EFUZION: A FRAMEWORK FOR CREATING AN EXTENSIBLE EDUCATIONAL PRESENTATION AND INTERACTION SYSTEM

BY

BORIS CAPITANU

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

ABSTRACT

This study presents eFuzion, a classroom presentation and interaction system aimed at assisting students and instructors in engaging in a more immersive classroom experience. The pedagogical benefits of eFuzion have been researched for almost four years at the University of Illinois at Urbana-Champaign, allowing the software to mature through the three different iterations of the development process. Drawing upon the experiences of its predecessors, eFuzion v3 breaks away from its more rigid counterparts by providing developers with an extensible platform and a simple programming model that can be used to easily customize or extend the application. Extensibility is achieved through the use of plug-ins (scripts) written in the Python language coupled with the use of an explicit application state encoding accessible to these scripts. This simple model permits regular users to assume the role of developers and make contributions to the software, having complete control of almost every aspect of the application. As a consequence of the system's modular design, and through the use of the Python scripting language, large portions of this application can be ported to other systems without modification. As an added benefit, this system has been built on top of a framework that is generic enough to permit many other types of applications to be created with it in a similar fashion, being restricted only by one's imagination. All these other applications will benefit from the same extensibility and portability characteristics as eFuzion.

To my beloved mother

ACKNOWLEDGEMENTS

This project would not have been possible without the guidance, patience, and support of my advisor, Professor Sam Kamin, to whom I owe a great debt of gratitude. Many thanks also to Chad Peiper for his useful comments on pedagogical issues. Thanks to the Python.NET and IronPython community for their troubleshooting support. Also thanks to my good friend Dragos Chirila for providing me with time-saving tips regarding the Python language. And finally, thanks to my mother, my brother, and my wife for their support throughout this process.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Introduction

The eFuzion project was started in Fall of 2001 at the University of Illinois at Urbana-Champaign by two undergraduate students, as an experiment meant to explore the use of technology in classrooms. The initial version of eFuzion was used during the Spring 2002 semester in a pilot course (CS 300) at the university, and ran on regular laptop computers. Through the use of a custom-built vector graphics drawing library, the software offered students the ability to take notes and collaborate graphically with the instructor. The instructor was able to load lecture materials on the computer and annotate on them using an early type of pen-based input device. All students in the class received the instructor's annotations on their machines and were able to take their own notes using the keyboard and mouse.

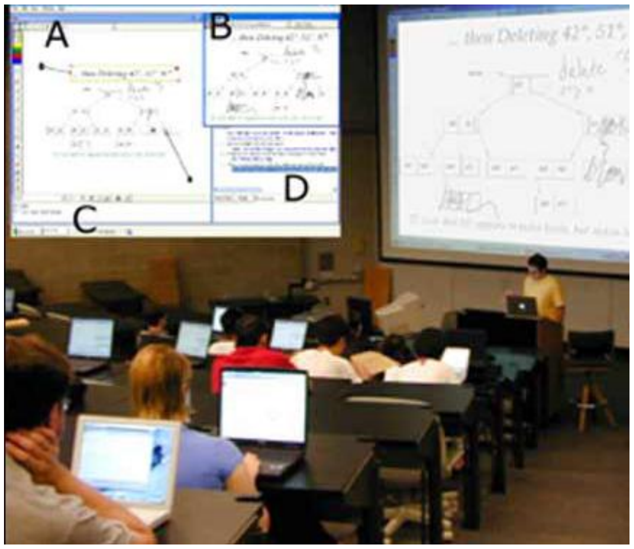


Figure 1.1 - Student's view of eFuzion v1. Areas include: instructor's slides and notes (B), student's clipboard (A), student notes (C), and graphical newsgroup (D)

Figure 1.1 shows a sample of the way the student's screen was partitioned. Additionally, the instructor was able to assess student's performance by sending out polls and quizzes to the class. The pedagogical value of eFuzion was proven during a comparative study conducted in the Summer of 2002 in a live classroom setting when the student's results showed a half letter grade improvement over a similar classroom taught without this technology [24]. This experience also emphasized the importance of pen-input in support of in-class note taking. To track the "under the hood" evolution of this software, this first version was a monolithic application written in the C# [2] language, featuring the aforementioned vector graphics drawing library as its most notable component.

The success of this initial version of eFuzion, coupled with the advent of Tablet PC technology from Microsoft [25] determined the now established Educational Technologies Research Group [7] to create a newer version of eFuzion that would embrace this new digital ink technology, and use it to provide students with a more natural note-taking experience. The major advantage came from the use of the built-in active digitizer that had superior sampling rate and included pressure sensitivity, something that traditional mouse devices lacked. Microsoft's digital ink technology made use of these additional capabilities while also providing vector-based drawing support, thus bringing the new eFuzion much closer to the natural feel of writing to which everyone was so accustomed. Under the hood this version was also a monolithic application that was hard to maintain by other people than its developers, was hard to extend without knowledge of the entire system, and was not portable at all to other non-Microsoft platforms. Still, there was no doubt that this newer version of eFuzion was better: it felt more natural, it incorporated some of the feedback from its users, and fixed a couple of the issues present in the first version. However things weren't as perfect as they could have been. Primarily the problem was that the system was hard to maintain and extend. The user base was very diverse, so there was no fixed set of requirements that could satisfy everyone. Different instructors needed different types of tools to aid their instruction, students weren't cohesive in their demands of the system, and it was completely unfeasible to create an application for each user needing a different feature. It was clear that a different strategy was needed.

The main focus that drove the next version of eFuzion, the one being described in this thesis, was extensibility. The realization was that the best way to please everyone was to empower the user population with the ability to make their own contributions to the software, as they are the ones that best know what they want. The same idea can be found in projects like Emacs [10] or TeX [11] that have enjoyed widespread recognition for their extensible architectures. From the point of view of the classroom "presentation and interaction system" the new eFuzion did not offer more functionality than its predecessors but, most importantly, it offered a solid framework for contributors to implement any desired functionality. Even though extensibility was the most important aspect of eFuzion, it wasn't the only improvement over the previous versions.

Portability was also a key consideration during the design of the framework. To this extent, under the hood the system still used the digital ink technology from Microsoft, but moved away from the traditional monolithic approach and was decomposed into a portable, platform-neutral, part and a non-portable platform specific core. Even though initially the platform-specific core accounted for more than 70% of the code base, it was expected that with the upcoming features and extensions the platform independent part to become the overwhelming majority of the code. This way, if the system needed to be ported to another platform, only the small platform-specific core had to be rewritten for that platform, everything else (the meat of the system) could be reused without modification. As for its most prominent characteristic, extensibility was achieved through a symbiotic combination of explicit state encoding (in XML [9]) and Python [21] scripting support, which made up the platform-neutral part. Everything regarding the system (the way it looked, the way it behaved) was exposed through this explicit state, and could be modified through simple APIs, requiring only an understanding of Python scripting and of the explicit state layout. Among the benefits of this approach was the fact that now extensions could be written, modified, and tested "in the field" without recompilation of the core, in principle requiring nothing more than a simple text editor, for skilled programmers. As an added benefit of using Python scripting, deployment of a newly created extension (script) amounted to just deploying a text file, thus avoiding the security concerns (viruses, Trojans, etc) usually associated with deploying executable files.

As it is well known that every solution has its tradeoffs, the new eFuzion had to accept responsibility for some issues that had not been of concern at the time, but for which possible workarounds exist: from a security standpoint, since the explicit state is publicly modifiable, a script cannot be easily stopped from (un)intentionally corrupting the state. A possible solution can be provided by script signing, such that in a production system if a new script needs to be used it would first have to go through a review authority that will give its stamp of approval if everything seems fine. A second problem exists when uncooperative, conflicting scripts, are used. To alleviate this situation, an "eFuzion scripting guidelines" section is provided in Appendix B that includes useful tips regarding what to do and what not to do when writing a script.

There is still some work to be done until this version can provide the complete feature set of its predecessors, primarily on the network model for which there is currently only a preliminary analysis.

Related Work

Many other universities have bought into the idea that using technology in classrooms will enable students to become more productive and as a result earn higher grades. Some notable research projects include:

UW Classroom Presenter [23] – a project that started at Microsoft Research in 2002 and was continued by University of Washington, which turned it into a valuable resource for networked instruction. The interesting observation about this project is that it relies on the ConferenceXP [3] platform for its network infrastructure, just like the first version of eFuzion.

ReMarkable Texts [22] – a Brown University effort following the same goals of technology-enabled instruction, providing functionality that enables conducting general and domain-specific note-taking research. Its communication infrastructure also relies on the ConferenceXP platform.

MIT InkBoard [16] – although this project is currently retired it is worth mentioning due to its ability to provide collaborative real-time sketching and audio/video conferencing between users. As with the previous two projects, ConferenceXP is used as the network transport mechanism.

All the above projects cash in on the benefits offered by the natural feel of note-taking provided by Tablet PCs.

Commercially, the only product that's directly geared towards live, in-class, collaboration is offered by DyKnow [6]. It, too, started as a research project and was initially developed at DePauw University.

Considering the technical approaches taken by eFuzion, related work can include projects like Emacs and TeX that rely heavily on scripting to achieve extensibility. With respect to the explicit state encoding in XML used by eFuzion, the closest related efforts existing include the XUL [17] platform from Mozilla, the XAML [8] platform from Microsoft (will be available in the upcoming Windows Vista product), and XUI [26] – a Java based open-source framework. The major difference between these projects and the approach taken by eFuzion is that eFuzion actually updates the XML representation of the state whenever its internal state changes, while all these other projects are more geared toward *defining* user interface elements and associating actions to them, but not updating their XML representation when internal changes occur.

CHAPTER 2: MOTIVATION

Motivation

The results of the comparative study done in the early stages of eFuzion motivated the Educational Technology Group to continue the research pioneered by eFuzion, fueled by a growing interest from the user population (instructors and students) and the underutilization of the potential of the Tablet PC platform. As a consequence, many of the traditional pedagogical models have been studied and, where possible, enhanced through the use of technology. As an example, it is well known that student engagement in classroom activities depends a lot on the student's ability to establish direct contact with the instructor and peers. The traditional collaboration model does not make it easy for students (especially in large classrooms) to share ideas with each other or easily address issues concerning the materials presented by the instructor. This is one area where eFuzion has concentrated a lot of effort: providing students with an unprecedented ability to collaborate and engage in classroom activities, as well as creating a direct communication channel between the students and faculty, through the use of a graphical newsgroup.

From a technical point of view the motivation behind the design of eFuzion v3 included the need for complete extensibility and potential for portability. The choice of Python as the scripting language for coding eFuzion extensions took into consideration the dynamic nature of the language, its widespread availability, and publicly recognized performance characteristics. Since the extensions needed to communicate with the application core, Python needed a way to interface with the C# language. This was done initially through Python.NET [20], an experimental bridge that allowed the passing of objects and method calls from one language to the other, and later through IronPython [12], a new project from Microsoft that provided more intimate ties between Python and .NET [1].

Design Considerations

The research performed during the development of the first two iterations of eFuzion together with the analysis of the feedback obtained from the users, led to the conclusion that the fluid nature of the user requirements makes a rigid design impractical. For this reason a different approach was needed that could easily adapt to the requirements fluctuation, while still providing the indispensible set of features present in the prior versions. The resulting design comes a long way towards achieving these goals, having as the central component a very generic framework that could be used as the foundation for many different types of applications. The only reason this application is a classroom presentation and interaction system is because of the scripts determining it to behave as such. The advantage of this approach lies in the fact that the key properties (extensibility and portability) are preserved irrespective of the type of application built on top of this framework.

Extensibility

The need for extensibility grew out of the understanding that a fixed set of features are insufficient to make everybody happy. Given that the goal of this application was to become a very useful educational tool, it was concluded that many of the most meaningful extensions to this application may come as contributions from the users themselves, as they are the ones most likely to discover what they need the application to do, how it should look, and what the interaction with it should be like. In order to facilitate the scenario where the users can assume the role of developers the application needed to provide a quick way for contributors to make customizations and add desired functionality. Scripting was the obvious choice.

After the decision was made to allow everyone to make contributions to this application via scripting, the next step was to decide on how the scripts were to interact with the application and modify its state. Having a rigid API that provides very good sandboxing properties is beneficial from a security / stability standpoint, however it creates hard limits around what the user can do, ultimately creating a tight leash around the user's imagination. As that contradicted the principle of allowing the user freedom of expression, it was decided to take the opposite approach and give the user full control over the application by exposing the entire application state publicly in XML form, and providing the means for scripts to make modifications to this state. As a consequence users can customize any part of the system they need and add functionality in an unrestricted way, provided that certain conventions regarding the XML state are followed. Since this is a research project, concerns regarding security and stability were put aside to achieve maximum extensibility. The goal was to see if this model was viable and if it indeed allows for a faster / better application customization experience, while harvesting the benefits of "on-the-fly" scripting and high portability.

Portability

As educational technology gains momentum in the race towards a more efficient educational system the underlying technology evolves continuously at a very rapid pace. It may not be far into the future when the laptops of today, weighing a healthy couple of pounds and only providing few hours of battery life, will be replaced by pocket-size full-featured mini computers that have more memory and processing power than many desktop computers today. It is equally plausible to think that many of the technologies that currently exist only on proprietary platforms will be made available to other (potentially open source) operating systems.

Even if that is not the case, it is still possible to design the application in a way that allows interoperation across different systems. For example, eFuzion currently relies heavily on the Microsoft inking technology, which is not available outside the Windows platform. This essentially locks eFuzion into being available only on Windows machines loaded with the Tablet PC SDK. However, in order to have access to a more diversified user base, it was decided that allowing the possibility of running eFuzion on non-Microsoft platforms should become a requirement. As a consequence the application had to be factored out into a non-portable, platform-specific part, and a portable, platform-independent part. The separation was done in a way that allowed for maximal code reuse across platforms. Python was chosen as the driving force behind the platform independent part, for its performance, portability, and dynamism. Combined with the explicit access to the XML representation of the application state (which is not platform-dependent), interoperation between different platforms could, at least theoretically, be achieved. For example, if it was desired to port this application to the UNIX environment, only the platform-specific part would need to be re-written (in Java, for example). Inking information could be rendered on the screen as a series of connected points at the coordinates encoded in the XML state, applying the specified thickness and color, while safely ignoring all other attributes that might not be reproducible without the Microsoft ink technology. One might wonder why Java was not used from the beginning, and the answer to that is that Microsoft ink technology was providing the most natural note-taking experience, and it was available only on the .NET platform.

Scalability

The discussion about scalability only makes sense in the context of networking, for which there is only a preliminary analysis. The prior versions of eFuzion have tried different approaches for the network transport layer: eFuzion v1 relied on ConferenceXP, a research project from Microsoft that provides the infrastructure needed to create distributed applications, while eFuzion v2 has resorted to a more traditional unicast-based transmission method. Both these solutions had problems that needed to be addressed: ConferenceXP was placing stringent constraints regarding the type of networking infrastructure needed (it required access to an Internet2 backbone), and it confined the application to a specific model that was not easy to work with. The unicast-based transmission employed by eFuzion v2 had a different problem: it did not scale well. As the number of clients increased, so did the load on the server machine. Since updates had to be sent separately to each client, the server had to work harder every time a new client was added to the session, and the application performance degraded correspondingly. The problem was compounded by the fact that all clients were connected wirelessly to the network, and due to the message replication the wireless medium was overloaded with data transfers, resulting in many dropped connections.

Based on the above observations the conclusion was that for eFuzion v3 a different network transport method has to be used. It was clear that unicast-based transmission will always result in a system that's not scalable, so multicast solutions were investigated. The difficulty in using traditional multicast comes from the fact that multicast is an unreliable, unguaranteed, connectionless protocol. In order for multicast to be useful for eFuzion, delivery guarantees needed to be added to the protocol, but this proved to be a rather time-consuming task. Consequently attention turned towards existing solutions for reliable multicast, and the salvation came in the form of the PGM protocol for which there was built-in support on the Windows platform. Experiments using this technology showed it being far superior to anything else used in the past, although it only alleviated the problem, not solved it completely. The reason is that while instructor-generated notes that needed to be sent to all the connected clients could use this mechanism for transmission very efficiently, if the reverse situation was needed (all clients to send their work to a server for recording, for example) then PGM would not provide any relief as all clients had to use the wireless medium to send their data to the server. If this situation is unacceptable for eFuzion, then this problem needs to be tackled at a different level, perhaps reconsidering the business rules of the application.

CHAPTER 3: SYSTEM ARCHITECTURE

Overview

The general idea and motivation behind the design of eFuzion v3 has been presented in the previous chapters. This chapter expands on that knowledge and provides an in-depth view under the hood of eFuzion v3.

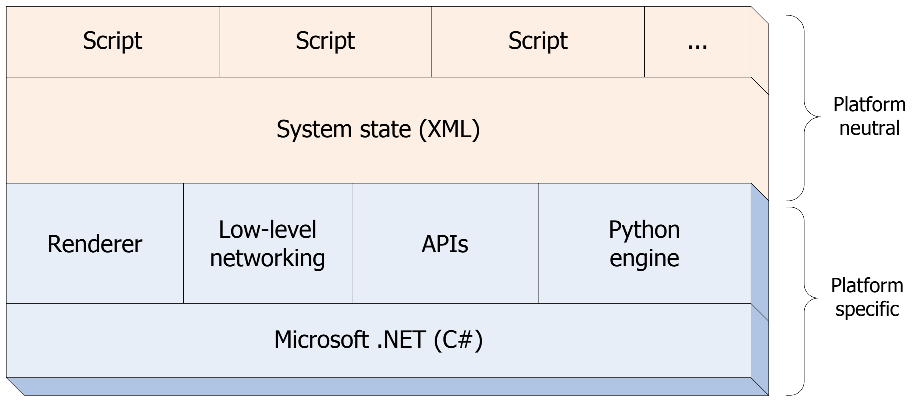


Figure 3.1 - High-level view of the system architecture

Figure 3.1 shows a high-level view of the system architecture, which consists of two parts: the platform-specific and platform-neutral parts. The platform-specific part constitutes the core of eFuzion: it includes components such as the renderer, low-level networking, Python engine, and extensibility APIs. The platform-neutral part is described by the explicit system state encoding in XML, and the collection of Python scripts that modify this state. The primary responsibility of the renderer is to update the display based on the information stored in the system state. Python scripts have access to modify this state through a set of generic XML-manipulating functions defined in the API. This allows for maximum extensibility as scripts can modify anything pertaining to the system state. Scripts can be loaded and unloaded from memory as needed because the system state includes the local state of each script, and removing a script from memory amounts to just removing its associated sections from the XML describing the state. Scripts are notified of user events by registering their interest with the objects triggering those events through the addition of the corresponding properties to the XML node describing those objects. Communication between scripts and the system core happens through the Python engine, which interfaces C# to the Python language.

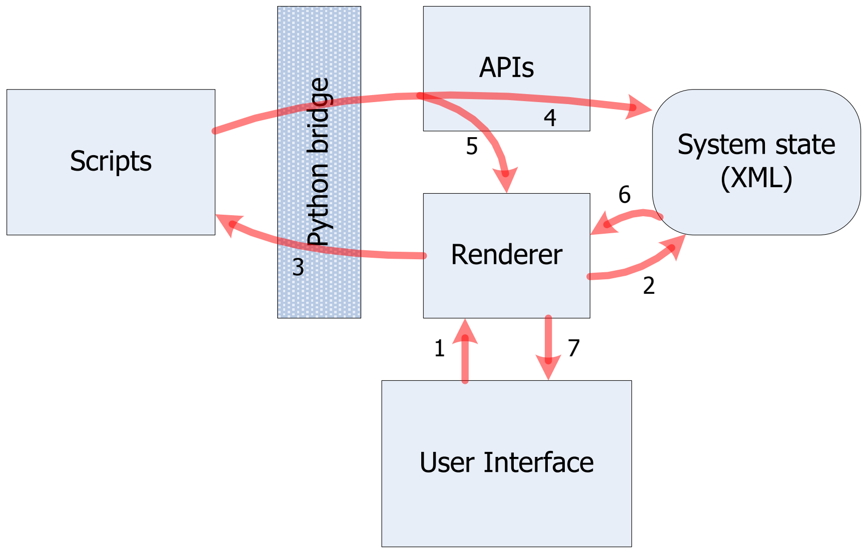


Figure 3.2 - System flow diagram

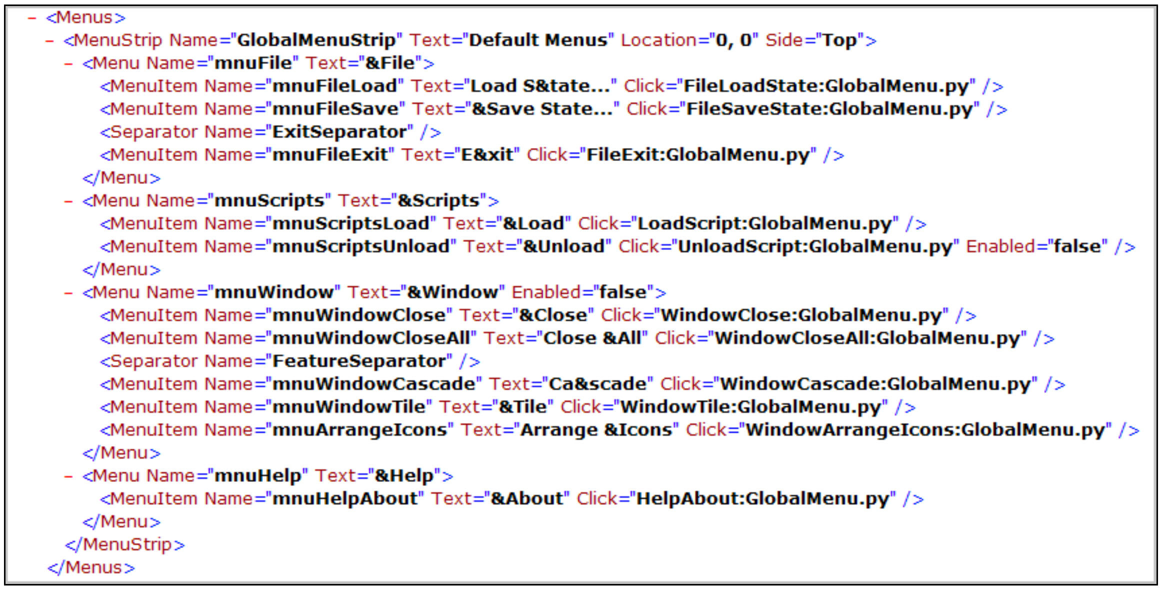
Figure 3.2 shows the communication paths between the different components as messages pass through the system. Events originating from the user interface are captured by the renderer (1), which in turn updates the system state accordingly (2). If any scripts have registered interest in the handling of these events, the renderer invokes the appropriate script functions through the Python bridge (3). In response to these events scripts may want to update the user interface, and to do so they use API functions to make the necessary modifications to the system state (4). Next, also using functions published in the API, they indicate to the system that rendering needs to be performed

(5). The renderer, in turn, reads the modifications in the state (6) and updates the user interface accordingly (7).

At startup, the application initializes the Python engine and loads the initial state of the system from an XML file with the same name as the application executable. The renderer is then invoked to "project" the logical system state onto the screen. Afterwards the application enters a wait state, expecting commands from the user. The system state can be saved explicitly into an XML file any time during the operation of the application, much like the way Windows saves state in preparation for hibernation. This way the application can be resumed at a later time, from the same state where it was left off.

System State

The need to encode the system state explicitly has been justified by the desire for extensibility. The decision to use XML as the mechanism to encode the system state was made in line with the necessity for portability. This XML representation of the state lives in the application memory while the application is running, but can be saved to permanent storage if desired. The structure of the state is dictated by the hierarchical organization of XML. Since eFuzion is centered on its user interface, this organization maps naturally to the way visual elements are constructed. As a consequence, anything that is visible on the screen pertaining to eFuzion can be traced back to a node in the XML state – this includes the menu, toolbars, ink, frames (windows), etc. For example, Figure 3.3 shows the way menus are encoded in the XML representation of the initial state of the system at startup.



```xml
- <Menus>
  - <MenuStrip Name="GlobalMenuStrip" Text="Default Menus" Location="0, 0" Side="Top">
    - <Menu Name="mnuFile" Text="&File">
        <MenuItem Name="mnuFileLoad" Text="Load S&tate..." Click="FileLoadState:GlobalMenu.py" />
        <MenuItem Name="mnuFileSave" Text="&Save State..." Click="FileSaveState:GlobalMenu.py" />
        <Separator Name="ExitSeparator" />
        <MenuItem Name="mnuFileExit" Text="E&xit" Click="FileExit:GlobalMenu.py" />
      </Menu>
    - <Menu Name="mnuScripts" Text="&Scripts">
        <MenuItem Name="mnuScriptsLoad" Text="&Load" Click="LoadScript:GlobalMenu.py" />
        <MenuItem Name="mnuScriptsUnload" Text="&Unload" Click="UnloadScript:GlobalMenu.py" Enabled="false" />
      </Menu>
    - <Menu Name="mnuWindow" Text="&Window" Enabled="false">
        <MenuItem Name="mnuWindowClose" Text="&Close" Click="WindowClose:GlobalMenu.py" />
        <MenuItem Name="mnuWindowCloseAll" Text="Close &All" Click="WindowCloseAll:GlobalMenu.py" />
        <Separator Name="FeatureSeparator" />
        <MenuItem Name="mnuWindowCascade" Text="Ca&scade" Click="WindowCascade:GlobalMenu.py" />
        <MenuItem Name="mnuWindowTile" Text="&Tile" Click="WindowTile:GlobalMenu.py" />
        <MenuItem Name="mnuArrangeIcons" Text="Arrange &Icons" Click="WindowArrangeIcons:GlobalMenu.py" />
      </Menu>
    - <Menu Name="mnuHelp" Text="&Help">
        <MenuItem Name="mnuHelpAbout" Text="&About" Click="HelpAbout:GlobalMenu.py" />
      </Menu>
    </MenuStrip>
  </Menus>
```

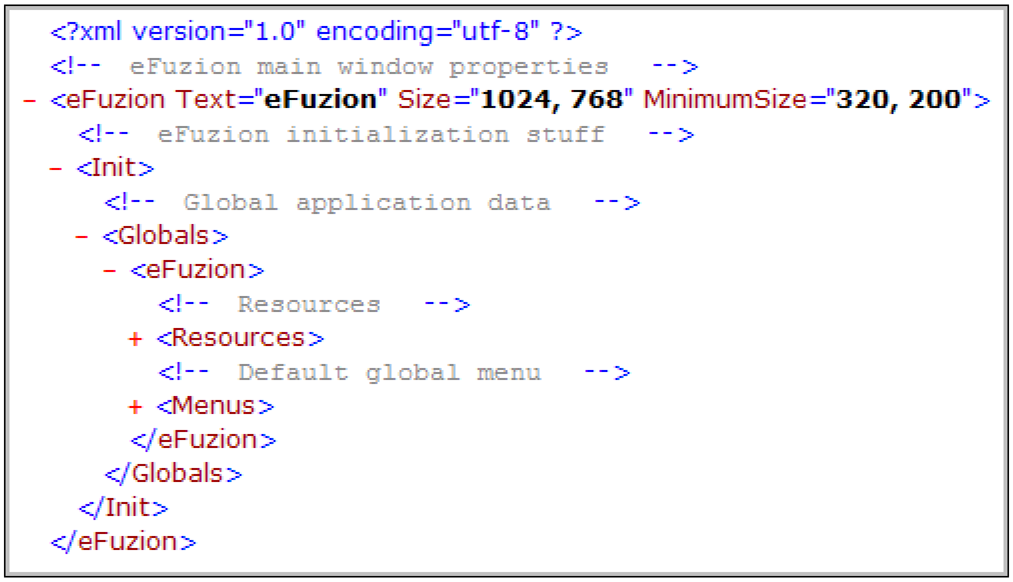
Figure 3.3 - XML encoding of the menus in the initial state of eFuzion

As has been mentioned before, the framework employed by eFuzion is very generic and allows the creation of many types of extensions on top of it. These extensions live inside scripts that can be loaded on top of the framework at any time. The framework allows multiple extensions to coexist at the same time. Currently the XML file containing the initial state of the system at startup defines only the functionality necessary for this framework to be usable. This includes functionality to load scripts, unload scripts, manage windows, save and load state, and exit. There is nothing in there regarding the "presentation and interaction" aspects of the educational application; in order to access that functionality the extension (script) providing it must be first loaded. Chapter 4 will present this extension in more detail.

Some of the objects defined in the system (the system includes the framework and any extensions loaded) generate events when triggered (such as buttons being clicked, ink strokes being completed, windows being resized, etc.). The XML state contains mappings from events to functions defined in scripts that get executed in response to these events. In Figure 3.3, for example, when the user clicks the menu item called "Load State… " the function "FileLoadState" defined inside GlobalMenu.py is executed. The labeling of many of the XML nodes in the state closely follows the naming of the corresponding visual elements from .NET. This allows developers to easily track the places inside the XML state where the visual elements they're interested in are defined. Furthermore, the labels used for describing events that objects can trigger correspond to the names of the events published by the .NET counterparts of these objects.

The state is not made up only of elements that have a visual representation; it can include partial results of calculations, the values of counters, and many other things applications might need to perform their bookkeeping. In fact anything that can be considered "state" should be encoded in the XML. The way the XML state is structured allows great flexibility regarding the place where extensions can store their local state. Only visual elements need to follow certain rules that ensure that the renderer can find them and display them on the screen.

The best way to explain the rules governing the structure of the state is by showing some examples of how the state information is organized in XML. Because the full XML hierarchy takes up a lot of space, it will be unfolded progressively as the discussion proceeds. Figure 3.4 shows the XML description of the initial state at application startup, with no extensions yet loaded. The root element describes the properties of the main form of the application. In accordance with the .NET structure for the Form class, the attributes defined in the XML map directly to the attributes of the form. Under the hood reflection is used to implement this mapping. This allows developers maximum flexibility in setting the properties of visual objects. The main application form is an MDI container that hosts the frames associated with extensions. A frame is an MDI child window defined by an extension. An extension can define multiple frames if needed.

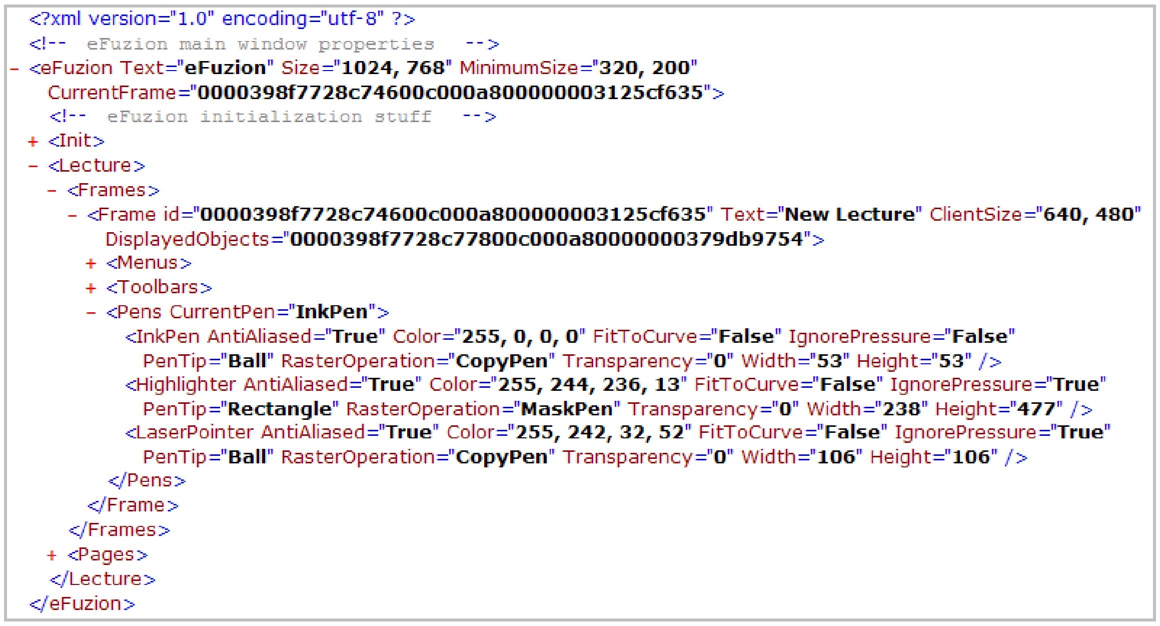


```
  <?xml version="1.0" encoding="utf-8" ?>
  <!--   eFuzion main window properties   -->
- <eFuzion Text="eFuzion" Size="1024, 768" MinimumSize="320, 200">
    <!--   eFuzion initialization stuff   -->
  - <Init>
      <!--   Global application data   -->
    - <Globals>
      - <eFuzion>
          <!--   Resources   -->
        + <Resources>
          <!--   Default global menu   -->
        + <Menus>
        </eFuzion>
      </Globals>
    </Init>
  </eFuzion>
```

Figure 3.4 - Initial XML state of eFuzion

The "Init" section of the XML contains information used during application initialization at startup; it contains a "Globals" node where extensions can optionally define resources, menus, and toolbars to be used globally, and a "Scripts" node that allows the application to keep track of the loaded scripts (not shown as there are no scripts loaded initially).

The evolution of the system state after the "Lecture" extension was loaded can be seen in Figure 3.5. The "Init" section of the state has been collapsed since its purpose was already explained. The "Lecture" section of the XML state contains information specific to the "Lecture" extension. If the extension needs to create a new frame inside the main application space (as most extensions will) then the "Frames" subsection needs to contain the definition of the frame.

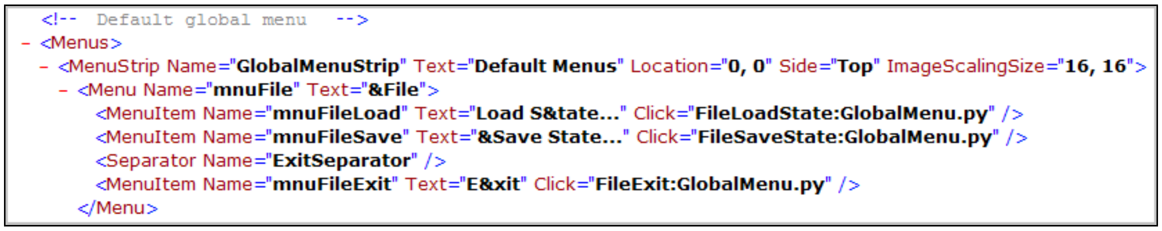


```
  <?xml version="1.0" encoding="utf-8" ?>
  <!--  eFuzion main window properties   -->
- <eFuzion Text="eFuzion" Size="1024, 768" MinimumSize="320, 200"
    CurrentFrame="0000398f7728c74600c000a800000003125cf635">
    <!--  eFuzion initialization stuff   -->
  + <Init>
  - <Lecture>
    - <Frames>
      - <Frame id="0000398f7728c74600c000a800000003125cf635" Text="New Lecture" ClientSize="640, 480"
          DisplayedObjects="0000398f7728c77800c000a80000000379db9754">
        + <Menus>
        + <Toolbars>
        - <Pens CurrentPen="InkPen">
            <InkPen AntiAliased="True" Color="255, 0, 0, 0" FitToCurve="False" IgnorePressure="False"
              PenTip="Ball" RasterOperation="CopyPen" Transparency="0" Width="53" Height="53" />
            <Highlighter AntiAliased="True" Color="255, 244, 236, 13" FitToCurve="False" IgnorePressure="True"
              PenTip="Rectangle" RasterOperation="MaskPen" Transparency="0" Width="238" Height="477" />
            <LaserPointer AntiAliased="True" Color="255, 242, 32, 52" FitToCurve="False" IgnorePressure="True"
              PenTip="Ball" RasterOperation="CopyPen" Transparency="0" Width="106" Height="106" />
          </Pens>
        </Frame>
      </Frames>
    + <Pages>
    </Lecture>
  </eFuzion>
```

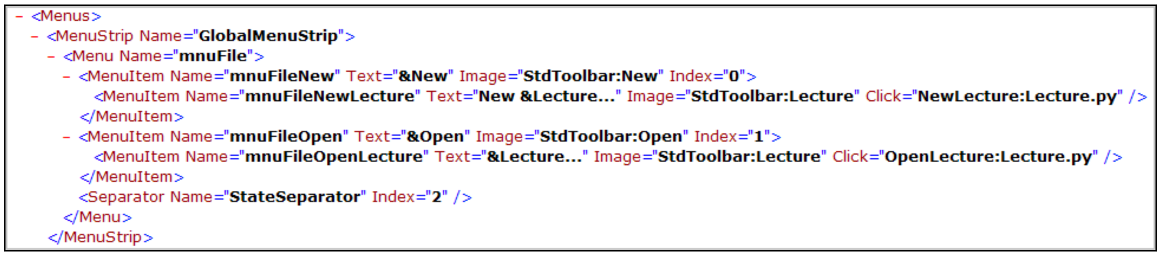
Figure 3.5 - System state after loading the "Lecture" script

The use of the "CurrentFrame" attribute on the root node of the XML state indicates that the frame corresponding to the referenced id is the currently focused frame of the application. If an extension defines more than one frame, switching between them programmatically happens by changing the value of the "CurrentFrame" attribute to point to the desired frame. Assigning an "id" value to the frame is the responsibility of the extension. Ids can be generated using the supplied "generate" function defined in the file GUID.py. The "DisplayedObjects" property is a renderer-required attribute of the frame whose role is to indicate the place in the XML state which describes what should be displayed in the frame. How and when the renderer makes use of this attribute will be covered in detail in the section titled "Renderer".

There are two levels of menus and toolbars: global and frame-specific. The way they are differentiated is through their placement inside the XML hierarchy. Items defined in the global menu (or toolbar) are available even when extensions are loaded and running. For example, the load state and save state functionality defined as global in the framework (see Figure 3.3 which shows the expanded "Menus" section from Figure 3.4) can be accessed at any time while eFuzion is running, even if other extensions have been loaded. The default global menu used by the framework is defined in the "eFuzion" subnode. Extensions can add to this menu (and/or create global toolbars) by adding a node labeled with the extension name as a child of the "Globals" node. Figure 3.6 (top) shows the default definition of the menu called "GlobalMenuStrip" with its sub-items (figure content part of the Init->Globals->eFuzion node). The bottom of the same figure shows the situation where an extension was loaded that adds two items to "GlobalMenuStrip" (figure content part of the Init->Globals->Lecture node).



```
  <!--  Default global menu   -->
- <Menus>
  - <MenuStrip Name="GlobalMenuStrip" Text="Default Menus" Location="0, 0" Side="Top" ImageScalingSize="16, 16">
    - <Menu Name="mnuFile" Text="&File">
        <MenuItem Name="mnuFileLoad" Text="Load S&tate..." Click="FileLoadState:GlobalMenu.py" />
        <MenuItem Name="mnuFileSave" Text="&Save State..." Click="FileSaveState:GlobalMenu.py" />
        <Separator Name="ExitSeparator" />
        <MenuItem Name="mnuFileExit" Text="E&xit" Click="FileExit:GlobalMenu.py" />
      </Menu>
```

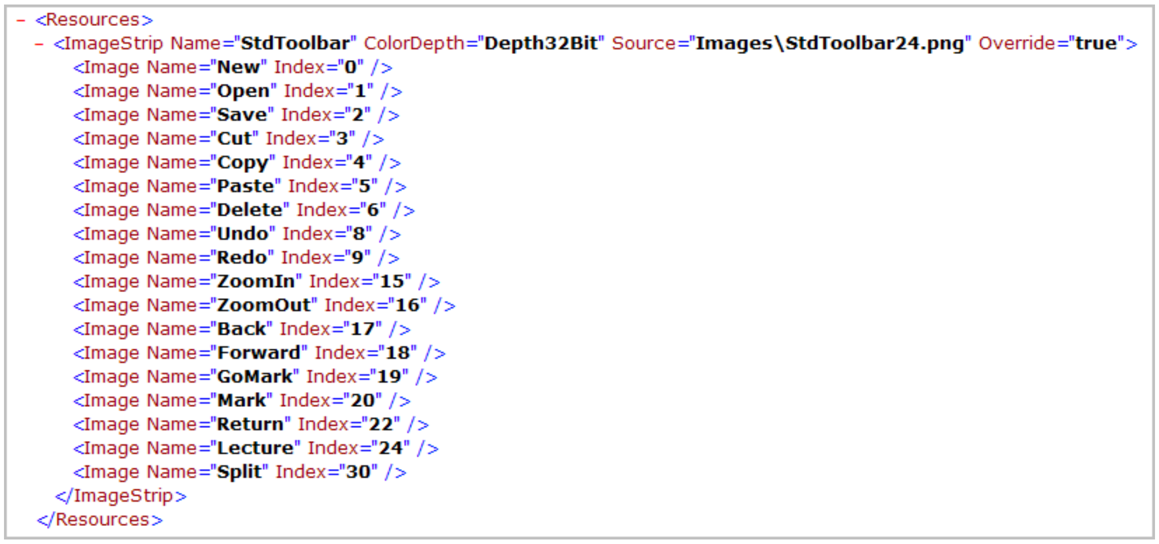


```
- <Menus>
  - <MenuStrip Name="GlobalMenuStrip">
    - <Menu Name="mnuFile">
      - <MenuItem Name="mnuFileNew" Text="&New" Image="StdToolbar:New" Index="0">
          <MenuItem Name="mnuFileNewLecture" Text="New &Lecture..." Image="StdToolbar:Lecture" Click="NewLecture:Lecture.py" />
        </MenuItem>
      - <MenuItem Name="mnuFileOpen" Text="&Open" Image="StdToolbar:Open" Index="1">
          <MenuItem Name="mnuFileOpenLecture" Text="&Lecture..." Image="StdToolbar:Lecture" Click="OpenLecture:Lecture.py" />
        </MenuItem>
        <Separator Name="StateSeparator" Index="2" />
      </Menu>
    </MenuStrip>
```

Figure 3.6 - Default menu definition (top), extension-augmented definition (bottom)

Menu (or toolbar) entries can be added to the global menu (or toolbar) definition, but they cannot override the behavior of the existing entries. Each entry is identified by a name. That means that since there is a menu item named "mnuFileLoad" as a child of "mnuFile" inside "GlobalMenuStrip", an extension could not redefine its meaning by creating a similar structure inside Init->Globals-> <extension name> and binding it to a different action. When making additions to an existing structure the position where the new item is inserted can be specified by setting the "Index" property to achieve the desired item order. Frame-specific menus and/or toolbars can be optionally defined as part of each frame. These have a temporary nature because they are removed once the parent frame is closed. When switching between frames the menu (and toolbars) change dynamically depending on what has been defined under the "Menus" and "Toolbars" sections of each frame. These sections can define items to be merged with existing structures.

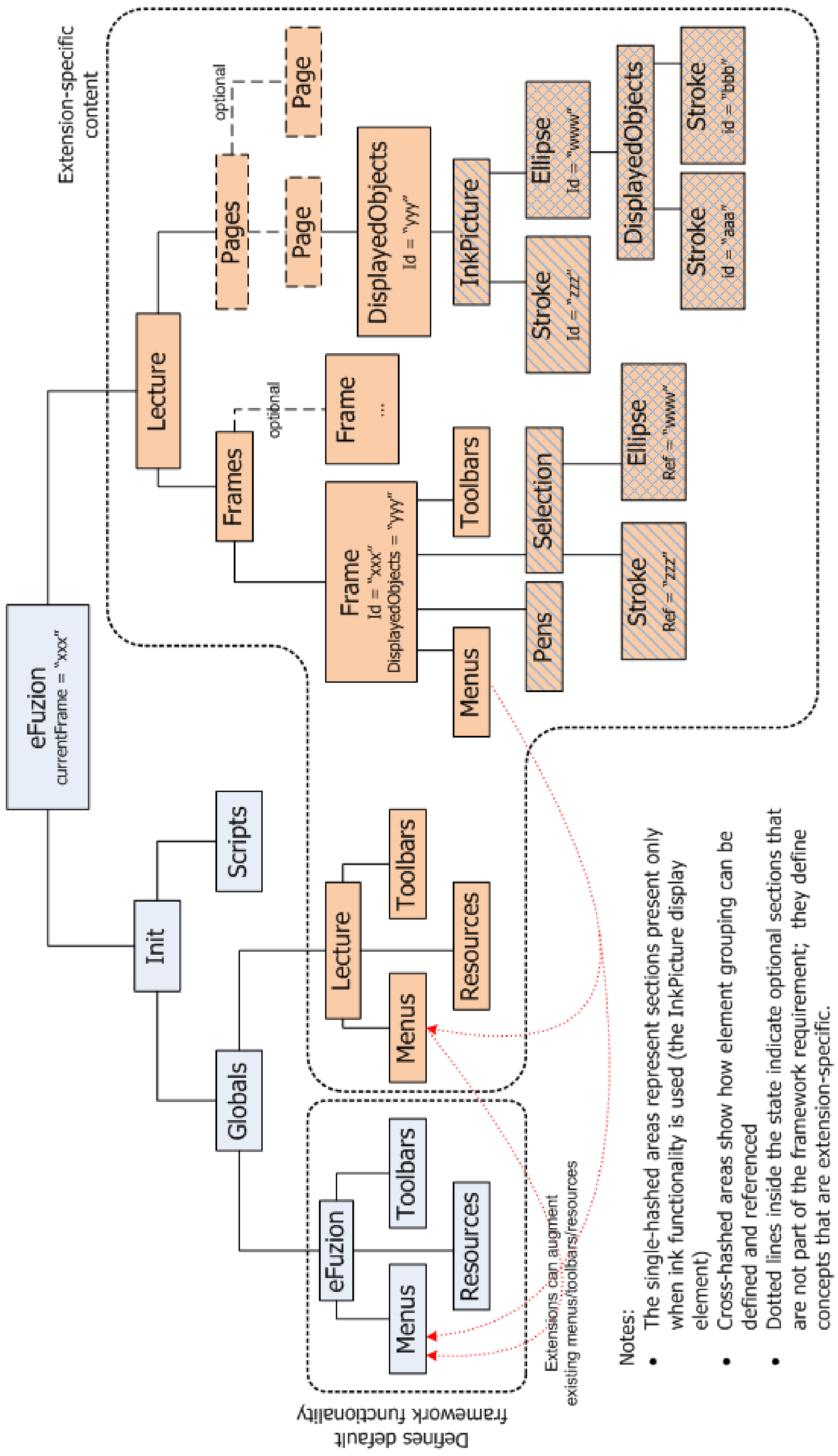
The "Resources" section provides developers with a way to define resources that will be used by extensions. The only type of resource currently supported is ImageStrip (mostly used for menus and toolbars), which can be defined as in Figure 3.7. The file specified in the "Source" attribute contains a series of images of same size on a horizontal strip. The child nodes of "ImageStrip" create labels for these images so they can be used later. The way to reference the images is by specifying the name of the image strip, followed by a colon, followed by the name of the image. Examples of this can be seen in the bottom part of Figure 3.6. The optional "Override" attribute indicates



```
- <Resources>
  - <ImageStrip Name="StdToolbar" ColorDepth="Depth32Bit" Source="Images\StdToolbar24.png" Override="true">
      <Image Name="New" Index="0" />
      <Image Name="Open" Index="1" />
      <Image Name="Save" Index="2" />
      <Image Name="Cut" Index="3" />
      <Image Name="Copy" Index="4" />
      <Image Name="Paste" Index="5" />
      <Image Name="Delete" Index="6" />
      <Image Name="Undo" Index="8" />
      <Image Name="Redo" Index="9" />
      <Image Name="ZoomIn" Index="15" />
      <Image Name="ZoomOut" Index="16" />
      <Image Name="Back" Index="17" />
      <Image Name="Forward" Index="18" />
      <Image Name="GoMark" Index="19" />
      <Image Name="Mark" Index="20" />
      <Image Name="Return" Index="22" />
      <Image Name="Lecture" Index="24" />
      <Image Name="Split" Index="30" />
    </ImageStrip>
  </Resources>
```

Figure 3.7 - An example of resource definition

whether extensions are allowed to replace this particular resource (default is “false”)



Notes:

- The single-hashed areas represent sections present only when ink functionality is used (the InkPicture display element)
- Cross-hashed areas show how element grouping can be defined and referenced
- Dotted lines inside the state indicate optional sections that are not part of the framework requirement; they define concepts that are extension-specific.
- Framework-defined sections are shown in solid color with continuous borders

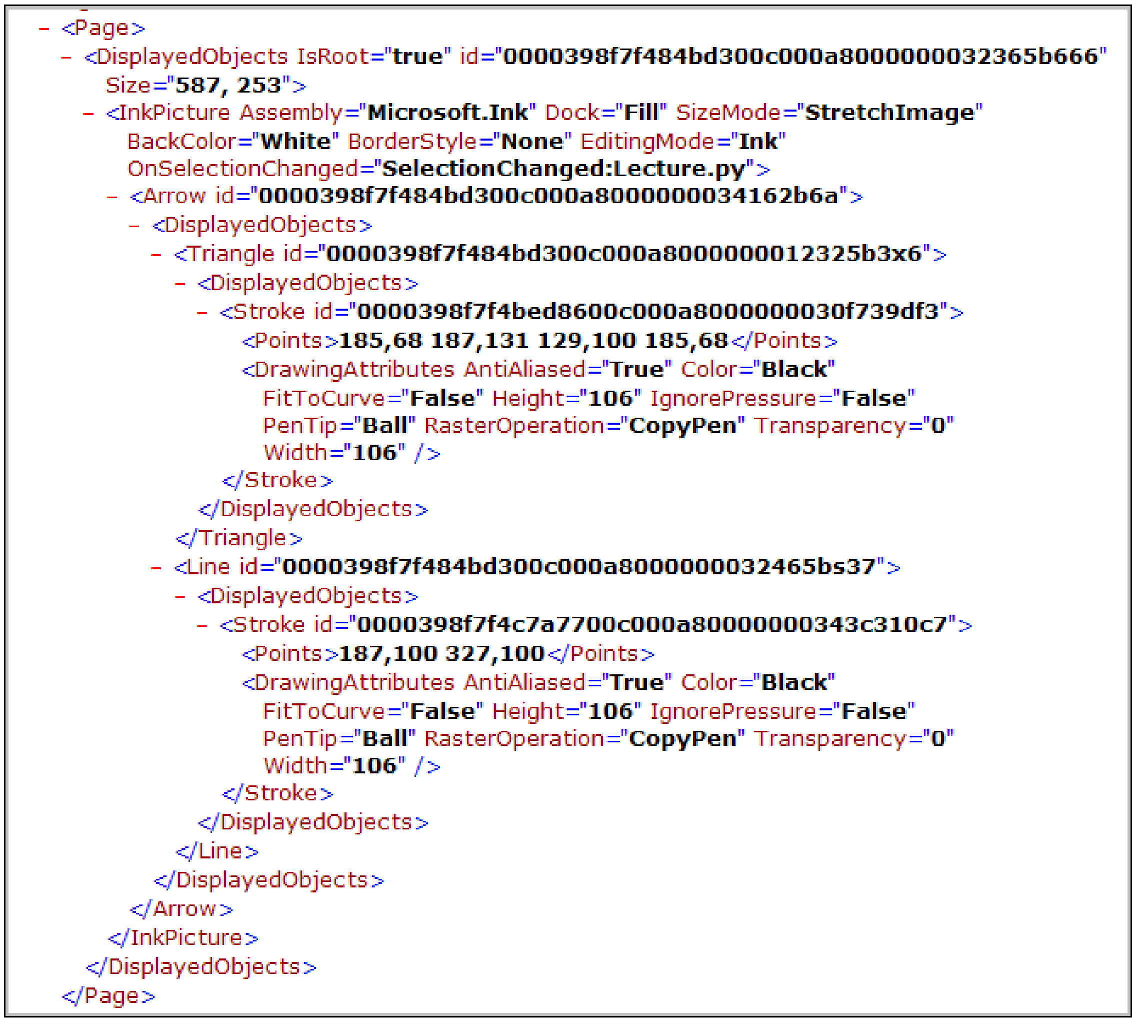
Figure 3.8 – System state organization diagram

As a summary of everything described so far regarding the explicit XML state layout, a diagram of the most important sections and attributes is presented in Figure 3.8. The section(s) descending from "Init" are not required for the application to be able to execute, but without them the functionality for loading/unloading of scripts and managing the MDI frames will be missing. When extensions are loaded by the framework, the extension-specific content is added to the XML state as depicted by the figure. It is not required that extensions define entries in the "Init->Globals" subtree. That section is used only when extensions want to add menu and/or toolbar items to the global menu/toolbar scheme, and when extension-specific resources need to be registered for later reference. Every extension has to have a unique name. The system denies the loading of an extension if another extension with the same name has been previously loaded. Part of the extension-specific content is the section titled "Lecture" on the right side of the figure. This section is optional unless the extension needs to provide any visual elements other than global menus/toolbars. When it is defined, the most notable subsections of it are the sections titled "Frames" and "DisplayedObjects". The "Frames" subsection is used to hold information regarding the frames (windows) that the extension uses to interact with the user. Each frame can optionally contain definitions for local menus and toolbars that are displayed only in association with that frame. Extensions define what should be rendered on top of the frame by adjusting the "DisplayedObjects" attribute to point to the XML section (also titled "DisplayedObjects") describing the desired content. There is no requirement of where in the XML state this section needs to be defined; extensions can organize those sections in any logical manner that makes sense to them. For example, in the right side of the diagram of Figure 3.8 the "Lecture" extension defines a frame that is used to host the contents of a document. The concept of pages is used to organize the data in the document. Each page contains an inkable surface (InkPicture [13]) that can display strokes and other data. When users navigate through the pages of the document, the document frame is updated to reflect the content of the current page. In order to implement this model the most logical placement of the "DisplayedObjects" section was as part of each page. Whenever page switching needs to occur, all that the script needs to do in order to show the content of the new page in the document frame is to point the "DisplayedObjects" attribute of the frame to the "DisplayedObjects" section corresponding to the new page, and then notify the renderer to update the display.

The last notable comment regarding the organization of the XML state is about custom-defined (composite) visual elements. Custom-defined elements are objects that the renderer cannot display without being told what they are composed of, in terms of the basic visual objects the renderer knows about. The renderer is told how to render composite elements by consulting the "DisplayedObjects" section descending immediately from the node describing the element. For example, in the diagram of Figure 3.8 the element titled "Ellipse" is one such composite object. The renderer can render an ellipse by displaying the two strokes that are composing it. Complex objects can be rendered by grouping different custom elements together. Strokes are the most basic graphical elements. Any shape can be constructed by grouping strokes appropriately. However, the renderer does not only know how to render strokes; it can render many other traditional visual elements taken from the .NET platform, as will be shown in the next section.

Renderer

The primary function of the renderer is to generate visual objects that can be displayed on the screen from the definition of the elements in the XML state. Whenever the renderer encounters an element that it does not know how to display, it looks inside the element definition for a “DisplayedObjects” section that defines what this element is composed of in terms of basic objects that the renderer knows about. This procedure continues recursively until all the basic components of the unknown element are displayed. Figure 3.9 shows the situation where the custom element “Arrow” is defined



```
- <Page>
  - <DisplayedObjects IsRoot="true" id="0000398f7f484bd300c000a8000000032365b666"
      Size="587, 253">
    - <InkPicture Assembly="Microsoft.Ink" Dock="Fill" SizeMode="StretchImage"
        BackColor="White" BorderStyle="None" EditingMode="Ink"
        OnSelectionChanged="SelectionChanged:Lecture.py">
      - <Arrow id="0000398f7f484bd300c000a8000000034162b6a">
        - <DisplayedObjects>
          - <Triangle id="0000398f7f484bd300c000a8000000012325b3x6">
            - <DisplayedObjects>
              - <Stroke id="0000398f7f4bed8600c000a8000000030f739df3">
                  <Points>185,68 187,131 129,100 185,68</Points>
                  <DrawingAttributes AntiAliased="True" Color="Black"
                    FitToCurve="False" Height="106" IgnorePressure="False"
                    PenTip="Ball" RasterOperation="CopyPen" Transparency="0"
                    Width="106" />
                </Stroke>
              </DisplayedObjects>
            </Triangle>
          - <Line id="0000398f7f484bd300c000a8000000032465bs37">
            - <DisplayedObjects>
              - <Stroke id="0000398f7f4c7a7700c000a80000000343c310c7">
                  <Points>187,100 327,100</Points>
                  <DrawingAttributes AntiAliased="True" Color="Black"
                    FitToCurve="False" Height="106" IgnorePressure="False"
                    PenTip="Ball" RasterOperation="CopyPen" Transparency="0"
                    Width="106" />
                </Stroke>
              </DisplayedObjects>
            </Line>
          </DisplayedObjects>
        </Arrow>
      </InkPicture>
    </DisplayedObjects>
  </Page>
```

Figure 3.9 - Example of custom object definition

as being composed of a triangle and a line (both being custom objects as well). The output from the renderer corresponding to this example is shown in Figure 3.10.

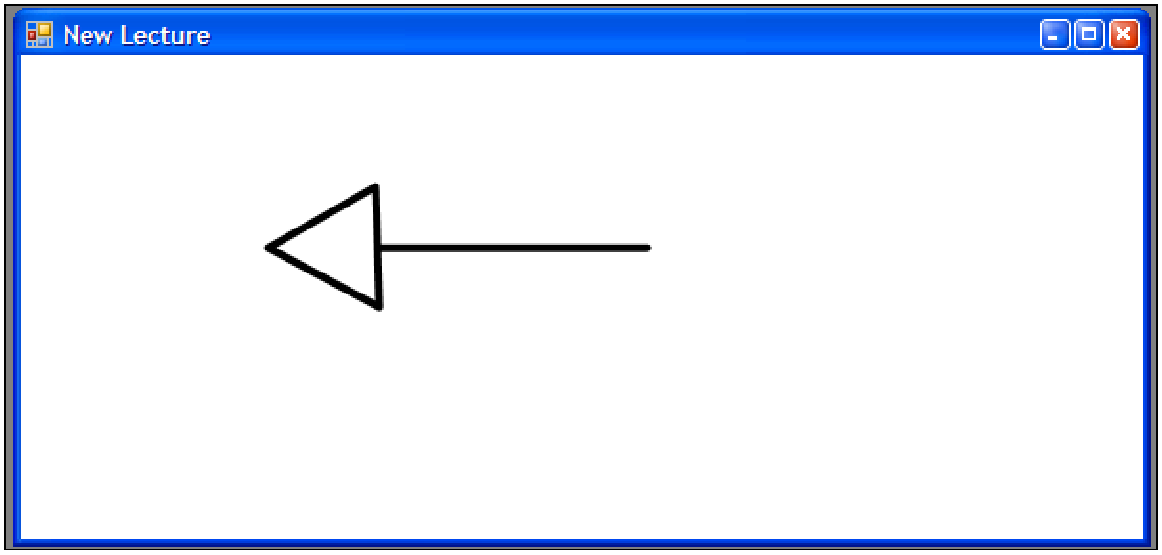


Figure 3.10 - Renderer output based on state shown in Figure 3.9

This type of grouping allows the user to select and manipulate the entire arrow as a whole object rather than through its component elements.

The second responsibility of the renderer is to update the XML state in response to events generated by the user interface. The user interface is composed of a collection of basic visual elements that exist in most GUI platforms (such as .NET, Java, Tk, etc). All these elements have some standard actions that are triggered in response to being manipulated by the user. For example, a Button must generate a "click" event in response to being pressed with the mouse; a Form (window) must generate events in response to being moved, resized, minimized, or closed, and so on. Since all these visual elements have a corresponding definition in the XML state, the properties of the actual objects and the properties encoded in the state need to be synchronized every time one or the other changes.

In order to do this synchronization the renderer maintains a two-way mapping between the objects on the screen and their XML representation in the state. By default the renderer registers handlers for the events indicating changes in the visual aspect of these objects. When an event is triggered by a change to a user interface object, the associated handler in the renderer is invoked which in turn figures out what changed and updates the XML state accordingly. The renderer has been implemented in a way that allows developers to use any visual object from the .NET Framework (called Control [5] in .NET terminology) as part of their extensions. This gives developers the ability to create functionally-rich extensions that leverage the resources of .NET. For an even greater flexibility the renderer was designed to permit the display of visual elements (controls) defined in external assemblies, as long as they are .NET compliant. This greatly increases the number of options available to developers when creating extensions.

Under the hood all these benefits are obtained through the use of reflection. The trained reader might immediately wonder how portable this solution is, and they would be justified to question that. Even though the XML state is inherently portable to a different platform, the renderer would have to be rewritten if the new platform does not support .NET. For these platforms the renderer would not be able to use reflection in the same way eFuzion currently does and would need to implement a translation layer that would map the definition from the XML description to visual objects available on that platform. Additionally, if an extension references a visual element from an external assembly, that element and the ability to load assemblies would have to be implemented as well on the new platform.

The third responsibility of the renderer is to allow extensions to also register handlers for events published by the visual elements. The renderer intercepts these events and invokes the appropriate handlers, passing information regarding the event and surrounding context in the arguments of the function call. What these arguments are and how scripts communicate with the renderer will be the focus of the next topic, titled "Scripts". Figure 3.2 shows a flow diagram where all three functions of the renderer can be seen, while Figure 3.9 shows how the "Lecture" extension registers interest in the "OnSelectionChanged" event of the "InkPicture" control, specifying that the function "SelectionChanged" (defined in the file Lecture.py) should be executed whenever the event is triggered.

Continuing along the execution path of eFuzion from the moment the initial XML state file is loaded the control is given to the renderer which is now responsible for projecting the system state on the display. Before that is accomplished the renderer loads the textual representation of the XML state into an object structure that can be programmatically manipulated. From now on any references to "node", "element" or "section" in the context of XML indicate the corresponding objects in this object structure. Next the renderer traverses the state structure and constructs a hash table that maps the id values of all the nodes that specify an id to the nodes themselves. This is done as an optimization to allow some elements in the state to reference other elements in the state at different locations. Only elements that may be referenced need contain an "id" attribute. While constructing the hash table the renderer checks that the ids specified are unique, and throws an error if this condition is not met. After the hash table is constructed the renderer sets up the container objects that will be used to host the eventual menus and toolbars in preparation for displaying the main application form. The properties of the main form are then set based on the attributes specified in the "eFuzion" root node. Next the renderer checks if there is an "Init->Globals" section defined in the state, and processes it as follows: first the "Resources" node is consulted (if it exists) to create a hash table that maps the name of the resource to the node defining it. The objects corresponding to those resources are constructed, and references to them added to the nodes defining them. This completes the mapping from resource name -> resource node -> resource object. Second, the "Menus" and "Toolbars" nodes are traversed (if they exist) and the corresponding objects created based on these definitions. The functions processing these two sections are also responsible for figuring out if any menu (or toolbar) merging needs to occur based on the menu (and toolbar) object structure that exists at that time and the new items that need to be created. Before any merging is accomplished checks are performed to determine if any global menu (or toolbar) defines items that override other global existing items, throwing an error exception if this situation is encountered. Next the renderer traverses the XML structure again and records in a list all the nodes that define frames. This list is needed by the renderer to know where to look when rendering the frames defined by extensions. The current active frame of the application is then identified by consulting the "CurrentFrame" attribute of the root node "eFuzion". Using the hash mapping constructed in the beginning, the node describing that frame is quickly identified. If there are frames defined in the state, the renderer renders them one by one onto the screen, and then gives focus to the one identified as current. The containers holding the menus and toolbars are placed on the application main form, which is then made visible on the screen.

Rendering of an individual frame follows a similar principle. First the form object corresponding to this frame is constructed, and then its properties set based on the attributes of the form's XML definition. Next the value of the mandatory attribute "DisplayedObjects" is checked against the hash mapping of ids, and if no error occurred the renderer proceeds to render the objects in that section. The types of objects supported by the renderer inside the "DisplayedObjects" section include images, strokes, grouped items, and controls. Images can reference external files or items defined in resources (by using the "ResourceName:ItemName" construct). Figures 3.9 and 3.10 show examples of the use of the "DisplayedObjects" construct and the resulting renderer output. Strokes can only be rendered on top of ink-enabled surfaces (such as InkPicture). Ink surfaces have a different resolution than that of the rest of the system. The following quote from the book "Building Tablet PC Applications" [14] provides the best explanation of the concept:

> *"All Ink objects and Stroke objects use the HIMETRIC coordinate system. A HIMETRIC unit represents 0.01mm where the measurement is derived from the screen's current DPI. The coordinate space is the usual Microsoft Windows style fourth quarter quadrant in which the origin (0,0) represents the upper left corner of the space and the x and y coordinates imply locations to the right and down respectively. The Stroke objects contained in the Ink object all share the coordinate space, so each Stroke objects packet's x,y values are based from a common (0,0) origin."*

The XML state uses the Windows coordinate system to record the x and y values of the strokes. The conversion to and from HIMETRIC is done automatically by the renderer.

After the "DisplayedObjects" section is rendered, the next step towards finishing the rendering of the frame involves hooking up the necessary events that allow the synchronization between the screen object and its XML representation to occur, and making the frame visible on the screen if instructed to by its properties. Frames containing the "IsModal" attribute set to "true" are shown in modal dialogs.

Controls that are part of the "DisplayedObjects" section are rendered in a similar fashion. First the renderer establishes if the control was created before by checking if a mapping exists between the associated XML object and the control. If none is found the control is created and its properties set based on the attributes in the XML definition. If the control was already created only its properties are updated. This mechanism prevents the renderer from creating a new control every time it is invoked, thus saving system resources. As with frames, the next step in the process involves rendering of the "DisplayedObjects" section associated with the control. This section can be omitted if nothing needs to be displayed on top of the control, as is the case with buttons. Afterwards the renderer checks if the control that was just created contains an inkable surface and if so it looks inside the "Selection" section of the corresponding frame to see if anything needs to be visually selected. If anything is present inside that section the renderer follows the value of the "ref" attribute to find the exact element that is referenced, which is then rendered accordingly. The last step in the rendering of controls involves making the control visible on the screen as defined, and hooking in the standard events that are associated with changes in the control's properties, so that the XML state can be updated to reflect these changes.

Scripts

Working hand-in-hand with the renderer are the scripts that control the behavior of the application. After the framework starts up all interaction with the application (such as invoking any of its functionalities) results in the appropriate scripts being executed that implement those functionalities. The communication between scripts and the renderer is realized through the primitives defined by the framework API. Six categories of primitives are provided:

1. *XML primitives* – contain functionality to manipulate the XML state (example: functions to add nodes, remove nodes, retrieve properties of nodes, etc.)
2. *Rendering primitives* – contain functionality to communicate with the renderer telling it which sections of the state should be rendered (example: RenderFrame, RenderControl, RenderMenus, etc.)
3. *Windows primitives* – contain functionality that allows the creation of standard system dialog windows (example: OpenFileDialog – helps to select which file(s) to open, ColorChooserDialog – helps select a color from a palette of colors, etc.)
4. *Image primitives* – contain functionality to manipulate images (example: GetImage – creates a snapshot of the contents of an inkable surface, ImportImages – extracts images from PDF or PowerPoint files, etc.)
5. *Application primitives* – contain functionality for shutting down the application and for obtaining the version number and other information
6. *Error-handling primitives* – contain functionality to display error messages

A detailed description of each of these categories and the functionality they offer can be found in Appendix C.

Scripts are written in the Python language. IronPython is currently used as the bridge that allows Python scripts to communicate with the framework API which is written in C#. Initially Python.NET was used for this purpose. It offered the most complete solution because it provided access to all the features of both languages. IronPython was in its early stages at the time and contained only partial support of the Python semantics. The major difference between the two is in the realm of debugging. The mechanism that Python.NET used to bridge the two languages prevented it from marshalling the full context of errors when they occurred. This made debugging very difficult because the error messages from the bridge were uninformative and did not point to the exact location of the error. For example, a syntax error in a script or an error thrown for accessing an undefined variable generated the same error message, confusing the developer. Not even the line number of the error was known. Furthermore, functionalities like single stepping through code, inspection of the values of variables, breakpoints and other useful debugging tools were missing altogether. The only way to debug a script was through the (ab)use of "print" statements. This made debugging very slow and frustrating. As time passed IronPython matured and once it provided enough support of the Python semantics the transition was made to replace Python.NET. IronPython uses a different mechanism to interface the two languages. It is actually a C# implementation of the Python language, thus running on top of the CLR [4]. As a consequence, debugging a Python script became as easy as debugging a C# program. All the debugging tools (from breakpoints to step-execution and variable inspections) are now available. The only frustration left is when developers want to use Python constructs that are not yet supported by IronPython, but this only temporary as IronPython sprints towards its 1.0 release.

The default functionality of the framework is accessible through the set of menu items displayed at application startup. Invoking any of these functionalities triggers the execution of the corresponding functions from the appropriate script. The list of scripts supplied with the framework is the following:

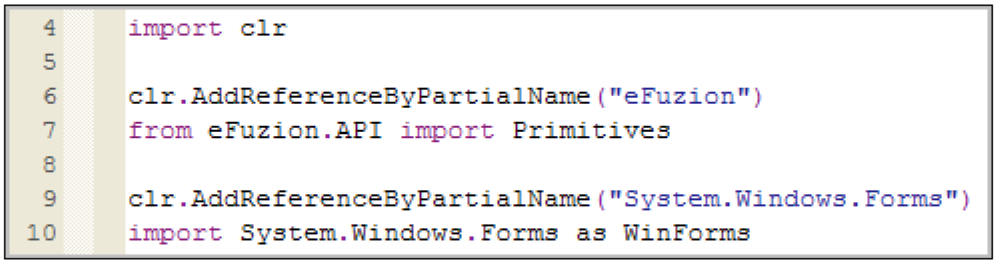
Common.py – provides access to the framework API through the “Primitives” object. All scripts that require access to the API must reference this file.

GlobalMenu.py – implements the logic corresponding to the set of menu items provided by the framework

GUID.py – provides functionality for generating unique ids (derived from the machine’s IP address, current time, and random bits). Offers a way to extract IP and time information from a GUID value.

PyUtilities.py – provides a set of useful functions for managing files and scripts

The content of the script *Common.py* is displayed in Figure 3.11 and shows the way C# functionality is accessed through IronPython. The “clr” module holds a link to the IronPython engine. The code in lines 6-7 directs the engine to import the “Primitives” object defined in the “eFuzion” assembly which is needed when scripts want to access



```
 4   import clr
 5
 6   clr.AddReferenceByPartialName("eFuzion")
 7   from eFuzion.API import Primitives
 8
 9   clr.AddReferenceByPartialName("System.Windows.Forms")
10   import System.Windows.Forms as WinForms
```

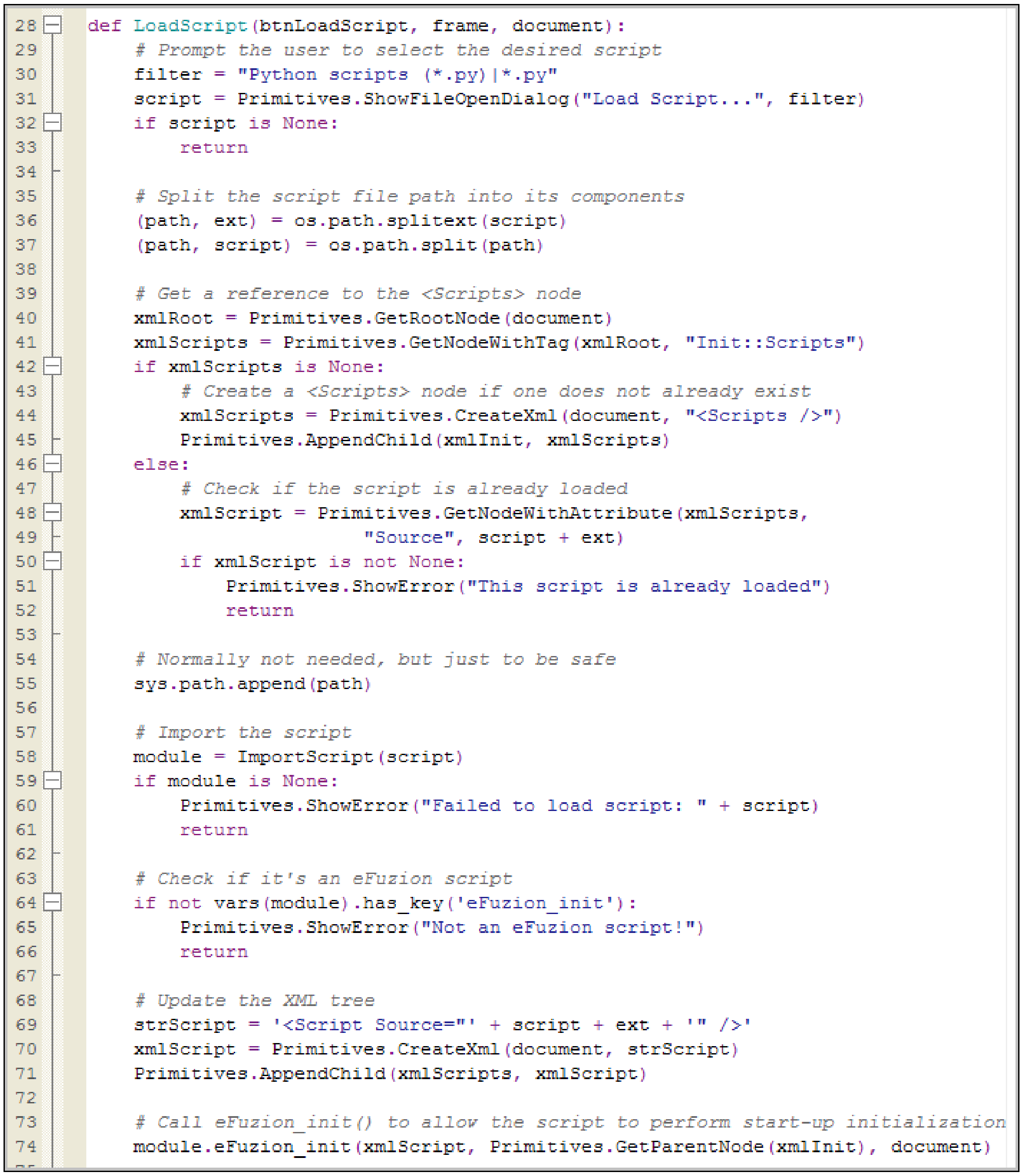
Figure 3.11 - Content of script Common.py

functions defined in the API.

When executing script functions the framework passes either three or four parameters in the function call, depending on the object generating the action, as follows:

- If the object is a .NET control (descending from System.Windows.Forms.Control, such as Button, Form, Label, etc.), then three parameters are used to represent, in order:
    - a reference to the XML node corresponding to the control that generated the event
    - a reference to the XML node corresponding to the current active frame
    - a reference to the object containing the entire XML state
- If the script is invoked in response to a stroke (or a group) being clicked, or due to an event generated by the InkPicture control, then four parameters are used, as follows:
    - a reference to the XML node corresponding to the element generating the event
    - a reference to the XML node corresponding to the InkPicture control
    - a reference to the XML node corresponding to the current active frame
    - a reference to the object containing the entire XML state

Figure 3.12 shows an example of the use of the “Primitives” object to implement the logic behind the functionality allowing the loading of extensions. Since the event responsible for triggering the execution of the code in the figure is generated in response to the user invoking the corresponding menu item (which is part of a .NET control), only three parameters are passed to the handling function as mentioned above. Line 31 invokes the "ShowFileOpenDialog" Windows primitive which prompts the user to select



```
28  def LoadScript(btnLoadScript, frame, document):
29      # Prompt the user to select the desired script
30      filter = "Python scripts (*.py)|*.py"
31      script = Primitives.ShowFileOpenDialog("Load Script...", filter)
32      if script is None:
33          return
34  
35      # Split the script file path into its components
36      (path, ext) = os.path.splitext(script)
37      (path, script) = os.path.split(path)
38  
39      # Get a reference to the <Scripts> node
40      xmlRoot = Primitives.GetRootNode(document)
41      xmlScripts = Primitives.GetNodeWithTag(xmlRoot, "Init::Scripts")
42      if xmlScripts is None:
43          # Create a <Scripts> node if one does not already exist
44          xmlScripts = Primitives.CreateXml(document, "<Scripts />")
45          Primitives.AppendChild(xmlInit, xmlScripts)
46      else:
47          # Check if the script is already loaded
48          xmlScript = Primitives.GetNodeWithAttribute(xmlScripts,
49                          "Source", script + ext)
50          if xmlScript is not None:
51              Primitives.ShowError("This script is already loaded")
52              return
53  
54      # Normally not needed, but just to be safe
55      sys.path.append(path)
56  
57      # Import the script
58      module = ImportScript(script)
59      if module is None:
60          Primitives.ShowError("Failed to load script: " + script)
61          return
62  
63      # Check if it's an eFuzion script
64      if not vars(module).has_key('eFuzion_init'):
65          Primitives.ShowError("Not an eFuzion script!")
66          return
67  
68      # Update the XML tree
69      strScript = '<Script Source="' + script + ext + '" />'
70      xmlScript = Primitives.CreateXml(document, strScript)
71      Primitives.AppendChild(xmlScripts, xmlScript)
72  
73      # Call eFuzion_init() to allow the script to perform start-up initialization
74      module.eFuzion_init(xmlScript, Primitives.GetParentNode(xmlInit), document)
```

Figure 3.12 – Implementation of the functionality allowing the loading of extensions (taken from GlobalMenu.py)

a file of the type specified in the “filter” argument. The value returned by the primitive contains the full path of the selected file, or *null* if nothing was selected. This value is checked in line 32 which determines if the execution of the function should continue. The code in lines 35-37 extracts the components of the file path (folder name, file name, extension) for later use. A handle to the root node of the XML state is obtained in line 40, and used to retrieve the handle to the “Scripts” node in line 41. Passing the string “Init::Scripts” as the second argument of the call to “Primitives.GetNodeWithTag” results in the following actions taking place under the hood:

1. the list of children of the node referenced by “xmlRoot” is searched for a node labeled “Init”
2. if one is found, its list of children is searched for a node labeled “Scripts”
3. if found the function returns a handle to the node, otherwise returns *null*

The code in lines 42-45 checks if the node labeled “Scripts” exists, and creates one if one can’t be found. In lines 48-52 the function determines if the script that’s about to load has already been loaded by consulting the content of the “Scripts” node, and shows an error message if it was already loaded. In lines 58-61 the function attempts to load the specified script and displays an error message if the operation fails. Next the function checks if the specified script is a valid eFuzion extension (lines 64-66) by determining if the script implements a function labeled “eFuzion_init” and an error message is displayed if this condition is not met. In lines 69-71 an entry is inserted in the “Scripts” section that allows the system to keep track of the loaded script. Finally line 74 passes control to the script by invoking the “eFuzion_init” function. More examples of the use of the platform API can be found in Chapter 4 when the “Lecture” script is presented.

CHAPTER 4: EXAMPLE

The “Lecture” Extension

Following on the footsteps of the previous versions of eFuzion this extension provides students and instructors with tools that can help them increase their productivity. From the point of view of the instructor the functionality provided by this extension allows them to create or annotate presentations that can be used as lecture materials during class sessions. Although the networking component is not yet done for this version of eFuzion, the materials created using this extension can be loaded by the previous eFuzion versions which provide networking support, if necessary. Students can use this extension to take notes during lectures, and to review their notes while at home. The discussion regarding this extension will now shift towards explaining what’s under the hood, but not before a screen shot of the “Lecture” script in action is shown (Figure 4.1).

After eFuzion is started the “Lecture” extension can be loaded using the provided functionality. As soon as the corresponding script is loaded, control is given to the “eFuzion_init” function which performs any initialization tasks required by the extension. These tasks include setting up the menus and toolbars and updating the display accordingly. Figure 4.2 shows the code behind “eFuzion_init”. The first parameter of the function represents the XML node inside the “Scripts” section corresponding to this extension. The function starts by setting the name of the script appropriately and retrieving the start-up path of the application.

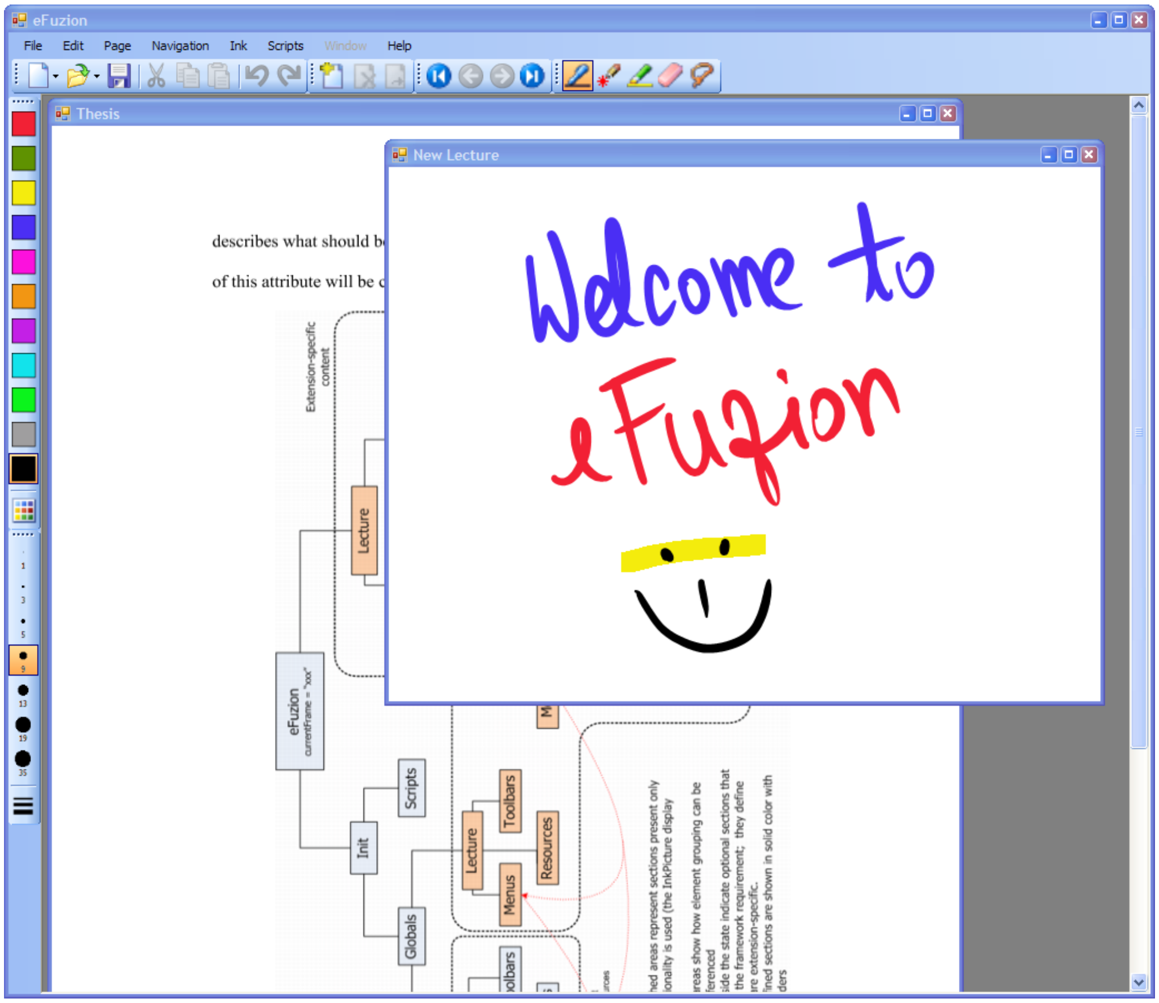
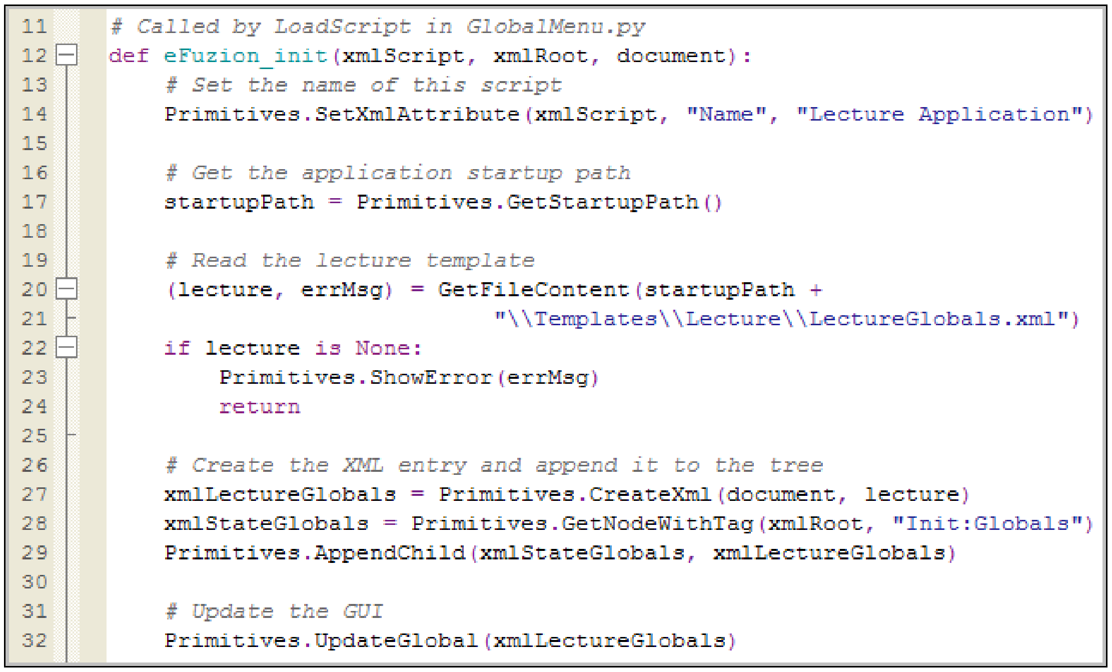


Figure 4.1 - Screenshot of the "Lecture" extension in action

Next the function loads the file "LectureGlobals.xml" which contains references to the resources used and the definitions of the global menu items and toolbar buttons provided to the user for accessing the extension's functionalities. These definitions are added to the XML state in lines 27-29. The call to "Primitives.UpdateGlobal" in line 32 tells the renderer to look inside the "eFuzion->Init->Globals" section and update the screen if any changes are found. After "eFuzion_init" finishes executing the application waits for further commands from the user. At this stage the screen looks as shown in Figure 4.3.



```
# Called by LoadScript in GlobalMenu.py
def eFuzion_init(xmlScript, xmlRoot, document):
    # Set the name of this script
    Primitives.SetXmlAttribute(xmlScript, "Name", "Lecture Application")

    # Get the application startup path
    startupPath = Primitives.GetStartupPath()

    # Read the lecture template
    (lecture, errMsg) = GetFileContent(startupPath +
                            "\\Templates\\Lecture\\LectureGlobals.xml")
    if lecture is None:
        Primitives.ShowError(errMsg)
        return

    # Create the XML entry and append it to the tree
    xmlLectureGlobals = Primitives.CreateXml(document, lecture)
    xmlStateGlobals = Primitives.GetNodeWithTag(xmlRoot, "Init:Globals")
    Primitives.AppendChild(xmlStateGlobals, xmlLectureGlobals)

    # Update the GUI
    Primitives.UpdateGlobal(xmlLectureGlobals)
```

Figure 4.2 - The code for "eFuzion_init" from Lecture.py

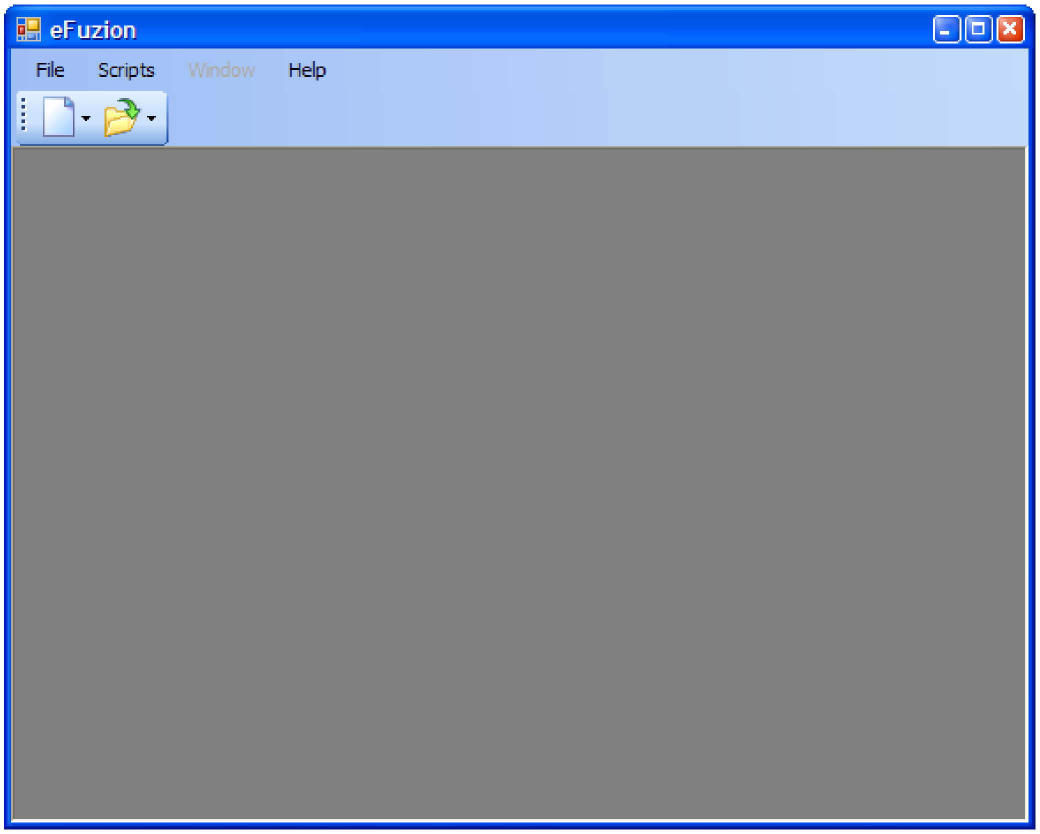
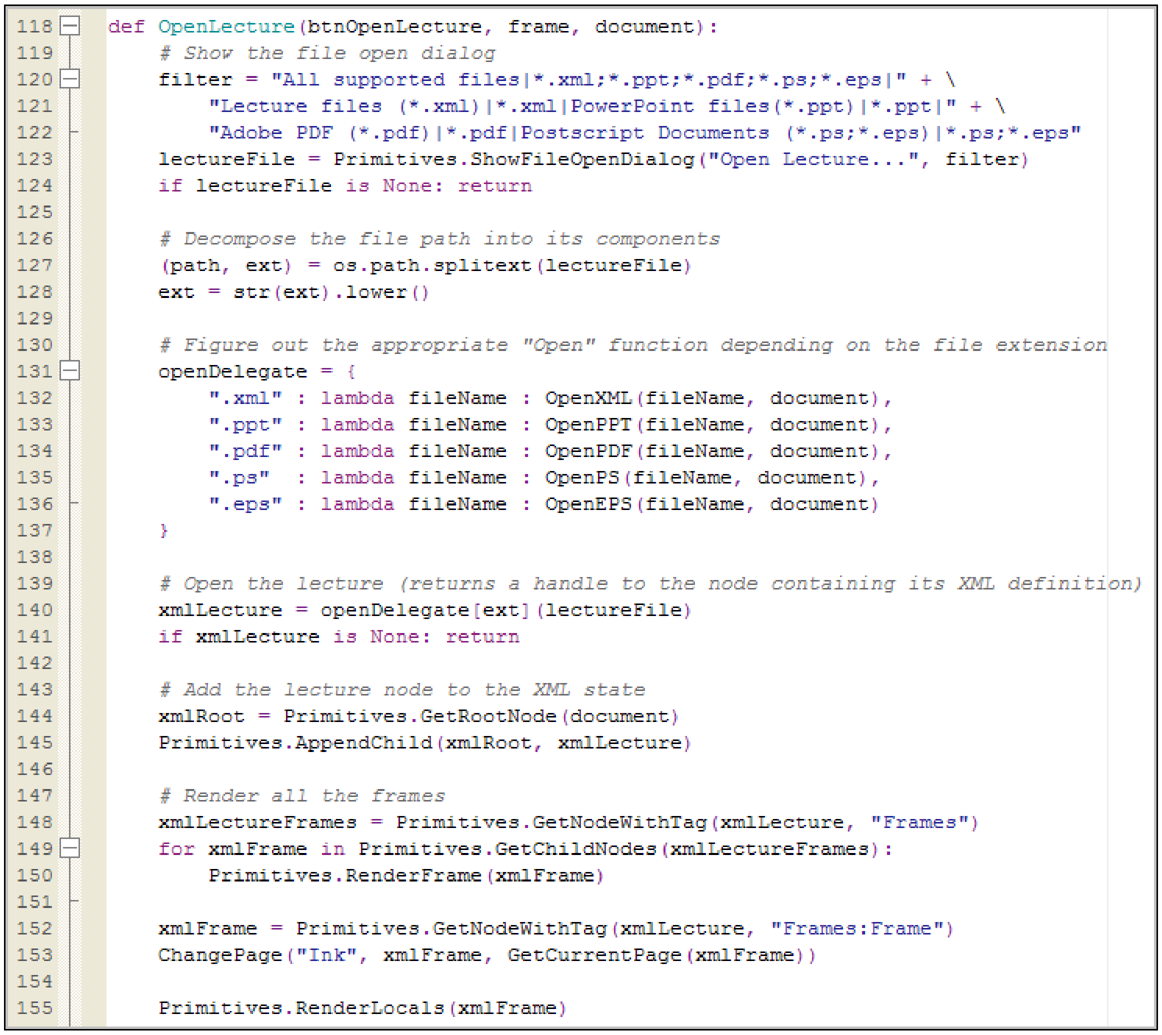


Figure 4.3 - Screenshot of eFuzion after the “Lecture” extension first loads

Users are now able to create new documents, save documents, and open existing documents by invoking the appropriate functionalities.

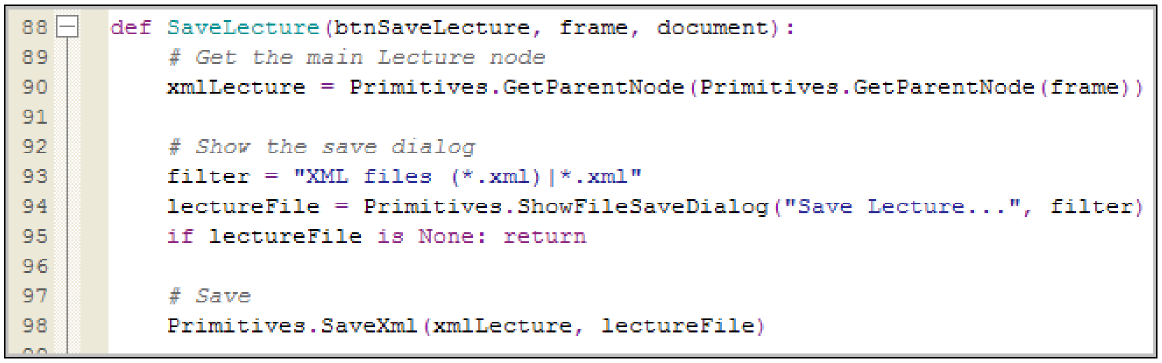
The code for creating a new document is a bit long because it involves a lot of XML structure manipulation, and thus will not be shown, but what it does is as follows: it first loads the content of the file "LectureContent.xml" which defines the lecture frame along with the corresponding menu and toolbar items, and then attaches it to the XML state. Next the title of the frame is adjusted and an id value is assigned to the frame's XML description. Then the content of the file "LecturePage.xml" is loaded which contains the definition of the "Pages" section, used for organizing the pages of the document. An id value is next added to the "DisplayedObjects" node inside the "Page" section, and the same value referenced through the "DisplayedObjects" attribute of the lecture frame's XML definition. This tells the renderer where to look to find what should be rendered inside the frame. The XML state is then updated and the display refreshed by a call to "Primitives.RenderFrame".

The code for opening an existing document is shown in Figure 4.4. In a nutshell it allows the user to select a file to open from a multitude of file types supported, then it figures out what the appropriate "Open" function should be that can handle that particular file type. The code in line 140 opens the document and obtains a handle on the node containing the XML content of the document. This XML section is then included in the application state and the display updated to reflect the new additions. The code in line 153 takes care of updating the XML state of the menu and toolbar items to reflect the current context. Finally the renderer is instructed to determine if any modifications were made to the “Menus” and “Toolbars” sections of the frame, and update the display accordingly.



```python
def OpenLecture(btnOpenLecture, frame, document):
    # Show the file open dialog
    filter = "All supported files|*.xml;*.ppt;*.pdf;*.ps;*.eps|" + \
        "Lecture files (*.xml)|*.xml|PowerPoint files(*.ppt)|*.ppt|" + \
        "Adobe PDF (*.pdf)|*.pdf|Postscript Documents (*.ps;*.eps)|*.ps;*.eps"
    lectureFile = Primitives.ShowFileOpenDialog("Open Lecture...", filter)
    if lectureFile is None: return

    # Decompose the file path into its components
    (path, ext) = os.path.splitext(lectureFile)
    ext = str(ext).lower()

    # Figure out the appropriate "Open" function depending on the file extension
    openDelegate = {
        ".xml" : lambda fileName : OpenXML(fileName, document),
        ".ppt" : lambda fileName : OpenPPT(fileName, document),
        ".pdf" : lambda fileName : OpenPDF(fileName, document),
        ".ps"  : lambda fileName : OpenPS(fileName, document),
        ".eps" : lambda fileName : OpenEPS(fileName, document)
    }

    # Open the lecture (returns a handle to the node containing its XML definition)
    xmlLecture = openDelegate[ext](lectureFile)
    if xmlLecture is None: return

    # Add the lecture node to the XML state
    xmlRoot = Primitives.GetRootNode(document)
    Primitives.AppendChild(xmlRoot, xmlLecture)

    # Render all the frames
    xmlLectureFrames = Primitives.GetNodeWithTag(xmlLecture, "Frames")
    for xmlFrame in Primitives.GetChildNodes(xmlLectureFrames):
        Primitives.RenderFrame(xmlFrame)

    xmlFrame = Primitives.GetNodeWithTag(xmlLecture, "Frames:Frame")
    ChangePage("Ink", xmlFrame, GetCurrentPage(xmlFrame))

    Primitives.RenderLocals(xmlFrame)
```

Figure 4.4 - The code for "OpenLecture" from Lecture.py



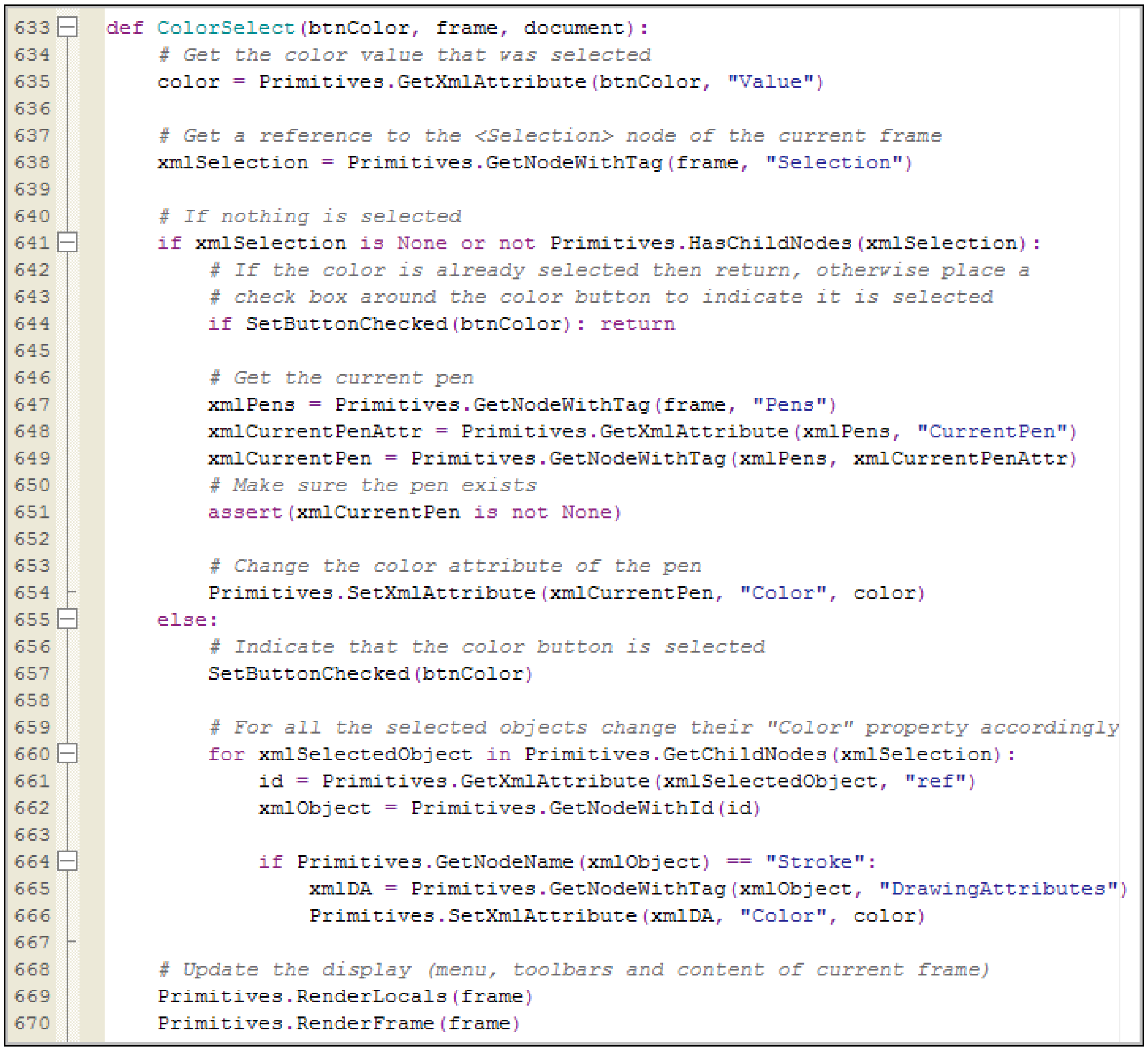
```python
def SaveLecture(btnSaveLecture, frame, document):
    # Get the main Lecture node
    xmlLecture = Primitives.GetParentNode(Primitives.GetParentNode(frame))

    # Show the save dialog
    filter = "XML files (*.xml)|*.xml"
    lectureFile = Primitives.ShowFileSaveDialog("Save Lecture...", filter)
    if lectureFile is None: return

    # Save
    Primitives.SaveXml(xmlLecture, lectureFile)
```

Figure 4.5 - The code for "SaveLecture" from Lecture.py

Next the code for saving a document is shown in Figure 4.5. Its logic is simple: show the user a dialog window where the name and location of the file to be saved can be selected (lines 90-95) and then place into that file the XML section associated with the document to be saved.

After the user creates or opens a document, the display will look similar to that shown in Figure 4.1. At this stage the user has access to a lot of functionality, like: selecting pen properties, changing colors, and navigating pages. Since the code implementing some of this functionality is quite long to be presented here only the simpler operations will be presented in detail; longer operations will be discussed in general terms.

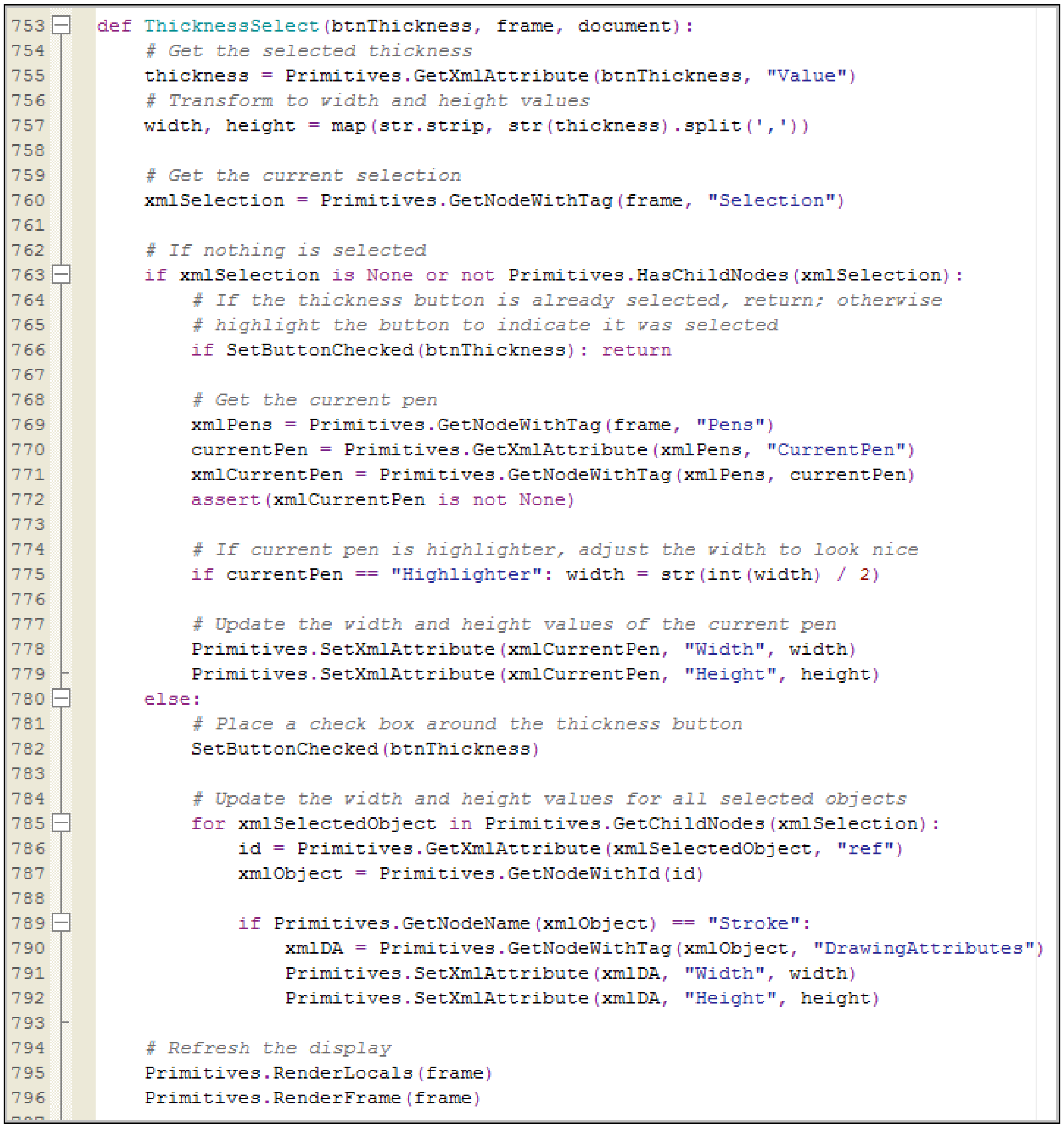
The code for changing colors is presented in Figure 4.6, and involves figuring out what the desired color value is (line 635), and if any objects are currently selected on the screen (lines 638-641). If nothing is selected, the next step is to determine if the color chosen was already selected, and if so return from the function as there is nothing further to do. Otherwise, place a check box around the button to indicate it is selected. Next, a reference to the node describing the current pen is obtained by inspecting the value of the "currentPen" attribute of the "Pens" node and searching for that value inside the "Pens" section (lines 647-649). After making sure the pen node exists (line 651), the "Color" attribute of the "Pen" node associated with the current pen is updated to reflect the new color chosen (line 654). On the other hand, if a selection exists (as indicated by the "Selection" section of the frame) then the function loops through all the objects selected and updates their color accordingly (lines 660-666). The function ends by instructing the renderer to update the display (lines 669-670).



```python
def ColorSelect(btnColor, frame, document):
    # Get the color value that was selected
    color = Primitives.GetXmlAttribute(btnColor, "Value")

    # Get a reference to the <Selection> node of the current frame
    xmlSelection = Primitives.GetNodeWithTag(frame, "Selection")

    # If nothing is selected
    if xmlSelection is None or not Primitives.HasChildNodes(xmlSelection):
        # If the color is already selected then return, otherwise place a
        # check box around the color button to indicate it is selected
        if SetButtonChecked(btnColor): return

        # Get the current pen
        xmlPens = Primitives.GetNodeWithTag(frame, "Pens")
        xmlCurrentPenAttr = Primitives.GetXmlAttribute(xmlPens, "CurrentPen")
        xmlCurrentPen = Primitives.GetNodeWithTag(xmlPens, xmlCurrentPenAttr)
        # Make sure the pen exists
        assert(xmlCurrentPen is not None)

        # Change the color attribute of the pen
        Primitives.SetXmlAttribute(xmlCurrentPen, "Color", color)
    else:
        # Indicate that the color button is selected
        SetButtonChecked(btnColor)

        # For all the selected objects change their "Color" property accordingly
        for xmlSelectedObject in Primitives.GetChildNodes(xmlSelection):
            id = Primitives.GetXmlAttribute(xmlSelectedObject, "ref")
            xmlObject = Primitives.GetNodeWithId(id)

            if Primitives.GetNodeName(xmlObject) == "Stroke":
                xmlDA = Primitives.GetNodeWithTag(xmlObject, "DrawingAttributes")
                Primitives.SetXmlAttribute(xmlDA, "Color", color)

    # Update the display (menu, toolbars and content of current frame)
    Primitives.RenderLocals(frame)
    Primitives.RenderFrame(frame)
```

Figure 4.6 - The code for "ColorSelect" from Lecture.py

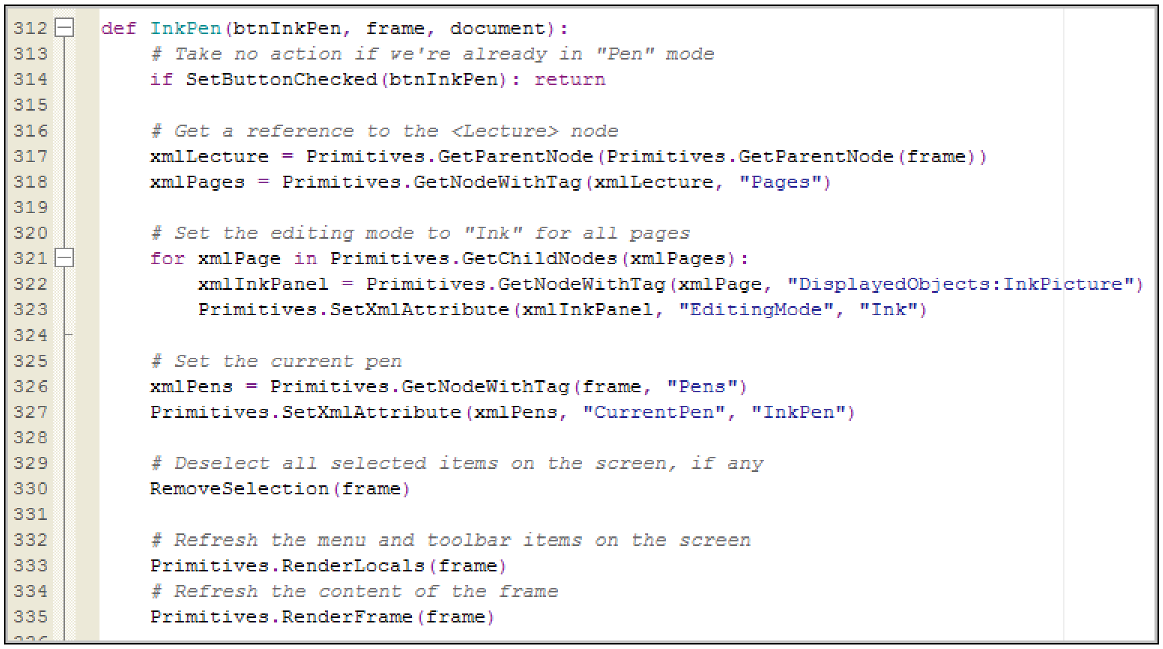
Changing the pen (or highlighter) thickness involves many of the same steps as before, and is shown in Figure 4.7. The code in lines 755-757 figures out what the width and height values of the desired thickness are, then obtains a reference to the “Selection” node which will be used to determine if anything was selected (line 760-763). Next, a reference to the node describing the current pen is obtained (lines 769-771), followed by a check for the situation where the current pen is the highlighter, in which case the width of the pen is adjusted by half to make it look nice on the screen (line 775). The width and height attributes of the current pen are finally updated in lines 778-779. If, however, some objects were selected on the screen, then the function loops through all the selected objects (as indicated by the children of the “Selection” section of the current frame), and their width and height attributes updated to reflect the desired thickness chosen (lines 785-792). The last step invokes the renderer to update the display (lines 795-796).



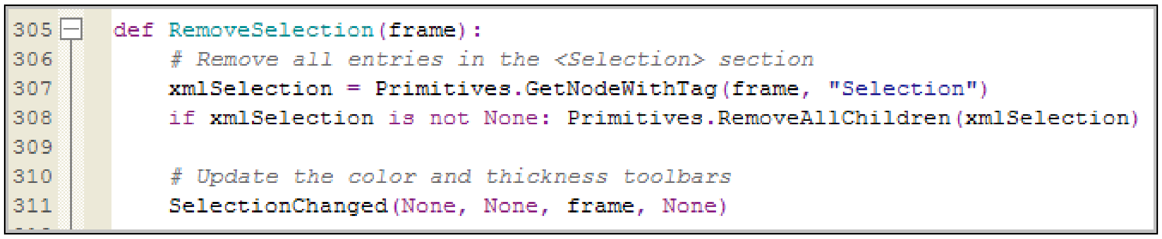
```
753 def ThicknessSelect(btnThickness, frame, document):
754     # Get the selected thickness
755     thickness = Primitives.GetXmlAttribute(btnThickness, "Value")
756     # Transform to width and height values
757     width, height = map(str.strip, str(thickness).split(','))
758 
759     # Get the current selection
760     xmlSelection = Primitives.GetNodeWithTag(frame, "Selection")
761 
762     # If nothing is selected
763     if xmlSelection is None or not Primitives.HasChildNodes(xmlSelection):
764         # If the thickness button is already selected, return; otherwise
765         # highlight the button to indicate it was selected
766         if SetButtonChecked(btnThickness): return
767 
768         # Get the current pen
769         xmlPens = Primitives.GetNodeWithTag(frame, "Pens")
770         currentPen = Primitives.GetXmlAttribute(xmlPens, "CurrentPen")
771         xmlCurrentPen = Primitives.GetNodeWithTag(xmlPens, currentPen)
772         assert(xmlCurrentPen is not None)
773 
774         # If current pen is highlighter, adjust the width to look nice
775         if currentPen == "Highlighter": width = str(int(width) / 2)
776 
777         # Update the width and height values of the current pen
778         Primitives.SetXmlAttribute(xmlCurrentPen, "Width", width)
779         Primitives.SetXmlAttribute(xmlCurrentPen, "Height", height)
780     else:
781         # Place a check box around the thickness button
782         SetButtonChecked(btnThickness)
783 
784         # Update the width and height values for all selected objects
785         for xmlSelectedObject in Primitives.GetChildNodes(xmlSelection):
786             id = Primitives.GetXmlAttribute(xmlSelectedObject, "ref")
787             xmlObject = Primitives.GetNodeWithId(id)
788 
789             if Primitives.GetNodeName(xmlObject) == "Stroke":
790                 xmlDA = Primitives.GetNodeWithTag(xmlObject, "DrawingAttributes")
791                 Primitives.SetXmlAttribute(xmlDA, "Width", width)
792                 Primitives.SetXmlAttribute(xmlDA, "Height", height)
793 
794     # Refresh the display
795     Primitives.RenderLocals(frame)
796     Primitives.RenderFrame(frame)
```

Figure 4.7 - The code for "ThicknessSelect" from Lecture.py

The code for selecting the pen instrument is shown in Figure 4.8 and amounts to finding the “Pages” section of the “Lecture” node (lines 317-318), navigating through all the “Page” subsections inside that section (line 321), getting a handle of the corresponding “InkPicture” node (line 322), adjusting the “EditingMode” attribute to indicate that the “Ink” pen is used (line 323), and finally updating the “CurrentPen” attribute of the “Pens” node to reflect the selection of the new instrument (lines 326-327). Line 330 makes a call to “RemoveSelection” (shown in the bottom of the same figure) which removes all entries inside the “Selection” section of the current frame in order unselect everything on the screen (lines 307-308). As always, the last step invokes the renderer to update the display (lines 333-335).



```
312  def InkPen(btnInkPen, frame, document):
313      # Take no action if we're already in "Pen" mode
314      if SetButtonChecked(btnInkPen): return
315  
316      # Get a reference to the <Lecture> node
317      xmlLecture = Primitives.GetParentNode(Primitives.GetParentNode(frame))
318      xmlPages = Primitives.GetNodeWithTag(xmlLecture, "Pages")
319  
320      # Set the editing mode to "Ink" for all pages
321      for xmlPage in Primitives.GetChildNodes(xmlPages):
322          xmlInkPanel = Primitives.GetNodeWithTag(xmlPage, "DisplayedObjects:InkPicture")
323          Primitives.SetXmlAttribute(xmlInkPanel, "EditingMode", "Ink")
324  
325      # Set the current pen
326      xmlPens = Primitives.GetNodeWithTag(frame, "Pens")
327      Primitives.SetXmlAttribute(xmlPens, "CurrentPen", "InkPen")
328  
329      # Deselect all selected items on the screen, if any
330      RemoveSelection(frame)
331  
332      # Refresh the menu and toolbar items on the screen
333      Primitives.RenderLocals(frame)
334      # Refresh the content of the frame
335      Primitives.RenderFrame(frame)
```



```
305  def RemoveSelection(frame):
306      # Remove all entries in the <Selection> section
307      xmlSelection = Primitives.GetNodeWithTag(frame, "Selection")
308      if xmlSelection is not None: Primitives.RemoveAllChildren(xmlSelection)
309  
310      # Update the color and thickness toolbars
311      SelectionChanged(None, None, frame, None)
```

Figure 4.8 - The code for "InkPen" (top), and "RemoveSelection" (bottom) from Lecture.py

The code for adding a page is a bit more complicated. It involves reading the "LecturePage.xml" file and creating a corresponding XML structure, generating a unique value for the "id" attribute of the "DisplayedObjects" node, and updating the attributes of the "InkPicture" node to match those of the current page. The final steps include attaching the resulting structure to the XML state, updating the state of the menu and toolbar items, and invoking the renderer to refresh the display. Deleting a page is easier and involves removing the associated XML section from the state, updating the menu and toolbars as necessary, and finally updating the display.

The code for navigating pages is a bit long, but only because it wants to be visually fancy and update the menu and toolbar items for navigation to reflect the surrounding context (i.e. visually disable the "previous" button if the current page is the first page, etc.). In a nutshell it involves playing with the value of the attribute "DisplayedObjects" of the current frame to point to the "DisplayedObjects" node corresponding to the new page.

There are still a lot of enhancements that can be added to this extension, namely: the ability to print notes, undo/redo operations, enter textual input, search for text through written notes, copy and paste using the system clipboard, and add networking support, but due to time constraints this task will be left to interested individuals.

CHAPTER 5: CONCLUSIONS

Although the application presented in this study is incomplete, it can be used as a proof-of-concept that validates the main ideas and motivation behind the design of eFuzion, proving them to be viable. There is still a lot to do before this application can be used in a production environment. Many details regarding the organization of the XML state will need to be revised, refined, and potentially replaced, to allow for a more flexible and natural extension-writing experience. One area that definitely needs improvement is with respect to the model used by developers to implement the functionality provided by their extensions. Currently a lot of grunt work needs to be done to accomplish even the most basic tasks due to the assembly-like programming model existent. To allow for a more rapid and natural extension development experience higher level abstractions need to be provided. For example, if developers want to create a new frame (window) for their extension, it should be possible to do so in one or two lines of code, instead of ten, following a principle similar to the traditional OO model.

Another situation that needs to be considered is that of synchronization. Even though objects can currently be “locked” using constructs provided by the language, if multiple scripts modify different nodes falling in the scope of the same frame, when the renderer is invoked to update the display it may encounter an inconsistent state and throw an error as a result. The locking of individual nodes does not help in preventing this situation to occur unless all the nodes falling in the scope of that frame are locked by the same function, and released upon its completion. Anyone can see that this is a very tedious process and a different method needs to be devised.

The platform-specific core can benefit from some additions and alterations too: for instance, the way scripts register interest in the events published by .NET objects needs to be reworked to allow the hooking of any event defined by that object. Currently only a fixed set of events can be hooked. Additionally, the mechanism for updating the XML state in response to changes done to the properties of objects needs to be improved to include all the properties of an object, and not just the few that are currently tracked.

Finally the network model needs to be implemented and tested. Decisions need to be made as to how the XML state should be changed to account for this addition, and exactly how much control scripts should be given over the network.

Overall there are many benefits provided by this design: it allows developers to create, test, and deploy extensions "on-the-spot", without worrying about viruses during the deployment process; it is extremely flexible and offers virtually unlimited possibilities for customization given the way the system state is encoded and made available to the scripts; finally, as a consequence of the separation of responsibilities between the core and pool of scripts, much of the system can be ported to other platforms without modification.

From the point of view of the "presentation and interaction" aspects of this application, the "Lecture" extension follows along the footsteps of its predecessors. It, too, needs a lot more work before it can be considered "complete", but shows that it is viable to build this type of applications on top of the eFuzion framework.

BIBLIOGRAPHY

[1] *.NET Framework*. Microsoft Corporation. July 2006. http://msdn.microsoft.com/netframework/

[2] *C# Language Reference*. Microsoft Developer Network. July 2006. http://msdn.microsoft.com/vcsharp/programming/default.aspx

[3] *ConferenceXP Project*. Microsoft Research. http://www.conferencexp.net/community/default.aspx

[4] *Common Language Runtime (CLR)*. Microsoft Corporation. http://www.microsoft.com/downloads/details.aspx?FamilyId=5D5FE65F-DAFC-4DB3-A6EF-A3CCE07B1B65&displaylang=en

[5] *Control class (System.Windows.Forms)*. Microsoft Developer Network. http://msdn2.microsoft.com/en-us/library/system.windows.forms.control.aspx

[6] *DyKnow Vision*. DyKnow Corporation. July 2006. http://www.dyknow.com/

[7] *Educational Technologies Research Group*. University of Illinois at Urbana-Champaign. http://edtech.cs.uiuc.edu/

[8] *Extensible Application Markup Language (XAML)*. Microsoft Corporation. June 2006. http://en.wikipedia.org/wiki/XAML

[9] *Extensible Markup Language (XML)*. World Wide Web Consortium (W3C). April 2006. http://www.w3.org/XML/

[10] *GNU Emacs*. Free Software Foundation. May 2006. http://www.gnu.org/software/emacs/

[11] *GNU TeXmacs*. Free Software Foundation. http://www.texmacs.org/tmweb/home/welcome.en.html

[12] *IronPython*. Microsoft Corporation. July 2006. http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

[13] *InkPicture Control*. Microsoft Developer Network. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tpcsdk10/lonestar/unmanaged_ref/inkpicture_ref/tbobjinkpicture.asp

[14] Jarret, R., Su, P. *Building Tablet PC Applications.* Redmond: Microsoft Press. 2003.

[15] *Microsoft Developer Network (MSDN).* Microsoft Corporation. http://msdn2.microsoft.com/en-us/default.aspx

[16] *MIT InkBoard.* Massachusetts Institute of Technology. Intelligent Engineering Systems Laboratory. http://ender.mit.edu/inkboard/

[17] *Mozilla XUL Project.* The Mozilla Foundation. December 2003. http://www.mozilla.org/projects/xul/

[18] Peiper, C. et all. *eFuzion: Development of a Pervasive Educational System.* Proc. Innovation and Technology in Computer Science Education. Monte de Caprica, Portugal, June 2005.

[19] *PGM Reliable Transport Protocol.* RFC 3208. Internet Engineering Task Force (IETF). http://www.ietf.org/rfc/rfc3208.txt

[20] *Python.NET.* http://pythonnet.sourceforge.net/

[21] *Python Programming Language.* Python Software Foundation. http://www.python.org/

[22] *ReMarkable Texts.* Brown University Computer Graphics Group. http://graphics.cs.brown.edu/research/ReMarkableTexts/

[23] *UW Classroom Presenter.* University of Washington Computer Science and Engineering. http://www.cs.washington.edu/education/dl/presenter/

[24] Wentling, T. et all. *Validation of e-Fuzion: a Wireless Classroom Technology.* Proc. World Conference on E-Learning in Corp., Govt., Health & Higher Ed., Vol. 2003, Issue. 1, 2003.

[25] *Windows XP Tablet PC Edition 2005.* Microsoft Corporation. http://www.microsoft.com/windowsxp/tabletpc/default.mspx

[26] *XML User Interface (XUI).* Xoetrope Company. http://www.xoetrope.com/zone/index.php?zone=XUI

APPENDIX A: SETTING UP THE BUILD ENVIRONMENT

The system requirements for creating and running eFuzion extensions are as follows:

Operating System:

Any flavor of Microsoft Windows™ (Windows XP Tablet PC Edition preferred when inking functionality is desired)

Development Tools:

Microsoft Visual Studio 2005, available at: http://msdn.microsoft.com/vstudio/

Tablet PC SDK 1.7, available at: http://msdn.microsoft.com/tabletpc

Python, available at: http://www.python.org/download/

IronPython, available at:
http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

Office 2003 Primary Interop Assemblies, available at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dno2k3ta/html/OfficePrimaryInteropAssembliesFAQ.asp

Ghostscript, available at: http://www.cs.wisc.edu/~ghost/

Installation Procedure:

Install, in order, the tools listed in "Development Tools".
For core development: unpack the eFuzion solution in an empty folder on the computer and launch the *eFuzion.sln* solution file.
For extension development only: use the Windows Installer-based distribution of eFuzion.

APPENDIX B: SCRIPTING GUIDELINES

Extension developers are encouraged to adhere to the following guidelines in order to maximize productivity and minimize frustration:

- Use many *try-catch* statements to detect and handle any anomalies during script execution; when exceptions occur, always remove any alterations done to the state that would leave it in an inconsistent state.
- Invoke *RenderFrame* only after all changes to the state that don't require immediate UI update have been made. This ensures a smooth visual experience and increases application performance.
- Use the tighter-scoped *Render[*]* functions whenever their behavior is sufficient: for example, after adding a menu item call *RenderMenuStrip* instead of *RenderFrame*. This will increase the application performance.
- If something goes wrong with a script it may be a good idea to enable the provided XML debug window by un-commenting the two lines in the *XmlRenderer* constructor (in *XmlRenderer.cs*); this will display a window containing a live view of the XML state; any subsequent modifications to the state will be visible immediately in that window. Note: the application performance will degrade when the debug window is enabled. The window can be closed at any time to return to the original performance characteristics.

APPENDIX C: THE EFUZION API

XML Primitives

```
string GetXmlAttribute(XmlNode xmlNode, string xmlAttrName)
```

Description:

Gets the value of an an attribute for a specified node.

Arguments:

xmlNode	– the node on which the operation should be performed
xmlAttrName	– the name of the attribute whose value should be returned

Returns:

The value of the specified attribute, or *null* if the node does not have an attribute by that name.

```
SetXmlAttribute(XmlNode xmlNode, string xmlAttrName,
      string value)
```

Description:

Sets the value of an an attribute for a specified node, or deletes the attribute if *value* is *null*.

Arguments:

xmlNode	– the node on which the operation should be performed
xmlAttrName	– the name of the attribute to be set
value	– the value of the attribute

```
XmlNode GetNodeWithTag(XmlNode xmlNode, string tagPath)
```

Description:

Retrieves searches the list of children of xmlNode for the node specified by *tagPath*
The construct *Tag1:Tag2:Tag3…* can be used to instruct the function to search following the path *xmlNode -> Tag1 -> Tag2 -> Tag3 …*

Arguments:

xmlNode	– the start node on which the operation is to be performed

tagPath	– the list of names of nodes to be traversed

Returns:

The node with the specified tag, or *null* if the path does not exist

Example:

Assuming the following XML structure:

```
<someNode>
   <foo … >
      <bar … />
   </foo>
</someNode>

Primitives.GetNodeWithTag(someNode, “foo:bar”)
```

returns the node `<bar … />`; any number of tags can be specified, delimited by ':'

```
XmlNodeList GetNodesByTag(XmlNode xmlNode, string tag)
```

Description:

Searches the children of node *xmlNode* for any node that has *tag* as a tag and returns the list of all found nodes matching this criteria.

Arguments:

xmlNode	– the node whose children are to be searched
tag	– the tag that is to be searched for

Returns:

A list of children of *xmlNode* that have the specified tag, or *null* if no children have that tag.

```
XmlNode GetNodeWithAttribute(XmlNode xmlNode,
      string attrName, string attrValue)
```

Description:

Searches the children of node *xmlNode* for the first node that has an attribute *attrName* whose value is *attrValue* and returns that node, or *null* if no such node is found.

Arguments:

xmlNode	– the node whose children are to be searched
attrName	– the attribute to be queried
attrValue	– the value of the attribute seeked

Returns:

The first child of *xmlNode* which has the attribute *attrName* with value *attrValue,* or *null* if none was found.

```
XmlNode GetNodeWithAttribute(XmlNode xmlNode, string attrName,
     string attrValue, bool recursive)
```

Description:

Searches all descendants of node *xmlNode* for the first node that has an attribute *attrName* whose value is *attrValue* and returns that node, or *null* if no such node is found.

Arguments:

xmlNode	– the node whose descendants are to be searched
attrName	– the attribute to be queried
attrValue	– the value of the attribute seeked
recursive	– *True* if the search should be done recursively, *False* otherwise

Returns:

The first descendant of *xmlNode* which has the attribute *attrName* with value *attrValue,* or *null* if none was found.

```
string GetNodeName(XmlNode xmlNode)
```

Description:

Returns the name (tag) of the specified node.

Arguments:

xmlNode – the node whose name is to be returned

Returns:

The name (tag) of the specified node.

Example:

Assume the following XML node: `<foo … />`

```
Primitives.GetNodeName(fooNode)
```

will return “foo”

`string GetNodeContent(XmlNode xmlNode)`

Description:

Returns the content of the specified node.

Arguments:

xmlNode – the node whose content is to be returned

Returns:

The content of the specified node.

Example:

Assume the following XML node: `<foo>This is some content</foo>`

```
Primitives.GetNodeContent(fooNode)
```

will return “This is some content”

`SetNodeContent(XmlNode xmlNode, string content)`

Description:

Sets the content of a node.

Arguments:

xmlNode – the node whose content is to be set

Example:

Assume the following XML node: `<foo>This is some content</foo>`

```
Primitives.SetNodeContent(fooNode, “New content”)
```

will change fooNode to read: `<foo>New content</foo>`

`XmlNode GetRootNode(XmlDocument document)`

Description:

Returns the root node of an XML document

Arguments:

document – the XML document

Returns:

The root node of *document*. Note: The parameter *document* contains a reference to the object holding the entire XML tree.

`XmlNodeList GetChildNodes(XmlNode xmlNode)`

Description:

Returns a list of all children of *xmlNode*.

Arguments:

xmlNode – the node whose children are to be returned

Returns:

A list of all children of *xmlNode*, or *null* if *xmlNode* has no children.

`bool HasChildNodes(XmlNode xmlNode)`

Description:

Returns a boolean value corresponding to whether *xmlNode* has children or not.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

True if *xmlNode* has at least one child node, *False* otherwise.

`XmlNode GetFirstChild(XmlNode xmlNode)`

Description:

Returns the first child of *xmlNode*, or *null* if *xmlNode* has no children.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

The first child of *xmlNode*, or *null* if *xmlNode* has no children.

```
XmlNode GetLastChild(XmlNode xmlNode)
```

Description:

Returns the last child of *xmlNode*, or *null* if *xmlNode* has no children.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

The last child of *xmlNode*, or *null* if *xmlNode* has no children.

```
XmlNode GetNextSibling(XmlNode xmlNode)
```

Description:

Returns the next sibling of *xmlNode*, or *null* if *xmlNode* has no next sibling.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

The next sibling of *xmlNode*, or *null* if *xmlNode* has no next sibling.

Example:

Assume the following XML structure:

```
<foo>
   <bar />
   <har />
   <zar />
</foo>
```

```
Primitives.GetNextSibling(harNode)
```

will return *zarNode*.

```
XmlNode GetPreviousSibling(XmlNode xmlNode)
```

Description:

Returns the previous sibling of *xmlNode*, or *null* if *xmlNode* has no previous sibling.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

The previous sibling of *xmlNode*, or *null* if *xmlNode* has no previous sibling.

```
XmlNode GetParentNode(XmlNode xmlNode)
```

Description:

Returns the parent of *xmlNode*, or *null* if *xmlNode* has no parent.

Arguments:

xmlNode – the node on which the operation is to be performed

Returns:

The parent of *xmlNode*, or *null* if *xmlNode* has no parent.

```
XmlNode GetNodeWithId(string id)
```

Description:

Returns the node that has the attribute "id" with value *id*.

Arguments:

id – the unique id that identifies the node seeked

Returns:

The node that has the attribute "id" with value *id*.

```
AppendChild(XmlNode xmlParent, XmlNode xmlChild)
```

Description:

Appends a node at the end of the child list of another node.

Arguments:

xmlParent	– the node that will gain a new child
xmlChild	– the node that is to be inserted

`PrependChild(XmlNode xmlParent, XmlNode xmlChild)`

Description:

Inserts a node at the beginning of the child list of another node.

Arguments:

xmlParent	– the node that will gain a new child
xmlChild	– the node that is to be inserted

`InsertBefore(XmlNode xmlRefNode, XmlNode xmlNode)`

Description:

Inserts a node before another node (the inserted node will become the previous sibling of the reference node).

Arguments:

xmlRefNode	– the reference node
xmlNode	– the node that is to be inserted

`InsertAfter(XmlNode xmlRefNode, XmlNode xmlNode)`

Description:

Inserts a node after another node (the inserted node will become the next sibling of the reference node).

Arguments:

xmlRefNode	– the reference node
xmlNode	– the node that is to be inserted

`ReplaceChild(XmlNode xmlParent, XmlNode xmlNew, XmlNode xmlOld)`

Description:

Replaces one of the children of a node with another node.

Arguments:

xmlParent	– the node on whose children the operation is to be performed
xmlNew	– the node that is to replace the old node
xmlOld	– the node that is to be replaced by the new node

```
RemoveNode(XmlNode xmlNode)
```

<u>Description:</u>

Removes a node.

<u>Arguments:</u>

xmlNode – the node that is to be removed

```
RemoveAllChildren(XmlNode xmlNode)
```

<u>Description:</u>

Removes all the children of a node.

<u>Arguments:</u>

xmlNode – the node whose children are to be removed

```
XmlNode CloneNode(XmlNode xmlNode, bool deep)
```

<u>Description:</u>

Creates a clone of a given node. The clone could be shallow or deep, depending on the value of the parameter *deep*.

<u>Arguments:</u>

xmlNode	– the node that is to be cloned
deep	– *True* if deep clone should be performed (copy all children), *False* otherwise

<u>Returns:</u>

A new node that is a clone of the specified node.

```
XmlNode CreateXml(XmlDocument xmlDocument, string strXml)
```

<u>Description:</u>

Creates a node object from a given string representation.

<u>Arguments:</u>

xmlDocument	– the object representing the XML document
strXml	– the string value containing the raw XML definition

<u>Returns:</u>

A new node that has been created based on the information provided in *strXml*.

`SaveXml(XmlNode xmlNode, string fileName)`

Description:

Saves the XML subtree rooted at *xmlNode* in the file specified by *fileName*.

Arguments:

xmlNode	– the node that represents the root of the subtree that is to be saved
fileName	– the file path where the XML data is to be saved

`SaveState(XmlDocument xmlDocument, string fileName)`

Description:

Saves the XML document in the file specified by *fileName*.

`LoadState(string fileName)`

Description:

Loads the XML state from the file specified by *fileName* and switches to the new state.

Rendering Primitives

`RenderFrame(XmlNode xmlFrame)`

Description:

Renders the contents of the frame specified by *xmlFrame*. The menus and toolbars are **not** rendered. See `RenderLocals` for a function that renders both menus and toolbars.

Arguments:

xmlFrame – the node specifying the frame that is to be rendered

---

`RenderMenuStrip(XmlNode xmlMenuStrip)`

Description:

Renders a menu strip.

Arguments:

xmlMenuStrip – the node specifying the menu strip that is to be rendered

---

`RenderToolStrip(XmlNode xmlToolStrip)`

Description:

Renders a toolbar strip.

Arguments:

xmlToolStrip – the node specifying the toolbar strip that is to be rendered

---

`RenderMenus(XmlNode xmlMenus)`

Description:

Renders all the menu strips contained in *xmlMenus*.

Arguments:

xmlMenus – the node containing the menu strips that are to be rendered

---

`RenderToolbars(XmlNode xmlToolbars)`

Description:

Renders all the toolbar strips contained in *xmlToolbars*.

Arguments:

xmlToolbars – the node containing the toolbar strips that are to be rendered

---

`RenderLocals(XmlNode xmlFrame)`

Description:

Renders the local menus and toolbars.

Arguments:

xmlFrame – the node specifying the frame whose menus and toolbars are to be rendered

```
RenderControl(XmlNode xmlControl, XmlNode xmlParent)
```

Description:

Renders a control on the specified "surface".

Arguments:

xmlControl –	the node describing the control that is to be rendered
xmlParent –	the node describing the control on top of which rendering is done

---

```
RenderDisplayedObjects(XmlNode xmlDisplayedObjects,
     XmlNode xmlParent)
```

Description:

Renders a DisplayedObjects node on the specified "surface".

Arguments:

xmlDisplayedObjects	– the DisplayedObjects node that is to be rendered
xmlParent	– the node describing the control on top of which rendering is done

---

```
UpdateGlobal(XmlNode xmlGlobal)
```

Description:

Renders the content of the "Globals" node specified by *xmlGlobal.* This updates resources and renders the menus and toolbars defined in that section.

Arguments:

xmlGlobal	- the node referencing the "Globals" section

Windows Primitives

```
string ShowFileOpenDialog(string caption, string filter)
```

Description:

Prompts the user to select a file based on the specified filter.

Arguments:

caption	– the title of the dialog window
filter	– specifies the file types allowed

Returns:

The fully qualified path name of the selected file, or *null* if the Cancel button was pressed.

```
string ShowFileSaveDialog(string caption, string filter)
```

Description:

Prompts the user to select the file that will be used as a destination for the save operation.

Arguments:

caption - the title of the dialog window
filter - specifies the file types allowed. More info: here

Returns:

The fully qualified path name of the selected file, or *null* if the Cancel button was pressed.

```
string ShowColorChooserDialog(string selectedColor)
```

Description:

Prompts the user to select a color from the ones provided in the dialog.

Arguments:

selectedColor - the color that the dialog will show as selected, or *null* to use the default selection

Returns:

The selected color as a string with format "A, R, G, B", or *null* if the Cancel button was pressed.
[A = transparency (255 fully opaque, 0 fully transparent), R = red, G = green, B = blue]

```
string ShowThicknessDialog(string value)
```

Description:

Prompts the user to select the desired pen thickness.

Arguments:

value - the thickness value that will show as selected

Returns:

The selected thickness value.

Image Primitives

```
int GetImageWidth(Image image)
```

Description:

Returns the width of a given image.

Arguments:

image – the image object whose width is to be returned

Returns:

The width of the specified image.

```
int GetImageHeight(Image image)
```

Description:

Returns the height of a given image.

Arguments:

image – the image object whose height is to be returned

Returns:

The height of the specified image.

```
SaveImage(Image image, string filePath)
```

Description:

Saves an image to a file.

Arguments:

image	– the image object that needs to be saved
filePath	– the file where the image is to be saved

```
ArrayList ImportImages(string fileName)
```

Description:

Returns a list of images "captured" from the specified file. Currently supported are PowerPoint, PDF, Postscript, and EPS files.

Returns:

A list of images representing "screenshots" of the pages contained in the specified file.

Application Primitives

`string GetStartupPath()`

Description:

Returns the startup path of the application.

`ApplicationExit()`

Description:

Exits the application.

`AboutPyFuzion()`

Description:

Displays the "About" box containing application version information and copyright notice.

Error-handling Primitives

`ShowError(string errorMsg)`

Description:

Shows a dialog box containing the specified error message.

Arguments:

errorMsg – the error message that should be displayed