

PRESENT LIVELY: A MODULAR DYNAMIC PRESENTATION SYSTEM

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

---

Philip Hanson

Date: Mar 6, 2009

Approved:

---

Professor George Heineman, Advisor

# Table of Contents

1 Introduction.....	4
2 Related Work and Motivation.....	4
3 Design.....	6
3.1 Object Model.....	8
3.1.1 Slides.....	8
3.1.2 Slide Sets.....	9
3.1.3 Slide Shows.....	10
3.2 Database Interaction.....	11
3.3 Enhanced Presentation Flow.....	12
3.3.1 Presentation Control.....	12
3.3.2 Content Management.....	13
3.4 Distinctive Features.....	13
3.4.1 Tagging.....	13
3.4.2 Dynamic Content.....	13
3.5 Modular Design.....	14
4 Implementation.....	14
4.1 Database Structure.....	15
4.2 User Interface.....	18
4.3 Module Management.....	21
4.3.1 Class Structure.....	22
4.3.2 Module Loading and Activation.....	22
4.3.3 Message Bus.....	23

4.3.4 Database and Display Interaction.....	24
4.4 Core Modules.....	31
4.4.1 Slide Editor.....	31
4.4.2 Browser.....	32
4.4.3 Set Display.....	32
4.4.4 Set Editor.....	33
4.4.5 Show Display.....	33
4.4.6 Web Object Module.....	34
5 Evaluation: Creating a New Module.....	35
5.1 Module Class Code.....	36
5.2 User Control.....	38
6 Summary and Future Work.....	40
Appendix A: Creating a Slide Layout.....	44

# 1 Introduction

As computers have become increasingly integrated into everyday life and business operations, it has become commonly desired to use computers to display information - usually on a large screen - as a method of supporting communication with others. From this need arose specialized presentation software based on the existing paradigm of slide and transparency projectors.

Modern presentation software restricts the user to present information on the application's terms. Existing systems, while allowing increasingly rich forms of media within a slide, still enforce a linear ordering of slides. Once a presentation is under way it is “on rails” and cannot be modified. As a result, most people use tools such as PowerPoint in the same way one would use flash cards: as a written record of their talk [1]. In his e-book “Really Bad PowerPoint,” marketing consultant Seth Godin argues that while this tactic facilitates information transfer, a written report would be just as effective. What is lacking in these types of presentations is *communication* [2]. This project, PresentLively, will produce presentation software that enables dynamic communication by allowing presenters to convey ideas in a more natural way – presenting on human terms.

## 2 Related Work and Motivation

The most well-known digital presentation software application is Microsoft PowerPoint [3], which is used by a majority of presenters. PowerPoint allows users to create slide “shows” or “decks” that are linear arrangements of slides intended to be shown in order without alteration. As mentioned earlier, this linear structure makes presentations inflexible and tends to encourage bad communication habits. Slide decks cannot be changed at presentation time without

retreating to the editing interface, which takes the audience's attention away from content the presenter is trying to show. This limitation can be circumvented, but it requires a fair amount of technical expertise and time to do so. One method of doing this will be discussed later in this section.

Since its introduction in 1987, PowerPoint has continually been maintained and updated to support new content formats and more impressive visual effects. The latest version, PowerPoint 2007, allows users to include text, images, and videos in slides. Objects within a slide can be animated with a variety of effects ranging from a simple fade in or out to flying around the screen. One can include hyperlinks to the web, other files, or even other slides within the same presentation. However, none of these actions can be performed at presentation time.

Third parties have attempted to rectify the problems inherent in the design of PowerPoint, resulting in techniques such as the Relational Presentation methodology promoted by Aspire Communications [4]. These approaches focus on modifying the use of PowerPoint to support real communication between the presenter and audience, but they remain limited by the program design and require extensive preparation. Presentations developed using these techniques, though easier to navigate during the presentation, are more difficult to modify due to their complex structure. This structuring also makes such presentations more likely to have broken or missing references than normal presentations since their content is spread throughout multiple locations.

Rather than competing based on breadth of features, PresentLively addresses the more basic problem of communication by implementing and extending methods such as Relational Presentation directly to make them manageable and encourage good communication techniques.

In addition to basic presentation features, the use of dynamic content and navigation will set PresentLively presentations apart.

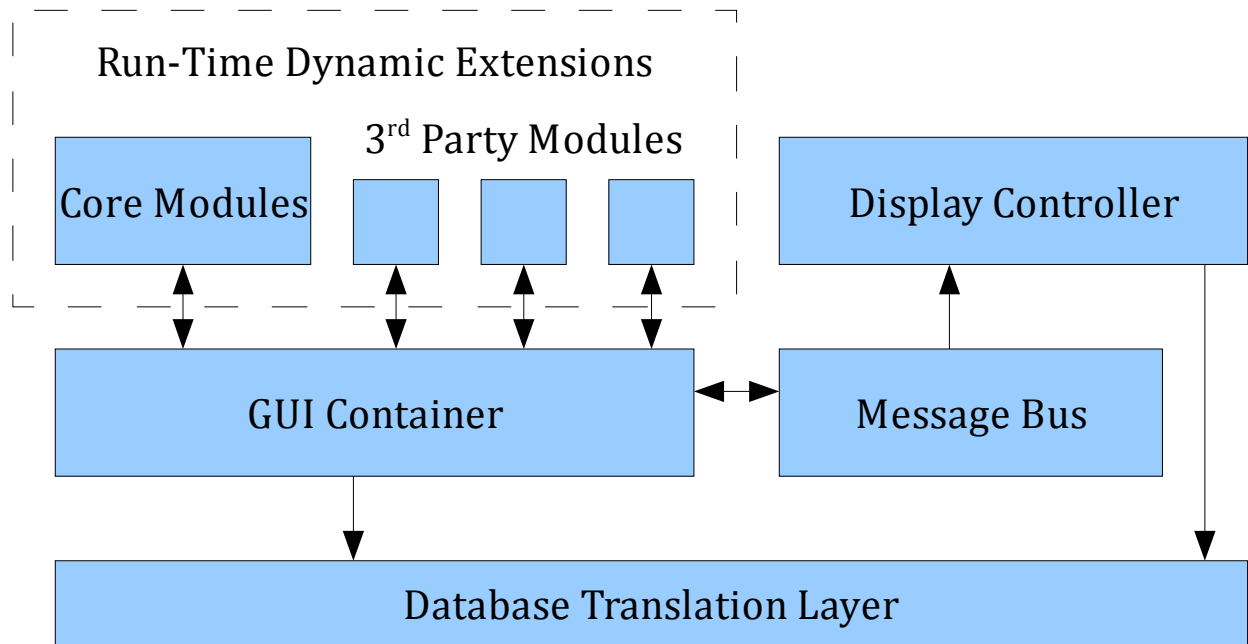
### **3 Design**

PresentLively is designed to accomplish two objectives: The first objective is to allow users to break out of presentation linearity and thereby present information in a more natural way. The second objective is to increase the dynamism of presentation software by including active content as well as search and content discovery controls within the presentation interface. PresentLively is designed for users that have semi-structured content or a desire to integrate dynamic content in structured presentations.

At a high level of abstraction, PresentLively supports the display of linear and nonlinear slide shows, random-access retrieval and display of slides, and direct transfer of content to the screen. For the purposes of this document, let us consider linear slide shows as a special “branchless” case of nonlinear shows.

Figure 1 depicts the overall PresentLively system architecture. The diagram is divided into two sections, “PresentLively” (unlabeled) and “Run-Time Dynamic Extensions.” Items in the PresentLively portion are elements contained in the main PresentLively executable files. Those items are required elements that are always present. Together, the items in the PresentLively portion compose a framework for managing and presenting slides, sets, and shows. However, none of the functionality of this framework is accessible to users through the basic PresentLively user interface.

At run-time the PresentLively framework dynamically loads all available modules. These are shown in Figure 1 in the “Run-Time Dynamic Extensions” box. All user interactions with the system take place through module interfaces. The process by which module interfaces are loaded and presented to users is described more fully in Section 4.3.2 .



*Figure 1: System Architecture Diagram*

As may be inferred from the diagram of Figure 1 and the above description, functions and user interface elements of PresentLively described in this report are contained within a set of “Core Modules” that are described in greater detail in Section 4.4 .

The remainder of this section describes the various features and qualities of the PresentLively software as well as design challenges encountered in its creation.

## 3.1 Object Model

The standard model for presentation software is that a presentation consists of slides and shows. While parts of this project defy even that notion, PresentLively retains in its data model the idea of slides. Non-linear shows, described in Section 3.1.3 below, are substituted for the traditional variety, and a new concept, the slide set, is added. The design of these objects is described in the following sections.

### 3.1.1 Slides

The basic unit of a presentation is a *slide*. A slide in PresentLively consists of one or more content items and a reference to a predefined layout description. Raw content items include text, images, and dynamic objects.

The layout description includes formatting, location, and boundary information for an unbounded number of item “containers,” which are matched at run time with any content items in a slide. Once items are matched with their containers, they are ready to be displayed, as both content and display details are known. Note here that content and layout information are always stored separately.

Each container within a layout has a name associated with it. A container has a defined font, foreground and background colors, X and Y screen locations, and width and height values. When a content item and a container share the same name, they are matched. Once matched, the content item is displayed in an appropriate widget that is formatted and arranged according to the properties of the container.

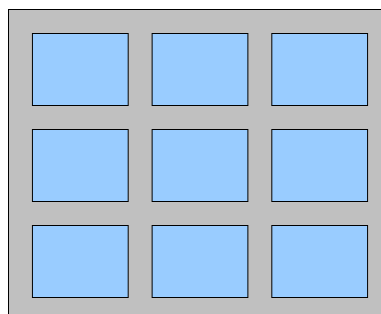
For instance, a slide might use the "ContentSlide" layout, which has a Title container at the top and a Content container below it. If the slide has a text content item named Title with the value "Sample Slide" and a file content item named Content with the value of an image filename, the slide would look similar to Figure 2:



*Figure 2: Sample Slide*

### **3.1.2 Slide Sets**

Sets are organizational units that contain slides in an unordered fashion. They can be defined extensionally by listing slides contained in the set. Set definitions are stored in the database and contain a list of slides in the set and criteria to use (if any) to order the list of slides.



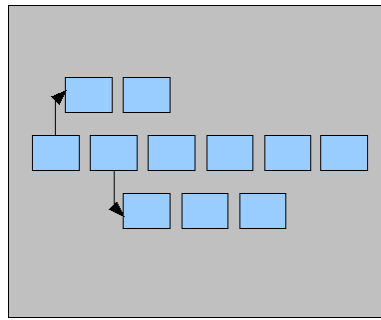
*Slide Set Icon*

We could create a set that includes the sample slide of Figure 2 above, along with any number of other slides, such as a title slide and two other content slides. The resulting set can be presented in any order.

### **3.1.3 Slide Shows**

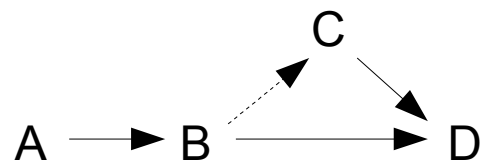
Shows in PresentLively are flexible in nature. Unlike “shows” or “decks” in other presentation software, which use strict linear ordering with implicit transitions, PresentLively slide shows can have any number of transition types spreading out into a directed graph, where each slide is a node and edges are the potential transitions. A show begins at a single designated start slide. From the starting slide, a “default,” or unlabeled transition defines the main “trunk” of the show, while any labeled transitions lead to alternate paths or sidebars, which are called “branches.” Branches not connected to the trunk are also allowed, thus the graph may be unconnected.

Each branch in a nonlinear slide show can be considered as its own show, with a trunk of its own that follows default transitions. Branches can still make use of labeled transitions, of course, and may have transitions back to the main trunk or even other branches. Note also that slides within the trunk of a show or branch may also have transitions to other trunk (or branch) slides. The trunk/branch ontology is useful in discussing the flow of a presentation, but it does not restrict the transitions.



*Slide Show Icon*

To produce a slide show using the slides from our slide set example of Section 3.1.2 , we would add all the slides to our slide show: one title slide and three content slides. For convenience, we will refer to the title slide as A and the three content slides as B, C, and D. For the slide show, we define A to be the start of the show. Suppose B contains information we want to present next. We would then add a default transition from A to B. Slide C contains optional information, so we will create a labeled transition from B to C with the label "extra." Slide D should always be presented, so we create default transitions from B to D and from C to D. The final slide show would have the structure shown in Figure 3:



*Figure 3: Sample Show*

### **3.2 Database Interaction**

To protect data integrity and maintain validity of data objects in memory, all interaction with the database is passed through a translation layer. Components above the translation layer – including, at the current time, the entire PresentLively application and modules – only interact with data objects like those defined in Section 3.1 .

All organizational objects (slides, sets, and shows) and text content items are stored in the PresentLively database. Other types of content are stored in the database as references or URIs. Thus, aside from actual referenced content, all of the data used by PresentLively is stored in the database.

### **3.3 Enhanced Presentation Flow**

Because the presentation style of PresentLively is quite different from those of other applications, there are additional considerations to be made regarding both presentation control and content navigation.

#### **3.3.1 Presentation Control**

Slide set and show navigation at presentation time is managed through an on-screen navigation display. While such a display does use valuable space on-screen, the primary advantage of easy presentation control is that it eliminates the need to return to the main user interface while the user is giving a presentation.

For sets, the on-screen display format is inspired by a technique of Relational Presentation as described by Aspire Communications [4]. A thumbnail of each slide in the set is shown in a selection area at the bottom of the screen. By clicking on a thumbnail, the presenter will be taken to that slide.

For shows, a representation of the show structure is displayed. Presenters can jump around the show by clicking on nodes within the show, as with the set display. This reminder of the show structure should also be useful when a presenter needs to remember which path a certain transition will lead her down.

### **3.3.2 Content Management**

Other on-screen displays include “slide search” panels which allow users to search names, tags, and slide contents to find slides he may want to display. Results are displayed in a format similar to that described for slide sets in the previous section, and clicking on a slide thumbnail will likewise display that slide.

This feature essentially enables on-the-fly set creation based on slide content. It could also be used to quickly retrieve slides for explanatory purposes when a full slide show is unneeded or impractical (e.g. due to time constraints).

## **3.4 Distinctive Features**

Beyond the distinctive branching presentation and set-based paradigms introduced above, PresentLively has several other features that distinguish it from other presentation packages.

### **3.4.1 Tagging**

Slides, sets, and shows can be “tagged” using text strings. This is similar to tagging mechanics used on web sites such as Flickr [5] and del.icio.us [6], and provides another way to organize and search through content. Tags can be applied to any item that is stored in the PresentLively database.

### **3.4.2 Dynamic Content**

While normal (static) content links are managed through the PresentLively database, modules can be added to the system that display dynamic content. These modules can be added to a slide

so that it will e.g. play streaming video from the Internet, provide an interactive web search box, or display all pictures from a designated folder on the user's hard drive.

### **3.5 Modular Design**

One of the main ideas embodied in the design of PresentLively is that it should be easy to extend. We wanted third parties to have the ability to create extensions that would add substantial new functionality to the application. To accomplish this, we designed a modular framework that leverages the .NET framework's *reflection* features.

Reflection is a feature of many programming languages by which a program can observe and modify its own structure. In C#, objects of type `Type` can be used to retrieve information about classes and dynamically loaded assemblies at run-time. In this manner, a module library, which is a kind of .NET assembly, can be dynamically loaded and inspected to discover the classes it contains as well as their supported interfaces and methods.

Every module is a class that implements the `IModule` interface. On start-up, PresentLively scans the module directory for managed DLLs and attempts to examine them. Using reflection, a list of classes that implement `IModule` is produced. This list is then used to initialize each module.

## **4 Implementation**

Although the design described in Section 3 is the final design of PresentLively and was influenced by development, it naturally does not reflect all the challenges encountered. In this section we list more of the technical details and low-level design decisions involved in implementing PresentLively.

## 4.1 Database Structure

Because PresentLively advances the state of presentation software through new presentation methods and organizational paradigms, we have to carefully manage the often dynamic information that makes up the presentations. Many of the operations necessary for the full functionality and benefit of a dynamic presentation system, such as searching for a slide or aggregating the tags for a particular item, are database-like in nature.

In choosing the storage mechanism to be used for PresentLively, we had to choose whether the software would manage files on disk itself – and essentially perform database operations on those files – or whether an external database management system could be used. This is why, as Section 3.2 describes at a high level, everything is stored in the database – unless it's not. Actually, there is only a well-defined subset of information that is not contained within the database. This set of information is limited to file system objects referenced by file content items and external resources identified by URIs associated with URI content items.

Content external to PresentLively is only referenced in the database. Figure 4 represents the database structure using an entity-relation diagram.

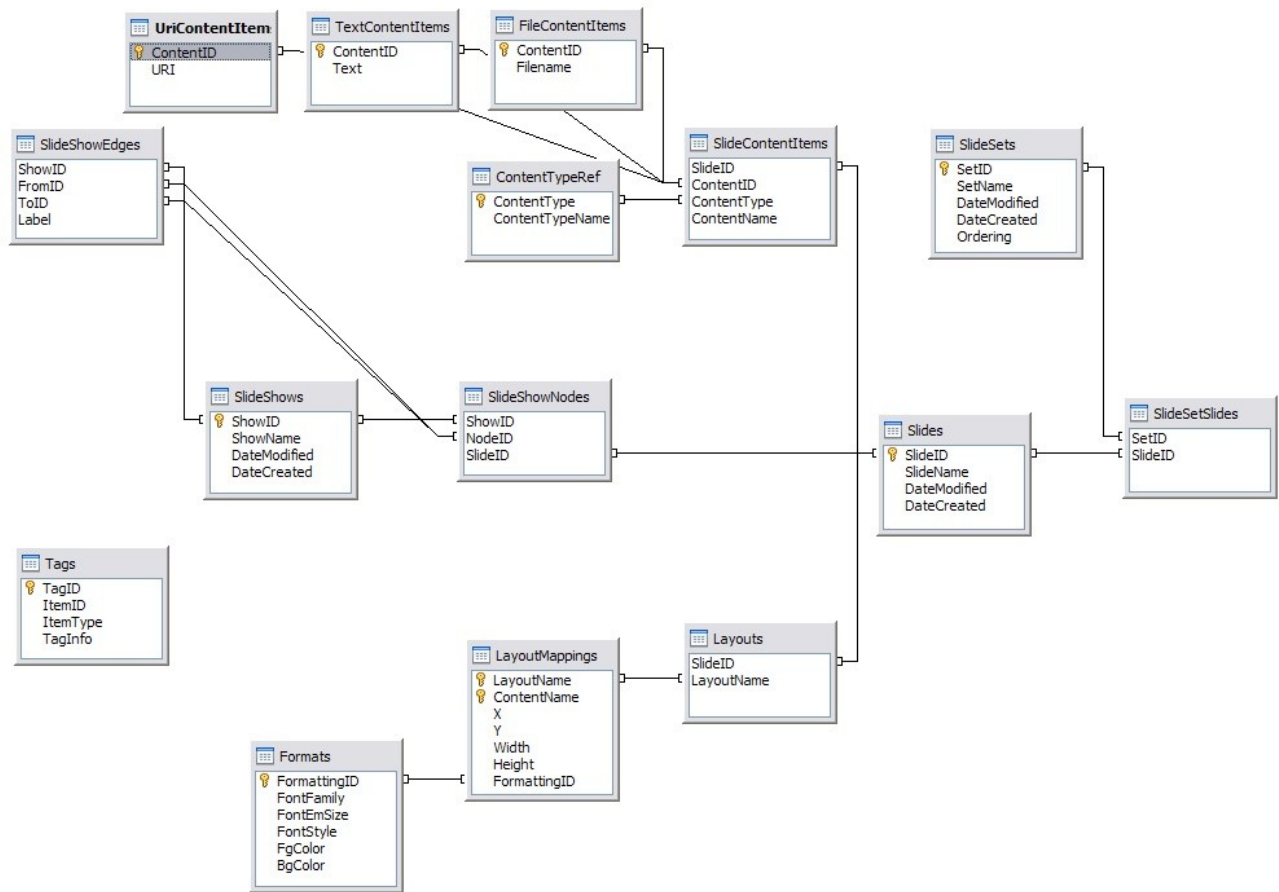


Figure 4: Entity-Relation Diagram

Slides are the building blocks of presentations. Therefore, the **Slides** table is central to the database schema. Slides have a name, creation and modification dates, a collection of content that is part of the slide, and layout information. Only the name, modification date, and creation date are stored in the **Slides** table itself. Content and layouts are stored separately and linked to individual slides via other tables.

Content storage (or reference storage in the case of files and URIs) is straightforward: each type of content (or reference) is stored in its own table with an ID. These tables are **TextContentItems**, **FileContentItems**, and **UriContentItems**.

Content is linked to a slide via the **SlideContentItems** table. Each record in **SlideContentItems** associates one piece of content, referenced using the content ID and a ContentType value, with a single slide and a “content name”. ContentType values are resolved using the **ContentTypeRef** table. Content names are text identifiers used to associate content with content containers in a layout.

Slide layouts are a challenge to store. In the original design, a system of “super-layouts” with a consistent naming scheme for content containers was planned. This would allow the user to change the super-layout of a slide and automatically rearrange the content in predefined containers, while still including other content that will not change. Because of implementation difficulties and the complexity of such a system, we changed the design so that all layouts are predefined. All layouts will act like the old super-layouts, but extra content cannot be included. Slides are assigned a layout via the **Layouts** table. Each record in the table simply associates a Slide ID and a Layout Name.

Each tuple in the **LayoutMappings** table contains location, size, and formatting information (formatting ID, lookup in the Formats table) for a single content container. This information is associated with a Content Name and a Layout Name, which together form the primary key. A **Layout** is defined as a collection of **Layout Mappings**. There are no separate records kept regarding individual layouts, although the application-level object model does include a Layout class. This class contains only the layout's name and a collection of mappings, which adequately reflects the layout structure.

Slide Sets are a simple affair: each set is described by an entry in the **SlideSets** table (name, modified, created, set ordering) and slides are included in a set by entries in the **SlideSetSlides** table.

Slide Shows are more complex. Because every show is a directed graph, it's necessary to store the graph structure in the database. In addition to a **SlideShows** table (name, modified, created) there are two tables used to describe the graph structure: **SlideShowNodes** and **SlideShowEdges**. Nodes have a node ID that is unique within a show and contain a slide ID. Edges, because they are directed, reference source (FromID) and destination (ToID) nodes. Edges also have a "label," which identifies the type of edge. If the label is empty or NULL, the edge is considered the default edge for the source node and is used when the presenter does not specify an edge to follow when moving through the slide show. If only one edge is present, it is considered the default edge; otherwise the presenter is given a choice between available edges (which may be displayed as the destination nodes they point to). If there are multiple default edges, the presenter must select which one to follow.

## 4.2 User Interface

While user interface design is not the focus of this project, we had to address several interface issues in PresentLively. The first of these issues is dynamic retrieval and display of module UI within the main PresentLively user interface.

Every module that supports user interaction outside of presentation mode must implement the `IModuleUI` interface. This interface signifies to PresentLively that the module has UI to display and defines a set of methods to describe the module's interface. The `IModuleUI` interface is shown in Listing 1 below:

### Listing 1. IModuleUI Interface

```
/// <summary>
/// Interface for modules needing GUI interaction outside Presentation Mode.
/// </summary>
public interface IModuleUI
```

```

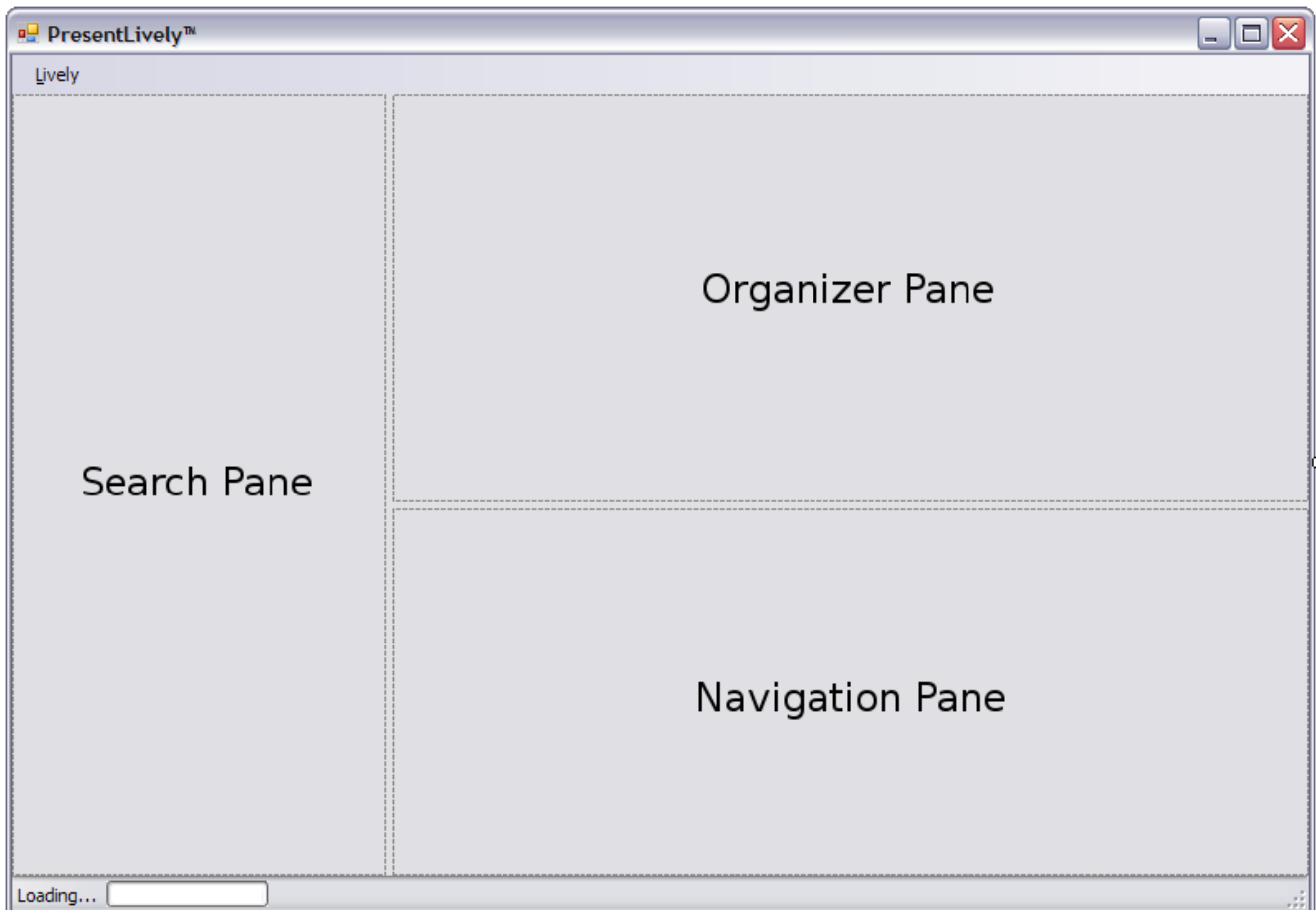
{
    /// <summary>
    /// Returns module user interface
    /// </summary>
    UserControl GetModuleUI();

    /// <summary>
    /// Returns name to display on tab, window, etc. containing module UI
    /// </summary>
    string GetName();

    /// <summary>
    /// Returns preferred on-screen location.
    /// </summary>
    ModuleUILocations GetLocation();
}

```

The `GetModuleUI` method returns a `UserControl` object that can be inserted into the `PresentLively` interface directly. The `GetName` method returns a string that is used as the title of that module's tab within the interface. The `GetLocation` function returns an enumeration value from `ModuleUILocations` that indicates what portion of the screen the module would like its UI to occupy. Figure 5 shows the three values of `ModuleUILocations` used within `PresentLively`:



*Figure 5: PresentLively User Interface*

As each module is loaded using the process described in Sections 3.5 and 4.3.2, it is inspected to see if the module implements the `IModuleUI` interface. If so, the module's preferred location is retrieved and mapped to one of the panes within the `PresentLively` interface. A new tab is created within the pane, along with a new tab container if necessary, and the module's UI name is used to identify the tab to the user. Finally, the `UserControl` supplied by the module is placed inside the newly created tab and made visible.

Another issue encountered in implementation was the process of loading dynamic object modules. Known types, such as text content items or file content items, use the built-in `TextWidget` and `FileWidget` view objects, respectively. Since `PresentLively` can't know how a third-party module wants to interact with the screen, some process was required by which modules which are unknown at compile time could be given direct access to the screen.

This issue was resolved in much the same way as module UI: every module that supports display of dynamic objects must implement the `IDynamicObjectModule` interface, which is shown in Listing 2 below:

### Listing 2. `IDynamicObjectModule` Interface

```
/// <summary>
/// Interface for modules which can display URI content items.
/// </summary>
public interface IDynamicObjectModule
{
    /// <summary>
    /// Returns a widget that displays the content associated with the given URI
    /// </summary>
    DisplayWidget GetWidget(string uri);
}
```

Because of prior design decisions, `TextWidget` and `FileWidget` were already child classes of a general `DisplayWidget` class, which in turn inherits from `Control`, the basic UI building block of the Windows Forms display environment used in .NET development. To allow modules to create

their own widgets, the `GetWidget` method of `IDynamicObjectModule` returns a `DisplayWidget`. Modules should create classes that inherit from `DisplayWidget` to display objects on-screen.

Without exception, dynamic objects are stored as URI content items. Modules are specified within a URI by using the string `"lively://host/modules/moduleName"`, where `'host'` is the server from which to obtain the module and `'moduleName'` is the class name of the desired module. Currently only local requests are supported, i.e., `host` is equal to `'localhost'`.

Once the correct module is identified (assuming it exists) and a widget is obtained, the entire URI is placed in the widget's `Text` property. Thus, additional information for the module can be placed at the end of the URI as a query string. For example, the `WebObjectModule` class in the Core Modules set (see Section 4.4 ) supports URI content items of the form `'lively://localhost/modules/WebObjectModule?url=X'`, where `X` is the address of a web page to be displayed in a web browser control.

### **4.3 Module Management**

The design of `PresentLively` is based entirely on system modularity. Thus, even the core components of the system are implemented as modules. These are stored in `CoreModules.dll`. Because all but the most basic functions are handled in modules, there is some assurance that future module developers will be able to add significant new functionality through modules. Additionally, this structure gives us the freedom to remove, rewrite or replace any individual module without breaking functions outside that module.

### 4.3.1 Class Structure

The `IModule` interface defines an `Init` function with a single argument of type `IModuleCallback`. This callback is the module's only method of communication with the parent process, which will generally be an instance of `PresentLively`. Using the callback, a module can access the data layer and the message bus.

In the initial design, modules did not pass information among themselves and could be isolated in their own “information silos.” As development progressed, however, we recognized the need for at least an event-notification level of module communication, possibly including the ability to transmit and broadcast data. To remedy this, we introduced the `MessageBus` class. This class, as introduced, is simply a pass-through mechanism with a single event, `Message`, and a single function, `PostMessage`. Modules register with the message bus by connecting to the `Message` event using the C# operator “+=” in conjunction with an event handler defined by the module. The event handler is called every time a message is sent. While this approach is less efficient than filtering the messages, it is simple to write and understand, and the volume of messages within a single `PresentLively` instance is expected to be relatively small. A full description of the `MessageBus` class is given in Section 4.3.3 .

### 4.3.2 Module Loading and Activation

When `PresentLively` is run, all DLLs within the current working directory are scanned for classes that implement the `IModule` interface. All class types found using this method are then initialized as active modules. Initially `PresentLively` stored only the `Type` of a module until an instance was needed, at which point it was created on-the-fly. This proved to be troublesome and

the decision was eventually abandoned once we realized that, due to the message bus architecture, all modules needed to be initialized on startup.

Using the old strategy, a module can't register to receive "wakeup" messages such as `DisplayItem`. Not only that, but not initializing modules immediately meant that `PresentLively` tended to do things like look through the master list of `Type` objects and instantiate all modules of a certain type when needed. This could lead to having multiple instances of a single module, which is probably an unexpected event for module developers. The final designed process ensures that only one instance of a module will exist at a time, and that all modules receive the same callback object (and thus connect to the same message bus).

### **4.3.3 Message Bus**

To facilitate inter-module communication, we created the `MessageBus` class. The message bus is a simple pass-through subsystem, as described briefly in Section 4.3.1 . Several implementation details are of note, however.

Systems using a message bus architecture typically define the bus operations using an interface, but in this case the message bus makes use of a .NET framework "event." Events are defined using a delegate function type. When an event fires, all delegate functions of the proper type that have been associated with the event are called with appropriate arguments. While events can be defined for interfaces, delegate function types cannot. Therefore, the message bus is defined as a class rather than an interface.

Note that while the event handler is called in a different thread and must therefore be thread-safe, this is standard practice in C# development; all event handlers must be thread-safe. In the

case of UI controls, a run-time exception is generated if a non-thread-safe operation is attempted. Thus, this method of notification does not introduce additional opportunities for data corruption.

#### 4.3.4 Database and Display Interaction

Section 3.2 described the abstraction model used by PresentLively, namely that all in-memory operations are performed on object types and that these types are then translated by the `DataLayer` class to and from an appropriate representation in the database. This translation layer is useful both because modules do not need to incorporate redundant database access code and because it can be used to guard against database corruption. Modules have direct access to the Data Layer via the module callback. The `DataLayer` class is an implementation of the `IDataLayer` interface, which is shown in Listing 3 below:

##### Listing 3. IDataLayer Interface

```
/// <summary>
/// Translation layer for interfacing with a database.
/// </summary>
public interface IDataLayer
{
    /// <summary>
    /// Returns a full list of slides in the database.
    /// </summary>
    Slide[] GetSlides();

    /// <summary>
    /// Returns the slide with the given ID
    /// </summary>
    Slide GetSlide(int SlideID);

    /// <summary>
    /// Returns a full list of slide sets in the database.
    /// </summary>
    SlideSet[] GetSlideSets();

    /// <summary>
    /// Returns a list of the slides contained in the slide set with the given ID.
    /// </summary>
    Slide[] GetSlideSetSlides(int SetID);
}
```

```

/// <summary>
/// Returns a full list of slide shows in the database.
/// </summary>
SlideShow[] GetSlideShows();

/// <summary>
/// Returns the slide show with the given ID.
/// </summary>
SlideShow GetSlideShow(int ShowID);

/// <summary>
/// Returns a list of the text content items contained in the slide with the given ID.
/// </summary>
TextContentItem[] GetSlideTextContentItems(int SlideID);

/// <summary>
/// Returns a list of the file content items contained in the slide with the given ID.
/// </summary>
FileContentItem[] GetSlideFileContentItems(int SlideID);

/// <summary>
/// Returns a list of the URI content items contained in the slide with the given ID.
/// </summary>
UriContentItem[] GetSlideUriContentItems(int SlideID);

/// <summary>
/// Creates a new slide in the database
/// </summary>
int CreateSlide(string SlideName, string LayoutName);

/// <summary>
/// Saves a slide along with accompanying content items.
/// Will not create slides or content items (use the Create* functions for that).
/// </summary>
void SaveSlide(Slide s, TextContentItem[] tci, FileContentItem[] fci, UriContentItem[]
uci);

/// <summary>
/// Creates a new text content item in the database
/// </summary>
/// <param name="data">Text</param>
/// <returns>Content ID</returns>
int CreateTextContentItem(string data);

/// <summary>
/// Creates a new text content item in the database
/// </summary>
/// <param name="data">Filename</param>
/// <returns>Content ID</returns>
int CreateFileContentItem(string data);

/// <summary>
/// Creates a new text content item in the database
/// </summary>
/// <param name="data">URI</param>

```

```

/// <returns>Content ID</returns>
int CreateUriContentItem(string data);

/// <summary>
/// Associates the given text item with a slide.
/// </summary>
void AddTextContentToSlide(int SlideID, int ContentID, string ContentName);

/// <summary>
/// Associates the given file item with a slide.
/// </summary>
void AddFileContentToSlide(int SlideID, int ContentID, string ContentName);

/// <summary>
/// Associates the given URI item with a slide.
/// </summary>
void AddUriContentToSlide(int SlideID, int ContentID, string ContentName);

/// <summary>
/// Saves the data of a content item.
/// Will not create an item (use the Create* functions for that).
/// </summary>
void SaveContentItem(TextContentItem i);
void SaveContentItem(FileContentItem i);
void SaveContentItem(UriContentItem i);

/// <summary>
/// Delete a slide and all text content items associated with that slide.
/// </summary>
void DeleteSlide(int SlideID);

/// <summary>
/// Delete a slide set. Slides belonging to the slide set are NOT deleted.
/// </summary>
void DeleteSlideSet(int SetID);

/// <summary>
/// Delete a slide show. Slides belonging to the slide show are NOT deleted.
/// </summary>
void DeleteSlideShow(int ShowID);

/// <summary>
/// Gets a list of all valid layout names.
/// </summary>
string[] GetLayoutNames();

/// <summary>
/// Gets the layout with the given name.
/// </summary>
Layout GetLayout(string LayoutName);

/// <summary>
/// Saves the slide set with the given ID and sets the contents
/// to be the given list of slides (by ID).
/// </summary>

```

```

void SaveSet(int id, int[] slides);

/// <summary>
/// Saves the slide show.
/// </summary>
void SaveShow(SlideShow show);

/// <summary>
/// Creates a new set in the database.
/// </summary>
/// <param name="name">Set name</param>
/// <returns>Set ID</returns>
int CreateSet(string name);

/// <summary>
/// Creates a new show in the database.
/// </summary>
/// <param name="name">Show name</param>
/// <returns>Show ID</returns>
int CreateShow(string name);

/// <summary>
/// Removes text content with the given ID from a slide
/// </summary>
/// <returns>True if removed</returns>
bool RemoveSlideTextContentItem(int SlideID, int ContentID);

/// <summary>
/// Removes file content with the given ID from a slide
/// </summary>
/// <returns>True if removed</returns>
bool RemoveSlideFileContentItem(int SlideID, int ContentID);

/// <summary>
/// Removes URI content with the given ID from a slide
/// </summary>
/// <returns>True if removed</returns>
bool RemoveSlideUriContentItem(int SlideID, int ContentID);

/// <summary>
/// Adds the tag to the given slide
/// </summary>
void AddTagToSlide(int SlideID, string Tag);

/// <summary>
/// Adds the tag to the given set
/// </summary>
void AddTagToSet(int SetID, string Tag);

/// <summary>
/// Adds the tag to the given show
/// </summary>
void AddTagToShow(int ShowID, string Tag);

/// <summary>

```

```

/// Gets the tags associated with the specified slide
/// </summary>
/// <returns>List of tags</returns>
string[] GetSlideTags(int SlideID);

/// <summary>
/// Gets the tags associated with the specified set
/// </summary>
/// <returns>List of tags</returns>
string[] GetShowTags(int ShowID);

/// <summary>
/// Gets the tags associated with the specified show
/// </summary>
/// <returns>List of tags</returns>
string[] GetSetTags(int SetID);

/// <summary>
/// Remove the tag from the specified slide
/// </summary>
void RemoveSlideTag(int SlideID, string Tag);

/// <summary>
/// Remove the tag from the specified set
/// </summary>
void RemoveSetTag(int SetID, string Tag);

/// <summary>
/// Remove the tag from the specified show
/// </summary>
void RemoveShowTag(int ShowID, string Tag);

/// <summary>
/// Gets a list of all tags in the database
/// </summary>
string[] GetTags();

/// <summary>
/// Gets a list of all slide IDs associated with the given tag
/// </summary>
int[] GetTaggedSlides(string Tag);

/// <summary>
/// Gets a list of all set IDs associated with the given tag
/// </summary>
int[] GetTaggedSets(string Tag);

/// <summary>
/// Gets a list of all show IDs associated with the given tag
/// </summary>
int[] GetTaggedShows(string Tag);
}

```

Once the MessageBus of Section 4.3.1 was implemented, all communication between the framework and modules or between modules was routed through the message bus. By defining generic message types, the message bus has allowed communication between modules to take on the same flavor as interaction with the data translation layer: a module is not concerned with how a message is processed or which module processes it, only that it gets sent.

Figure 6 depicts a typical user interaction in which the user selects a slide set, chooses to display the set, then uses the set display module to select another slide from the set to display.

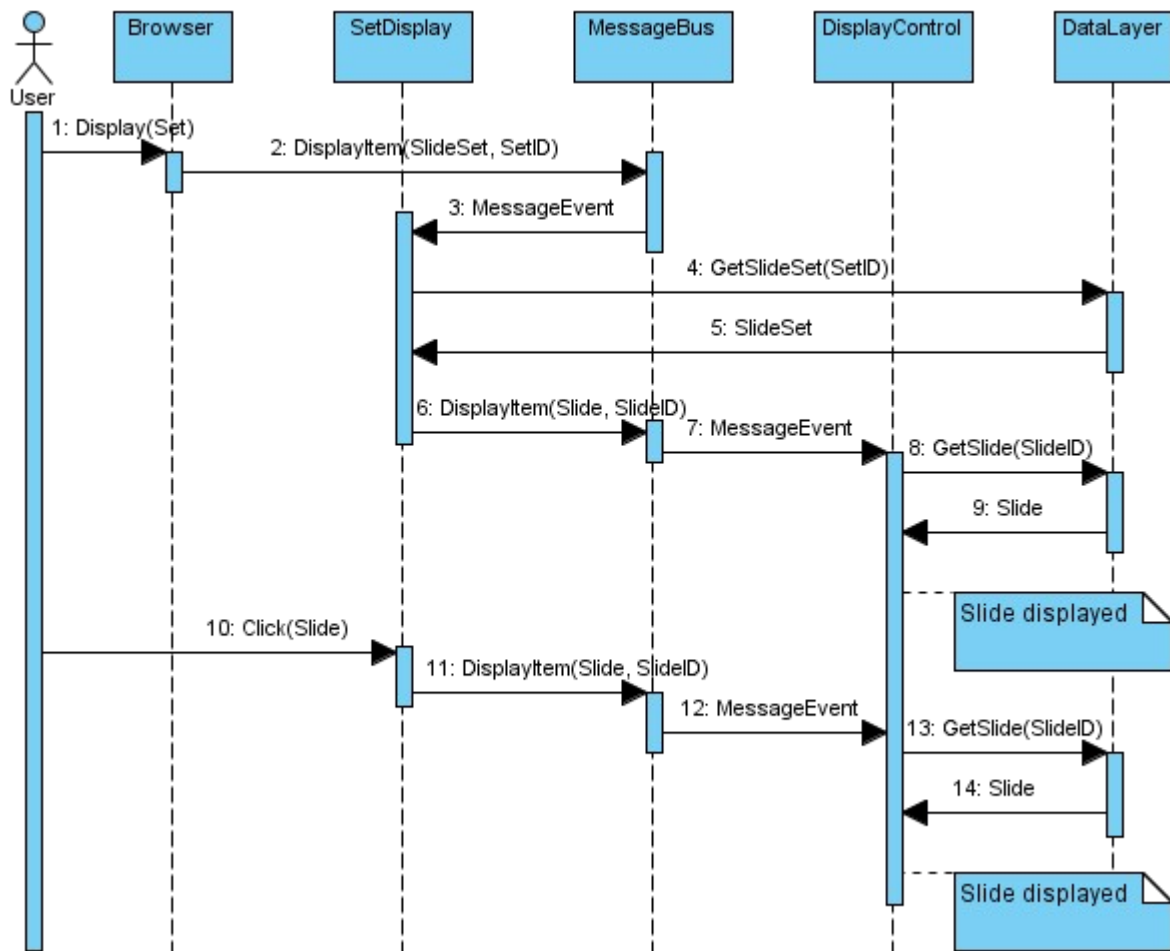


Figure 6: Module Communication

This interaction will provide a good example of all types of module interaction. For clarity, the text descriptions in Figure 6 describe only the semantics of an action or function call and do not

represent specific functions. The interaction begins when the user right-clicks a set in the Browser module (see Section 4.4.2 for a description) and selects the option to “Show” the slide set. The Browser module then sends a “DisplayItem” message to the message bus with parameters for the item type and item ID. In this case, the type is SlideSet and the ID is the SetID of the selected set. After sending the DisplayItem message, the user's interaction with the Browser module is complete.

The Set Display module is registered with the message bus and listens for the DisplayItem message. When the message bus sends out the message generated by the Browser module, the Set Display module inspects it and determines based on the message parameters that it should take action. The Set Display module then calls the GetSlideSet method on the Data Layer object to obtain a copy of the slide set. It selects a slide from the slide set to display, and sends its own DisplayItem message with parameters “Slide” and the selected slide's ID.

This message illustrates an important point: rather than allow modules to fight over the display area, PresentLively regulates what is displayed on the screen using the DisplayControl class. This display controller is also registered with the message bus, and it responds only to DisplayItem messages with an item type of “Slide.”

After the display controller receives the slide display message, it communicates with the data layer to retrieve the slide and its contents. It then puts PresentLively into presentation mode by covering the screen with a non-modal window on which it displays the slide and any active INavigationModule modules.

Because SetDisplay implements INavigationModule and set itself to active upon receiving the SlideSet typed DisplayItem message, the Set Display interface (see Section 4.4.3 for a description)

is displayed in the navigation area. This interface allows the user to select other slides from the same show. In this example, the user selects another slide to be displayed.

When the user clicks on a slide in the Set Display navigation interface, the Set Display module sends a DisplayItem message to the message bus with arguments specifying an item type of "Slide" and the SlideID of the newly selected slide. Upon receiving this new message, the display controller remains in presentation mode, pulls the new slide and content from the database, and displays the new slide on-screen.

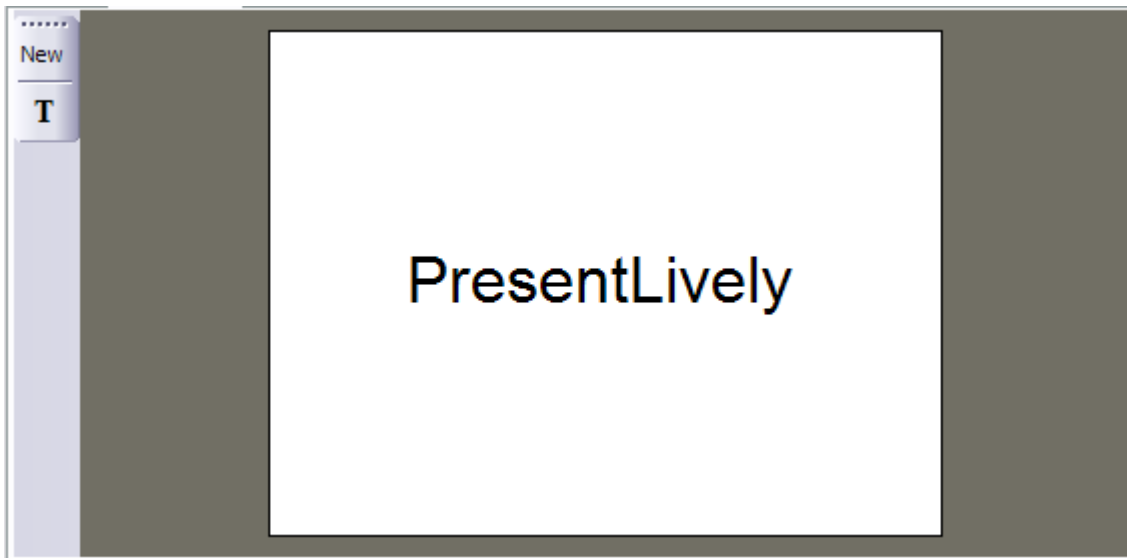
The above example illustrates every type of interaction available for modules to use. Note that since modules cannot access the PresentLively database directly, it is impossible for modules to add metadata to the database. Instead, modules must use a separate database or file operations to store persistent data. One method of storage is the .NET framework's Settings mechanism, which assemblies can leverage to automatically serialize and deserialize data.

## **4.4 Core Modules**

As described in Sections 3 and 4.3, the core functionality of PresentLively is contained in modules which are stored in CoreModules.dll. This section describes the content and functionality of the core modules.

### **4.4.1 Slide Editor**

The slide editor allows users to create and modify slides. The user interface for this module is shown in Figure 7 below:



*Figure 7: Slide Editor Interface*

Clicking on the "New" button in the Slide Editor panel presents the user with a dialog where they can select the layout of the new slide and enter its name. The "T" (or Text) button presents a dialog which allows the user to select a content container from the layout and associate text.

Layouts are defined in the database and cannot be altered using the PresentLively interface. All changes made to a slide using the Slide Editor are immediately written to the database.

#### **4.4.2 Browser**

The browser displays all slides, sets, and shows in the database and allows users to organize, edit, view, rename, or delete them.

#### **4.4.3 Set Display**

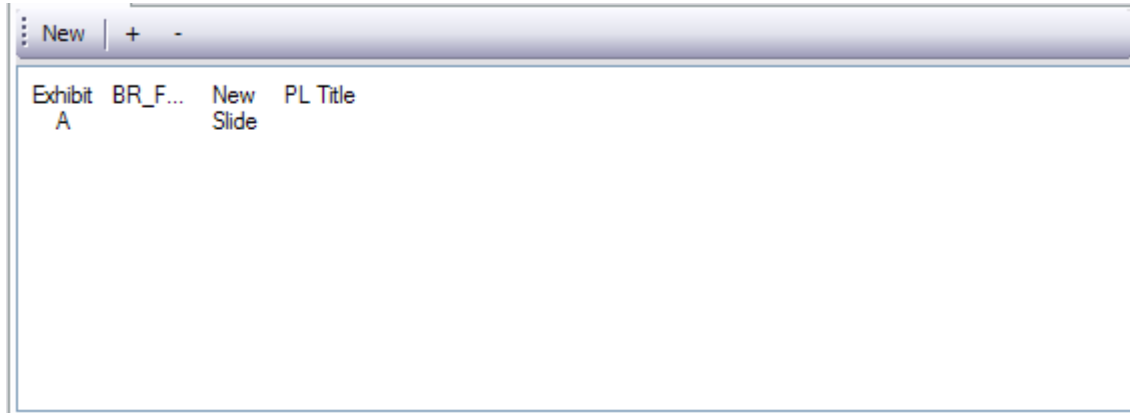
While the set display module has a user interface, it is the first module we will discuss that is not an IModuleUI module. Instead, it implements the INavigationModule interface, which alerts

PresentLively that when a set is being displayed on the screen, this module's navigation interface should be shown.

The set display navigation interface arranges the title of each slide in the set in rows across the bottom of the screen. Clicking on a slide title will display that slide.

#### 4.4.4 Set Editor

The set editor allows users to create new sets and alter the composition of existing sets. Figure 2 depicts the user interface of the set editor.



*Figure 8: Set Editor Interface*

The "New" button prompts the user for the name of the new set to create. The buttons with plus and minus symbols on them add and remove slides, respectively. All changes are immediately saved to the database.

#### 4.4.5 Show Display

Another INavigationModule, the Show Display module performs a similar role for slide shows that the Set Display module performs for slide sets. The show display navigation interface displays a visualization of the slide show as a directed graph. Recall that, conceptually, the slide

show is a directed graph of slides, so this representation makes sense. The next section will describe how users build slide shows visually as a directed graph, so displaying a graph at presentation time also increases consistency and perceived system cohesion.

Presenters can click on a graph node (slide) in the show navigation interface to display that slide, or they can select one of the available transition types for the current slide to display the slide that is connected to the current slide via that transition. The 'default' transition type may or may not be included in this list.

A slide with no outbound transitions generally constitutes the end of the show, so it would be logical to include a 'default' transition on such slides that ends the show, however it may be the case that the presenter has followed a dead-end branch and wishes to return to the trunk or even switch to a different presentation entirely. For this reason, the show display module does not insert any artificial transitions. Users must explicitly exit presentation mode.

The show display module uses controls from the Northwoods Software GoDiagram code library [7] to display the show diagram.

#### **4.4.6 Web Object Module**

As mentioned in Section 4.2 , the WebObject module is designed to generate widgets for certain URI content items. When the content item's URI is of the proper form, the WebObject module will display a web page embedded in the slide using the .NET framework's WebBrowser control.

This functionality allows PresentLively to do things no other desktop presentation application can do. Aside from the obvious benefit of referencing live web pages, the WebObject module also supports Java applets, Flash objects, dynamic JavaScript, and even more exotic web technologies

such as Microsoft Silverlight. Using PresentLively, users can embed YouTube videos and Java physics demonstrations directly into their slides.

## 5 Evaluation: Creating a New Module

Creating a new module for PresentLively only requires a copy of the PresentLively.Libraries DLL and a compiler for .NET DLLs. This section demonstrates the ease in creating new modules independently of the main PresentLively codebase. This tutorial is described assuming that Microsoft Visual C# Express Edition 2008 will be used to develop a slide search interface for use during presentation mode.

The choice of C# as a programming language is arbitrary; since modules are stored in .NET DLLs (Assemblies), a module could just as easily be built using Visual Basic, Managed C++, J#, or even Python via the IronPython project. All instructions will assume use of C# in the Visual Studio programming environment.

To begin development, create a new Class Library project called "SearchNavigationModule". Create a class within the project called SearchNavigationModule. This class will be the actual module type. Now we need to implement IModule. In order to access the interface definition, place a copy of PresentLively.Libraries.dll in the main SearchNavigationModule solution directory. It may be necessary to "Save All" before the solution directory is created.

Add a reference to the DLL by right-clicking on "References" below the project title and selecting "Add Reference...". In the Browse tab, find the SearchNavigationModule solution directory and select PresentLively.Libraries.dll. Make the interface definitions accessible by adding the following lines to the "using" declarations in your class file:

```
using PresentLively.Libraries.Interfaces;  
using PresentLively.Libraries.Interfaces.Modules;
```

Because navigation modules interact with the system via UserControls, the class also needs to reference the System.Windows.Forms namespace. Add this reference using the .NET tab in the Add Reference dialog.

Add the IModule and INavigationModule interfaces to the SearchNavigationModule class and have Visual Studio insert method stubs for each interface. Now the basic outline of our module is present. For convenience, declare a local variable "cb" and a public get-only property "Callback", both of type IModuleCallback.

Also create a user control named SearchNavigationControl and define a constructor for it that takes an instance of SearchNavigationModule as an argument. We'll fill in the details of the user control later.

## 5.1 Module Class Code

Let's fill in the module class code function by function. Listing 1 at the end of the section contains a listing of all the code described in this section.

**Init:** Initialization for this module is simple: store the callback that we receive.

**URL:** This property defines a URL from which to download module updates. For now we will return NULL, as there are no versions of the module that can be downloaded.

**Active:** When a navigation module's Active property is set to true, the display controller will show that module in the lower portion of the screen. Some modules, such as the Set Display and

Show Display modules, are only active when presenting a set or show, respectively. Our module is useful all the time, so this property can simply return true.

**ClearingDisplay:** This function is called when the user exits presentation mode. Modules like the Set and Show Display modules mentioned above will usually need to change their Active property when the display is cleared. SearchNavigationModule can safely do nothing here.

**GetName:** Return a name that the user will recognize when it shows up on a tab in presentation mode. In this case, return "Search".

**GetNavigationUI:** Return a new instance of SearchNavigationControl, passing in the current SearchNavigationModule instance as an argument. This facilitates communication.

These modifications will result in module code similar to that in Listing 4:

#### Listing 4. SearchNavigationModule.cs

```
using System;
using System.Collections.Generic;
using System.Text;

using PresentLively.Libraries.Interfaces;
using PresentLively.Libraries.Interfaces.Modules;

namespace SearchNavigationModule
{
    /// <summary>
    /// Module allowing users to search for and display slides from the database
    /// while in presentation mode.
    /// </summary>
    public class SearchNavigationModule : IModule, INavigationModule
    {
        private IModuleCallback cb;

        public IModuleCallback Callback
        {
            get { return cb; }
        }

        #region IModule Members

        public void Init(IModuleCallback callback)
        {
```

```

        cb = callback;
    }

    public string URL
    {
        get { return null; }
    }

#endregion

#region INavigationModule Members

    public bool Active
    {
        get { return true; }
    }

    public void ClearingDisplay()
    {
    }

    public string GetName()
    {
        return "Search";
    }

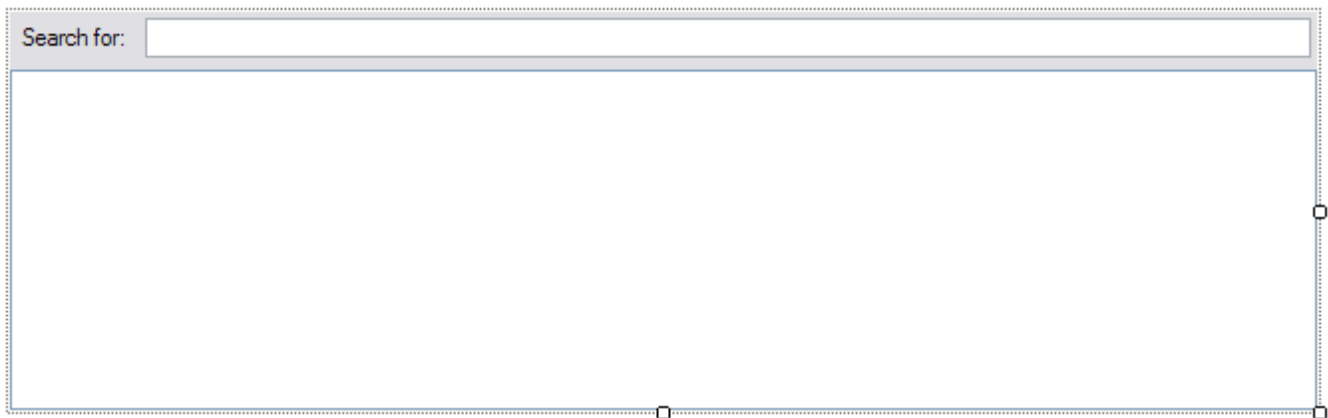
    public System.Windows.Forms.UserControl GetNavigationUI()
    {
        return new SearchNavigationControl(this);
    }

#endregion
}
}

```

## 5.2 User Control

The interface will be simple: a search box and a list of slides. Open SearchNavigationControl in design view and add a TextBox control and a ListView control to the container. We also added a label to clarify the purpose of the text box. Adjust the various properties of the controls to taste; note particularly the Anchor property, which allows relative positioning of control edges. Also ensure that the ListView control has its View property set to "List." The result of these changes is shown in Figure 9:



*Figure 9: Search Navigation User Control (Design View)*

There are improvements that can be made to the interface in Figure 9, but it will serve well enough for demonstration. Make sure that the user control constructor saves a reference to the parent module (an instance of `SearchNavigationModule`). Add handlers for the text box's `KeyDown` event and the list view's `ItemSelectionChanged` event.

Within the text box `KeyDown` event handler, add a simple check on `e.KeyCode` to detect the Enter key and call a private function, `doSearch(string searchText)` when the user presses Enter. Add a private variable of type `List<Slide>` named 'results' to hold search results and create a new private function, `refreshResults()`, to handle updating the view. As for `doSearch`, we handle the search operation in a simple manner using an anonymous delegate, as shown in Listing 5:

#### **Listing 5. doSearch Function**

```
private void doSearch(string searchText)
{
    results = new List<Slide>(parentModule.Callback.DataLayer.GetSlides()).FindAll(
        delegate(Slide s) { return s.SlideName.Contains(searchText); });
    refreshResults();
}
```

Again, there are many optimizations which could be performed. For instance, the module could cache the list of slides and add or remove slides according to messages it receives. The module's usefulness could be improved by allowing searches on fields other than the slide title. These are

areas which can be improved once one is familiar with the architecture. The refreshResults function is equally sparse, as Listing 6 shows:

### Listing 6. refreshResults Function

```
private void refreshResults()
{
    resultView.Items.Clear();
    ListViewItem i;
    foreach (Slide result in results)
    {
        i = new ListViewItem(result.SlideName);
        i.Tag = result.SlideID;
        resultView.Items.Add(i);
    }
}
```

This code just discards the current set of displayed results and re-generates the current view objects with the Slide ID as the item tag. These view objects are selectable by the user and, by convention, selection is a command that means "show this slide." In order to accomplish this, we must implement the list view's ItemSelectionChanged event handler. Inside the handler, simply find the first selected item (if any) and post a "DisplayItem" message to the message bus with arguments specifying a Slide item type and the item's tag as the Slide ID.

Once all the above tasks have been completed, the project can be compiled and the resulting DLL dropped into PresentLively's module directory. The code comprises a fully functional module.

## 6 Summary and Future Work

We began this report by describing the state of presentation software and the ways in which we desired to advance it through PresentLively. The primary objectives of this project were to allow users to break out of presentation linearity and to increase the dynamism of presentations,

including the dynamism of the content being presented. We have described in this report how the creation, organization, and content of presentations is more dynamic in PresentLively than in existing systems. This accomplishes the second objective. By producing the PresentLively framework, we have also enabled the first objective through our design of non-linear slide shows and slide sets. While our system currently has no interface for editing slide shows, the framework concepts have been proven. Future work in completing the user interface will accomplish the first objective.

PresentLively's automatic update feature for modules will also pose challenges. To ensure proper module functionality, some form of module versioning will need to be implemented. Additionally, a convention for managing and publishing module versions will have to be established to prevent inconsistencies and unnecessary retransmission of up-to-date modules.

Currently the database translation layer performs no caching of any kind. All operations are performed directly on the database. To improve performance we could write a caching layer that implements the `IDataLayer` interface. This caching layer could then be passed on to the modules without changing any module code, thereby improving all areas of the application, including third-party extensions.

One planned feature that was never implemented is thumbnail generation for slides. The current implementation uses lists of slide names for selection, but the original design called for thumbnail images of the slides to be displayed so users could more easily discern which slide they are selecting and whether it is the slide they want to display. This feature would have to be added both to the object model and the database translation layer.

While usability was not the focus of this project, future work in theming or similar user interface improvements for PresentLively would be easier if module UI were implemented in XAML or a similar design language. Many advanced visual effects are included in or enabled by the Windows Presentation Foundation (to which XAML is closely tied) which would be ideal for inclusion in PresentLively.

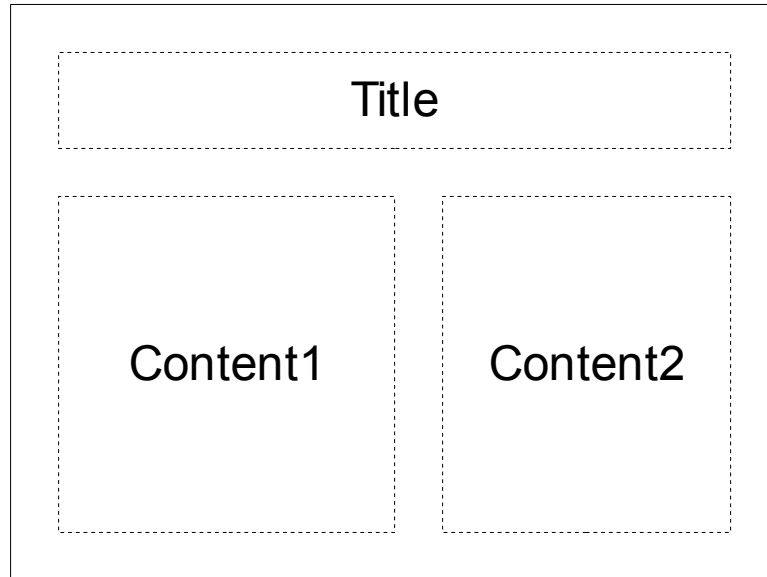
There are many situations in the business world where multiple users might want to share the same slides and even have them automatically updated. Additionally, although Internet access is widespread and there are many on-line file storage services available, there is still a demand for on-line presentation software. This state suggests an extension or adaptation of PresentLively to an Internet-based content organization and presentation system.

## References

- 1: Arundel, Rikki, "People should need a license to use PowerPoint", September 2008,  
<http://speakingandmarketingtips.blogspot.com/2007/01/people-should-need-licence-to-use.html>
- 2: Godin, Seth, "Really Bad PowerPoint", 2001
- 3: Microsoft Corporation, Microsoft PowerPoint Home Page, January 2009,  
<http://office.microsoft.com/en-us/powerpoint/>
- 4: Aspire Communications, Aspire Communications Website, September 2008,  
<http://www.aspirecommunications.com>
- 5: Yahoo! Inc., Flickr, January 2009, <http://www.flickr.com>
- 6: Yahoo! Inc., Delicious Bookmarking, January 2009, <http://del.icio.us>
- 7: Northwoods Software, GoDiagram web site, March 2009, <http://www.northwoods.com>



in layout creation. There is no "layout" object in the database. Instead, layouts are built dynamically using entries in the LayoutMappings table. To demonstrate the process of creating a new layout, we will use the following sample layout: a three section design with the title at the top and two content boxes side by side below the title. Figure 10 depicts this sample layout:



*Figure 10: Sample Layout*

In the database, our sample layout will have three entries in the LayoutMappings table, one for each container. We have chosen a name for each container, as well as the name "Sample Layout" for the layout. The remaining values for the new entries are the X and Y location - measured from the top-left corner of the slide to the top-left corner of the container - and the container's width, height, and formatting information. All four dimension values are specified using a value between 0 and 1000 that is scaled to the size of the slide when displayed.

Formatting information is stored in the Formats table and consists of font settings along with colors for the foreground and background. The FontFamily field is a string containing the font name, FontEmSize is an integer size, and FontStyle is a numeric indicator of font style, such as regular, italic, bold, etc. Colors are six-character strings in hexadecimal "RRGGBB" format. It will

be necessary to reference a Format entry from our containers. We will create two format entries to support the sample layout, as shown in Listing 6:

### Listing 6. Creating Formats

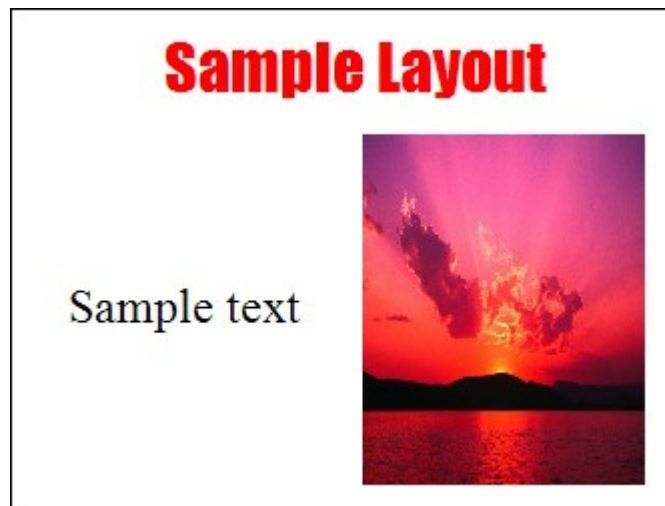
```
mysql> INSERT INTO Formats (FontFamily, FontEmSize, FontStyle, FgColor, BgColor)
VALUES ("Impact", 24, 0, "FF0000", "FFFFFF"), ("Times New Roman", 18, 0,
"000000", "FFFFFF");
```

After performing a SELECT statement to find the FormattingID of each format, we are ready to create the containers. This process is shown in Listing 7:

### Listing 7. Creating Containers

```
mysql> INSERT INTO LayoutMappings (LayoutName, ContentName, X, Y, Width, Height,
FormattingID) VALUES ("Sample Layout", "Title", 50, 50, 950, 150, 4), ("Sample
Layout", "Content 1", 50, 250, 425, 700, 5), ("Sample Layout", "Content 2",
525, 250, 425, 700, 5);
```

After completing the above operations successfully, our "Sample Layout" will appear in PresentLively as an option during new slide creation. Users can then use the slide editor to produce slides like that of Figure 11:



*Figure 11: Slide Using Sample Layout*