brief illustration of figure 5, the concept of conventional MMI development [5], MMI designers have to compose MMI requirement documents with screen layouts or menu tree, etc, and system designers compose functional and other requirement documents. According to these requirement documents, programmers implement MMI, functional and other programs, and compile them together and produce applications. However, these applications are implemented for specific environments, e.g. *Booting Utility* in BIOS / EFI environment, *Pre-OS Utility* in EFI Shell or Win-PE environments. It is hard to design reusable programs for all MMI utilities of storage systems. Usually their MMI programs mix with functional programs and others. Therefore, programmers have to take long time to modify, or develop new programs when MMI requirements were changed. Also, we need much effort to maintain these utilities for serial products and various customers.

2.2 Visual-Based UI Construction Methodology

The Software Engineering Laboratory of NCTU had developed ***Visual-Based User Interface Construction Methodology*** [6]. We illustrate the concept of this methodology in figure 6 briefly.

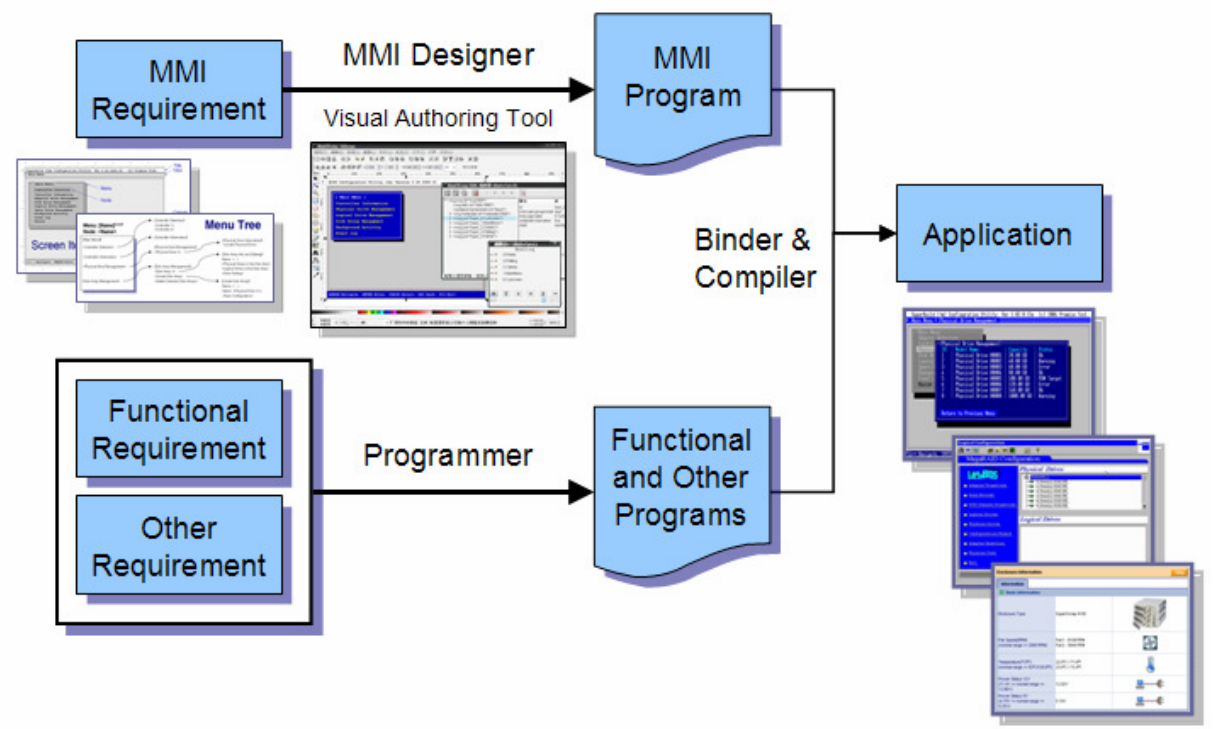


Figure 6. The Concept of Visual-Based UI Construction Methodology

In this methodology, MMI designers use visual authoring tool to compose MMI prototype according to MMI requirements. The visual authoring tool can generate MMI programs. Programmers implement functional and other programs as API library for MMI programs. We can bind the related functions to MMI components, and then produce application utilities very efficiently.

Figure 7 illustrates the detail framework of *Visual-Based User Interface Construction Methodology*. This methodology supports the ***UI Visual Requirement Authoring System*** for UI designers to produce GUI-based requirement scenario and specifications. It also supports the ***Program Generator*** to generate the target application system as specified in the visual requirement representation. The programmer can produce the target application system, base on the function binding features provided in the program generator to bind each GUI component with the associated application function.

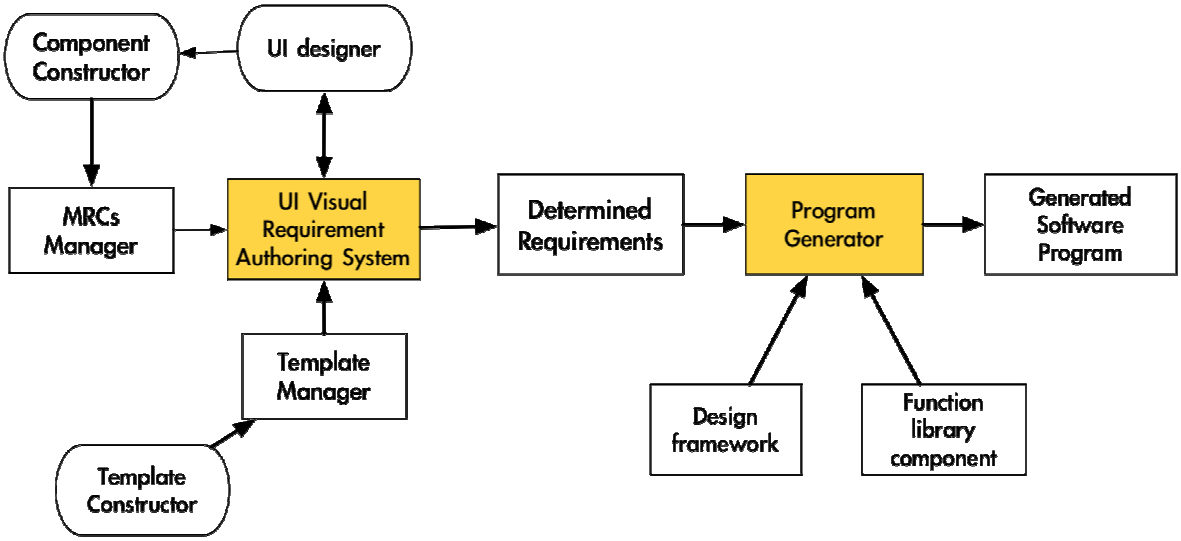


Figure 7. The Framework of Visual-Based UI Construction Methodology

The main components of this system are described as follows:

1) ***UI Visual Requirement Authoring System***: It is a visual-based authoring tool. The UI designers use the UI Visual Requirement Authoring System to create a prototype of user look and feel. Then one can edit this prototype by adding more text or buttons. The UI designers can preview the prototype and modify it. Once the authoring process is completed, the UI designer has generated a target UI system.

2) ***MRCs Manager***: MRC (Multimedia Reusable Components) is the basic component in visual requirement presentation, contains multimedia data by the object-oriented method. It has its own attributes, and accepts external signal to trigger actions. The UI designer uses the existing MRCs to produce the visual UI prototype via visual UI authoring tool. If there are no existing MRCs, the UI designer uses the component constructor to produce new MRCs and then adds it to the *MRCs Manager* for further use in the visual UI authoring tool.

3) ***Template Manager***: It is a database management system for managing UI templates, e.g. structure template, layout template, and style template. It provides an interface for adding or deleting UI templates as well as for retrieving an existing UI template. Template constructor can make new UI templates and then stores them into UI template database through the *Template Manager*.

4) ***Program Generator***: It is a function binding system that generates the program for target system based on the binding of a UI icon to an API function for the underlying device. When the design of user look and feel is satisfied, the UI designers use the program generator to produce the source code of application. The program generator produces the source code according to the visual representation generated by the Visual UI Authoring System. The program generator glues the UI components to the software design framework, and binds function library component with each UI component defined in the generated visual representation.

5) ***Design Framework***: It is used to generate software system architecture for the target application system. The *Program Generator* applies the software framework to generate the source code. A software framework is a platform for representing the visual representation that is generated by the authoring system. Programmers can instantiate a software framework from the generic software framework.

6) ***Function Library Component***: It is a set of pre-defined library. Programmers develop associated functions based on API (Application Program Interface) library functions, and implement them according to the hardware specification. The *Program Generator* applies this function library to produce source code for target UI system.

This *UI Visual Requirement Authoring System* is especially suitable for the UI designer. The visual-based authoring system helps the UI designers to create a prototype of MMI in an efficiency way. The UI designers can edit and preview this prototype and verify its functionality on PC. After the design of MMI is frozen, the UI designer can apply this authoring system to produce the target UI program without writing any textual code. The authoring system uses the code generator to translate the visual representation to source code. The code generator resolves the relationship between the MMI and the application functions of device drivers. It applies the designed framework and function library in code generation phase.

The ***Visual-Based User Interface Construction Methodology*** can improve the productivity, quality, and maintainability of MMI software efficiently [7]. This methodology claims for the following benefits:

1) Replace voluminous textual documentations
2) Support rich multimedia UI components such video, audio, image, and animation for UI designer to easily and quickly authoring visual UI prototype.
3) Generate visual MMI prototype without writing textual based program
4) Independent work between MMI designer and programmer
5) Provide more natural means of communication to reduce misunderstanding between MMI designer and programmer
6) Based on the experiment results, it depicts that using this methodology can reduce the development and maintenance time during the construction of MMI system [7]
7) Elicit an early feedback from user, more expressive in describing user's demands
8) Based on the experiment results, it depicts that using the proposed methodology can reduce the development and maintenance time during the construction of UI system

In figure 8, some experimental data in the dissertation, "*Generating User Interface for Mobile Devices Using Visual-Based User Interface Construction Methodology*" [7], are recalled here.

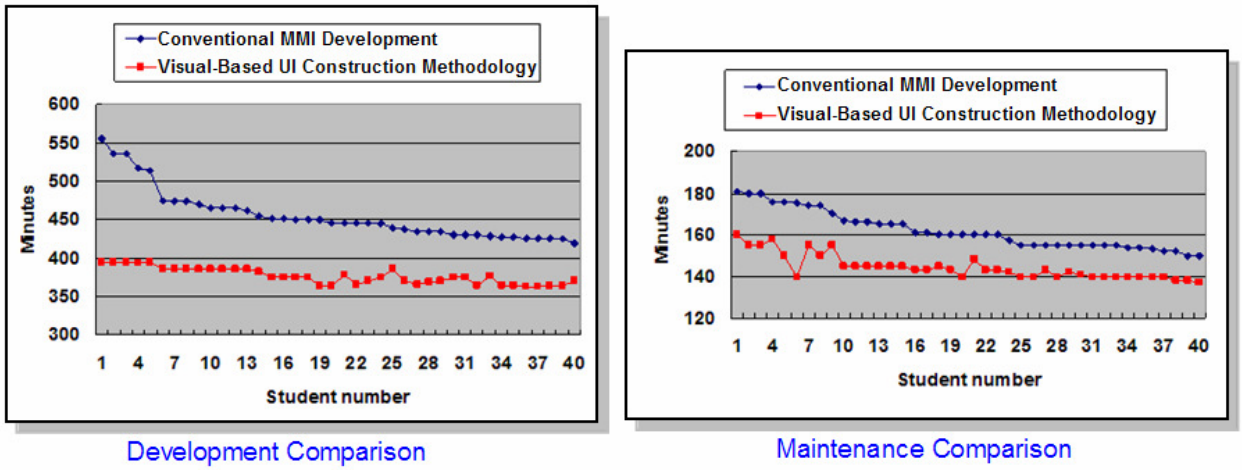


Figure 8. Comparisons between Conventional and Visual-based UI Construction [7]

In the left chart of figure 8, the experiment results compare the development time between the *Conventional MMI Development* and *Visual-Based User Interface Construction Methodology*. Based on the collected data, the upper line with circle dots represents the development time that students use the *Conventional MMI Development,* and the average time spent is 420 minutes. The lower line with square dots represents the development time that

students use *Visual-Based User Interface Construction Methodology*, and the average time spent is 376 minutes.

In the right chart of figure 8, the experiment results compare the maintenance time between the *Conventional MMI Development* and *Visual-Based User Interface Construction Methodology*. The upper line with circle dots represents the maintenance time that students use the *Conventional MMI Development*, and the average time spent is about 154 minutes. The lower line with square dots represents the maintenance time that students use *Visual-Based User Interface Construction Methodology*, and the average time spent is about 140 minutes.

Both charts in figure 8 reveal the development and maintenance time of *Visual-Based User Interface Construction Methodology* are less than the *Conventional MMI Development*.

2.3 Conventional Software Framework for MMIs of Storage Systems

We analyze conventional software framework for MMIs of storage systems, and generalize four layers: 1) ***MMI Program***, 2) ***Management API***, 3) ***Driver Layer***, and 4) ***Embedded Firmware***.

Here is a short example to describe the relationship between these layers. The end-users use the *MMI Program* to display the information or configuration of storage systems. Before *MMI Program* displays these data, it calls the functions of *Management API* to get the particular data, and then it arranges the data into the display screen. *Management API* is a function library, and its functions can submit management commands to get the information or set the configuration of storage systems. These commands will pass to *Driver Layer* because this is the software layer to transfer commands to hardware devices. In the internal side of storage systems, *Embedded Firmware* handles commands from host side and response them with particular results.

Figure 9 illustrates *the Conventional Software Framework for MMIs of Storage Systems* with these four layers.

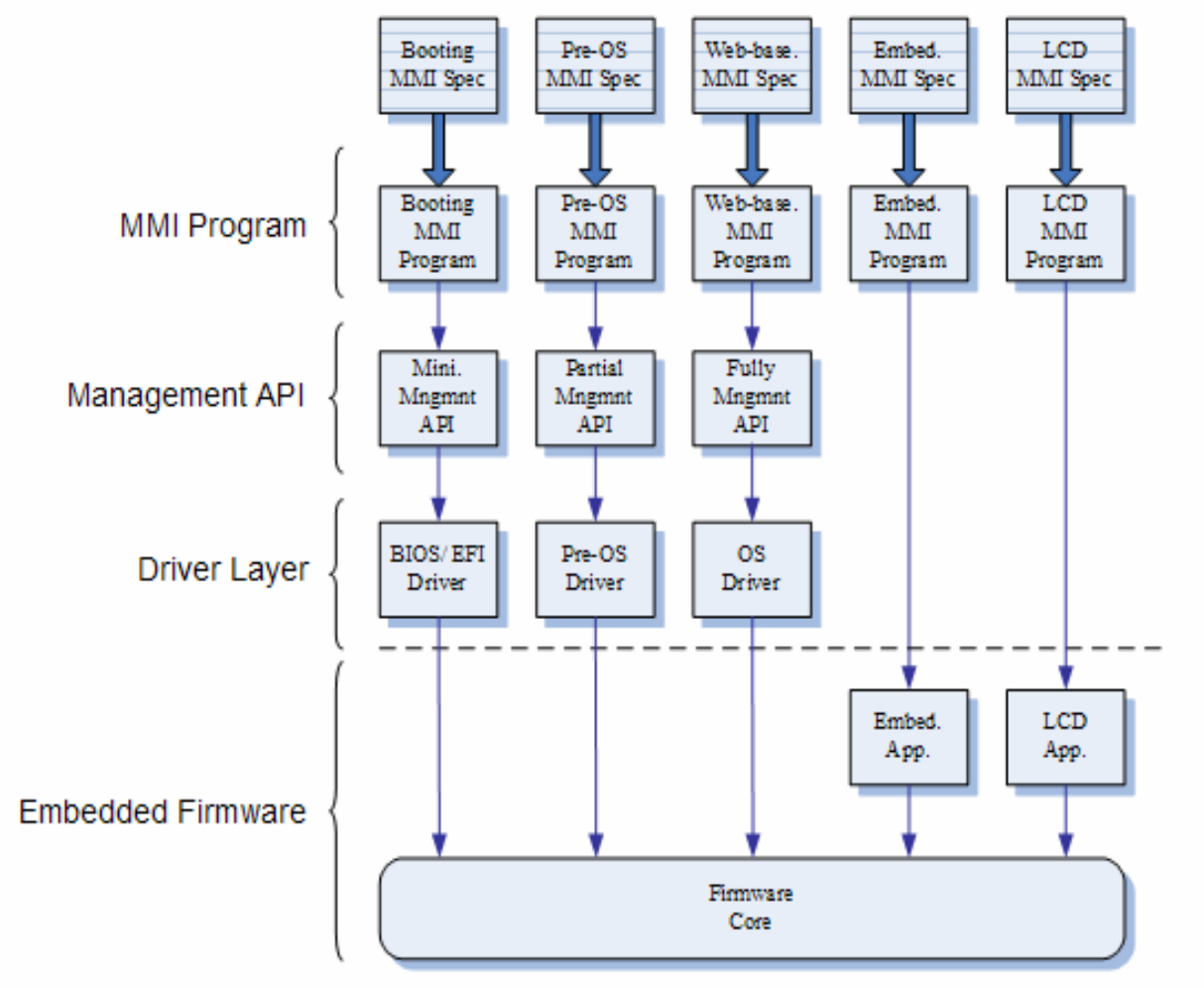


Figure 9. Conventional Software Framework for MMIs of Storage Systems

We describe these four layers in the following paragraphs.

1) ***MMI Program***: According to MMI specifications, programmers implement MMI programs for particular utilities. For instance, they implement the MMI program in assembly language for *Booting Utility*, C language for *Pre-OS Utility*, Java for *Web-based Utility*, etc. These *MMI Programs* call *Management API* to get information or change the setting of storage systems. For *Embedded Utility* and *LCD Panel* of storage systems, there are related applications embedded in firmware side.

2) ***Management API***: *Management API* is a library with many functions for utilities to get information, fetch strings, or change the setting of storage systems. For example, function *MngApi_GetCtrlInfo( )* can get controller information, and function *MngApi_GetPdInfo( )* can get physical drive information. The other functions can set logical driver's cache policy, etc.

First, the *Booting Utility* of storage systems includes basic functions only, so it usually uses minimal *Management API*. Second, the *Pre-OS Utility* is similar to *Booting Utility*, but it includes some advanced functions, and needs some additional *Management API* for these advanced functions. Third, the *Web-based Utility* includes full functions of storage systems, and of course, it uses full *Management API* of storage systems.

3) ***Driver Layer***: Driver provides the capability to submit commands to hardware devices. Usually *Management API* submits a set of management commands, or called IOCTRL (IO Control) commands, to *Driver Layer*. No matter BIOS / EFI driver, Pre-OS driver or OS driver will pass these commands to the embedded firmware of storage systems.

4) ***Embedded Firmware***: In the internal side of storage systems, there are embedded operating system and firmware core running. Firmware core handles commands from host side, transfer data to the buffer of *Management API* by DMA (Direct Memory Access), and return the status of commands to host drivers. For *Embedded Utility* and *LCD Panel* of storage systems, the related applications are running in embedded OS also.

Before MMI programs display the information of storage systems, they must call the functions of *Management API* to get the wanted information. *Management API* provides many functions for MMI programs and submits management commands to firmware by passing through the *Driver Layer* to get data. Driver passes management commands to firmware and returns the status of commands for *Management API*. Embedded firmware takes the responsibility to handle management commands and take the actions for them. For example, *Embedded Firmware* collects information of adapter controller or hard disk drives, or changes the setting of disk array, etc.

Chapter 3
System Design and Implementation

In this chapter, we describe the design and implementation of the *Visual MMI Development for Storage Systems*, and derive the *Generic Software Framework for the MMI Generation of Storage Systems*.

3.1 Visual MMI Development for Storage Systems

In this section, we design the model of ***Visual MMI Development for Storage Systems***. This model uses similar concept of *Visual-Based User Interface Construction Methodology*, MMI designers use visual authoring tool to compose MMIs, and programmers write functional and other programs separately. Figure 10 illustrates this concept.

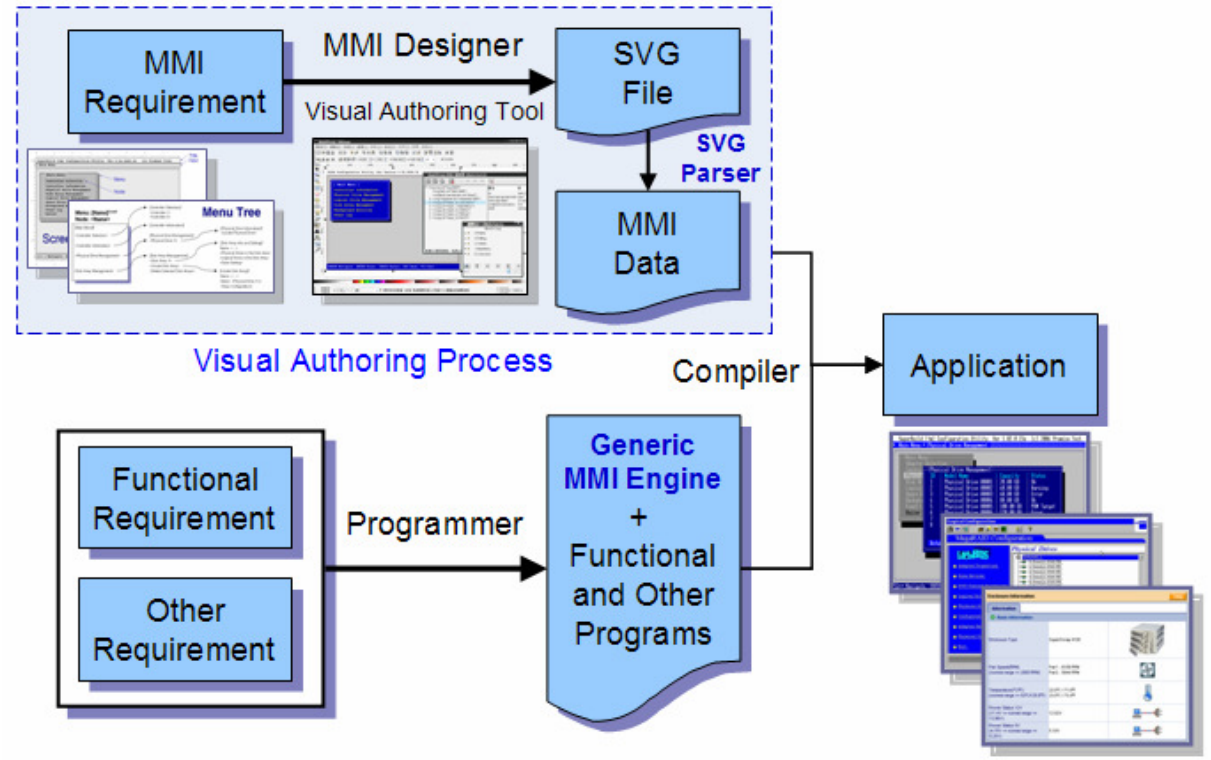


Figure 10. The Concept of Visual MMI Development for Storage Systems

However, the difference between *Visual-Based UI Construction Methodology* and *Visual MMI Development for Storage Systems* is the output of the *Visual Authoring Process*. In the previous methodology, *Visual Authoring Tool* generates MMI program, and programmer binds functions from functional library to MMI program and generate MMI applications.

We change the output of the *Visual Authoring Process* becomes MMI data, and let *Generic MMI Engine* to manipulate this MMI data directly. This change is due to the consideration on some limitations for MMI utilities of storage systems. With this change, we can implement the *Generic MMI Engine* for various environments.

Figure 11 illustrates the framework of ***Visual MMI Development for Storage Systems***.

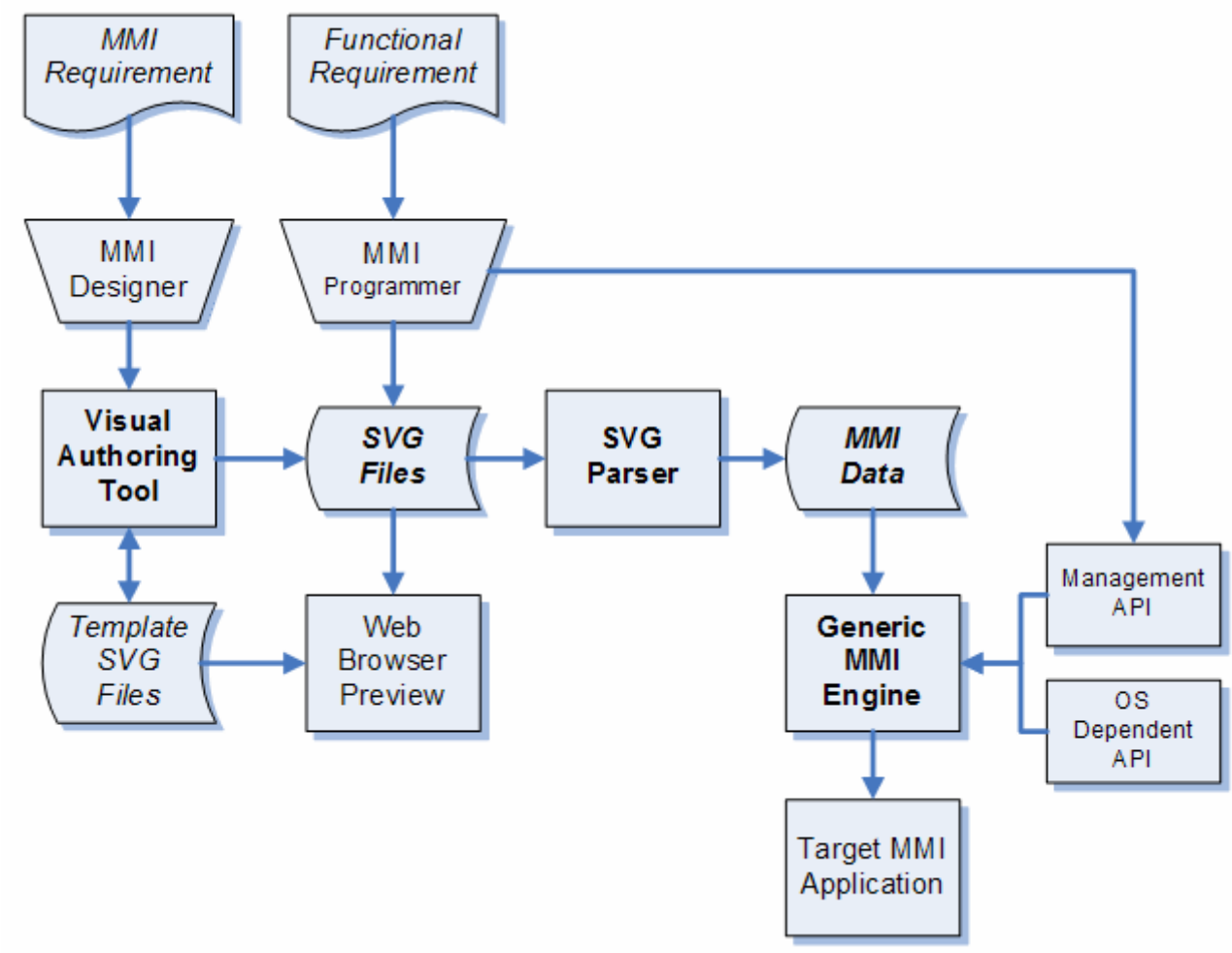


Figure 11. The Framework of Visual MMI Development for Storage Systems

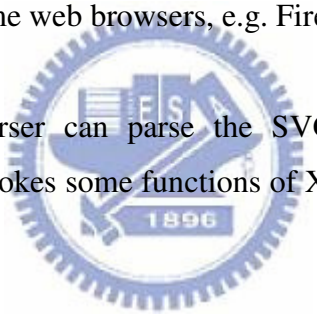
This framework includes the following major parts:

1) ***Visual Authoring Tool***: MMI designers use the *Visual Authoring Tool* to compose wanted MMIs according to MMI requirement documents. During implementation, we choose Inkscape as this tool. Please refer to section 3.3.1 Visual Authoring Tool -- Inkscape.

2) ***SVG Files***: These files are the output of *Visual Authoring Tool*. We use the file format in *SVG* (Scalable Vector Graphics) that is a standard language to describe two-dimensional graphics and graphical applications in XML (Extensible Markup Language). Please refer to section 3.3.2 SVG Files.

3) ***Template SVG Files***: These files are the *SVG files* with templates of MMI elements. To speed up the development of MMIs, we can create these files for MMI designers first. All SVG files can be previewed in some web browsers, e.g. Firefox.

4) ***SVG Parser***: This parser can parse the SVG structure, and obtain required information from SVG files. It invokes some functions of XML parser. Please refer to section 3.3.3 SVG Parser.



5) ***MMI Data***: The output data of *SVG Parser* is *MMI Data*. It is also the results of *Visual Authoring Process*. *Generic MMI Engine* can manipulate the *MMI Data* directly. Please refer to section 3.3.4 MMI Data.

6) ***Generic MMI Engine***: This is the kernel component of *Visual MMI Development for Storage Systems*. It needs the *Management API* and *OS Dependent API* to accomplish the functionalities of target MMI application. Please refer to section 3.3.5 Generic MMI Engine.

7) ***Management API***: This API (Application Program Interface) is depended on the features of storage systems, and it provides functions to manage underlying storage devices. MMI programmers have to program these functions according to functional requirement documents, and *Generic MMI Engine* will invoke these functions.

8) ***OS Dependent API***: *Generic MMI Engine* requires some functional APIs but they are OS dependent. For instance, it needs *Input API* to get the input of keyboard or mouse, and *Display API* to display the MMIs to the monitor screen of computers. We separate the OS dependent code to *OS Dependent API*, and let the implementation of *Generic MMI Engine* become more easily for various environments.

9) ***Target MMI Application***: After MMI designers compose MMIs with *Visual Authoring Tool*, save them as *SVG files*, and use *SVG Parser* to transform *SVG files* to *MMI data*, we can generate *target MMI application* with *MMI data*, *Generic MMI Engine*, *Management API* and *OS Dependent API.*

To use the model of ***Visual MMI Development for Storage Systems***, we describe its flow in the following steps:

Step 1) MMI designers use *Visual Authoring Tool* to compose wanted MMIs according to MMI requirement documents.

Step 2) Save the results of *Visula Authoring Tool* as *SVG Files*.

Step 3) To speed up the development of MMIs, we can create *Template SVG Files* for MMI designers.

Step 4) For developers or customers, they can preview these *SVG Files* in web browsers.

Step 5) MMI programmers add additional information in *SVG Files.*

Step 6) Use *SVG Parser* to transform *SVG Files* to *MMI Data*.

Step 7) MMI programmers program related functions in *Management API* according to functional requirement documents.

Step 8) Generate the target MMI applications with *MMI Data*, *Generic MMI Engine*, *Management API* and *OS dependent API*.

3.2 Generic Software Framework for the MMI Generation of Storage Systems

The conventional software framework for the MMI generation of storage systems lacks of the support of a visual authoring system. We therefore incorporate a visual authoring system into the software framework for MMIs of storage systems. The incorporated *Visual Authoring* and *MMI Engine* are shown in figure 12. We call the new software framework as ***Generic Software Framework for the MMI Generation of Storage Systems***, since it can be used to generate Man Machine Interfaces of many storage systems.

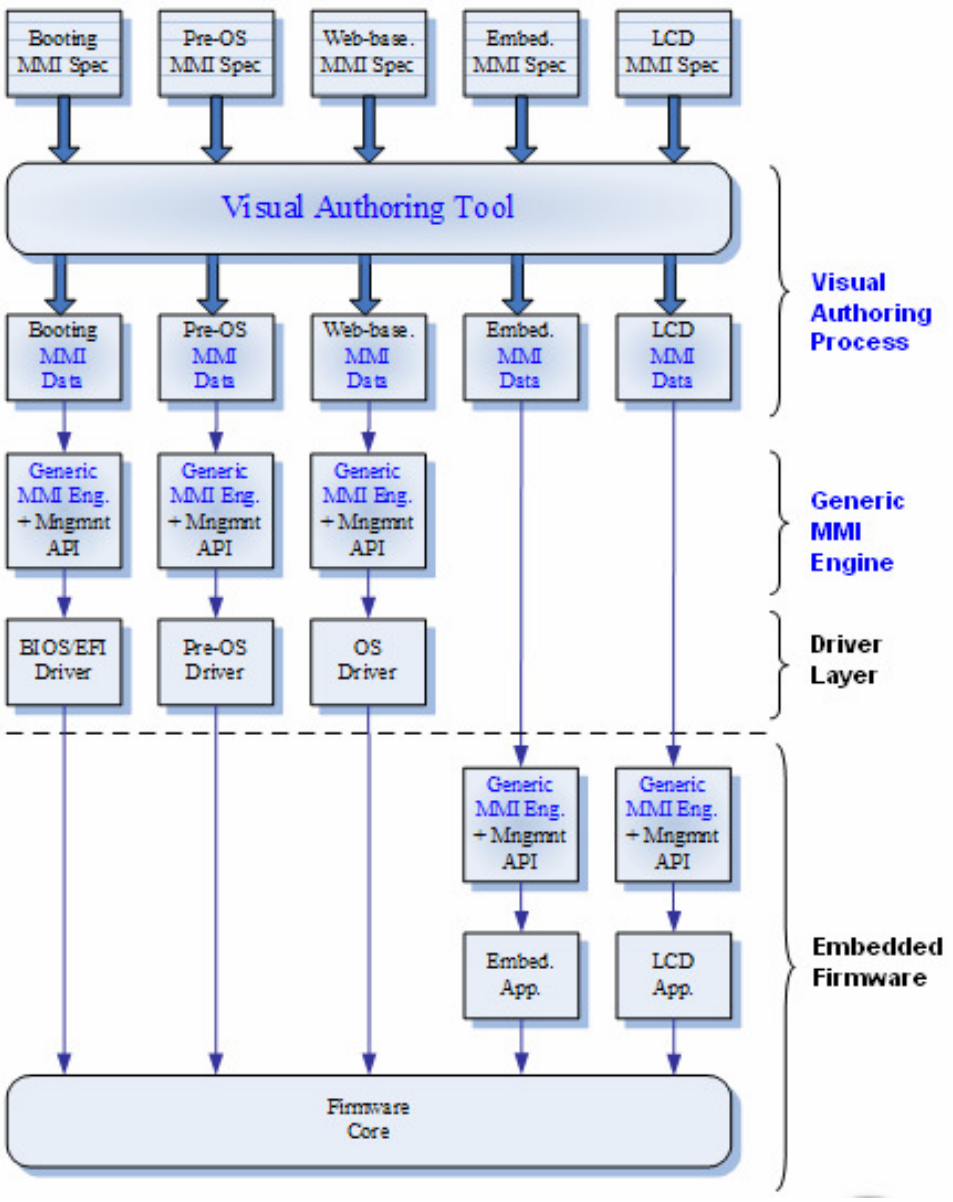


Figure 12. Generic Software Framework for the MMI Generation of Storage Systems

There are also four layers in *Generic Software Framework for the MMI Generation of Storage Systems*: 1) ***Visual Authoring Process***, 2) ***Generic MMI Engine***, 3) ***Driver Layer***, and 4) ***Embedded Firmware***. We introduce the new layers:

1) ***Visual Authoring Process***: MMI designers use *Visual Authoring Tool* to compose each MMI according to the MMI requirement specification for particular utility. The authoring tool is visualized and very user-friendly. After save the results in *SVG files* and use *SVG Parser* to parse these files, we can get the *MMI data* for *Generic MMI Engine* at the end of *Visual Authoring Process*. For instance, MMI designers use *Visual Authoring Tool* to compose the MMIs of *Booting Utility* and *Pre-OS Utility*, and then generate the *Booting MMI Data* and *Pre-OS MMI Data*.

2) ***Generic MMI Engine***: Programmers do not need to write any MMI programs, but they have to program the related functions for MMIs. We design the *Generic MMI Engine* as common code, and it invokes these MMI related functions. This engine also cooperates with the *Management API* of storage systems. The design of *Generic MMI Engine* is OS independent, and we separate the OS dependent functions to *OS Dependent APIs*. This design is helpful to implement utilities in various environments.

The 3) ***Driver Layer*** and 4) ***Embedded Firmware*** are the same as the ones in the conventional software framework for MMIs of storage systems.

After MMI designers finish the *Visual Authoring Process* and programmers write the related functions for MMIs, we can come out the particular utility for storage systems with the *MMI Data*, *Generic MMI Engine*, and *Management API*. This utility still needs *Driver Layer* to pass commands to *Embedded Firmware* of storage systems.

3.3 Implementation of Visual MMI Development for Storage Systems

To implement the *Visual MMI Development for Storage Systems*, we must consider five key components: 1) ***Visual Authoring Tool – Inkscape***, 2) ***SVG Files***, 3) ***SVG Parser***, 4) ***MMI Data***, and 5) ***Generic MMI Engine***.

The first phase of *Visual MMI Development for Storage Systems* is *Visual Authoring Process*. We illustrate this procedure in figure 13, and describe the details of this process in the following sections.

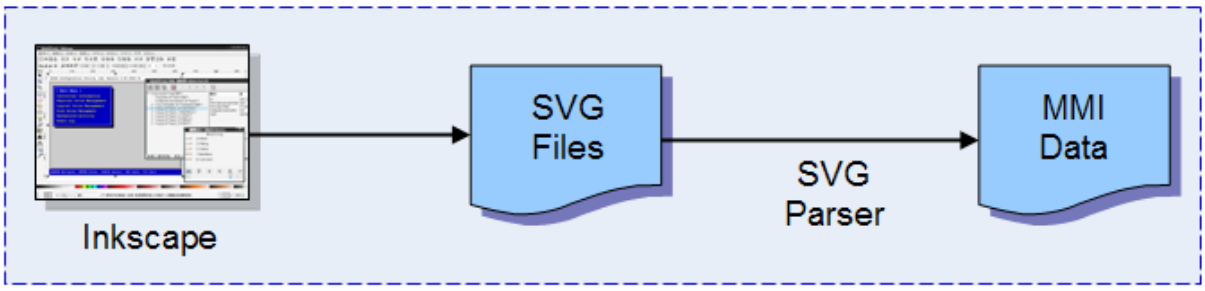


Figure 13. Visual Authoring Process

3.3.1 Visual Authoring Tool -- Inkscape

In our implementation, we choose ***Inkscape*** [12] as *Visual Authoring Tool*. Inkscape is an open source vector graphics editor, with capabilities similar to Illustrator, Freehand, CorelDraw, or Xara X using the W3C [13] standard Scalable Vector Graphics (SVG) file format. Supported SVG features include shapes, paths, text, markers, clones, alpha blending, transforms, gradients, patterns, and grouping. Inkscape also supports Creative Commons meta-data, node editing, layers, complex path operations, bitmap tracing, text-on-path, flowed text, direct XML editing, and more. It imports formats such as JPEG, PNG, TIFF, and others and exports PNG as well as multiple vector-based formats.

The main goal of Inkscape is to create a powerful and convenient drawing tool fully compliant with XML, SVG, and CSS standards. Therefore, it is very suitable for the *Visual Authoring Process of Visual MMI Development for Storage Systems*.

In figure 14, it is a snapshot of the Inkscape usage.

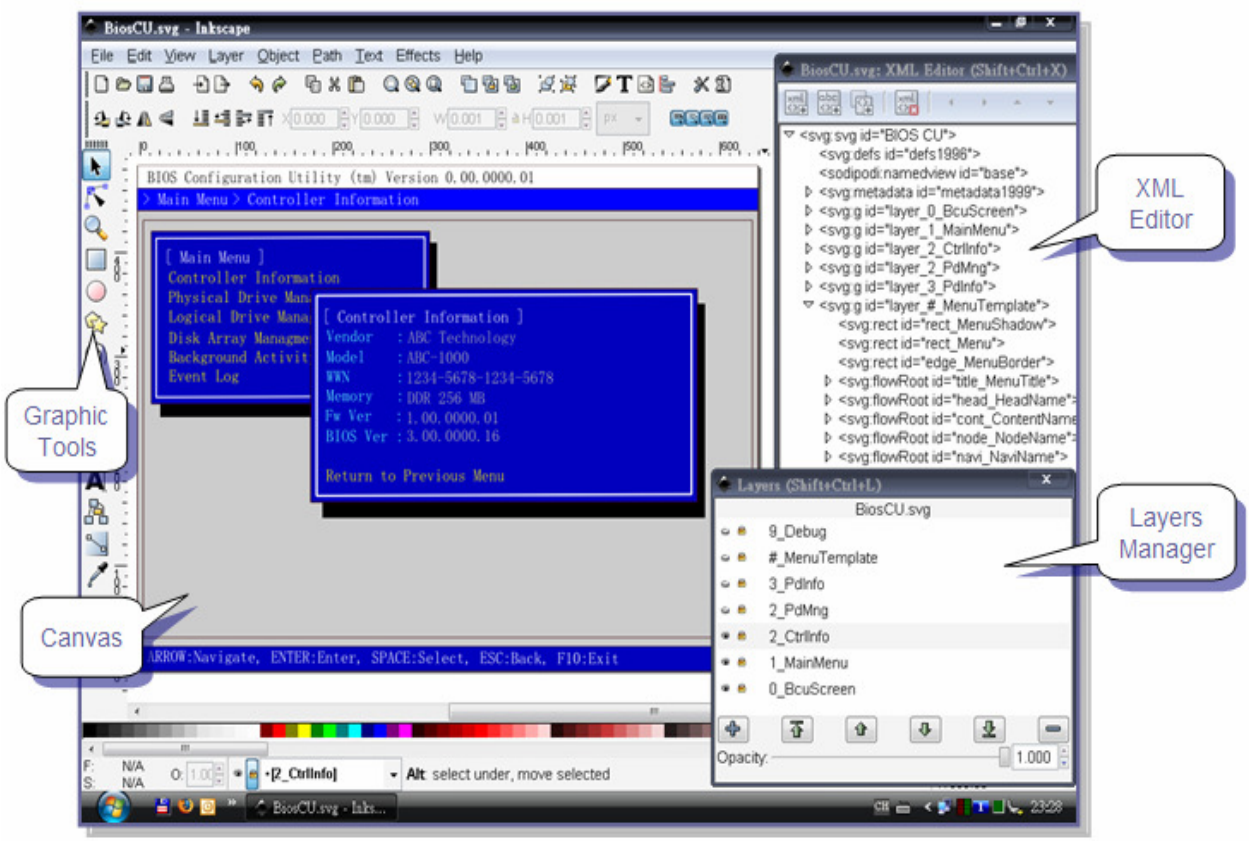


Figure 14. Visual Authoring Tool -- Inkscape

MMI designers can use the ***Graphic Tools***, ***Canvas***, and ***Layers Manager*** of Inkscape to implement MMIs for storage systems. To simplify and formalize the implementation of MMIs for storage systems, we use *Graphic Tools* to create rectangles and texts only, and put them in the *Canvas* area. To compose more complex vector graphics, MMI designers can use the *Layer Manager* to overlap many layers of graphics.

On the other hand, MMI programmers can use the ***XML Editor*** of Inkscape to modify SVG files conveniently. They can modify the name, attributes of SVG elements, and define the relationship between lots of SVG elements, or add the related functions in the functional or other programs into SVG elements.

Figure 15 illustrates one SVG file in plan text and the *XML Editor* of Inkscape. We recommend MMI programmers to use the *XML Editor* of Inkscape to add information in SVG files.

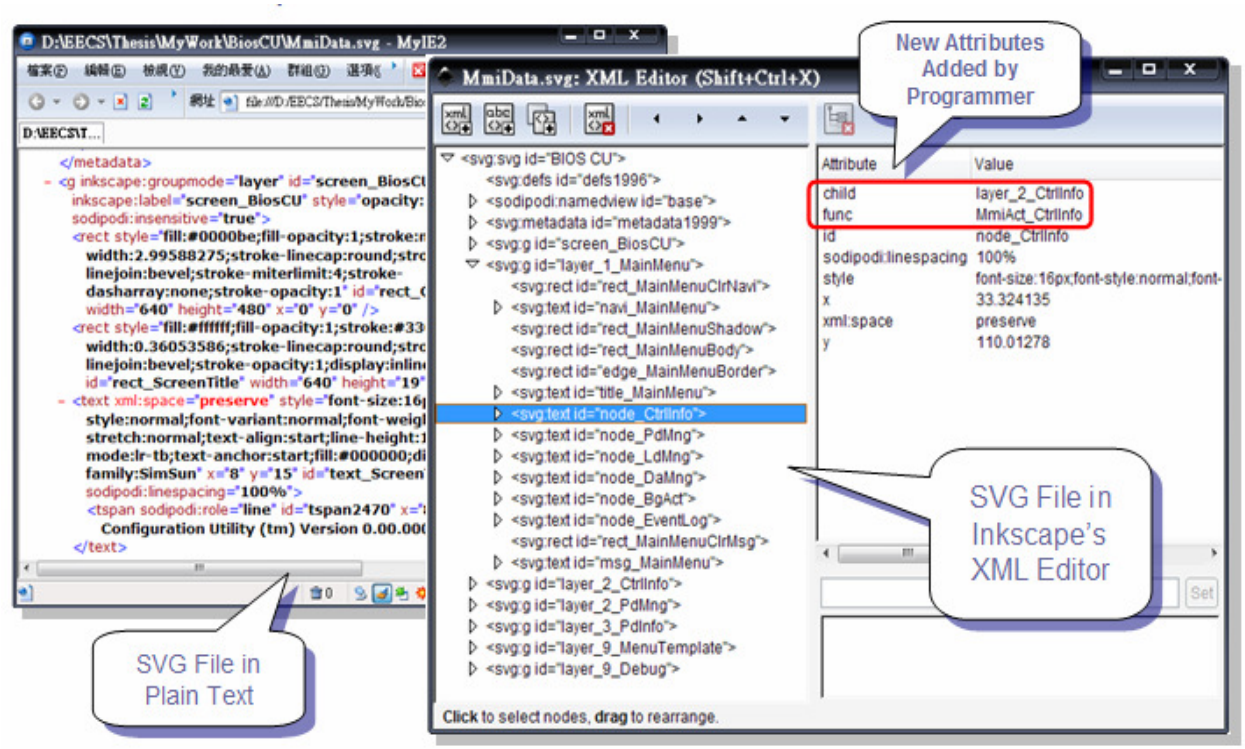


Figure 15. Edit SVG File in the XML Editor of Inkscape

To implement man machine interfaces by Visual Authoring Process, MMI designers or programmers are free to learn the SVG language. MMI designers use Inkscape to compose wanted MMI graphics and save them to the files in SVG format. When programmers want to modify the SVG files, they do not need to handcraft the SVG file in plain text. On the contrary, programmers can use the XML editor of Inkscape to edit SVG file easily and user-friendly.

3.3.2 SVG Files

SVG (Scalable Vector Graphics) [14] is a language for describing two-dimensional graphics and graphical applications in ***XML*** (Extensible Markup Language) [15]. XML is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

SVG is used in many business areas including Web graphics, animation, user interfaces, graphics interchange, print and hardcopy output, mobile applications and high-quality design. It has two parts: an XML-based file format and a programming API for graphical applications. Key features include shapes, text and embedded raster graphics, with many different painting styles. It supports scripting through languages such as ECMAScript and has comprehensive support for animation.

SVG is a royalty-free vendor-neutral open standard developed under the W3C Process. It has strong industry support; Authors of the SVG specification include Adobe, Agfa, Apple, Canon, Corel, Ericsson, HP, IBM, Kodak, Macromedia, Microsoft, Nokia, Sharp and Sun Microsystems. SVG viewers are deployed to over 100 million desktops, and there is a broad range of support in many authoring tools.

SVG builds upon many other successful standards such as XML (SVG graphics are text-based and thus easy to create), JPEG and PNG for image formats, DOM for scripting and interactivity, SMIL for animation and CSS for styling.

SVG is interoperable. The W3C release a test suite and implementation results to ensure conformance. Applications of SVG in industry are graphics platform for mobile phone, embedded system and print industry, web applications, design and interchange, GIS (Geographic Information Systems) and mapping, etc.

We adopt SVG files as the inter-media of *Visual Authoring Process* because SVG has above advantages. For example, we can preview the MMI in SVG files by web browser, and MMI programmers can add more information about MMI programs into SVG files conveniently. Furthermore, we can apply the functionality of XML parser to implement the specific *SVG Parse* very easily for the SVG files of MMI.

To simplify the implementation, we use three SVG elements, ***Layer***, ***Rect*** and ***Text***, to implement the MMI elements for storage systems. Please refer to the relationship between SVG elements and MMI elements in the following table. Using the same SVG element, we define one or more MMI elements with different attributes sot that we can manipulate them very conveniently in *Generic MMI Engine*.

Table 2. Implement MMI Elements by SVG Elements

SVG Elements	MMI Elements
Layer	***Screen, Layer***
Rect	***Rect, Edge***
Text	***Text, Title, Head, Cont, Navi, Msg, Node***

1) Using SVG ***Layer*** element to implement Screen, Layer MMI elements: The Layer Manger of Inkscape can control SVG Layer element to display or hide, free-to-modify or lock-up. We define ***Screen*** element to compose the base screen layout, and ***Layer*** element to place one menu and its related components, e.g. some items in this menu, or help messages for it.

2) Using SVG ***Rect*** element to implement Rect, Edige MMI elements: The rectangle in SVG graphics can be solid or hollow rectangles. We use solid rectangle to define ***Rect*** MMI element that is used for menu's body and shadow, and hollow rectangle to define ***Edge*** MMI element that is the edge or border of menu's body.

3) Using SVG ***Text*** element to implement Text, Title, Head, Cont, Navi, Msg, Node MMI elements: The SVG Text element has many attributes, like font size or color. We use the same font size for all text MMI elements. Nevertheless, we define many text MMI elements with different colors. They are used for different purposes: The ***Text*** MMI element is a common string and can be used in base screen layer. The ***Title*** MMI element is used for the title of base screen or menus. The ***Head*** MMI element is used for the head of items in menus. The ***Cont*** MMI element is used for the content of items in menus. The ***Navi*** MMI element is used for the string on Navigation Bar. The Navigation Bar is the row between screen title and canvas and it can show the path of current menus. The ***Msg*** MMI element is used for the string on Message Bar. The Message Bar is the lowest row in screen and it can display the message for current menu. The latest and important MMI element, ***Node***, can carry the information to child menu and the related function to take actions, but MMI programmers must add the child menu and related function for each Node manually.

3.3.3 SVG Parser

After using Inkscape to create SVG files, we need a parser to parse SVG structure and obtain required information from SVG files. We call this parser as ***SVG Parser***. Before describe the SVG parser, we introduce the concept of XML ***DOM*** (Document Object Model) first.

In the right part of figure 16, it is a simple XML in plain text. This XML describes a bookstore with a couple of books.

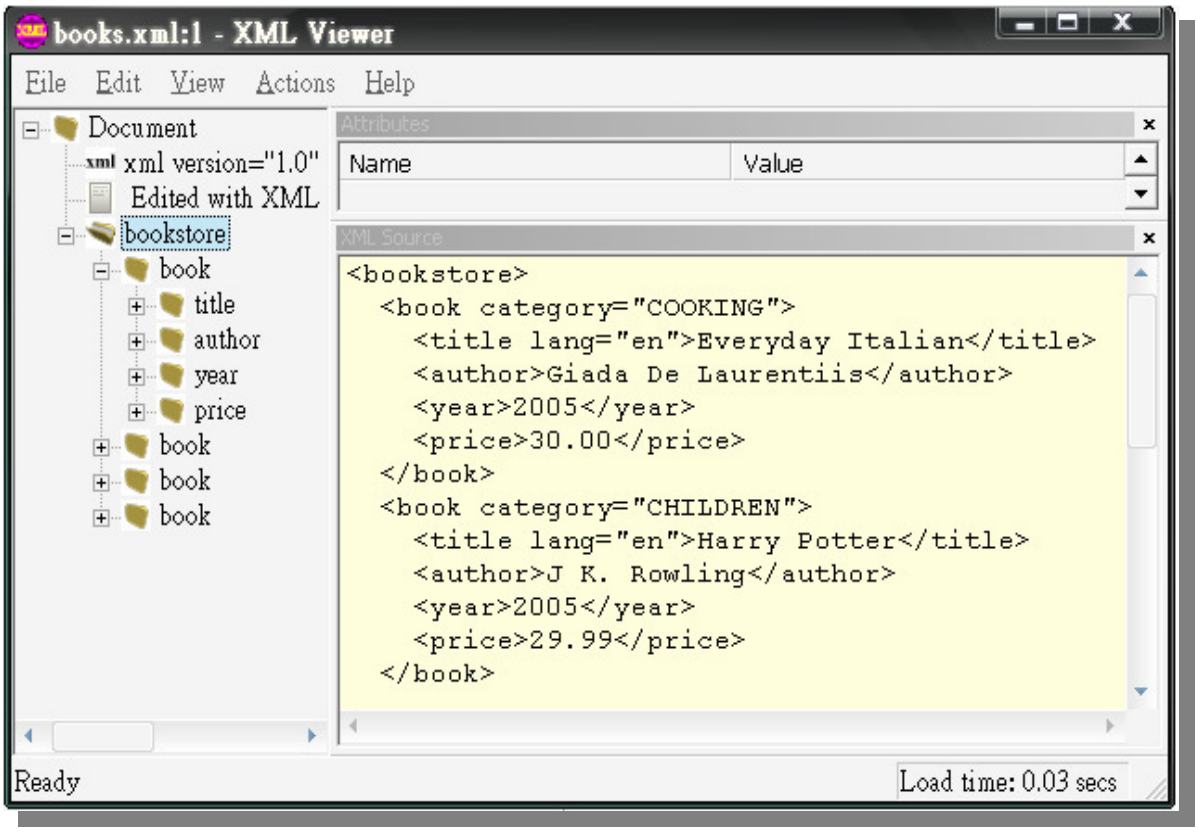


Figure 16. XML in Plain Text

Each book has four elements: title, author, year and price. For example, first book's title is "Everyday Italian" in English language, because its attribute is lang="en", author is "Giada De Laurentiis", year is 2005, price is 30.00. In the left part of figure 16, there is a tree to describe the hierarchy of that XML. It is very easy to understand compared with the plain text format.

Figure 17 illustrates a using example of XML ***DOM*** (Document Object Model) for the above XML of bookstore.

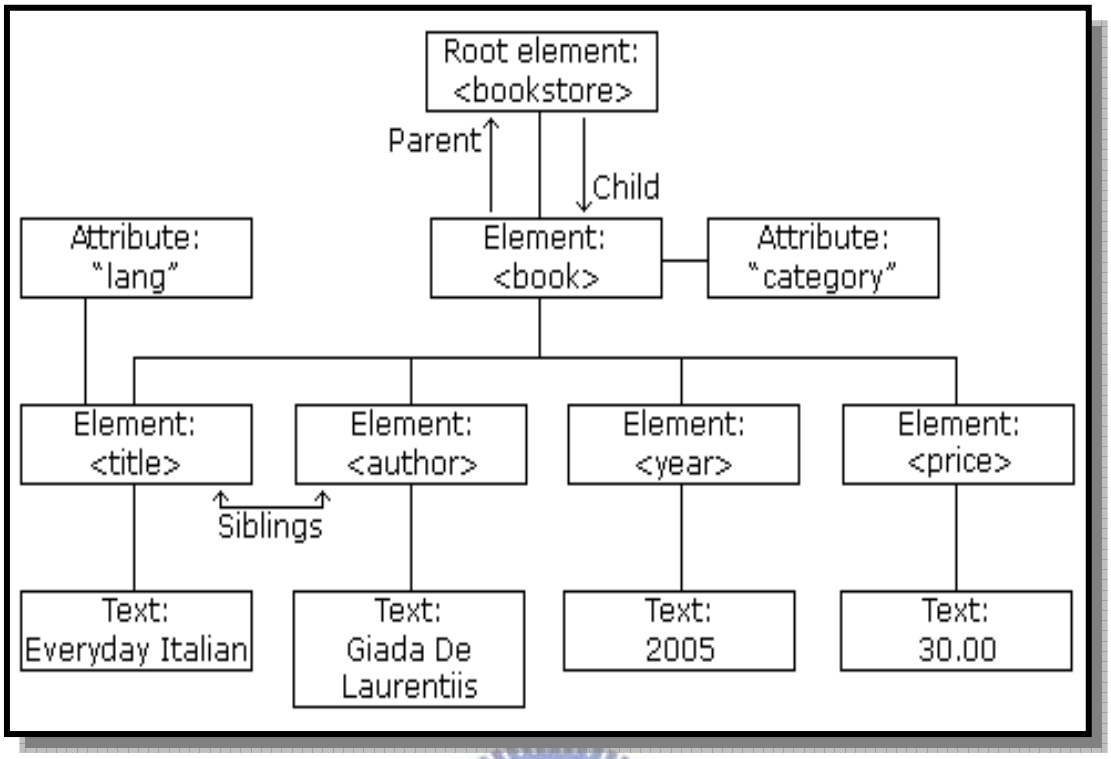


Figure 17. XML DOM

The *Document Object Model* is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page..

Using XML DOM, a *XML Parser* can traverse the node tree, access the nodes and their attribute values, insert and delete nodes, and convert the node tree back to XML. In our implementation, we choose ***Expat XML Parser*** [19].

Expat is an XML parser library written in C. It is a *stream-oriented* parser in which an application registers handlers for things the parser might find in the XML document (like start tags).

In figure 18, it is a function reference for *Expat XML Parser*. Expat provides many functions for parsing XML.

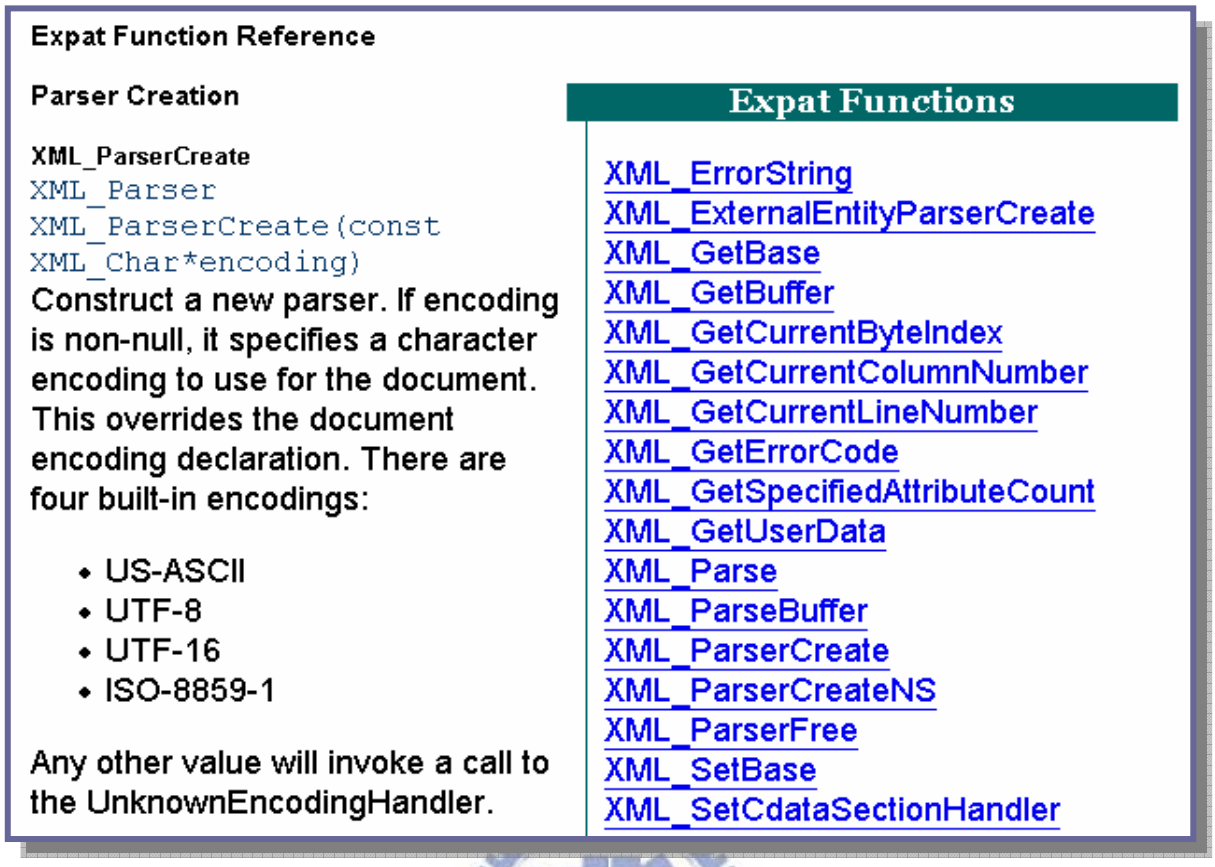


Figure 18. Expat XML Parser

Actually, we implement our ***SVG Parser*** and just invoke the following functions of *Expat XML Parser*. They are summarized in Table 3.

Table 3. SVG Parser Uses Functions of Expat XML Parser

Function Name	Description
XML_ParserCreate( )	Construct a new parser
XML_SetElementHandler( )	Set handlers for start and end tags
XML_SetCharacterDataHandler( )	Set a text handler
XML_Parse( )	Parse some more of the document in a buffer

In the main function of *SVG Parser*, we invoke ***XML_ParserCreate( )*** to construct a new XML parser, and invoke ***XML_SetElementHandler( )*** to set handlers for SVG elements and ***XML_SetCharacterDataHandler( )*** for element's text processing. In the body of main functions, we allocate a buffer, use a loop to read a part of SVG file into this buffer, and invoke ***XML_Parser( )*** to parse data repeatedly until the whole SVG file is processed.

3.3.4 MMI Data

The output of *Visual Authoring Tool* is SVG files. We use *SVG Parser* to get out the information from SVG files and store as ***MMI Data*** format. To simplify the usage of *MMI Data*, we implement the format of *MMI Data* in a C header file, and correspondent macros in the code of *Generic MMI Engine*.

To implement MMI elements, we define the following macros:

1) For ***Screen*** and ***Layer*** MMI elements, we define *CREATE_SCREEN( )*, *END_OF_SCREEN( )*, *CREATE_LAYER( )*, *END_OF_LAYER( )* macros. Because *Screen* and *Layer* MMI elements can contain other MMI elements, we use *CREATE_XXX*( ) and *END_OF_XXX( )* macros to clip them.

2) For ***Rect*** and ***Edge*** MMI elements, we define *CREATE_RECT( )* and *CREATE_EDGE( )* macros. These two macros need the attributes: name, X coordinate, Y coordinate, width, height, color of solid or hollow rectangles.

3) For ***Text-like*** MMI elements, we define *CREATE_TEXT( )*, *CREATE_TITLE( )*, *CREATE_HEAD( ), CREATE_CONT( ), CREATE_NAVI( ), CREATE_MSG( ), CREATE_NODE( )* macros. For ***Text, Title, Head, Cont, Navi, Msg*** MMI elements, they need the attributes: name, X coordinate, Y coordinate, color and string. Besides the previous ones, ***Node*** MMI element needs two additional attributes: child menu, related function. The attribute, child menu, tells Generic MMI Engine which menu layer is combined to this node. And the attribute, related function, invokes the functions in *Generic MMI Engine* to take the related actions for this node. The scenario is when users press this node, the related function will be invoked to do something and then the child menu will be popped up.

Figure 19 demonstrates a partial *MMI Data* in the output file of *SVG Parser* -- MmiData.h.

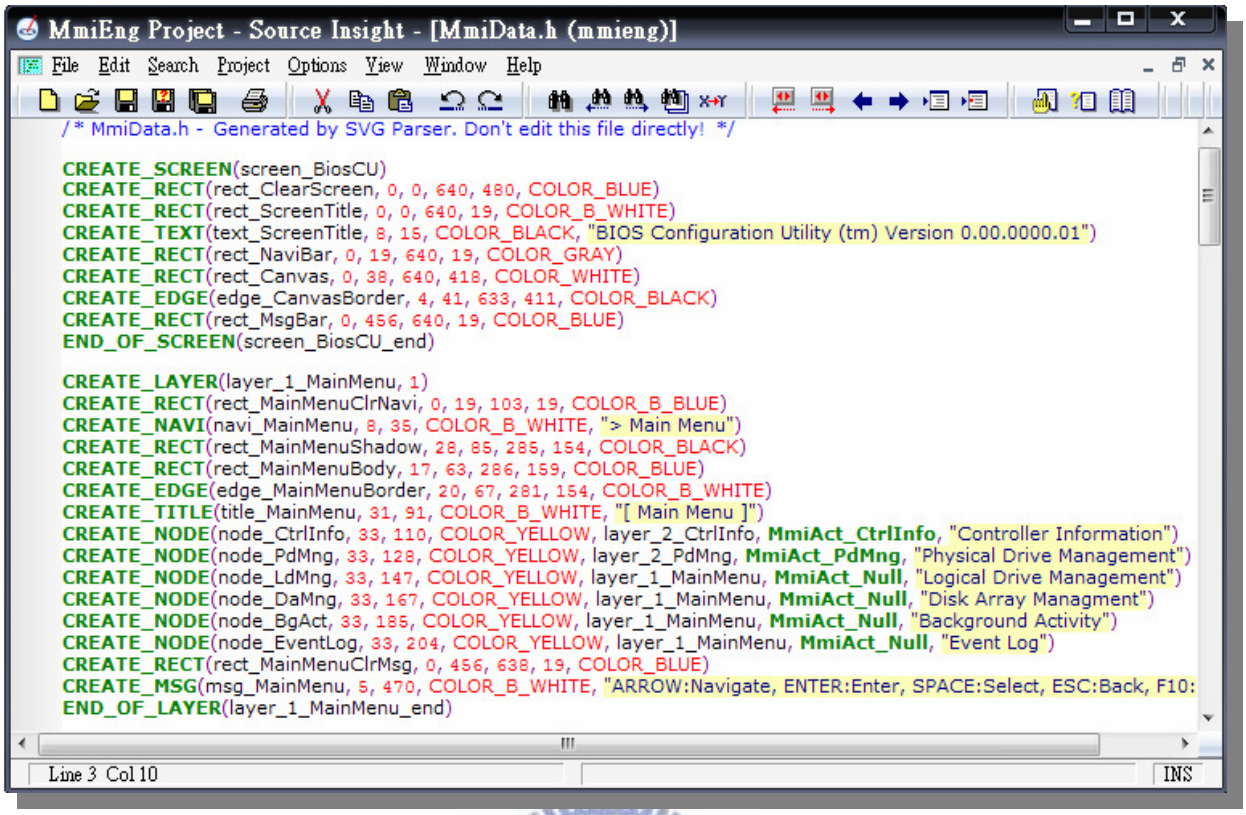


Figure 19. MmiData.h – The Output File of SVG Parser

In this example, we know that the screen ***screen_BiosCU*** contains MMI elements: *rect_ClearScreen, rect_ScreenTitle, text_ScreenTitle, rect_NaviBar, rect_Canvas, edge_CanvasBorder, rect_MsgBar*. The ***rect_ClearScreen*** is the first rectangle to clear the whole screen. The ***rect_ScreenTitle*** is the background of screen title, and the ***text_ScreenTitle*** is the string of screen title. The ***rect_NaviBar*** is the background of Navigation Bar. The ***rect_Canvas*** and edge_CanvasBorder form the area of canvas to place menus. The ***rect_MsgBar*** is the background of Message Bar.

About the display order of SVG elements, *Inkscape* draws the leading element first, and then draws the following elements in sequence. Therefore, the following elements overlap on the previous ones. In our implementation of *Generic MMI Engine*, we also use the same display order for MMI elements. Hence, the MMI element *text_ScreenTitle* is displayed over the *rect_ScreenTitle.*

On the other hand, the display order of graphic layers, Inkscape draws the lowest layer first, and then draws the upper layers one bye one. So, the upper layers overlap on the lowest

one. *Generic MMI Engine* also displays layers for MMI elements in the same order. Thus, the *layer_1_MainMenu* is displayed over the *scree_BiosCU*.

In the bottom half of figure 19, the menu layer ***layer_1_MainMenu*** contains MMI elements: *rect_MainMenuClrNavi, navi_MainMenu, rect_MainMenuShadow, rect_MainMenuBody, edge_MainMenuBorder, title_MainMenu, node_CtrlInfo, node_PdMng, node_LdMng, node_DaMng, node_BgAct, node_EventLog, rect_MainMenuClrMsg, msg_MainMenu*. The ***rect_MainMenuClrNavi*** is the background of navi_MainMenu. The ***navi_MainMenu*** is the string on Navi Bar for main menu. The ***rect_MainMenuShadow*** is the shadow of main menu's body. The ***rect_MainMenuBody*** is the body of main menu. The ***edge_MainMenuBorder*** is the border of main menu's body. The ***title_MainMenu*** is the title of main menu. The ***node_CtrlInfo*** is the first node of main menu, and its child menu is *layer_2_CtrlInfo* and related function is *MmiAct_CtrlInfo( )*. The ***node_PdMng*** is the second node of main menu, and its child menu is *layer_2_PdMng* and related function is *MmiAct_PdMng( )*. The ***node_LdMng, node_DaMng, node_BgAct, and node_EventLog*** are not functional nodes, so their child menu is layer_1_MainMenu and related function is *MmiAct_Null( )*. The ***rect_MainMenuClrMsg*** is the background of *msg_MainMenu*. The ***msg_MainMenu*** is the string on Msg Bar for main menu.

For the *MMI Data* in the format of MmiData.h, we write the macros for each MMI elements. The details of these macros are revealed in the following description:

1) For ***Screen*** MMI element, we use *CREATE_SCREEN( )* and *END_OF_SCREEN ( )* macros. The *Screen* MMI element can contain *Rect*, *Edge*, and *Text* MMI elements. Because of this, we can see *CREATE_RECT( )*, *CREATE_EDGE( )*, and *CREATE_TEXT( )* macros between *CREATE_SCREEN( )* and *END_OF_SCREEN( )* macros.

```
#define CREATE_SCREEN(name)  \
    mmi_screen_t name =  \
    {   \
        mmi_elem_screen, \
    };  \


#define END_OF_SCREEN(name)  \
    U16  name = mmi_elem_screen_end;
```

2) For ***Layer*** MMI element, we use *CREATE_LAYER( )* and *END_OF_LAYER ( )* macros. The *Layer* MMI element can contain *Rect, Edge, Text, Title, Head, Cont, Navi, Msg* and *Node* MMI elements. Because of this, we can see *CREATE_RECT( )*, *CREATE_EDGE( )*, *CREATE_TEXT, CREATE_TITLE, CREATE_HEAD, CREATE_CONT, CREATE_NAVI, CREATE_MSG* and *CREATE_NODE( )* macros between *CREATE_LAYER( )* and *END_OF_LAYER( )* macros.

```
#define CREATE_LAYER(name, level)  \
    mmi_layer_t name =  \
    {   \
        mmi_elem_layer, \
        level,          \
    };  \


#define END_OF_LAYER(name)  \
    U16  name = mmi_elem_layer_end;
```

3) For ***Rect*** and ***Edge*** MMI elements, we use *CREATE_RECT( )* and *CREATE_EDGE ( )* macros. They need the same parameters: name of MMI element, coordinate X, coordinate Y, width, height, and color.

```
#define CREATE_RECT(name, coordX, coordY, width, height, color)  \
    mmi_rect_t name =   \
    {   \
        mmi_elem_rect,  \
        coordX,         \
        coordY,         \
        width,          \
        height,         \
        color,          \
    };  \


#define CREATE_EDGE(name, coordX, coordY, width, height, color)  \
    mmi_edge_t name =   \
    {   \
        mmi_elem_edge,  \
        coordX,         \
        coordY,         \
        width,          \
        height,         \
        color,          \
    };  \
```

3) For ***Text-like*** MMI elements, we use *CREATE_TEXT( ), CREATE_TITLE( ), CREATE_HEAD( ), CREATE_CONT( ), CREATE_NAVI( ),* and *CREATE_MSG( )* macros. They need the same parameters: name of MMI element, coordinate X, coordinate Y, color, and string.

```
#define CREATE_TEXT(name, coordX, coordY, color, str)  \
    mmi_text_t name =   \
    {   \
        mmi_elem_text,  \
        coordX,         \
        coordY,         \
        color,          \
        str,            \
    };  \


#define CREATE_TITLE(name, coordX, coordY, color, str)  \
    mmi_title_t name =  \
    {   \
        mmi_elem_title, \
        coordX,         \
        coordY,         \
        color,          \
        str,            \
    };  \


#define CREATE_HEAD(name, coordX, coordY, color, str)  \
    mmi_head_t name =   \
    {   \
        mmi_elem_head,  \
        coordX,         \
        coordY,         \
        color,          \
        str,            \
    };  \


#define CREATE_CONT(name, coordX, coordY, color, str)  \
    mmi_cont_t name =   \
    {   \
        mmi_elem_cont,  \
        coordX,         \
        coordY,         \
        color,          \
        str,            \
    };  \


#define CREATE_NAVI(name, coordX, coordY, color, str)  \
    mmi_navi_t name =   \
    {   \
        mmi_elem_navi,  \
        coordX,         \
        coordY,         \
        color,          \
        str,            \
    };  \


#define CREATE_MSG(name, coordX, coordY, color, str)  \
    mmi_msg_t name =    \
    {   \
        mmi_elem_msg,   \
        coordX,         \
        coordY,         \
```

```
        color,          \
        str,            \
    };  \
```

4) For ***Node*** MMI elements, we use *CREATE_NODE( )* macro. This macro needs the parameters: name of MMI element, coordinate X, coordinate Y, color, child layer, related function name, and string.

```
#define CREATE_NODE(name, coordX, coordY, color, child, func, str)  \
    mmi_layer_t child;  \
    mmi_node_t name =   \
    {   \
        mmi_elem_node,  \
        coordX,         \
        coordY,         \
        color,          \
        &child,         \
        func,           \
        str,            \
    };  \
```

To distinguish MMI elements more easily, we implement macros and data structures for each MMI elements. These macros put the attributes of MMI elements into the correspondent data structures so that *Generic MMI Engine* can manipulate *MMI data* very easily.

We define constants for MMI elements, and these constants are used in previous macros. The enumerated type collects these constants as below:

```
typedef enum mmi_elem_e
{
    mmi_elem_start = 0xE0,

    mmi_elem_screen,        /* main screen */
    mmi_elem_screen_end,

    mmi_elem_layer,         /* layer with following elements */
    mmi_elem_layer_end,

    mmi_elem_rect,          /* solid rectangle */
    mmi_elem_edge,          /* hollow rectangle */

    mmi_elem_text,          /* text on screen */
    mmi_elem_title,         /* text of menu title */
    mmi_elem_head,          /* head of item */
    mmi_elem_cont,          /* content of item */
    mmi_elem_navi,          /* text on navi bar */
    mmi_elem_msg,           /* text on msg bar */

    mmi_elem_node,          /* node to other layer */
```

```
    mmi_elem_end
} mmi_elem_t;
```

We also define the data structures for MMI elements. They are:

1) For ***Screen*** MMI element, we use the data structure ***mmi_screen_s***.

```
typedef struct mmi_screen_s
{
    U16  mmi_elem_type;
} mmi_screen_t;
```

2) For ***Layer*** MMI element, we use the data structure ***mmi_layer_s***. It has the *level* field to indicate the level of this layer.

```
typedef struct mmi_layer_s
{
    U16  mmi_elem_type;
    U16  level;
} mmi_layer_t;
```

3) For ***Rect*** and ***Edge*** MMI elements, we use the same data structure ***mmi_rect_s***. For the *mmi_elem_type* field, *CREATE_RECT( )* macro places the value of *mmi_elem_rect* for *Rect* MMI element, and *CREATE_EDGE( )* macro places the value of *mmi_elem_edge* for *Edge* MMI element.

```
typedef struct mmi_rect_s
{
    U16  mmi_elem_type;
    S16 coordX;
    S16 coordY;
    S16 width;
    S16 height;
    S16 color;
} mmi_rect_t;


typedef mmi_rect_t mmi_edge_t;
```

4) For ***Text-like*** MMI elements, we use the same data structure ***mmi_text_s***. For the *mmi_elem_type* field, *CREATE_TEXT( )* macro places the value of *mmi_elem_text* for *Text* MMI element, and *CREATE_TITLE( )* macro places the value of *mmi_elem_title* for *Title* MMI element, etc.

```
typedef struct mmi_text_s
{
    U16 mmi_elem_type;
    S16 coordX;
    S16 coordY;
    S16 color;
    pU8 str;
} mmi_text_t;

typedef mmi_text_t mmi_title_t;
typedef mmi_text_t mmi_head_t;
typedef mmi_text_t mmi_cont_t;
typedef mmi_text_t mmi_navi_t;
typedef mmi_text_t mmi_msg_t;
```

5) For ***Node*** MMI element, we use the data structure ***mmi_node_s***. *CREATE_NODE( )* macro places the address of child layer into *child* field, and the related function address in *func* field for *Node* MMI element.

```
typedef struct mmi_node_s
{
    U16  mmi_elem_type;
    S16 coordX;
    S16 coordY;
    S16 color;
    pU8 child;
    MmiAct_func_t func;
    pU8 str;
} mmi_node_t;
```

In summary, the ***Rect*** and ***Edge*** MMI elements use the same data structure *mmi_rect_s*. The ***Text, Title, Head, Cont, Navi, Msg*** MMI elements use the same data structure *mmi_text_s*. The data structure for ***Node*** MMI element is *mmi_node_s*, and it has two additional fields, i.e. *child* and *func*.

3.3.5 Generic MMI Engine

The kernel component of the *Visual MMI Development for Storage Systems* is ***Generic MMI Engine***. This engine can manipulate the data from *Visual Authoring Process*. Figure 20 illustrates the block diagram of *Generic MMI Engine*.

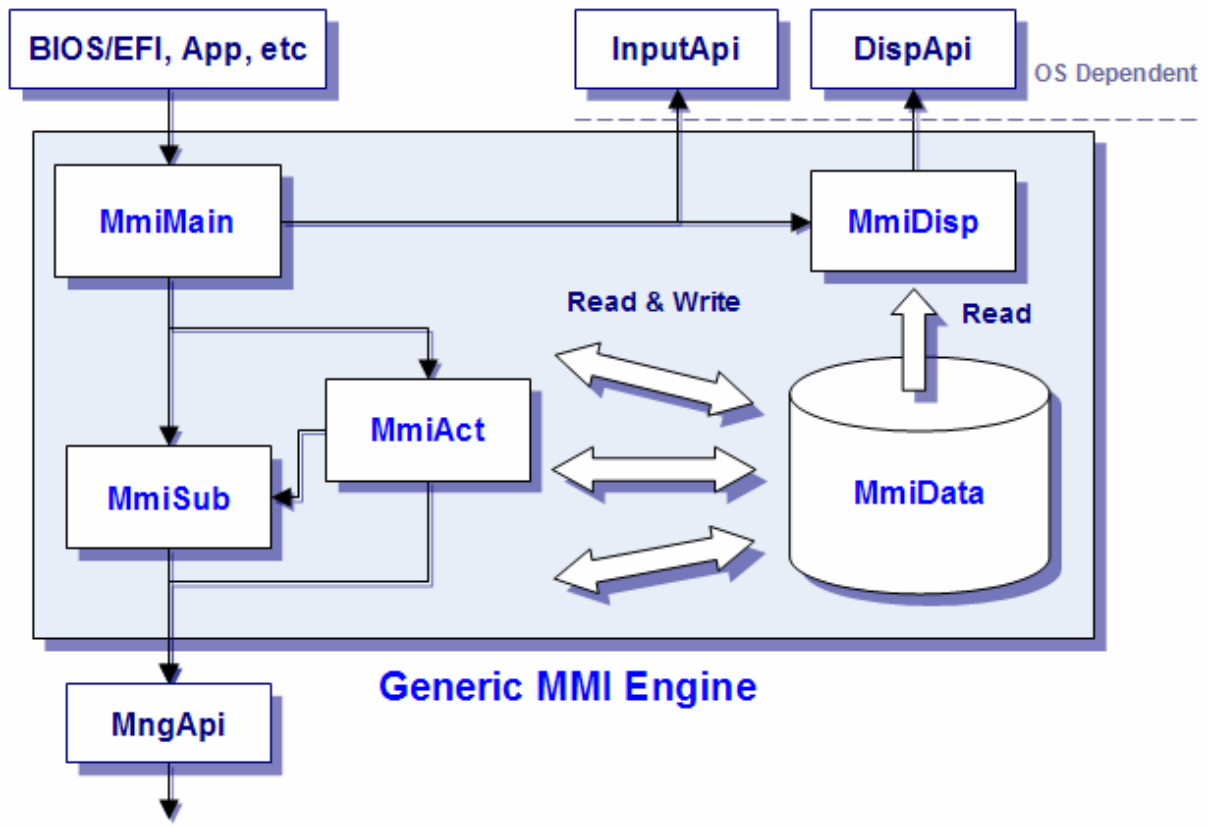


Figure 20. The Block Diagram of Generic MMI Engine

We design and implement five modules of *Generic MMI Engine*. They take different responsibilities. Table 4 summarizes these modules, and we describe the details design of these different modules in the following paragraphs.

Table 4. Modules of Generic MMI Engine

Module Name	Description
MmiMain	Main functions of MMI engine
MmiData	MMI data from visual authoring
MmiDisp	MMI display functions
MmiAct	Interface between MmiMain and MmiData
MmiSub	Interface between MmiMain and MngApi

1) ***MmiMain Module***: This module contains main functions of Generic MMI Engine. *MmiMain_Entry( )* is the entry function of engine and it invokes three sub-functions: *mmiMain_Initialization( ), mmiMain_NormalMode( ), and mmiMain_Finalization( )*. All

initialization will be done in the ***mmiMain_Initialization( )***. In this function, we init the data pointer to MMI Data, set active layer and node, and initialize MmiDisp and MmiSub modules. The ***mmiMain_NormalMode( )*** function is the core of Generic MMI Engine. It invoke functions of MmiDisp module to display screen and menus, and functions of InputApi to get input from users repeatedly. After get any input, it dispatch tasks to take correspondent actions. In the ***mmiMain_Finalization( )*** function, we finalize something before leaving, e.g. finalize MmiSub module and restore the display mode, etc.

2) ***MmiData*** Module: The MmiData.h is the output of SVG Parser. During compilation phase, we use SVG Parser to read the wanted SVG file and elicit MmiData.h automatically. In Generic MMI Engine, it reads and writes the data structures in MmiData module by MmiAct and MmiSub modules, but read only by MmiDisp module.

3) ***MmiDisp Module***: This module contains functions for display. For instance, the ***MmiDisp_DrawScreen( )*** draws the Screen MMI element, and the ***MmiDisp_DrawLayer( )*** function draws the Layer MMI element. These functions of MmiDisp module need low-level functions support, i.e. they invoke the functions of Display API (DispApi). We separate the display functions with OS dependence to Display API.

4) ***MmiAct Module***: This module contains functions for the ***Node*** MMI elements. MMI programmers must add the related functions into MmiAct module, and edit the SVG file to add the related function name into Node MMI elements, e.g. the ***MmiAct_CtrlInfo( )*** function for *node_CtrlInfo*, and the ***MmiAct_PdMng( )*** function for *node_PdMng*, etc.

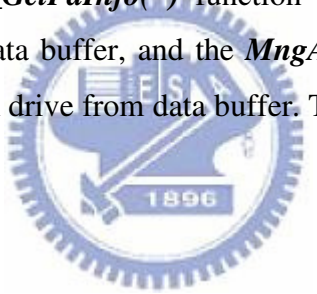
5) ***MmiSub Module***: Functions in this module are sub-functions for MmiMain and MmiAct modules. These functions cooperate with Management API (MngApi) to get information from Embedded Firmware. For example, the ***MmiSub_PdEnum( )*** function gets the ID of physical hard disk drives and enumerate all of them in the storage system. The ***MmiSub_FillPdInfo( )*** function gets the information of physical hard disk drives from Management API (MngApi) and fill the strings for MmiAct module.

Besides above modules, there are some API libraries will cooperate with Generic MMI Engine. They are:

1) ***Input API (InputApi)***: Input API provides functions to get input from users, e.g. what key is pressed, or the coordinates of mouse cursor, etc. This API is OS dependent. For the Booting Utility of storage systems, we write assembly sub-routines or invoke software interrupts of system BIOS.

2) ***Display API (DispApi)***: This API provides functions to display MMI elements. Using more powerful Display API, we can implement more functional Generic MMI Engine to display complex SVG graphics. For instance, the ***DispApi_GetDisplayMode( )*** function gets current display mode, and the ***DispApi_CursorDisable( )*** function can hide cursor. Of course, this API is also OS dependent.

3) ***Management API (MngApi)***: The Management API is the library has capability to communicate with the Embedded Firmware of storage systems. It allocates a private data buffer to get information from firmware, and provides related functions to fetch items in that buffer. E.g. the ***MngApi_GetPdInfo( )*** function gets the information of assigned physical hard disk drive into data buffer, and the ***MngApi_FetchPdId( )*** function fetches the ID of that physical hard disk drive from data buffer. The Management API is dependent on storage systems.



Chapter 4

Simulation and Application Examples

In chapter 3, we describe the design and implementation of *Visual MMI Development for Storage Systems*. In this chapter, we will demonstrate simulation examples for the Booting Utility and application examples for the Pre-OS Utility of storage systems. We will follow the framework of *Visual MMI Development for Storage Systems*, and implement these examples step by step.

4.1 Simulation Examples for the Booting Utility of Storage Systems

Before implement the simulation and application examples, we define the ***MMI requirement specification*** for them in Figure 21. We simply define four menus to demonstrate the basic functions for MMIs of storage systems.

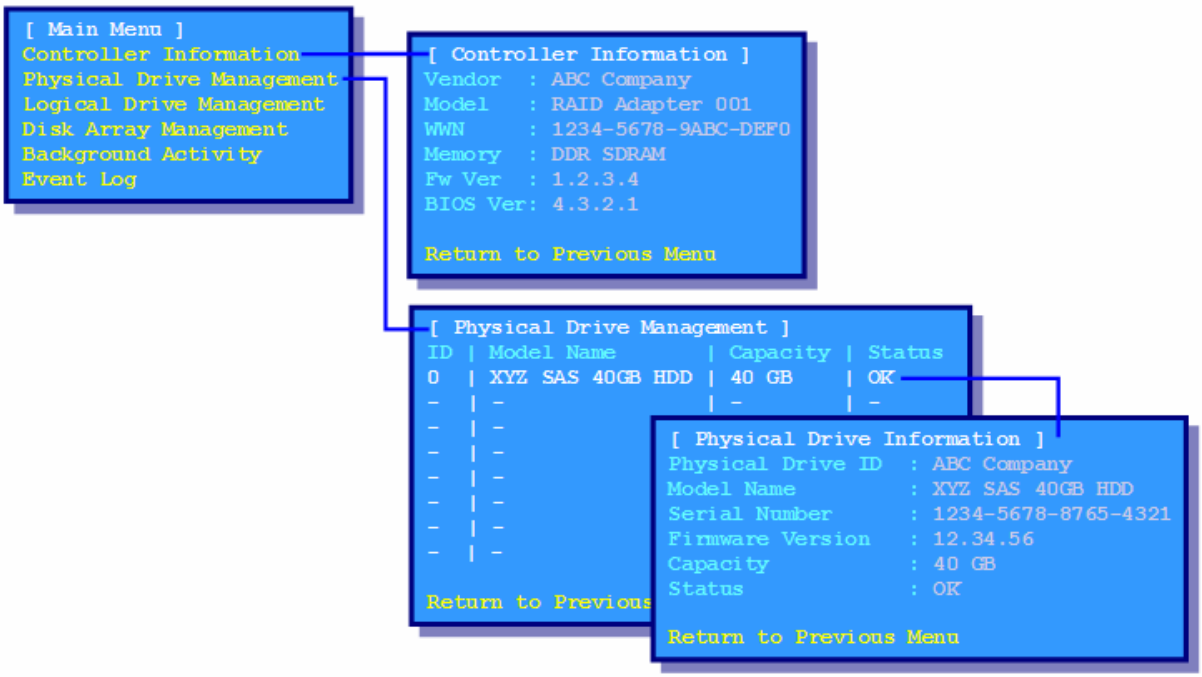


Figure 21. MMI Requirement Specification for Simulation and Application Examples

The ***usage scenario*** of this system is as below description. The first menu is main menu that contains some nodes, but only two nodes are functional. They are the node "*Controller Information*" and "*Physical Drive Management*". When users press the node "*Controller Information*", it will pop up the menu "*Controller Information*" that displays the information of adapter controller. When users press the node "*Physical Drive Management*", it will pop up the menu "*Physical Drive Management*" that has a table lists present physical hard disk drives. When users want to read the information of certain physical hard disk drive, they can press it and the menu "*Physical Drive Information*" will pop up.

To implement this simulation example for the Booting Utility of storage systems, we follow the steps in section 3.1 Visual MMI Development for Storage Systems.

Step 1) MMI designers use Visual Authoring Tool to compose wanted MMIs according to MMI requirement documents. We compose wanted MMIs in *Visual Authoring Tool -- Inkscape*. In figure 22, we compose the screen layout first, because it is the root layer.

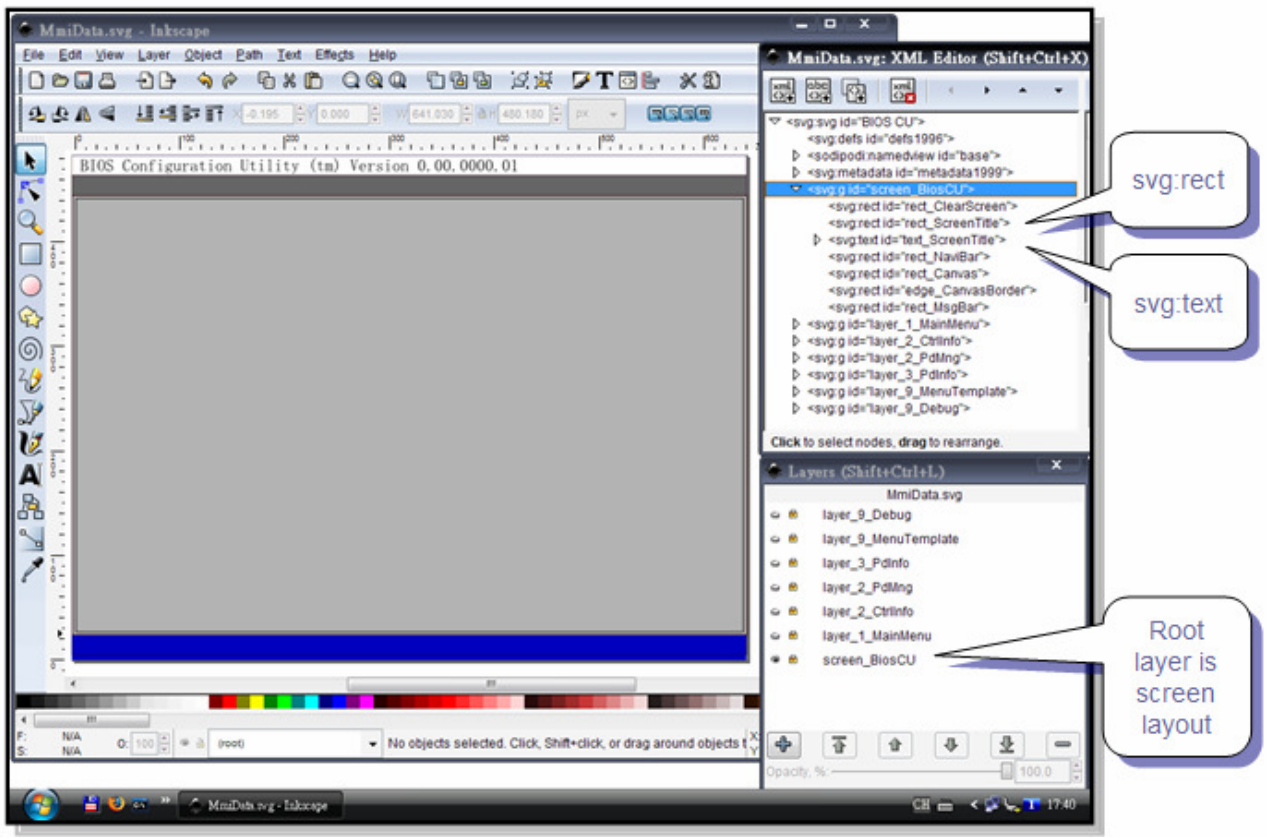


Figure 22. Compose Screen Layout by Inkscape

The first row of the screen layout is the screen title, and the second row in dark gray background color is Navigation Bar (*NaviBar*). In this bar, we can display the hierarchy of active menus, e.g. "*Main Menu* > *Physical Drive Management* > *Physical Drive Information*". The bottom row of screen is Message Bar (*MsgBar*). In this bar, we can display messages, e.g. the brief hint of hot keys. The rows between *NaviBar* and *MsgBar* are the canvas that we can place menus.

Step 2) Save the results of Visula Authoring Tool as SVG Files. In default setting, Inkscape saves files in SVG format. Figure 23 illustrates a partial SVG file in plain text.

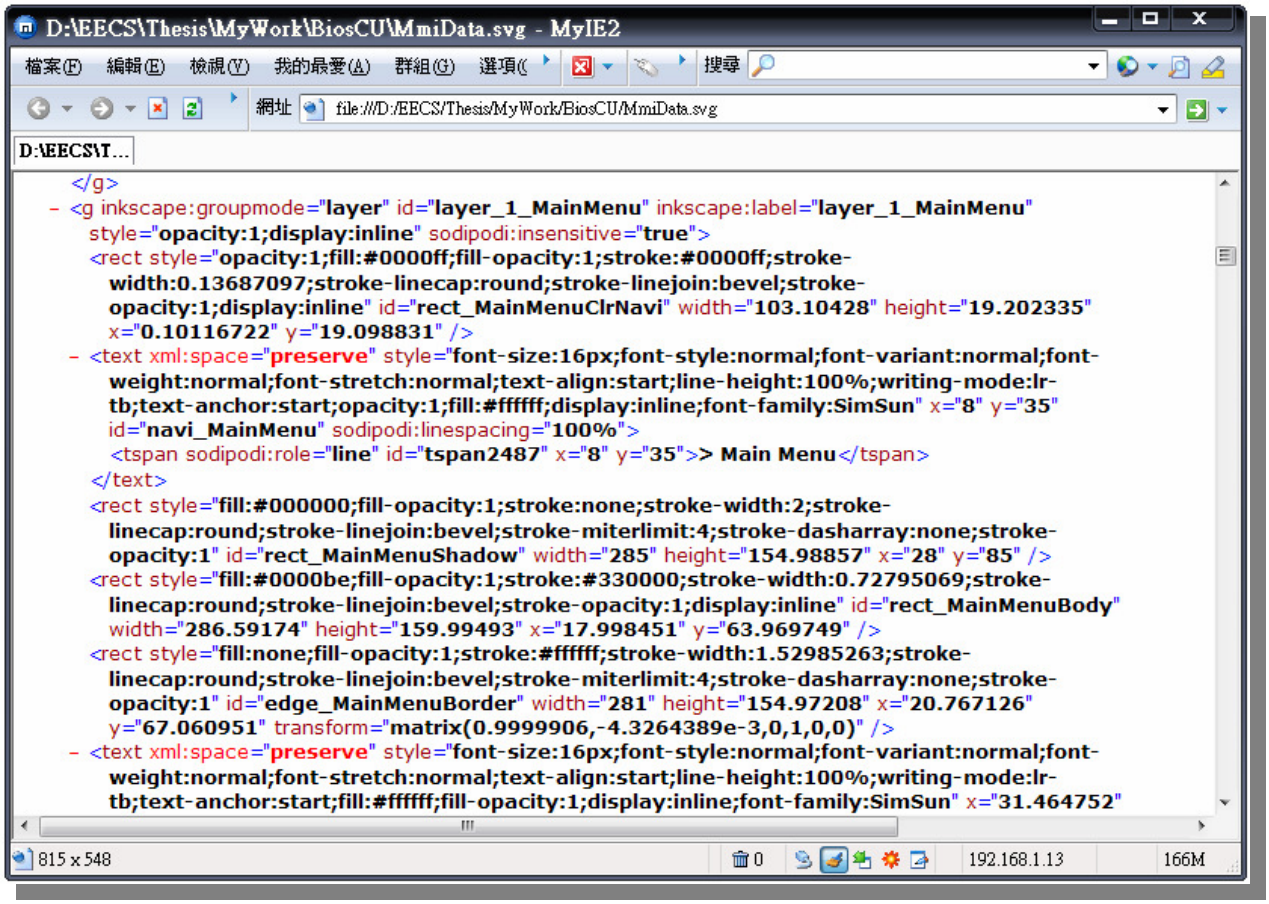


Figure 23. SVG File in Plain Text

Step 3) To speed up the development of MMIs, we can create Template SVG Files for MMI designers. We compose the *Template SVG File* in figure 24 with Menu Template (*layer_MenuTemplate*) that has key elements for each layer of menu.

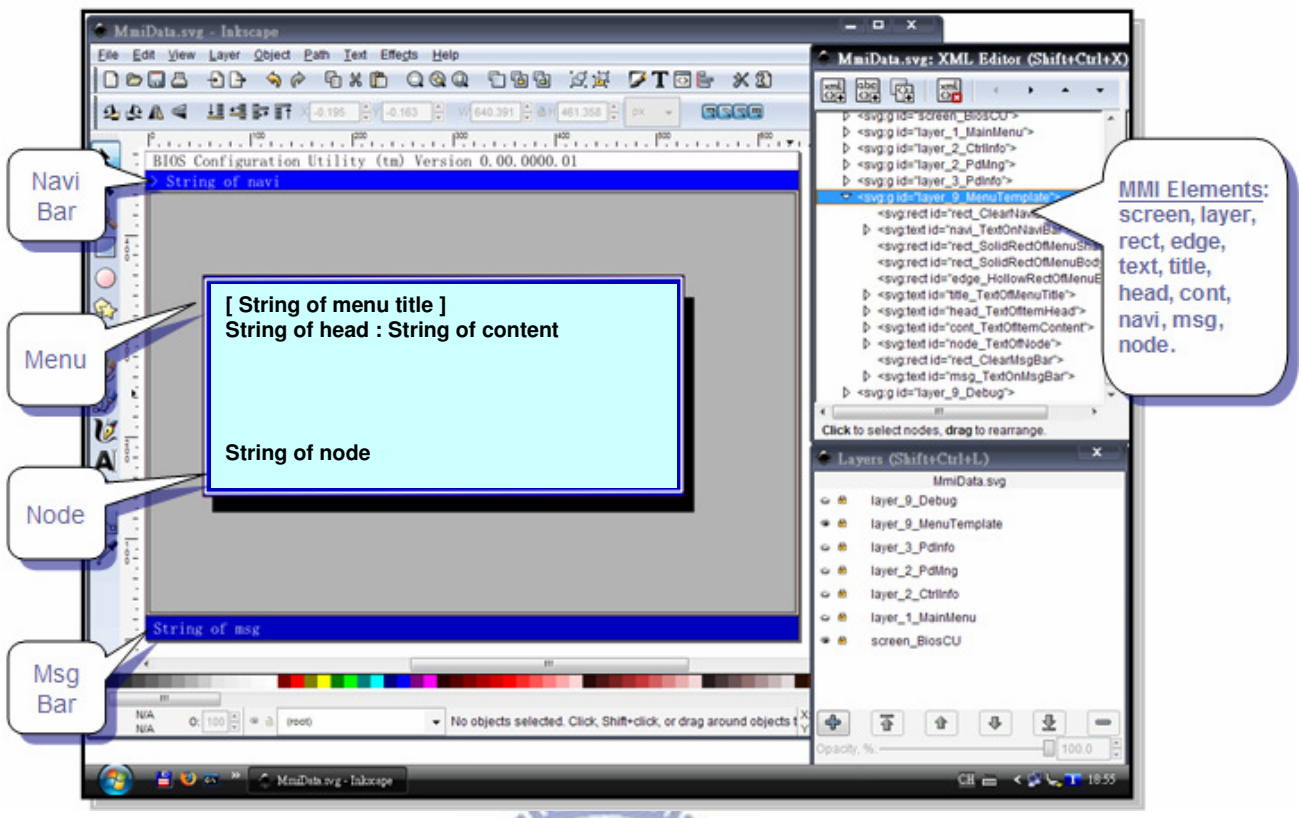


Figure 24. Compose the Template SVG File

These key elements are *Navi Bar, Menu, Node,* and *Msg Bar*. *Navi Bar* is a row for navigation, and it displays the hierarchy of active menus. *Menus* include menu title, heads of items, contents of items, and nodes. *Nodes* are special text elements with the ability to combine the function and child menu together. If users select one node in the running time, it will take the related function and then pop up its child menu. *Msg Bar* is a row to display messages or hints for users. For example, we can list hot keys for active menu particularly.

Using the layer of Menu Template (*layer_MenuTemple)* in the Template SVG file, MMI designers can compose the wanted menus quickly. Figure 25 illustrates three menus were been composed by using the *layer_MenuTemple*: Main Menu (*layer_1_MainMenu*), Physical Drive Management Menu (*layer_2_PdMng*), and Physical Drive Information Menu (*layer_3_PdInfo*), and other menu layers are hidden: Controller Information Menu (*layer_2_CtrlInfo*), Menu Template (*layer_9_MenuTemplate*), and Debug Layer (*layer_9_Debug*). In the right bottom corner of figure 25, the *Layer Manger* of Inkscape can control these layers.

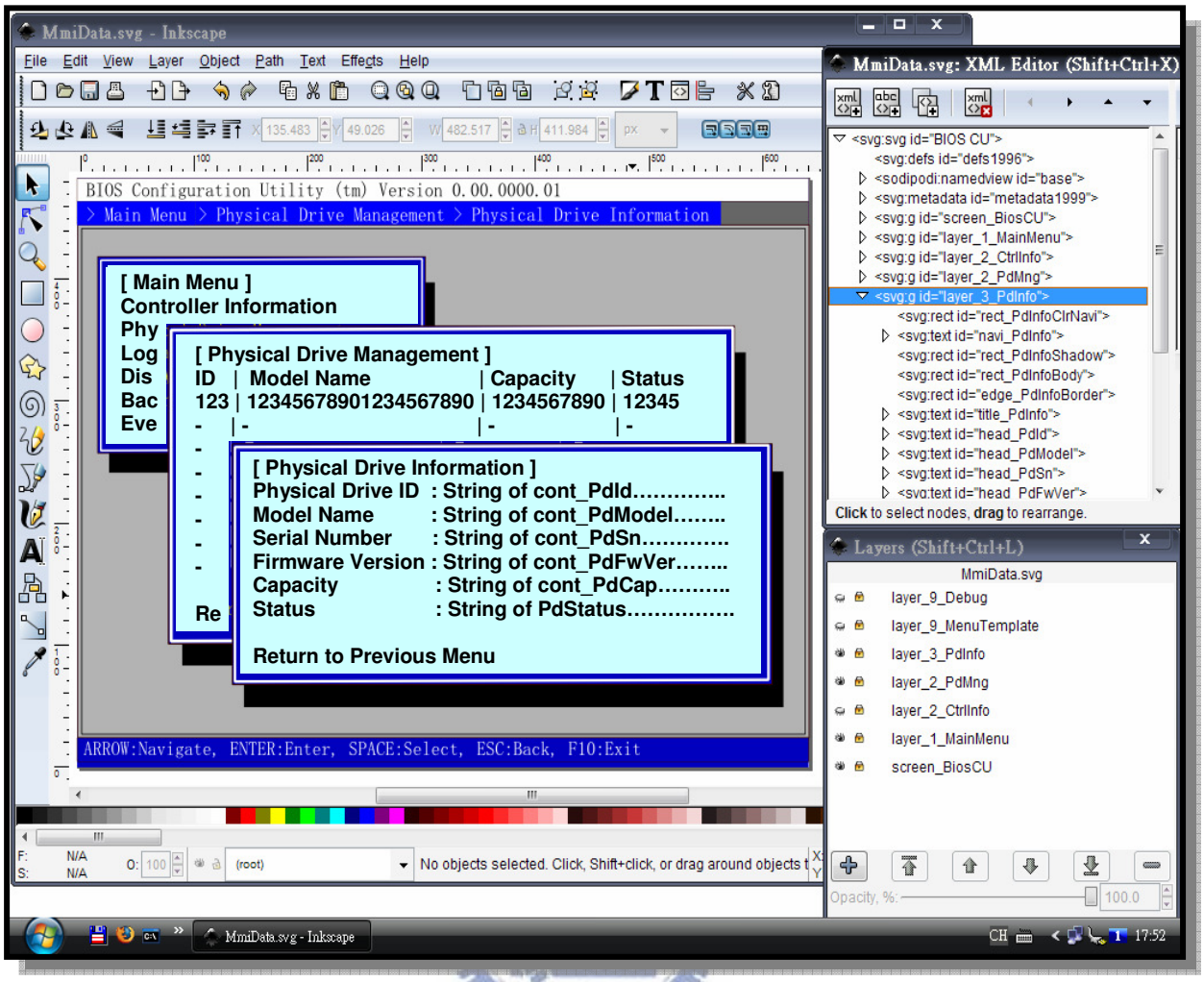


Figure 25. Compose Menus by Using the layer_MenuTemple

Step 4) For developers or customers, they can preview these SVG Files in web browsers. During the *Visual Authoring Process*, we can use web browser to preview SVG files. In order to let web browser displays texts in SVG files correctly, we have to set texts as Unflow. In Inkscape, select wanted texts and choose the function: Text > Unflow.

Step 5) MMI programmers add additional information in SVG Files. We use the *XML Editor* of Inkscape to edit the SVG elements, and give the meaningful names and attributes for MMI elements in Menu Template (*layer_MenuTemplate*). For instance, we give *layer_1_MainMenu* as the name of main menu layer, and the attribute of *layer_2_CtrlInfo* as the child menu for node *node_CtrlInfo*. This means that users press the node of *node_CtrlInfo*, and Generic MMI Engine will display the MMI elements in *layer_2_CtrlInfo.*

Step 6) Use SVG Parser to transform SVG Files to MMI Data. After we save the results in SVG files, we can use *SVG Parser* to generate *MMI Data* for *Generic MMI Engine* automatically. The output of *SVG Parser* is MmiData.h, and the following example is a partial code of MmiData.h.

```
/* MmiData.h - Generated by SVG Parser. Don't edit this file directly! */

CREATE_SCREEN(screen_BiosCU)
CREATE_RECT(rect_ClearScreen, 0, 0, 640, 480, COLOR_BLUE)
CREATE_RECT(rect_ScreenTitle, 0, 0, 640, 19, COLOR_B_WHITE)
CREATE_TEXT(text_ScreenTitle, 8, 15, COLOR_BLACK, "BIOS Configuration
    Utility (tm) Version 0.00.0000.01")
CREATE_RECT(rect_NaviBar, 0, 19, 640, 19, COLOR_GRAY)
CREATE_RECT(rect_Canvas, 0, 38, 640, 418, COLOR_WHITE)
CREATE_EDGE(edge_CanvasBorder, 4, 41, 633, 411, COLOR_BLACK)
CREATE_RECT(rect_MsgBar, 0, 456, 640, 19, COLOR_BLUE)
END_OF_SCREEN(screen_BiosCU_end)

CREATE_LAYER(layer_1_MainMenu, 1)
CREATE_RECT(rect_MainMenuClrNavi, 0, 19, 103, 19, COLOR_B_BLUE)
CREATE_NAVI(navi_MainMenu, 8, 35, COLOR_B_WHITE, "> Main Menu")
CREATE_RECT(rect_MainMenuShadow, 28, 85, 285, 154, COLOR_BLACK)
CREATE_RECT(rect_MainMenuBody, 17, 63, 286, 159, COLOR_BLUE)
CREATE_EDGE(edge_MainMenuBorder, 20, 67, 281, 154, COLOR_B_WHITE)
CREATE_TITLE(title_MainMenu, 31, 91, COLOR_B_WHITE, "[ Main Menu ]")
CREATE_NODE(node_CtrlInfo, 33, 110, COLOR_YELLOW, layer_2_CtrlInfo,
    MmiAct_CtrlInfo, "Controller Information")
CREATE_NODE(node_PdMng, 33, 128, COLOR_YELLOW, layer_2_PdMng, MmiAct_PdMng,
    "Physical Drive Management")
CREATE_NODE(node_LdMng, 33, 147, COLOR_YELLOW, layer_1_MainMenu,
    MmiAct_Null, "Logical Drive Management")
CREATE_NODE(node_DaMng, 33, 167, COLOR_YELLOW, layer_1_MainMenu,
    MmiAct_Null, "Disk Array Managment")
CREATE_NODE(node_BgAct, 33, 185, COLOR_YELLOW, layer_1_MainMenu,
    MmiAct_Null, "Background Activity")
CREATE_NODE(node_EventLog, 33, 204, COLOR_YELLOW, layer_1_MainMenu,
    MmiAct_Null, "Event Log")
CREATE_RECT(rect_MainMenuClrMsg, 0, 456, 638, 19, COLOR_BLUE)
CREATE_MSG(msg_MainMenu, 5, 470, COLOR_B_WHITE, "ARROW:Navigate,
    ENTER:Enter, SPACE:Select, ESC:Back, F10:Exit")
END_OF_LAYER(layer_1_MainMenu_end)
```

Step 7) MMI programmers program related functions in Management API according to functional requirement documents. We write the related functions for MMIs. For each *Node* MMI element, MMI programmers have to add the related functions into *MmiAct* and *MmiSub* modules of *Generic MMI Engine*. For example, we write the *MmiAct_CtrlInfo( )* function to get information of controller for *node_CtrlInfo*, and the *MmiAct_PdMng( )* function to get a list of physical hard disk drives for *node_PdMng*, etc.

Step 8) Generate the target MMI applications with MMI Data, Generic MMI Engine, Management API and OS dependent API. To simplify the complexity of demonstrations, we use two definitions to configure *Generic MMI Engine*. First, define *USE_MNG_API* to indicate if it needs to invoke Management API or not. Second, define *DRIVER_SIMULATOR* to indicate if it simulates the reactions of driver by itself or not.

If we set ***USE_MNG_API*** to OFF, *Generic MMI Engine* displays *MMI Data* directly. All the messages and texts are the same as the ones in *Visual Authoring Tool*. If we set *USE_MNG_API* to ON, *Generic MMI Engine* invokes the functions of Management API, and these functions submit related commands to driver directly.

If we set ***DRIVER_SIMULATOR*** to ON, the code of *Driver Simulator* in *Generic MMI Engine* takes actions to simulate reactions of real driver. If we set *DRIVER_SIMULATOR* to OFF, the real driver is required to pass commands to Embedded Firmware of storage systems.

In the simulation example, figure 26 demonstrates the Booting Utility of storage systems in the DOS Prompt of Windows OS.

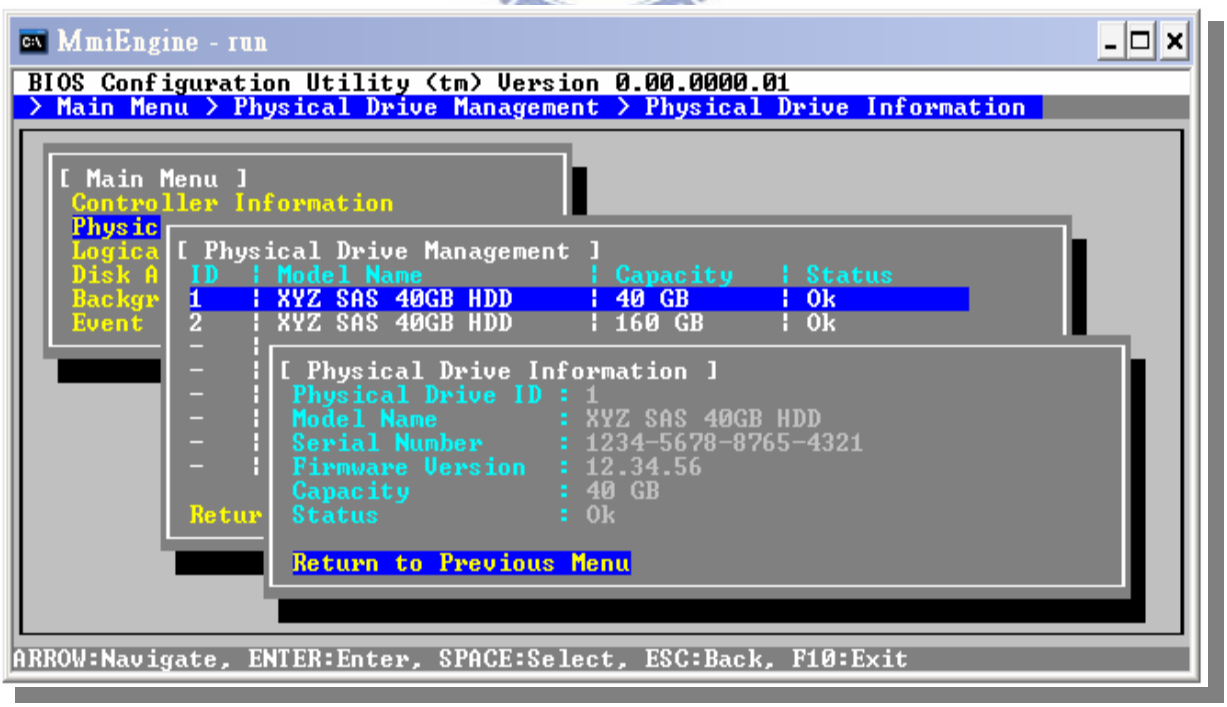


Figure 26. Simulation Examples for the Booting Utility of Storage Systems

In the case one of simulation examples, we set *USE_MNG_API* to OFF and *DRIVER_SIMULATOR* to ON. The utility will display MMI Data directly.

In the case two of simulation examples, we set *USE_MNG_API* to ON and *DRIVER_SIMULATOR* to ON. The utility will display imitative information from the *Driver Simulator* in *Generic MMI Engine*.

4.2 Application Examples for the Pre-OS Utility of Storage Systems

Before the demonstration of application examples, we have to setup a platform with an internal storage system. We use a ***HBA*** (Host Bus/Based Adapter) adapter and attach two hard disk drives. One hard disk drive is *SATA* (Serial ATA) drive and the other one is *SAS* (Serial Attached SCSI) drive.

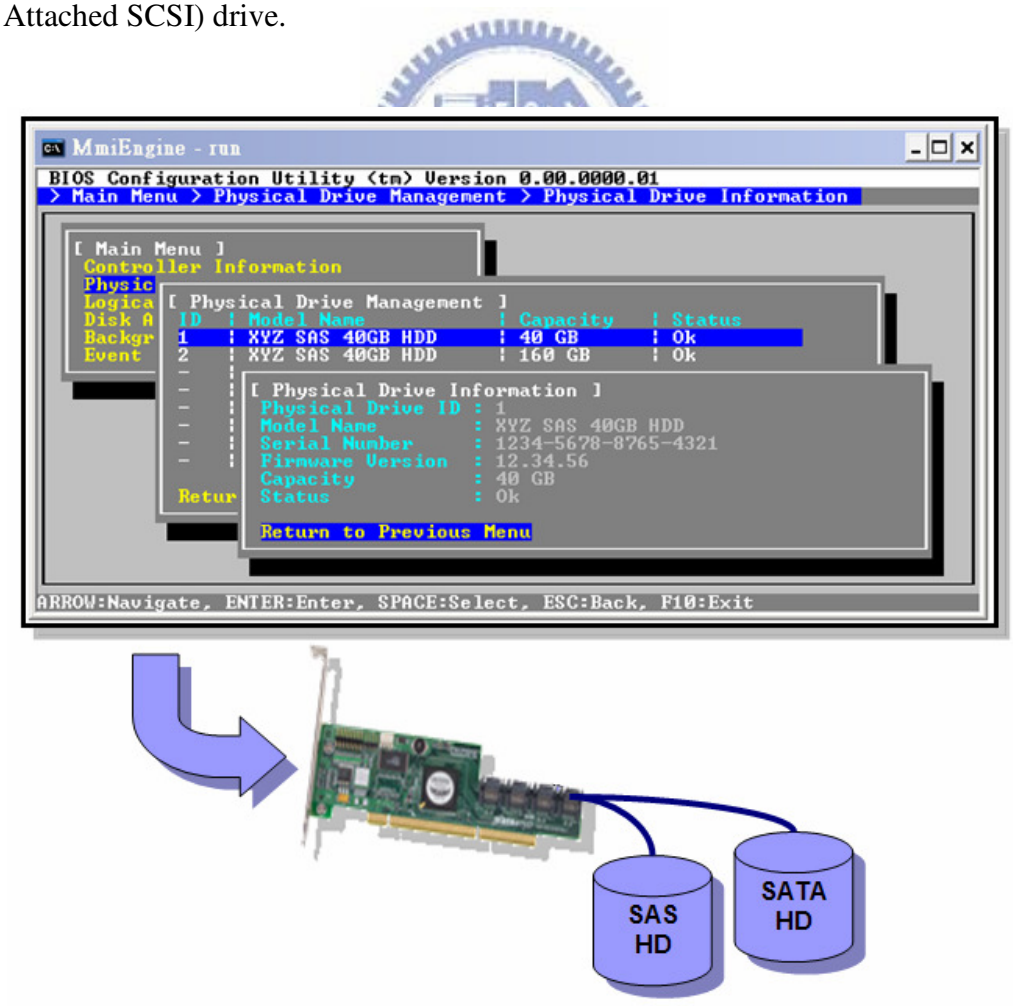


Figure 27. Application Examples for the Pre-OS Utility of Storage Systems

In order to get the real information from embedded firmware of storage system, we compile *Generic MMI Engine* with the Management API of storage system. The functions of the Management API have to allocate data buffer and submit management commands to driver. The driver will pass these commands to embedded firmware. After the embedded firmware handles management commands and gets the wanted results, it will transfer data by *DMA* (Direct Memory Access) to the data buffer of the Management API.

The *MmiSub module* of *Generic MMI Engine* uses the functions of Management API to fetch particular items in the data buffer, and copy them into *MMI data* of *Generic MMI Engine*.

In the application examples, we demonstrate the Pre-OS Utility of storage systems in the Pre-OS environment, e.g. pure *DOS* (Disk Operating System). Of course, we must set the configuration *USE_MNG_API* to ON and *DRIVER_SIMULATOR* to OFF.

For the case one of application examples, we execute this utility and enter the menu "*Controller Information*" to get the information of adapter controller, enter the menu "*Physical Drive Management*" to get a list of present physical hard disk drives, and the menu "*Physical Drive Information*" to get the information of assigned physical hard disk driver. To verify the correction, we can compare the information with the results displayed by the Embedded Utility of storage systems.

For the case two of application examples, we plug off some hard disk drives and check the list of present physical hard disk drives in the menu "*Physical Drive Management*" and "*Physical Drive Information*". Then, we plug in some hard disk drives and enter the above menus to check again.

Chapter 5
Conclusion and Future Work

In this chapter, we draw out the conclusion of this thesis to see the design and implementation can meet the motivation and goal or not, and then we point out the future work for *Visual MMI Development for Storage Systems*.

5.1 Conclusion of This Thesis

In this thesis, we design and implement the *Visual MMI Development for Storage Systems*, and derive the *Generic Software Framework for the MMI Generation of Storage Systems*. Figure 28 highlights the ***Visual Authoring Process*** *of Visual MMI Development for Storage Systems.*

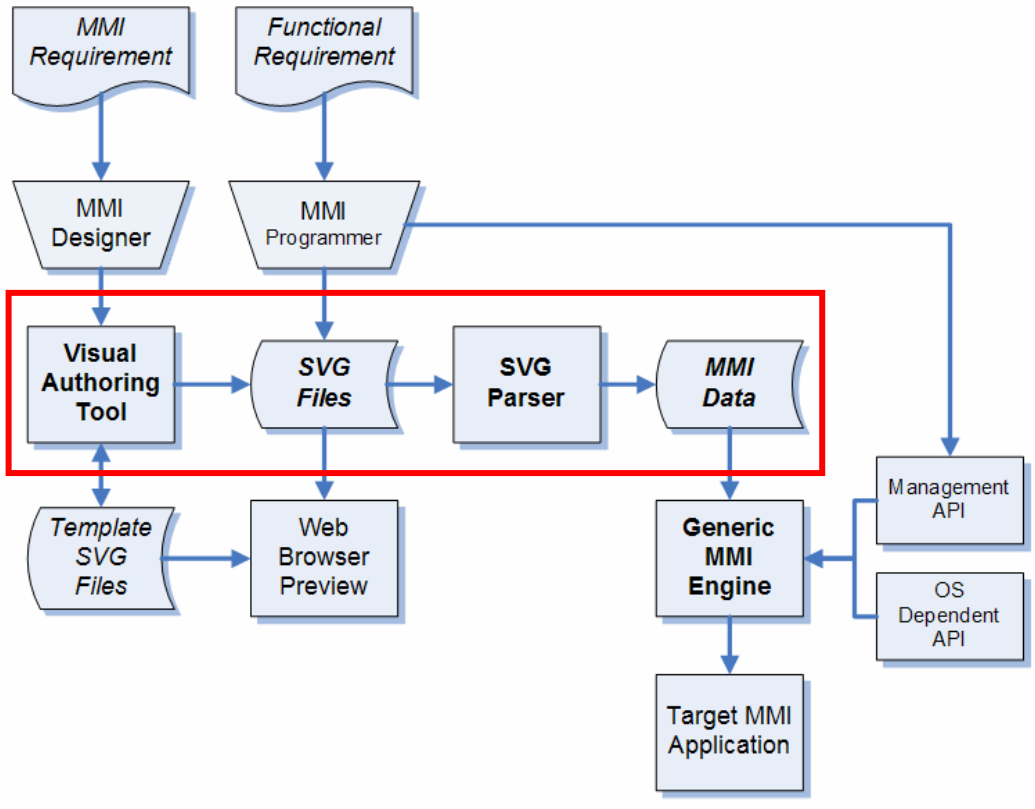


Figure 28. Visual Authoring Process of Visual MMI Development for Storage Systems

In the *Visual Authoring Process*, we use *Visual Authoring Tool -- Inkscape, SVG Files, SVG Parser, and MMI Data*. The benefits of *Visual Authoring Process* are:

1) MMI designers, e.g. art designer, or product's customer, use *Visual Authoring Tool -- Inkscape* to compose MMIs directly. Reduce any misunderstanding and the gap of knowledge between MMI designers and programmers.

2) Programmers can add related attributes in *SVG Files* directly, and use *SVG Parser* to generate *MMI Data* automatically. Therefore, *MMI Data* is seamless with the MMI requirements, and the time to program MMI elements is also saved.

Figure 29 highlights the ***Generic MMI Engine*** in the *Visual MMI Development for Storage Systems.*

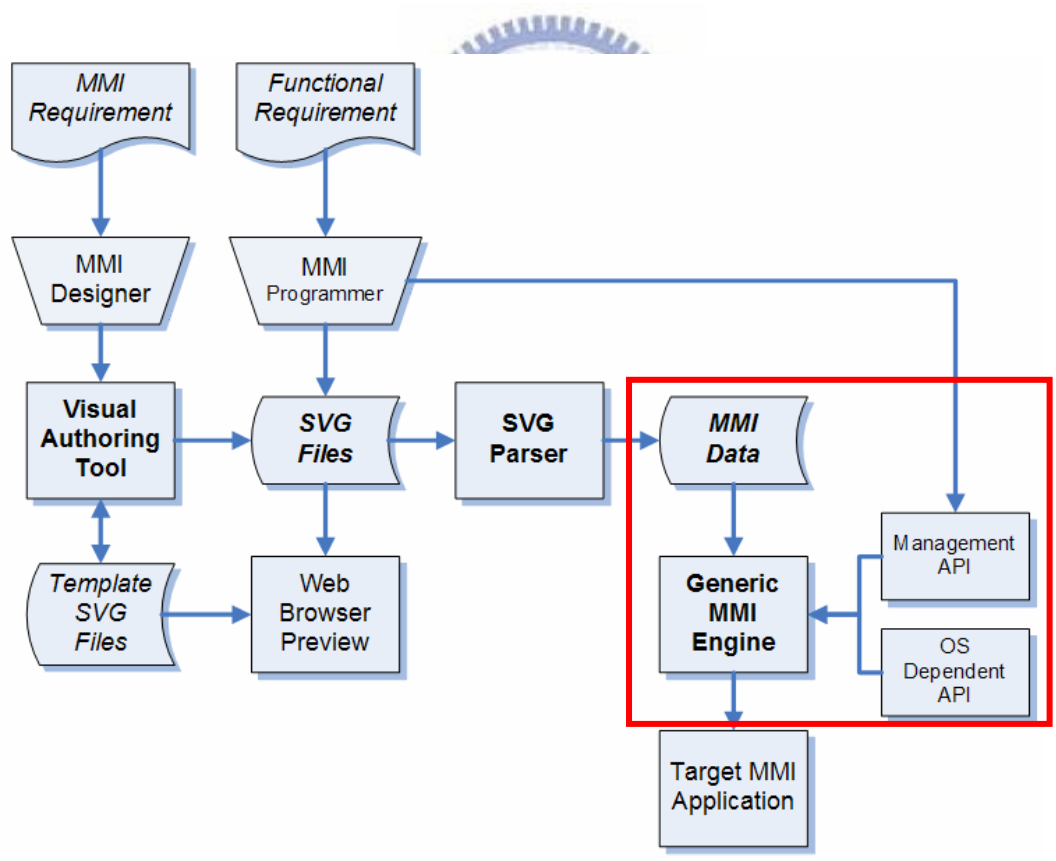


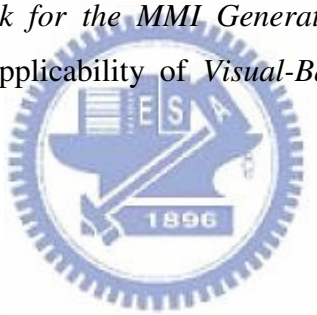
Figure 29. Generic MMI Engine for the MMI Generation of Storage Systems

To apply the *Generic Software Framework for the MMI Generation of Storage Systems*, we implement ***Generic MMI Engine*** that manipulates *MMI Data*, i.e. the output of *Visual Authoring Process*, and cooperates with *Management API*. The Benefits of *Generic Software Framework for the MMI Generation of Storage Systems* are:

1) *Generic MMI Engine* is a common code, and can manipulate multiple *MMI Data.* Therefore, programmers can maintain MMI programs easily for serial products and different customers.

2) The design of *Generic MMI Engine* is OS independent, and the OS dependent functions are separated to *OS Dependent APIs*. Therefore, we can implement it for various OS environments easily.

After the design and implementation of *Visual MMI Development for Storage Systems* and *Generic Software Framework for the MMI Generation of Storage Systems*, we also demonstrate the feasibility and applicability of *Visual-Based User Interface Construction Methodology*.



5.2 Future Work

In the design and implementation of *Visual MMI Development for Storage Systems*, we choose *Inkscape* as the *Visual Authoring Tool*. It is an open source software and powerful graphic editor, but we can improve it with the following features:

1) ***Support Replay Feature***: MMI designers compose wanted MMIs in *Inkscape*, but they are static graphics. If *Inkscape* can support replay feature to demonstrate the relationship between nodes and their child menus, we can get an interactive prototype more conveniently.

2) ***Provide Function Binder***: Currently MMI programmers use *XML Editor* of *Inkscape* to add the related functions for MMI elements. If *Inkscape* can provide the function binder to select the wanted function in API library and bind it to the related MMI element, this is more user-friendly for MMI programmers.

Reference

[1] DoSTOR 存儲入門：圖文闡釋 DAS、NAS、SAN
http://www.dostor.com/i/basic/2006-09-22/0006129755.shtml

[2] EFI (Extensible Firmware Interface) http://www.intel.com/technology/efi/

[3] Ian Sommerville, Software Engineering, 6th edition, Addison-Wesley, 2001

[4] Chwan-Hung Wang, "On the Enhancement of an Multimedia Authoring tool for the Visual-Based User Interface Requirement Representation", N.C.T.U. Taiwan, Master Thesis, 2002

[5] Jai-Chen Dai, "Visual-Based User Interface Generator", N.C.T.U. Taiwan, Master Thesis, 2002

[6] Shang-Ting Yang, "User look & Feel Design for Handset Devices Based on Visual Requirement Authoring and Program Generation Methodology", N.C.T.U. Taiwan, Master Thesis, 2004

[7] Ming-Jyh Tsai, "Generating User Interface for Mobile Devices Using Visual-Based User Interface Construction Methodology", N.C.T.U. Taiwan, Doctor Thesis, 2007

[8] Chien-Chung Lin, "A Generic DSC Software Framework in Handset Device" , N.C.T.U. Taiwan, Master Thesis, 2005

[9] Meng-Xi Zhuang, "XMMI – Extensible Man Machine Interface System", N.C.T.U. Taiwan, Master Thesis, 2004

[10] Ming-Chao Huang, "Extensible MMI system for mobile device and it's rapidly prototyping", N.C.T.U. Taiwan, Master Thesis, 2005

[11] Po-Chang Liu, "A Generic Software Framework for the Software System Architecture Design and Implementation of Handset Devices", N.C.T.U. Taiwan, Master Thesis, 2005

[12] Inkscape – Open Source SVG Editor http://www.inkscape.org/

[13] World Wide Web Consortium (W3C) http://www.w3.org/

[14] Scalable Vector Graphics (SVG) http://www.w3.org/Graphics/SVG/

[15] Extensible Markup Language (XML) http://www.w3.org/XML/

[16] XML Document Object Model (DOM) http://www.w3.org/DOM/

[17] W3C Schools – Online Tutorials http://www.w3schools.com/default.asp

[18] XML Parser http://www.w3schools.com/xml/xml_parser.asp

[19] Expat XML Parser http://expat.sourceforge.net/

[20] Compare SANs to Alternate Technologies
http://www.brocade.com/san/evaluate/compare_san.jsp

[21] Barkakati, The Waite Group's Turbo C Bible, 蔡明志譯, 松岡, 台北, 1992

[22] Robert Lafore, C Programming Using Turbo C++, 蔡明志譯, 松岡, 台北, 1992

[23] 施威銘, 80x86 MASM 6.x 組合語言實務, 旗標, 台北, 1996

Appendix A
External Storage system

In section 1.1 Overview of Storage Systems, we have introduced the overview of storage system. In this appendix, we explain the details of DAS (Direct Attached Storage), SAN (Storage Area Network), NAS (Network Attached Storage), and compare the difference between them. [20]

A.1 DAS (Direct Attached Storage)

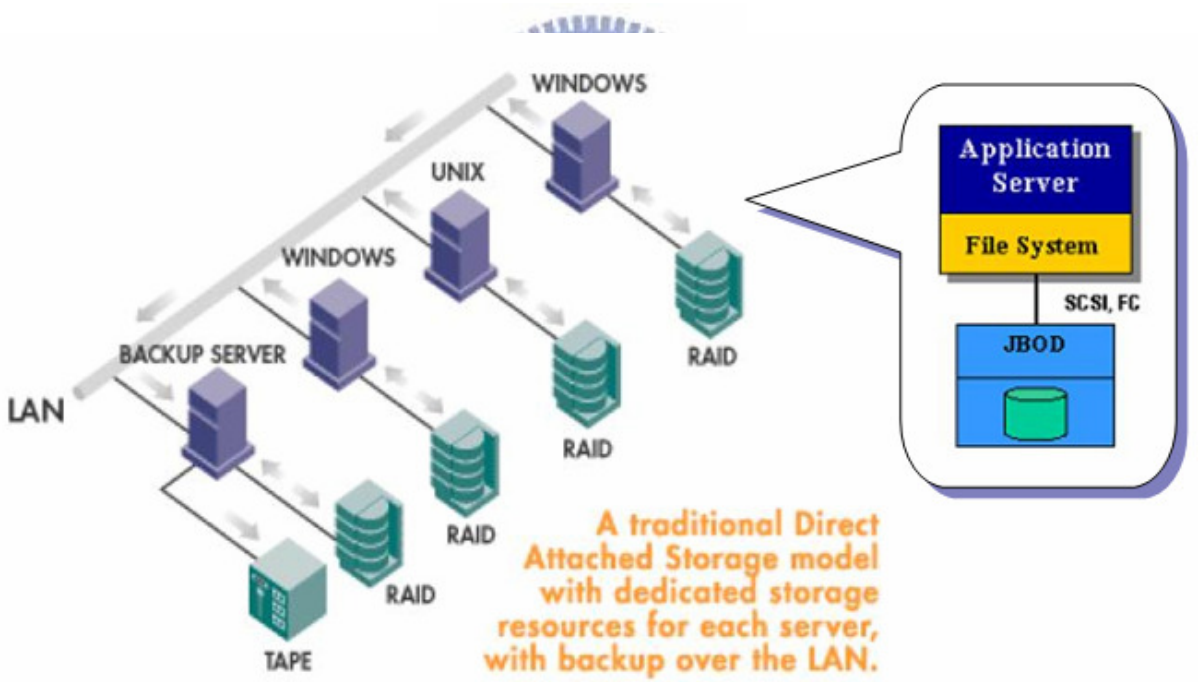


Figure 30. DAS (Direct Attached Storage)

Historically, the standard way of connecting hosts to storage devices has been direct, one-to-one SCSI attachments. As more and more storage and servers are added to meet demands, a DAS environment can cause a proliferation of server and storage islands, creating

a huge management burden for administrators, as well as inefficient utilization of resources. Data sharing in these environments is also severely limited.

A.2 SAN (Storage Area Network)

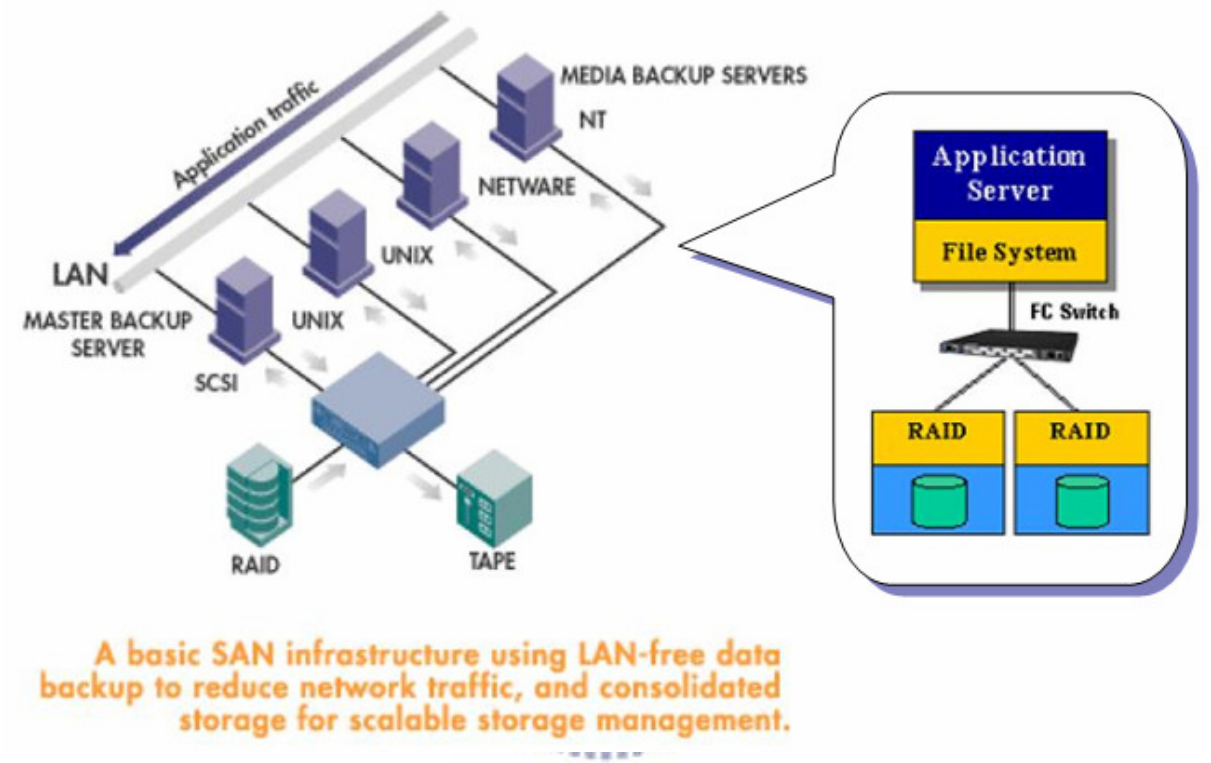


Figure 31. SAN (Storage Area Network)

The limitations and challenges of DAS are the reason many organizations today have chosen SAN or a combination of SAN and NAS solutions. The most effective SANs provide a wide range of benefits and advantages over DAS, including:

- More effective utilization of storage resources through centralized access
- Simplified, centralized management of storage, reducing administrative workload to save time and money
- Increased flexibility and scalability through any-to-any storage and server connectivity
- Improved throughput performance to shorten data backup and recovery time
- Reduced LAN congestion due to removal of backups from production network
- Higher data availability for business continuance through a resilient network design

- Excellent scalability and investment protection allowing you to easily add more storage as your business needs demand
- Superior security for storage environments
- Non-disruptive business operations when you add or re-deploy storage resources
- Proven short- and long-term return on investment (ROI)

A.3 NAS (Network Attached Storage)

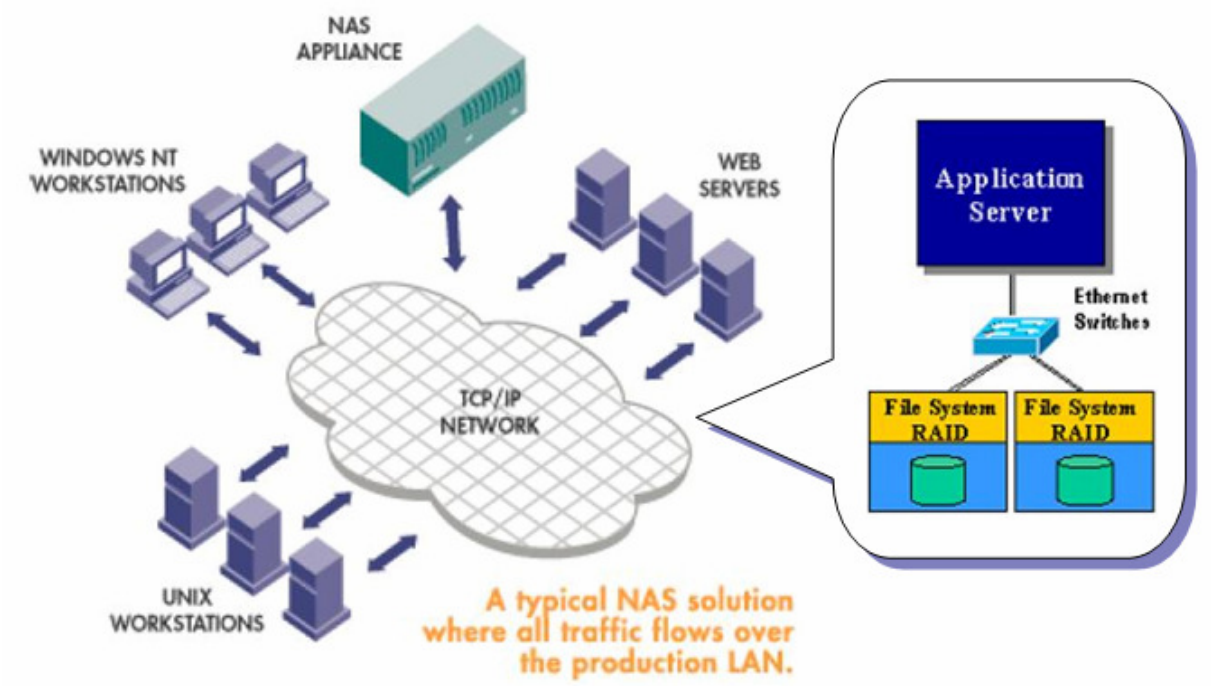


Figure 32. NAS (Network Attached Storage)

Unlike SANs that utilize a network of fiber channel switches, most NAS connections reside between workstation clients and the NAS file-sharing facility. These connections rely on the underlying corporate network infrastructure to function properly, which can lead to network congestion, particularly for larger data transfers. NAS solutions are typically configured as file-serving appliances accessed by workstations and servers through a network protocol such as TCP/IP and applications such as Network File System (NFS) or Common Internet File System (CIFS) for file access.

NAS storage scalability is often limited by the size of the self-contained NAS appliance enclosure. Adding another appliance is relatively easy, but sharing the combined contents is not. Because of these constraints, data backups in NAS environments typically are not centralized, and therefore are limited to direct attached devices (such as dedicated tape drives or libraries) or a network-based strategy where the appliance data is backed up to facilities over a corporate or dedicated LAN. Increasingly, NAS appliances are using SANs to solve problems associated with storage expansion, as well as data backup and recovery.

NAS does work well for organizations needing to deliver file data to multiple clients over a network. Because most NAS requests are for smaller amounts of data, data can be transferred over long distances efficiently.

A.4 Comparison of DAS / SAN / NAS

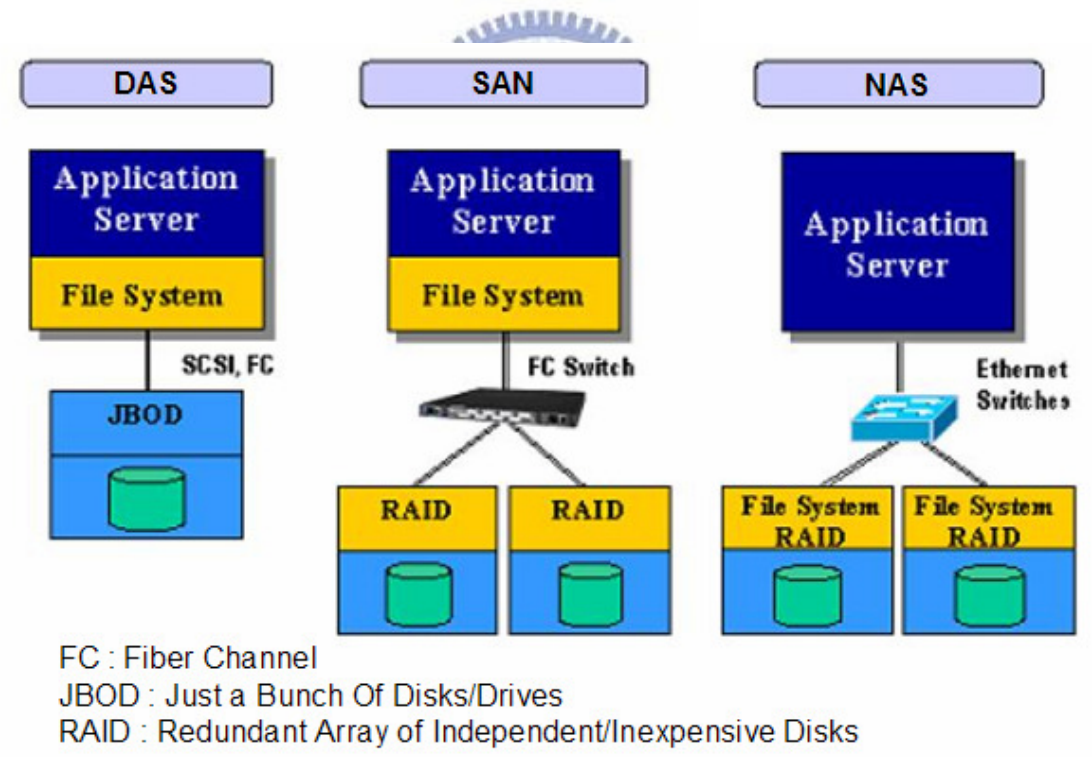


Figure 33. Comparison DAS / SAN / NAS

The major difference between DAS and SAN is the connection from application server to those storage systems. DAS usually use SCSI or fiber channel to connect to one

server, but SAN storage system utilizes the fiber channel switch to connect many servers at one. These servers can share the same storage system in Storage Area Network.

For the administrators of application servers that use DAS or SAN, the external storage system just like a logical drive under file system of operating system. They can partition the logical drive as a normal hard drive.

However, the operations on NAS storage system are not the same as DAS or SAN. The administrators of application servers that use NAS must mount the remote partition on the NAS storage system on network. This is because the NAS devices provide the servers of NFS (Network File System) or something like that. All servers in the same LAN can mount it with the same protocol very easily.

Appendix B
Application Instances of Inkscape

In section 3.2.1, we have introduced Inkscape graphic editor. In this appendix, we enumerate three kinds of application instances for Inkscape. First is using Inkscape to create vector graphics, and second is composing comics and maps. Finally, design web pages with static and dynamic graphics. After these instances, you can understand the power and important functionalities of the wonderful graphic editor.

B.1 Creating Vector Graphics

First instance is to create vector graphics. This is the basic functionality of Inkscape. Most people and I think that vector graphics should be the combination of simple geometric figures. After view the art works by the creative users of Inkscape, I realize the power of Inkscape and SVG language.

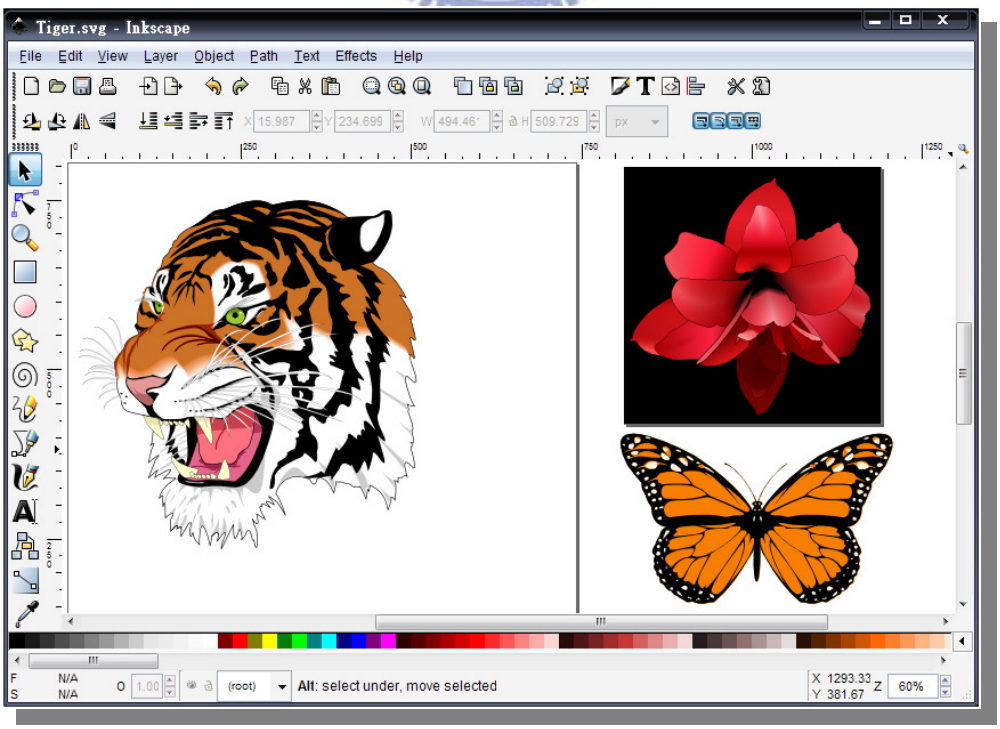


Figure 34. Complex Vector Graphics

You can view the famous tiger head in figure 34 and download its SVG file from internet. Use Inkscape to open it, and you will find that each part of this tiger head is one kind of SVG element. Inkscape provides plentiful graphic tools to draw basic shapes, freehand lines, Bezier curves and straight lines, calligraphic lines, etc.

Therefore, the users with art talent can create many beautiful two-dimension graphics and save them in SVG files. On the other hand, some web browsers, like Firefox, support SVG files. They can draw SVG files directly in web pages.

Besides artificial vector graphics, some people create photo realistic graphics by Inkscape. These graphics have the advantage of vector graphics – no distortion when their sizes were changed.

The Gaussian Blur filter support in Inkscape made possible some extremely photo-realistic art. The Lamborghini Gallardo, super car in figure 35, was created by Michael Grosberg based on a photo and he uses blurs extensively for soft shadows and halos around bright reflections. The SVG file is available in Inkscape distribution. The screenshot also shows a second window with an Outline view of the same file (green outlines are clipping paths).

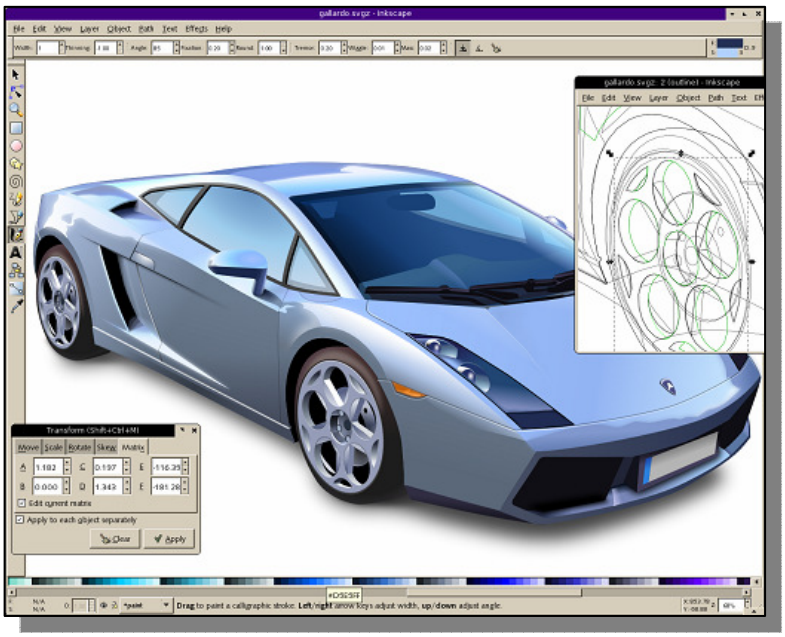


Figure 35. Photo-realistic Graphic

Inkscape includes many advanced graphic effects. For example, color effects can colors of graphics to be brighter, darker, more saturated, etc.

A group of extensions in the Color submenu of the Effects menu allows you to adjust all colors of a selection at once. These commands affect both fill and stroke colors, including gradients (but not bitmaps). The commands work recursively on groups. The only problem is that, being Python extensions, these commands may be quite slow on complex documents. Figure 36 demonstrate all color effects of Inkscape.

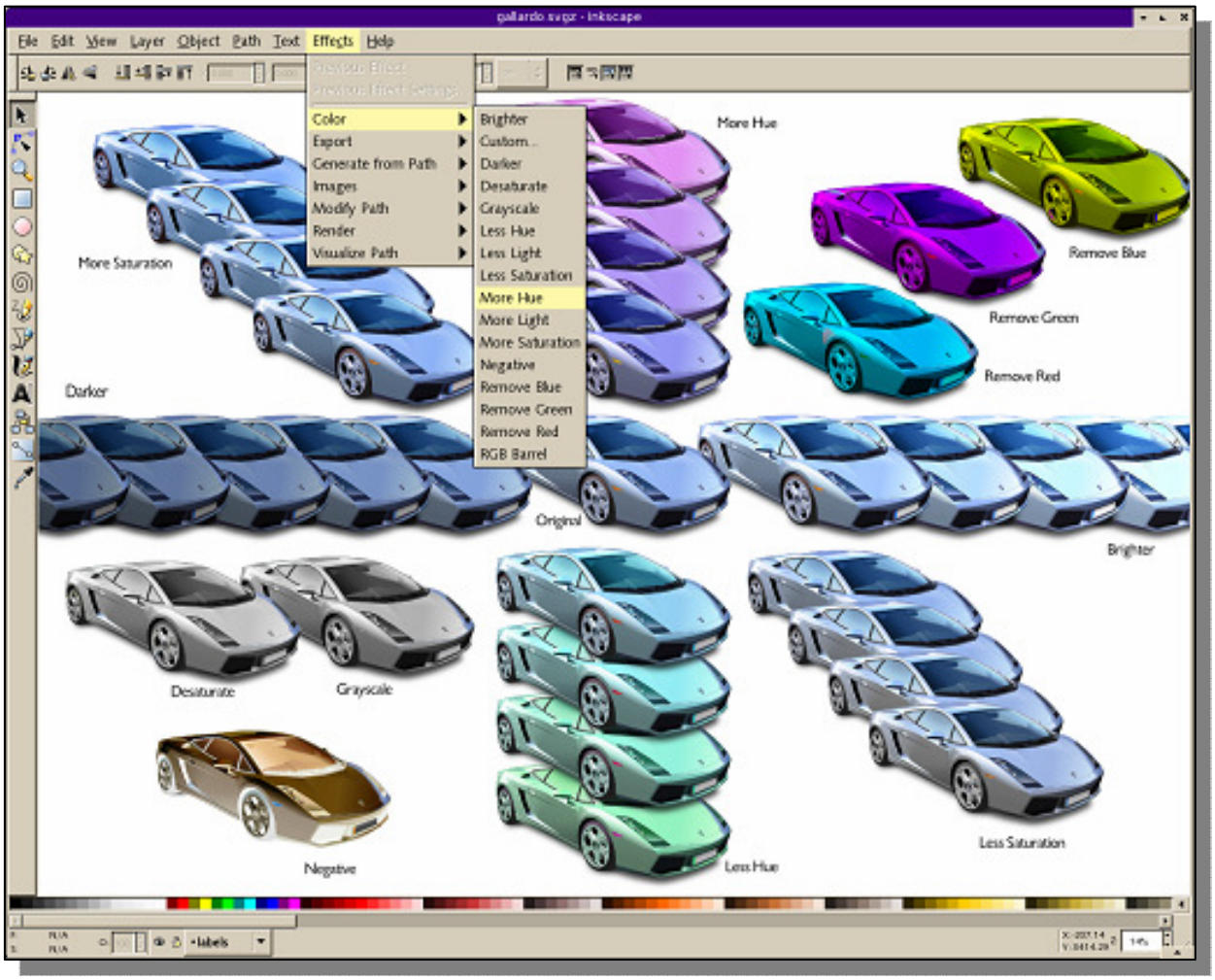


Figure 36. Advanced Graphic Effects

B.2 Producing Comics and Maps

Inkscape has layer manager that can hide or display layers and protect or unlock layers. This functionality can be used to compose comics and maps in the easy way.

A comic author, John Bintz, draw the graphic in figure 37 to demonstrate his use of Inkscape for producing comics. He scanned the original penciled pages and import them into Inkscape as the root layer. He created a new layer called “inks” and ink the drawing with the calligraphy tool. Then he used the node tool and select tool to clean up the drawing, reconnecting lines and smoothing or straightening paths as necessary. After that, he created a new layer under “inks” called “colors” and used Bezier tool to draw colored polygons for each area to be colored. Above “inks” layer, he created “balloons” layer and typed out the text, draw an ellipse around each text block.

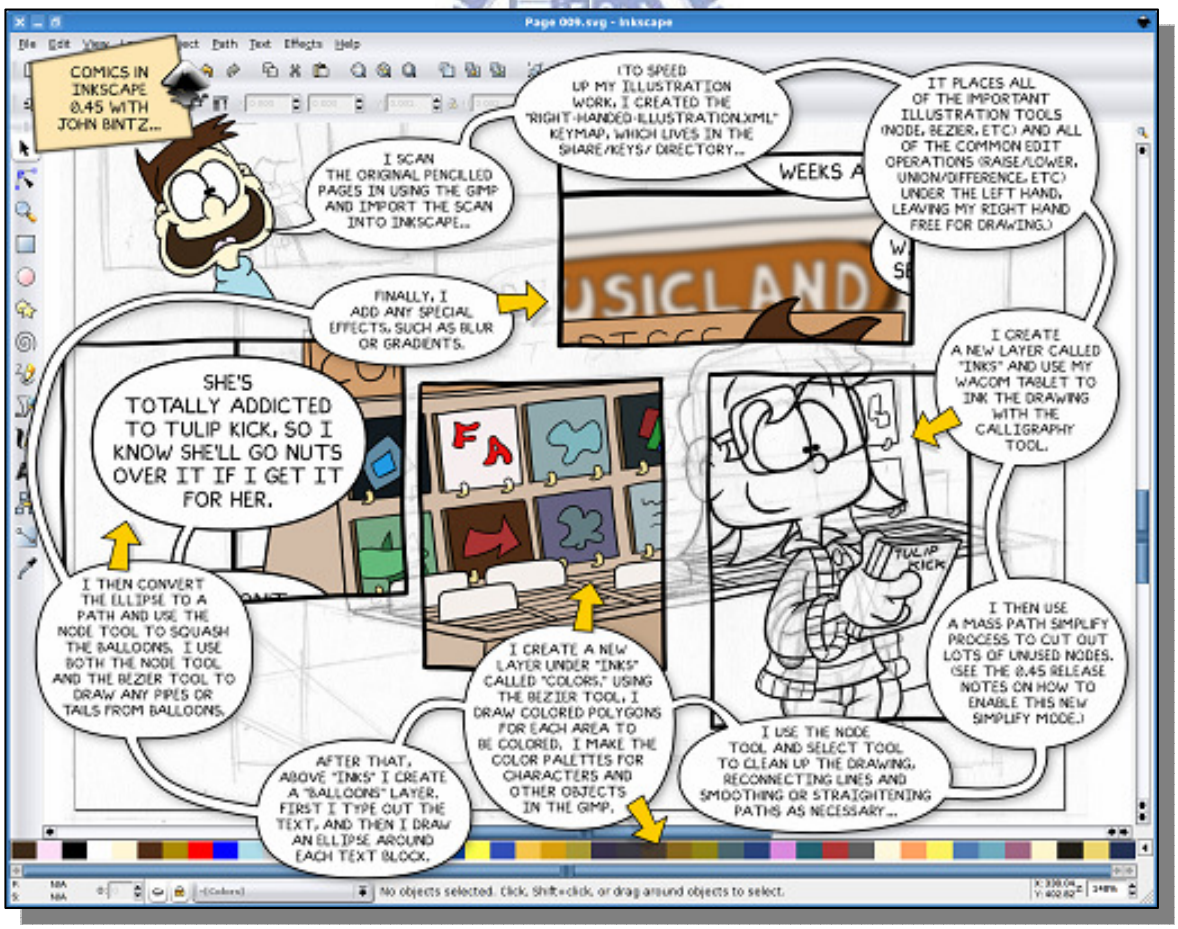


Figure 37. Producing Comics

In figure 38, the map of Lithuania was created by Andrius Ramanauskas. He used Inkscape's layer manager where we can view the layers of his drawing, as well as lock/unlock and hide/unhide them. Inkscape's layers can be hierarchical, so this dialog is not just a list but a tree whose branches can be expanded or collapsed.

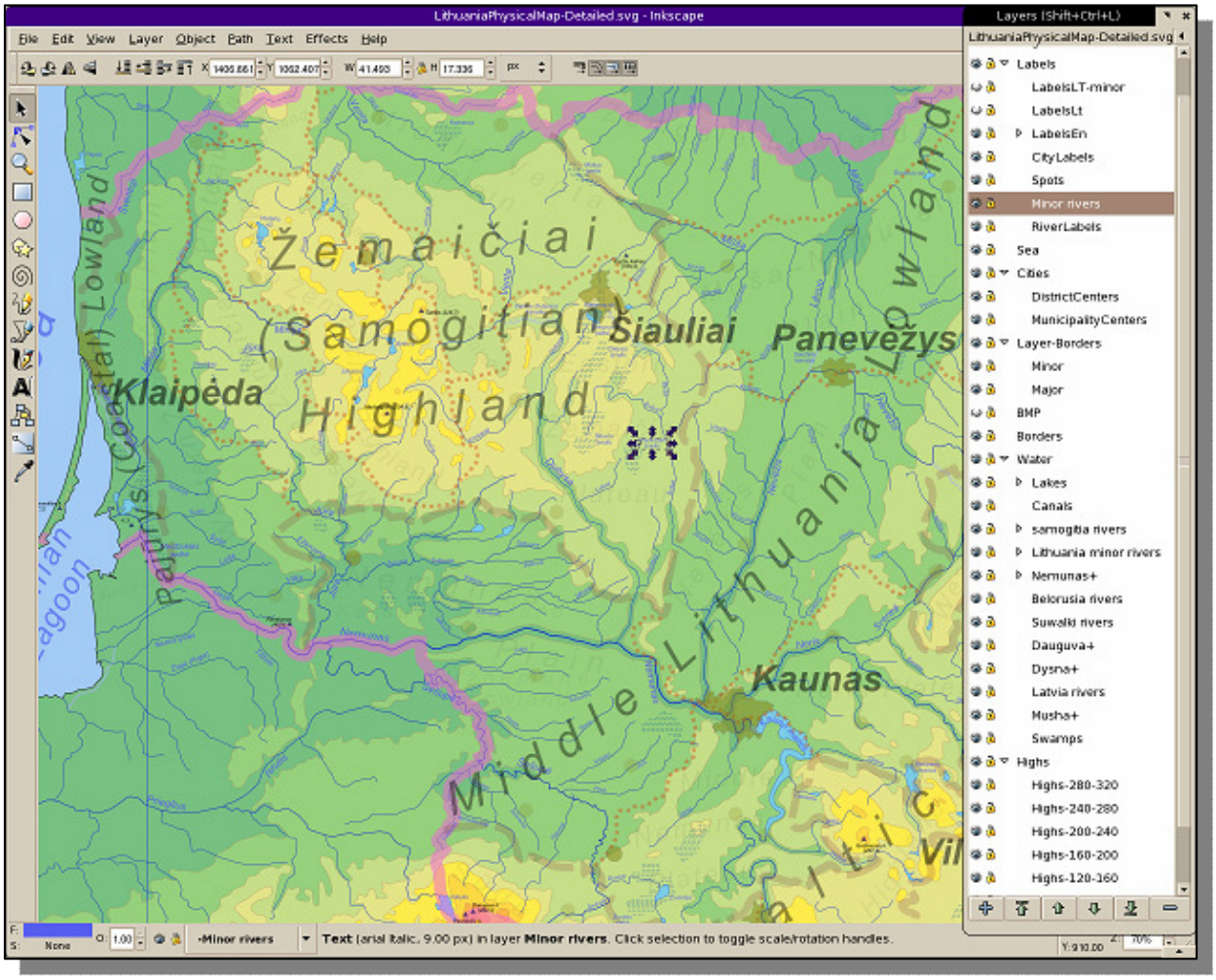


Figure 38. Producing Maps

B.3 Design of Web Pages

Currently some web browsers can draw SVG files directly, so we can use Inkscape to design web pages. Although the graphics created by Inkscape are static, we can add scripts in SVG files to let them become dynamic graphics in web pages.

Some interesting SVG files with scripts can create dynamic graphics. Please connect to the web page http://tavmjong.free.fr/INKSCAPE/ and open “Animated Mechanical Clocks with moving gears” by Firefox web browser. Clock.svg and Clock2.svg demonstrate the date and time of your system by graphic gears and these gears have hands to point the exact ones.

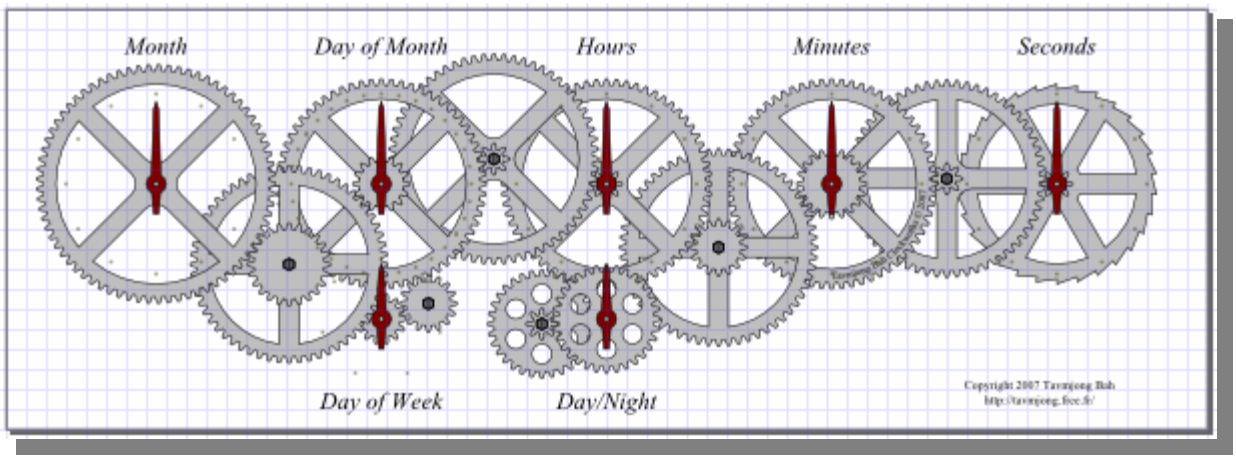


Figure 39. Animated Mechanical Clocks with Moving Gears

簡　歷

黃承一 (Cheng-Yi Huang) Dec. 1st, 1973

< 工作經歷 >

2003.10 ~ 迄今 喬鼎資訊 (Promise Technology)

目前擔任本公司軟體資深工程師，曾負責開發磁碟陣列卡之啟動程序與裝置驅動程式等工作，目前工作內容是維護並強化磁碟陣列之核心技術，以應用於各類型嵌入式儲存產品，包括內置型磁碟陣列卡及外置型磁碟陣列子系統。

1999.06 ~ 2003.10 矽統科技 (Silicon Integrated Systems)

擔任軟體工程師，主要負責撰寫各式主機板的系統啟動程序，對內協助晶片組的研發，對外提供 BIOS 公司與客戶技術上的支援。期間參與多項產品完整的開發流程，對資訊產業電腦硬體架構的演進，有更深入的瞭解；並於產品驗證偵錯的過程中，累積許多寶貴的經驗。