The Structural Analysis and Logical Controller Abstraction of Curator Software

Alex Staebell and Yi Liu
Department of Electrical Engineering and Computer Science
South Dakota State University
Brookings, SD 57007
alexander.staebell@jacks.sdstate.edu, yi.liu@sdstate.edu

Abstract

Since the inception of Software Engineering, computer systems and programs have been built with specific design decisions in mind. In a world where computing power and memory are inexpensive, customer needs change constantly, development cycles are shortened, and the number of consumer platforms steadily increase; software engineers must develop their systems to support modifiability, reusability, and flexibility design decisions. Curator, developed by the authors, is a web-centered management application made for professors and laboratory project leaders to manage files, budgets, instruments, and information of projects that they oversee. While Curator's design already exhibits strength in modifiability, reusability, and flexibility, this study examines methods to further improve these traits. This study analyzes the software architecture of Curator and improves its controller classes through logical controller abstraction. Through the abstraction, we establish patterns between each class by the method signatures they share. These patterns can then be quickly recognized by logically intelligent developers and through this recognition can more quickly learn how Curator works at the controller/view model level. Through an improved learning process, developers could then more easily modify Curator's functionality, reuse its code, and make it more flexible. This paper briefly presents the architecture of Curator and illustrates the details of the logical controller abstraction implementation. The paper further presents the logical intelligence theory and analyzes how the abstraction done in this study can effectively improve Curator's modifiability, reusability, and flexibility.

1 Introduction

Our study revolves around a program known as Curator. Curator is a web-centered data management application made for professors and laboratory project leaders to manage files, budgets, instruments, and info of projects that they oversee. The objective of this study is to research and implement architectural and design patterns to improve Curator’s modifiability, reusability, and flexibility. Throughout our research Curator proved to be well optimized in the aforementioned areas already. However, when examining Curator’s controller classes, we see an improvement opportunity through logical controller abstraction. Though the abstraction does little to improve Curator’s structure, there is evidence that this abstraction will improve a developer’s ability to learn how Curator works, which could make Curator easier to modify and reuse, and more flexible. This paper discusses background information in section 2, the architectural pattern in section 3, the case study detailing the logical controller abstraction implementation and accompanying logical intelligence theory in section 4, and giving a conclusion in section 5.

2 Background

Curator is useful for Professors and researchers who manage laboratory and research projects. For a single project created, a manager can create and manage a budget, track instruments being used, give a description, provide contact info, assign a status, post files, and specify user viewing and editing permissions for said project. Furthermore, this software can accommodate multiple projects for one or multiple managers. Most importantly, due the web-based nature of this application, clients can interact with the system in real time via the internet allowing them to view and/or edit data from nearly anywhere in the world. While there exists many programs and methods to do each task independently, Curator streamlines management overall by allowing users to perform all necessary tasks in one online program which constantly updates its data.

2.1 Curator Functionality

Curator’s Functions are summarized as follows according to Curator’s Software Requirements Specification [1]:

This software will provide project management to scientific research. The major uses of this software will be to assist Principal Investigators (PI) in managing and organizing the different projects they are involved with; be it past, present or future projects.

- An Authenticated User logs into the system and the initial view is of a list of projects that they are authorized to view and/or edit.
- The user then can select a project that they wish to access and the software will navigate the page to a different screen that displays the details of the project with a list of the different instruments that are connected with the project.

- Additionally, there will be a budget table that can be accessed, and edited, that the Authenticated User will be able to alter at any time [they] want.

2.2 Original Curator Design Decisions

When Curator was originally designed, its creators addressed reliability, availability, security, maintainability, and portability [1].

The goal regarding reliability was centered around a Mean Time Between Failures (MTBF) of once every 20,000 hours or 1 year with a Meant Time To Repair (MTTR) of 2 hours.

Curator was designed to be three nine's (99.9%) reliable, which equates allowing for roughly 9 hours of downtime per year.

The security decision was accomplished by implementing versioning and administrator-assigned security roles for users to protect data from being edited and/or viewed by unauthorized users.

Maintainability was secured by having code shared between the Curator client and server as well as have a test environment be built for function testing. It is noteworthy to state that extensibility and reliability improves through these design decisions as well.

While the first four decisions revolved around optimizing the attributes they were addressing, the last attribute, portability, was addressed as an attribute that could be implemented with the help of MONO, but portability was ultimately "not guaranteed and [would] not be tested".

3 Architectural Pattern

This section describes the architecture behind Curator. First we detail some system generalities behind Curator such as the system it was built on and the languages it utilizes. Secondly we examine Curator's model, view, view model or MVVM structure. Lastly we describe Curator's thin communication structure.

3.1 System Generalities

Curator was coded using two languages: C# and cshtml. C#, because of its object-oriented design, was used to code the model, controller, view model files for Curator. cshtml, html that is also capable of rendering ASP.net modules which provides handler mapping and MVC engine loading, was used to code the view files for Curator as described in [2]. Looking more at Curator's server side, its core database is MySQL and uses a Windows Communication Foundation (WCF) framework to encode user-to-server commands ferried by the client as expressed in [1]. To load its web content, Curator used server framework

IIS 7.5 Express as shown in [1]. Most noteworthy, Curator makes great use of ASP.NET 5 which has its MVC Web Pages and Web API was all merged together ensuring web page views can be created more quickly as described in [3].

3.2 MVVM Structure

Curator uses a variation of the Model View Controller (MVC) known as Model View View Model (MVVM) as its primary architecture. Figure 1 shows how MVVM parts work together as seen in [4]. The View portion of MVVM correlates to Curator's webpage coding that creates the interface to view and manipulate laboratory management data and files. Commands from the View are sent to the View model portion of MVVM where they are processed the view model code updates the appropriate models. View Model also formats data from the models so that the view code can interpret and display it correctly. Curator's model files outline the structure of each special data type to be managed in Curator's database and notifies the view model when the model state changes.

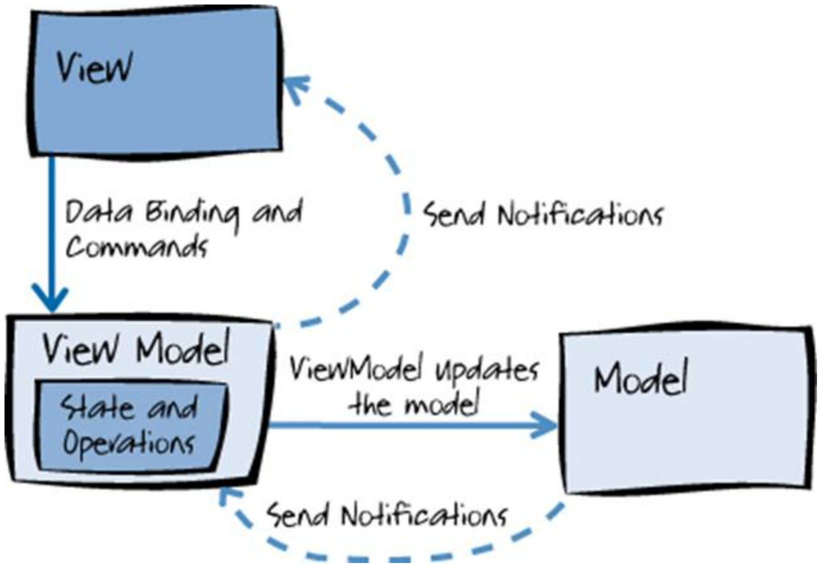


Figure 1: MVVM structure diagram from [4]

Curator's code files are different from a typical MVVM setup because it features both controller and view model sections rather than just a view model section. In Curator, the controller classes control the data access and mutation method logic while the view model classes define the complex data types used by the Curator controller and view classes to fulfill the view model duties defined in Figure 1.

3.3 Thin Communication Structure

While the underlying architectural structure of Curator is MVVM, it was defined conceptually as a thin client-server model as seen in Figure 2, which is described in [5]. All view web pages would be presented on the client-side while all viewmodel/controller processing and model data management would be done on Curator's server-side.

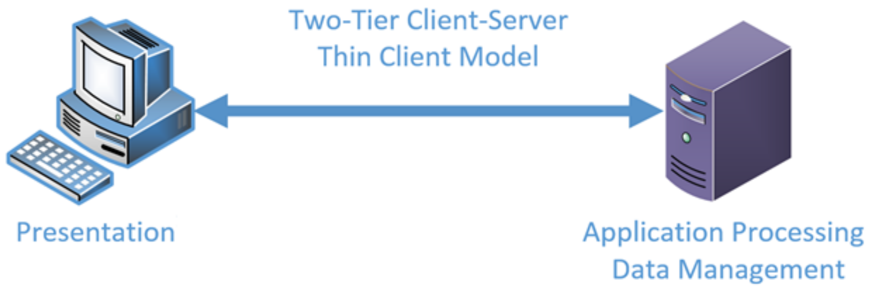


Figure 2: Curator's Thin Client-Server Model as described in [5]

4 Case Study: Implementing Controller Abstraction in Curator

This section leads with a quick preface on what other improvements were considered for Curator before establishing logical controller abstraction. We further describe what coding was implemented to make controller abstraction in Curator possible. The definition of logical learning, its connection to controller abstraction in Curator, and the theoretical benefits of implementing such a learning abstraction; are also discussed in this section.

Extensive research was done in fields of interfaces, generics, delegates, extension methods, and anonymous types to meet the Curator improvement goals. Unfortunately, none of the aforementioned methods could be applied to Curator's structure due to the specialized nature of Curator's controller and view model methods. However, through research into human learning, there is promise of improving modifiability, reusability, and flexibility through logical controller abstraction.

4.1 Logical Controller Abstraction Implementation

This logical controller abstraction relies on creating abstract classes to hold related method signatures that are shared between controller classes and then having the existing controller classes inherit and override the method signatures specified in the abstract classes.

Implementing logical controller abstraction required us to create four new abstract class files in our *Curator.Controllers* hierarchy that inherit the corresponding classes (note that *Controller.cs* is a pre-existing abstract class that must be inherited to ensure the existing controller classes can perform correctly). We also modified six existing class files in the *Curator.Controllers* hierarchy.

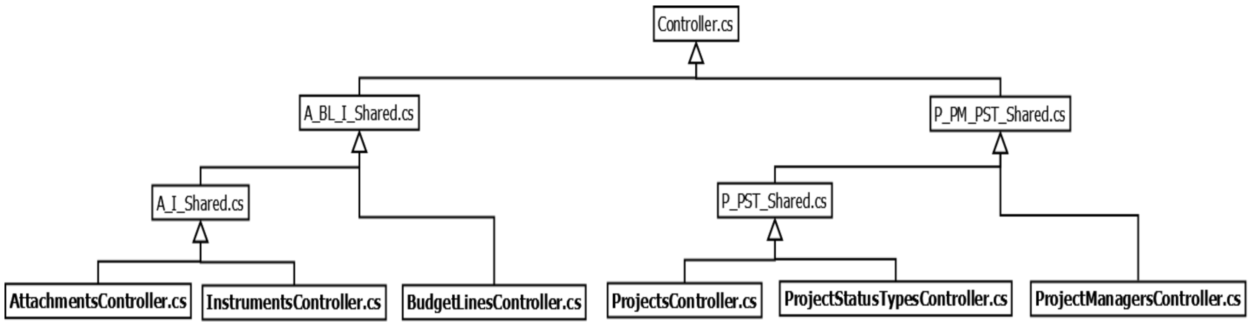
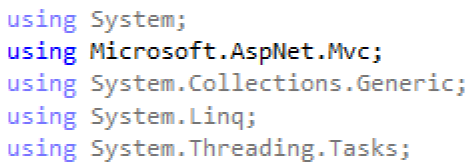


Figure 3: Logical Controller Abstraction File Hierarchy

Notice that the naming convention of the abstract class files correspond to the existing class files that share method signatures, for example in A_BL_I_Shared.cs:

• A	= Attachments
• BL	= Budget Lines
• I	= Instruments

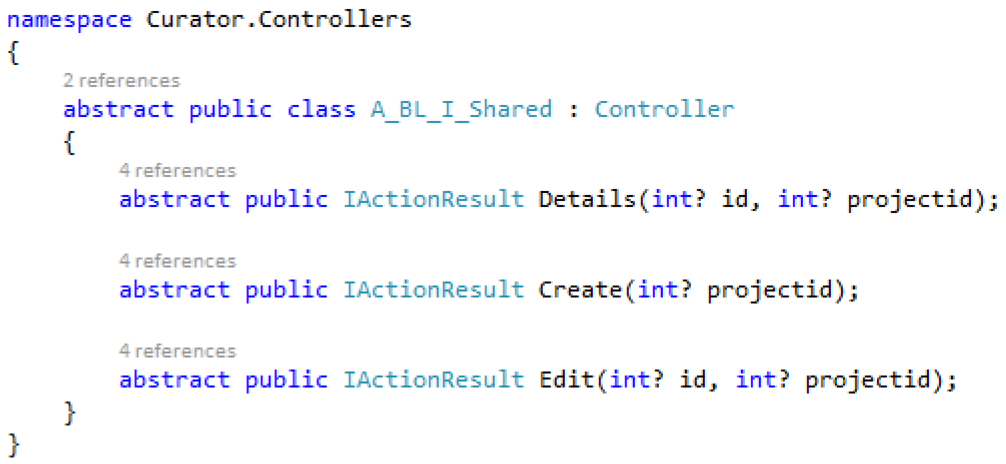
All abstract class files share the same using directive block as seen in Figure 4.



```
using System;
using Microsoft.AspNet.Mvc;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

Figure 4: Beginning Using Directive Block Used by All New Abstract Class Files

The new abstract class files are shown in Figures 5, 6, 7, and 8 in the order they were listed in the first bulleted list in section 4.1.

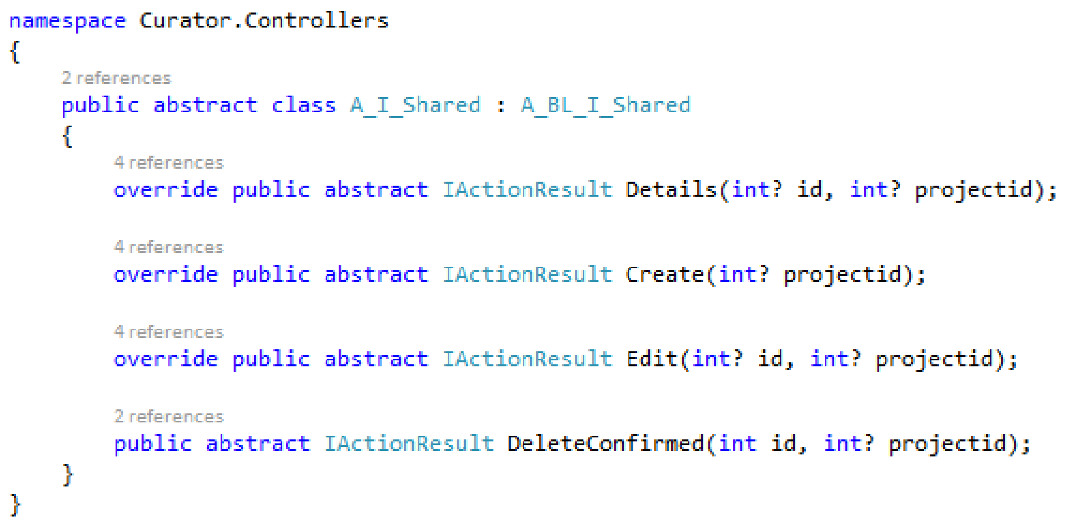


```
namespace Curator.Controllers
{
    2 references
    abstract public class A_BL_I_Shared : Controller
    {
        4 references
        abstract public IActionResult Details(int? id, int? projectid);

        4 references
        abstract public IActionResult Create(int? projectid);

        4 references
        abstract public IActionResult Edit(int? id, int? projectid);
    }
}
```

Figure 5: A_BL_I_Shared.cs Abstract Class



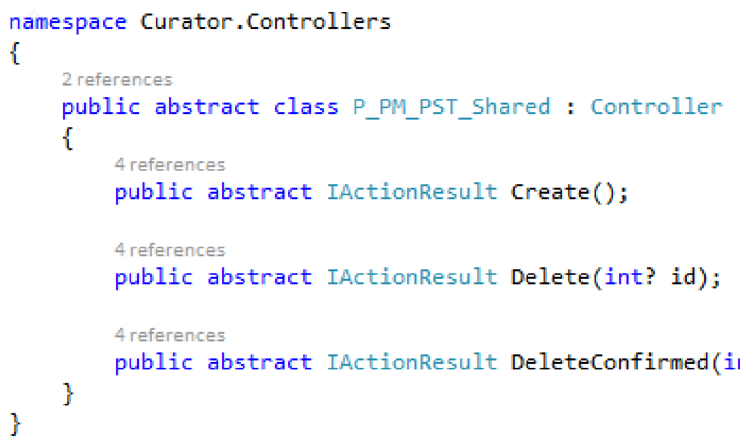
```
namespace Curator.Controllers
{
    2 references
    public abstract class A_I_Shared : A_BL_I_Shared
    {
        4 references
        override public abstract IActionResult Details(int? id, int? projectid);

        4 references
        override public abstract IActionResult Create(int? projectid);

        4 references
        override public abstract IActionResult Edit(int? id, int? projectid);

        2 references
        public abstract IActionResult DeleteConfirmed(int id, int? projectid);
    }
}
```

Figure 6: A_I_Shared.cs Abstract Class

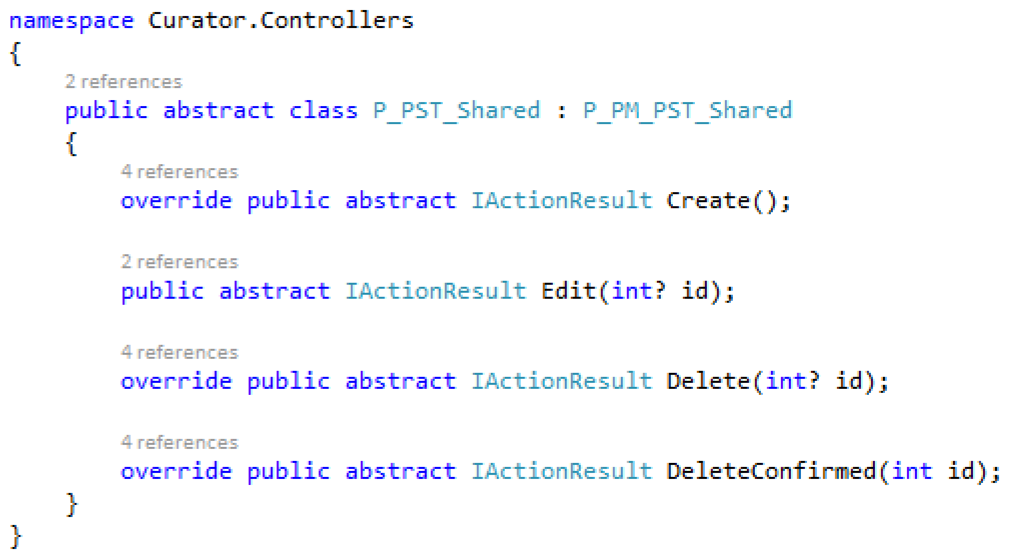


```
namespace Curator.Controllers
{
    2 references
    public abstract class P_PM_PST_Shared : Controller
    {
        4 references
        public abstract IActionResult Create();

        4 references
        public abstract IActionResult Delete(int? id);

        4 references
        public abstract IActionResult DeleteConfirmed(int id);
    }
}
```

Figure 7: P_PM_PST_Shared.cs Abstract Class



```
namespace Curator.Controllers
{
    2 references
    public abstract class P_PST_Shared : P_PM_PST_Shared
    {
        4 references
        override public abstract IActionResult Create();

        2 references
        public abstract IActionResult Edit(int? id);

        4 references
        override public abstract IActionResult Delete(int? id);

        4 references
        override public abstract IActionResult DeleteConfirmed(int id);
    }
}
```

Figure 8: P_PST_Shared.cs Abstract Class

All the modified controller class files that existed prior to these improvements only change in that they are told to inherit new parent files as specified in Figure 3 and have an *override* keyword adjacent to the methods being overridden.

4.2 Logical Intelligence and Logical Controller Abstraction

At first glance, tis logical controller abstraction seems novel. However, its significance is revealed through the idea of logical intelligence.

Logical/mathematical intelligence refers to an individual's ability to work with data, think logically, solve problems contemplatively, and see patterns as shown in [6]. People possessing this kind of intelligence gravitate towards activities like collections, math, graphs/charts, system analysis, working with numbers, brain teasers, ciphers, and creating computer programs; indicating that those working in computer science have strong logical intelligence as referenced in [7].

So, why create abstract classes for the already complete controller classes to inherit? It is all in the name: *logical* controller abstraction. By creating the four abstract classes above, we are establishing patterns between each class by the method signatures they share. These patterns can then be quickly recognized by logically intelligent developers and through this recognition can more quickly learn how Curator works at the controller/view model level. Through this improved learning process, developers could then more easily modify Curator's functionality, reuse Curator's code, and make Curator more flexible.

5 Conclusion

Curator is a fantastic system for helping professors and staff manage laboratory research projects with its structure already exhibiting promise in modifiability, reusability, and flexibility. The addition of logical controller abstraction further improves on those listed traits by making the system easier for logically intelligent developers to see patterns within Curator's controller methods and thus easier to modify and/or develop.

Furthermore, this logical controller abstraction case study shows promise for incorporating logical learning principles in software design to help simplify software modification for developers. By designing software that is easy to learn for logical thinkers, there can be benefits such as faster development times for inexperienced developers (crucial in a world with demanding development schedules, evolving client needs, and rising cyber security risks) and faster initiation-to-contribution timing for new developers (that is, allowing for new coders to contribute to their assigned teams much faster).

References

[1] C. Bodoh, *et al.*, "Software Requirements Specification (SRS) of Curator." Unpublished manuscript, South Dakota State University, Brookings, SD. Nov. 29, 2015.

[2] J. Harbison. *What is CSHTML*. http://johnharbison.net/what-is-cshtml/ . Last accessed: March 8th, 2017

[3] D. Roth, "ASP.NET 5: Introducing the ASP.NET," *Special Connect(),* vol. 29, no. 12A, 2014. https://msdn.microsoft.com/en-us/magazine/dn879354.aspx. . Last accessed: March 8th, 2017

[4] Microsoft Phone Developer Network. *The MVVM Pattern*. https://msdn.microsoft.com/en-us/library/hh848246.aspx. Last accessed: March 8th, 2017

[5] C. Bodoh, *et al.*, "Design Document of Curator." Unpublished manuscript, South Dakota State University, Brookings, SD. Dec. 6, 2015.

[6] E. Giles, S. Pitre, and S. Womack. *Multiple Intelligences and Learning Styles*. Department of Educational Psychology and Instructional Technology, University of Georgia, Athens, GA. http://epltt.coe.uga.edu/index.php?title=Multiple_Intelligences_and_Learning_Style. Last accessed: March 8th, 2017

[7] L. Miller. *Characteristics and Strategies for Different Learning Styles (Intelligences)*. California State University, Sacramento, Sacramento, CA. http://www.csus.edu/indiv/p/pfeiferj/edte305/learningstyle.html. Last accessed: March 8th, 2017