Degree project

A Journey Through the Land of Model-View-* Design Patterns

Author: Artem Syromiatnikov
Supervisor: Danny Weyns
External Supervisor: Fredrik Björn, Danfoss

Date: 2014-08-16

Course code: 5DV00E
Level: Master

Department of Computer Science

Abstract

Every software program that interacts with a user requires a user interface. Model-View-Controller (MVC) is a common design pattern to integrate a user interface with the application domain logic. MVC separates the representation of the application domain (Model) from the display of the application's state (View) and user interaction control (Controller). However, studying the literature reveals that a variety of other related patterns exists, which we denote with Model-View-* (MV*) design patterns. This thesis discusses existing MV* patterns classified in three main families: Model-View-Controller (MVC), Model-View-View Model (MVVM), and Model-View-Presenter (MVP). We take a practitioners' point of view and emphasize the essentials of each family as well as the differences. The study shows that the selection of patterns should take into account the use cases and quality requirements at hand, and chosen technology. We illustrate the selection of a pattern with an example of our practice. The study results aim to bring more clarity in the variety of MV* design patterns and help practitioners to make better grounded decisions when selecting patterns.

Keywords

patterns, model-view-controller, presenter, viewmodel, mvc, mvp, mvvm, separation of concerns, flow synchronization, observer synchronization

Acknowledgements

I would like to express my gratitude to all those people who supported me during the work on this Thesis.

My supervisor *Danny Weyns*, for tireless guiding, fair criticism and committing tons of energy to fight any sign of my laziness and ensuring everything's done right.

Professor *Jonas Lundberg* for sheer optimism, readiness to help, and for truly friendly atmosphere he brings wherever he comes.

Fredrik Björn, for flawless supervising my work with Danfoss and lots of interesting discussions.

Danfoss team—*Claes-Göran*, *Torbjörn* and *Mattias*—for great collaboration and fun teamwork experience.

WICSA 2014 organizers and guests, for a wonderful experience in Sydney, constructive feedback and interesting discussion.

Contents

1 Introduction **1**
1.1 The Goal 1
1.2 Methodology 1
1.3 Scope 2
1.4 Structure 2

2 Project Plan **3**
2.1 Danfoss 3
2.2 The Task 3
2.3 Requirements 3
2.4 High-level Analysis 4
2.5 Architecture 5

3 Desktop Application **7**
3.1 User Interface 7
3.1.1 News 7
3.1.2 Updates 7
3.1.3 User Preferences 8
3.1.4 Settings 9
3.2 UI Scenarios 9
3.2.1 Scanning for Updates 9
3.2.2 Running System Update 11
3.2.3 Retrieving User Preferences 13
3.3 The Problem 14

4 The Study **16**
4.1 Introduction 16
4.2 Methodology 17
4.3 The Land of MV* Design Patterns 18
4.3.1 Families of MV* Design Patterns 18
4.3.2 Flow Versus Observer Synchronization 19
4.4 Patterns 21
4.4.1 Widget-based User Interfaces (or Forms and Controls) 21
4.4.2 MVC: Model-View-Controller 22
4.4.3 MVVM: Model-View-View Model 27
4.4.4 MVP: Model-View-Presenter 31
4.5 Case Study 37
4.6 Conclusions 39

5 The Solution **41**
5.1 Pattern Selection 41
5.2 Pattern Overview 42
5.2.1 Structure 42
5.2.2 Collaboration 42
5.3 Consequences 42
5.4 How Does the Pattern Solve the Problem 43

6 Conclusions **47**

References **50**

List of Figures

2.1 High-level Architecture 6
3.1 News tab 7
3.2 Updates tab in Idle mode 8
3.3 Updates tab in Scan Results mode 8
3.4 User Preferences tab 8
3.5 Settings tab 9
4.1 The Land of MV* Design Patterns 20
4.2 Smalltalk'80 MVC pattern 23
4.3 Web MVC pattern 26
4.4 Application Model pattern 28
4.5 Microsoft MVVM pattern 30
4.6 Dolphin Smalltalk MVP pattern 32
4.7 Supervising Presenter pattern 34
4.8 Passive View pattern 35
4.9 Our realization of the MVP pattern 37
4.10 Screen shot of the desktop application UI 38
5.1 ViewModel scheme 44
5.2 VM Mapping: Updates tab while scanning 45
5.3 VM Mapping: Updates tab in Scan Results mode 45

List of Tables

4.1 Summary of pros and cons of MV* pattern families 39
5.1 Application States 44

1 Introduction

Keeping application user interface (UI) consistent often is non-trivial task. Each user action could trigger some changes within application UI (e.g. input fields could become read-only, buttons could be disabled or enabled back, etc.). The more usage scenarios application supports, more complex UI logic becomes. At some point, the efforts for keeping UI consistent may exceed amount of efforts for feature development. The code becomes convoluted and hard to maintain.

UI architectures provide a way to fix the issue. They give a structured, flexible way for building rich and consistent UI. However, a variety of UI architectures exists, each one with its own benefits and drawbacks. It this study we analyze and classify a set of patterns to give a guidance for selecting correct pattern for any particular needs.

1.1 The Goal

The goal of the study is to analyze existing patterns and practices for building applications with graphical user interface (GUI), MV* patterns in particular. We assess several families of such patterns, namely Model-View-Controller (MVC), Model-View-ViewModel (MVVM) and Model-View-Presenter (MVP), analyze key features and assess differences of patterns in these families. We use this knowledge to get an insight on UI architectures classification, indicate sphere of use, advantages and drawbacks of each pattern family. We aim to bring more clarity to the world of UI architectures and give some recommendations for UI pattern selection.

1.2 Methodology

In order to conduct the study, we collected a representative set of UI patterns and analyzed them in detail. Our representative set includes at least two patterns from each of major families: MVC, MVVM, MVP. In order to understand pattern evolution, we included some early patterns as well as modern ones. We studied influential materials that introduced the original patterns (including Smalltalk'80 MVC, VisualWorks Application Model, Taligent Programming model, Dolphin Smalltalk). Then, as our study was inspired by practical problem, we assessed how those pattern were implemented in a number of modern frameworks (Microsoft ASP.NET MVC, Spring, Grails, Microsoft WPF, Silverlight, Windows Forms).

We have written sample applications to get better understanding of patterns of each family. Based on this, we created component diagrams for each pattern, and used them for in-depth analysis and patterns comparison. This allowed us to identify common feats of patterns within each family and their optional details, identify their place in classification and provide some conclusions and recommendations regarding their area of use.

1.3 Scope

The study was initiated by practical problem, so we anticipate that it does not follow a systematic approach, rather a pragmatic one. Since first UI patterns were introduced in 1980's, hundreds of variations were implemented up to modern days. We focus on rather limited set of patterns from three most popular families to keep the scope manageable. The set of patterns we have studied was sufficient to solve the initial problem as well as to reach our goals declared above, namely: to provide a better picture of variety of UI architectures, give an insight of their capabilities and area of use, and give an advice how to choose a UI pattern for particular needs.

1.4 Structure

This report will first describe the initial task that led to the study of MV* architectures. It was the research and development (R&D) project with Danfoss company, where we had to design and implement software system for automatic downloading and installation of software updates for Danfoss customers. The report will cover high-level system structure and then focus on the desktop application and the problems we met in UI integration.

The next part of the report provides the study itself. In fact, it is copy of a paper 'A Journey Through the Land of Model-View-* Design Patterns' [22] we presented on WICSA 2014 conference. It takes a step away of the initial problem and shifts focus on the UI architectures research itself. This chapter is a core contribution of this thesis work.

In the last section, we move back to the problem. We discuss how did the study help us to organize the UI architecture for our application, and how did we benefit from UI pattern introduction. We summarize our knowledge and give recommendations regarding picking up the right UI pattern.

2 Project Plan

In this section we provide an overview of the initial task.

2.1 Danfoss

Danfoss is a global manufacturer of hydraulics and electronics for off-highway vehicles. The company has offices in Sweden, Denmark, Germany, USA, China and other countries. It produces products for such industries as agriculture, construction, road building, material handling, municipal, forestry and others. The product line includes series of motors, valves, microcontrollers, displays and many more.

2.2 The Task

Danfoss supports its hardware products with all kinds of software. It includes drivers, maintenance utilities, documentation and even full-featured development environment. These resources are periodically updated with new features, security and performance improvements.

All software is published on Danfoss website, and customers have to find, download and install it manually. When updates are released, customers are informed by e-mail. They can follow the link, download and install an update—once again, manually. Performing plenty of manual operations takes significant time and affects customer satisfaction.

Our task is to develop a solution that helps users manage installed Danfoss software, notify users about new versions available and install updates with minimal user interaction.

Apart from that, solution should provide Danfoss content managers way to manage published content: upload new files, retract undesirable content, and publish some content-related news and security messages.

2.3 Requirements

As a Danfoss engineer, I would like to:

- Have a clear overview of all the content we made available for download, including all published versions of each item.
- Manage published content:
  - Manage the content catalogue.
  - Publish new piece of content.

    - Publish new version of a content.
    - Remove particular content version from repository.
    - Remove a piece of content from repository.

- Manage news: publish, edit, delete information messages.

As a Danfoss customer, I would like to have a tool able to:

- Perform automatic scanning and detect Danfoss resources currently installed on my computer.
- Let me see Danfoss resource catalogue and specify which content I am interested in.
- Detect if there are any Danfoss resources available for installation/updating.
- Perform automated downloading and installation of the resources I have selected.
- Run scanning for updates periodically and show notification if any updates are found.
- Display news feed.

2.4 High-level Analysis

Based on the Requirements, we have come to following set of decisions.

The system should be structured as client-server system. The server software is responsible for maintaining the content repository, while client software should provide system users way to access the repository. Also, we have a clear separation of Danfoss Engineer and Danfoss Customer roles—the first one is content provider, while the second one is consumer. They work with repository in different ways and require different tools for their needs.

Content providers need a way to overview and manage the repository, publish new content and manage news. Neither of these operations involve cooperation with user's local machine, which makes Web application a good options for these needs.

The content consumers, on the contrary, require software capable of interacting with client operating system. It should be capable of scanning Windows registry and file system in order to detect installed software. It should also provide background task to run the scan periodically. This requirements make the desktop application an appropriate choice.

We use .NET Framework technology stack for development. .NET provides rich capabilities for building all system components we need, and fits us perfectly as Windows is our only target platform. Furthermore, Danfoss already uses .NET framework infrastructure in some products; this fact allows to reduce technology integration and maintenance cost.

Technology stack:

- **C#** — Primary development language;
- **Microsoft SQL Server** — the database;
- **ASP.NET MVC** — for web application;
- **WCF** — for web service (API for desktop application);
- **Windows Forms** — for desktop application.

2.5 Architecture

The high-level architecture overview is presented on the Figure 2.1. Short description of each component is provided below.

- **Server components.** The necessary software components running on the server-side.
  - **Database.** Relational DB for storing system data (excluding files).
  - **File storage.** Storage for the distributable files: software installers, update packages, documentation and other resources.
  - **Business logic.** The component encapsulates all business logic, such as content and version management, update detection, etc. Operates with Database and Files storage.
  - **Web interface.** The Web application for content providers. Provides UI for content publishing and management.
  - **WCF service.** The Web API for desktop client. Provides methods for update detection, file downloading, etc.
- Client components. These software components run on user's machine and give user a way to interact with server.
  - **Guide Update Center.** The desktop application for content consumers. Supports scanning for updates, update downloading and installation, etc. Interacts with WCF service on the server-side.

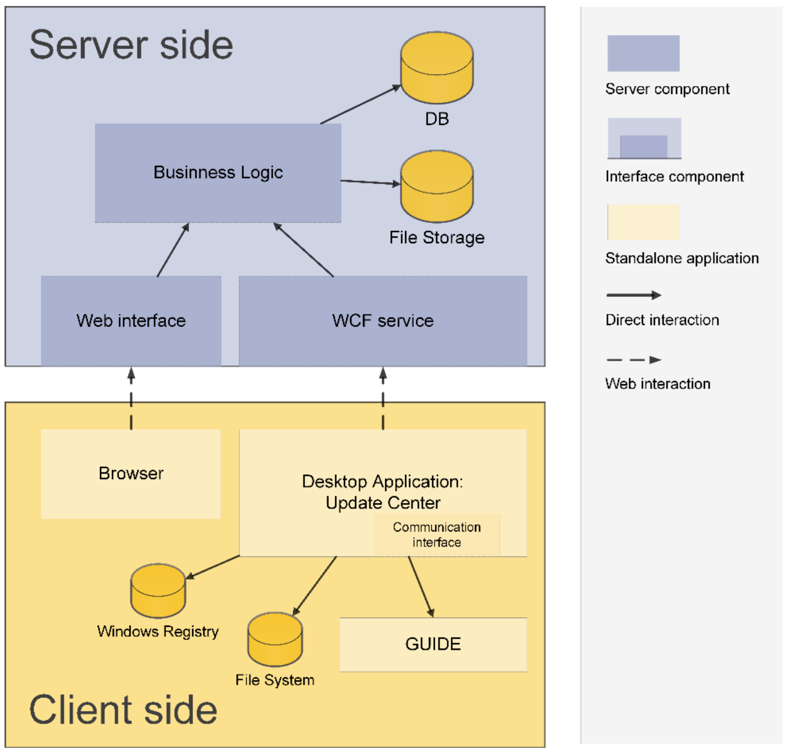


Figure 2.1: High-level Architecture

- **Windows registry.** Contains Windows OS configuration and data, including list of installed applications.
- **File system.** Guide Update Center interacts with file system to store the content downloaded from the server.
- **GUIDE.** Danfoss integrated development environment. Certain packages are installed within GUIDE as plug-ins. We interact with GUIDE to detect installed plug-ins and their versions.
- **Browser.** Content publishers need web browser for interaction with the Web application.

Further we focus on the Guide Update Center—the desktop application, where we faced problems while building the graphical user interface.

3 Desktop Application

In this section we provide an overview of the desktop application UI and functionality, perform detailed analysis of UI synchronization routine and identify the problem we faced.

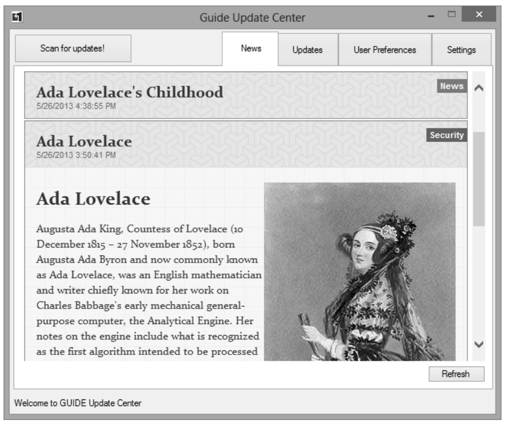
3.1 User Interface

In this section we overview the user interface of Guide Update Center.

The application window has 4 tabs with views for main software features, 'Scan for updates' button and status bar. *News* tab presents the latest notifications published by Danfoss. *Updates* tab provides interface for scanning for updates, their downloading and installation. *User Preferences* tab allows user to specify which content he is interested in. *Settings* tab allows to change application settings.

Status bar indicates currently running activity and short information message about application status.

3.1.1 News



Ada Lovelace's Childhood
5/26/2013 4:38:55 PM

Ada Lovelace
5/26/2013 3:50:41 PM

Ada Lovelace

Augusta Ada King, Countess of Lovelace (10 December 1815 – 27 November 1852), born Augusta Ada Byron and now commonly known as Ada Lovelace, was an English mathematician and writer chiefly known for her work on Charles Babbage's early mechanical general-purpose computer, the Analytical Engine. Her notes on the engine include what is recognized as the first algorithm intended to be processed

Welcome to GUIDE Update Center

Figure 3.1: News tab

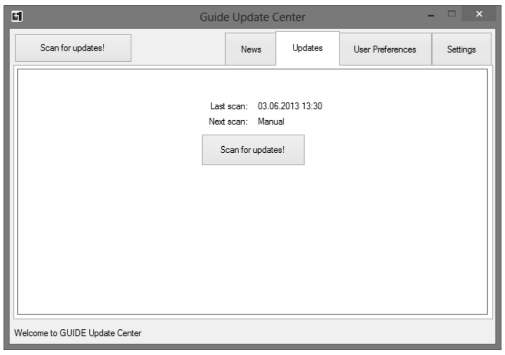
The News tab (Figure 3.1) displays the latest news published by Danfoss. The news category (e.g. Information, Advertisement, Security message) is marked on the right hand side. The news list can be reloaded by clicking the Refresh button.

3.1.2 Updates

The Updates tab (Figures 3.2, 3.3) is a core view of the application. It displays last scan time and planned time for the next scan (if scheduled). User can run scanning for updates manually at any moment.

When the scan is triggered, progress bar is shown here and status bar is updated to reflect running activity. If no updates are found, the view returns to the original state (Idle mode). Otherwise the table with results is shown (Scan Results mode).

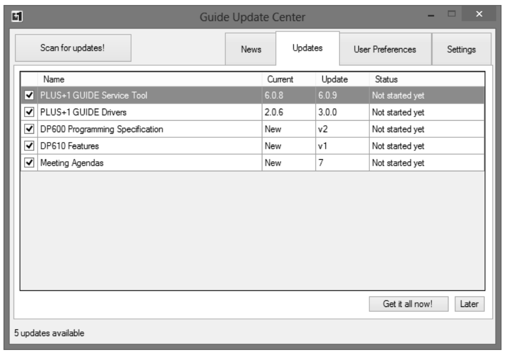
The scan results are shown as a table, where the package name, current and newly found version names are displayed. It allows to select the packages user wants to update. All the packages are selected by default.



Last scan: 03.06.2013 13:30
Next scan: Manual

Welcome to GUIDE Update Center

Figure 3.2: Updates tab in Idle mode



	Name	Current	Update	Status
✓	PLUS+1 GUIDE Service Tool	6.0.8	6.0.9	Not started yet
✓	PLUS+1 GUIDE Drivers	2.0.6	3.0.0	Not started yet
✓	DP600 Programming Specification	New	v2	Not started yet
✓	DP610 Features	New	v1	Not started yet
✓	Meeting Agendas	New	7	Not started yet

5 updates available

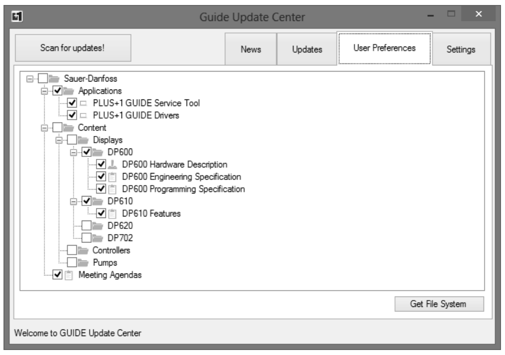
Figure 3.3: Updates tab in Scan Results mode

The update process is started by pressing 'Get it all now!' button. The system will download all the packages and install them. The installers are put to the default downloading location that could be set on the Settings tab. Downloading and installation progress is dynamically displayed in the last column of the table; short status message is also displayed in the status bar. 'Later' button allows to postpone updating and return the view to original state.

When updating is finished, the view is reset, and Windows notification is shown in system tray.

3.1.3 User Preferences

User Preferences tab (Figure 3.4) allows user to select the packages he is interested in. For example, user may need drivers for displays, but not for joysticks. The application retrieves current content catalog from server on the first launch; it could be refreshed manually in any moment. Scan for updates will be performed only for the packages the user selected, it would ignore others.



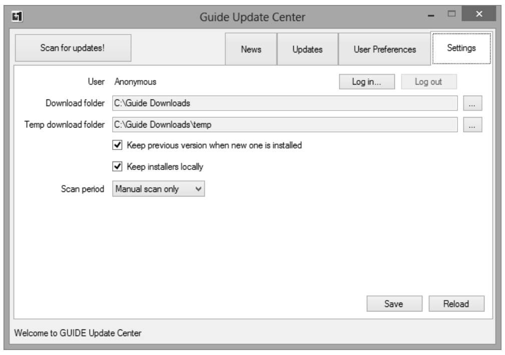
- ☐ Sauer-Danfoss
  - ✓ Applications
    - ✓ PLUS+1 GUIDE Service Tool
    - ✓ PLUS+1 GUIDE Drivers
  - ☐ Content
    - ☐ Displays
      - ✓ DP600
        - ✓ DP600 Hardware Description
        - ✓ DP600 Engineering Specification
        - ✓ DP600 Programming Specification
      - ✓ DP610
        - ✓ DP610 Features
      - ☐ DP620
      - ☐ DP702
    - ☐ Controllers
    - ☐ Pumps
  - ✓ Meeting Agendas

Welcome to GUIDE Update Center

Figure 3.4: User Preferences tab

User could instantly navigate to location of any downloaded package. The right mouse button click on any node of a File System tree opens a context menu that allows to open target location in Windows Explorer or to copy its path.

3.1.4 Settings



User Anonymous
Download folder C:\Guide Downloads
Temp download folder C:\Guide Downloads\temp
☑ Keep previous version when new one is installed
☑ Keep installers locally
Scan period Manual scan only
Welcome to GUIDE Update Center

Figure 3.5: Settings tab

The Settings tab (Figure 3.5) contains configuration options. User can set location for downloaded files and for pending downloads. User can decide whether Guide Update Center would uninstall older versions of software before installing new ones. He can also specify if he wants to keep all the installers in his file system.

The periodical scanning interval could be set or disabled completely. In this case, scanning for updates could be run only by manual clicking the 'Scan for updates!' button.

3.2 UI Scenarios

In this section we discuss main scenarios of Guide Update Center usage. We analyze how they interfere with each other and which problems does it create.

The general requirements to the User Interface are following:

- The user interface must remain responsive at any point of time.
- The user interface must disallow user to perform actions that are not possible in any given situation.
- The Windows task bar notifications should notify the user when new updates are available.
- Simultaneous server calls are not allowed.
- The user interface must dynamically display the updating progress, including listing of updating items, their status (waiting, downloading, installing), and download progress in bytes.

3.2.1 Scanning for Updates

Scanning for updates is one of the main routines of the Guide Update Center. It could be triggered by clicking Scan for Updates buttons on the top left or on the Updates tab. It could be called automatically by the scheduler as well.

Scanning for updates is not an instant operation, it includes interaction with server, so it takes noticeable amount of time. UI should display scanning progress and remain responsive at any moment of time. While scanning is running, we should deny user running it one more time; we should also turn some controls (that could potentially affect scanning results) to read-only mode. So the general plan is: update the UI, perform scanning, and update UI once again.

Before scan begins:

- General
  - Disable ‘Scan for Updates!’ button
  - Update message in Status bar
  - Display progress indicator in Status bar
- User Preferences tab
  - Turn content catalog tree to read-only mode (so user is unable to select new packages while scan is running)
  - Disable ‘Get File System’ button
- Updates tab
  - Hide ‘Scan for Updates!’ button
  - Display progress indicator at the center of panel

After scan is finished:

- General
  - Enable ‘Scan for Updates!’ button
  - Update message in Status bar
  - Hide progress indicator in Status bar
- User Preferences tab
  - Make tree view interactive
  - Enable ‘Get File System’ button
- Updates tab
  - Show ‘Scan for Updates!’ button

    - Hide progress indicator
    - Update ‘Last scan’ time label
    - If any updates were found:
        - Turn the Updates tab from Idle to Show Scan Results mode (Figure 3.3)
        - Update status bar message with ‘N updates available’
        - Display Windows task bar notification

3.2.2 Running System Update

Scan results view presents table with all found updates, ‘Get it all now!’ and ‘Later’ buttons. User can select which updates he is interested in. ‘Get it all now!’ button is available if at least one item is selected. ‘Later’ button skips update installation and switches Update tab back to Idle mode (Figure 3.2).

Updating is not an instant operation as well. It includes downloading multiple installer files and running installation process for each of them. UI should dynamically display the status for each of the packages: Not started, Downloading, Installation, Done, etc. While updating is in progress, we should forbid running Scan for updates and retrieving content catalog tree for User Preferences tab.

Before update begins:

- General
    - Disable ‘Scan for Updates!’ button
    - Update message in Status bar to ‘Updating GUIDE system’
    - Display progress indicator in Status bar
- User Preferences tab
    - Turn content catalog tree to read-only mode
    - Disable ‘Get File System’ button
- Updates tab
    - Turn table with selected updates to read-only mode
    - Disable ‘Get it all now!’ button
    - Disable ‘Later’ button

While update is running:

- For each selected package display its status:
  - Not started—processing of this package has not started yet;
  - Downloading... [progress]—file downloading is in progress;
  - Downloaded—new version installer is downloaded but is not installed yet;
  - Installation—installation is in progress;
  - Done—update installation is successfully complete;
  - Error—error appeared on any step of package updating.

When updating is finished, the results are shown to user. They appear in the same table where user selected packages to update. It is presented in read-only mode; each affected package should have 'Done' or 'Error' status message.

- General
  - Enable 'Scan for Updates!' button
  - Update message in Status bar to 'Update complete!'
  - Hide progress indicator in Status bar
  - Display Windows task bar notification 'Your GUIDE is up-to-date now!'
- User Preferences tab
  - Make tree view interactive
  - Enable 'Get File System' button
- Updates tab
  - Keep table with selected updates in read-only mode
  - Keep 'Get it all now!' button disabled
  - Rename 'Later' button to 'Done' and enable it

3.2.3 Retrieving User Preferences

Third important routine is retrieving content catalog from the server. It is a hierarchical representation of all content published by Danfoss. It is catalogued and looks similar to your OS file system. It is displayed on the User Preferences tab and allows user to select which content he is interested in (Figure 3.4).

When Danfoss releases new device (say, new display model), it would publish drivers and documentation for it. This new pieces of content will appear in the User Preferences tree, so user could select it and download.

File system is retrieved automatically on application startup. However, user can retrieve the latest version at any moment of time by clicking 'Get File System' button on the User Preferences tab.

Before file system retrieving begins:

- General
  - Disable 'Scan for Updates!' button
  - Update message in Status bar to 'Retrieving server file system…'
  - Show progress indicator in Status bar
- User Preferences tab
  - Turn content catalog tree to read-only mode
  - Disable 'Get File System' button
- Updates tab
  - Idle mode:
    - Disable 'Scan for Updates!' button
  - Scan Results mode:
    - Turn table with selected updates to read-only mode
    - Disable 'Get it all now!' button
    - Disable 'Later' button

After file system is retrieved:

- General
  - Enable 'Scan for Updates!' button
  - Update message in Status bar

  - Hide progress indicator in Status bar

- User Preferences tab
  - Render new file tree in tree view, restore selected items
  - Make tree view interactive
  - Enable 'Get File System' button

- Updates tab
  - Idle mode:
    * Enable 'Scan for Updates!' button
  - Scan Results mode:
    * Make table with selected updates interactive
    * Enable 'Get it all now!' button
    * Enable 'Later' button

3.3 The Problem

We have just observed three main operations performed by Guide Update Center and noticed that a set of UI-related operations should be performed before and after each business operation for maintaining consistent UI. Every situation includes a number of updates through application UI, and developer should pay attention to each of them to ensure all usage paths are covered.

Though we tried to mention as much details as possible while discussing three scenarios above, we still missed at least one use case. It happens when scan for updates is over and the table with available updates is shown to the customer, but user decides to start scanning for updates again. In this case Updates tab should be first reset and switched to Idle mode, and only then proceed with described routine.

Introduction of a new feature would require us to review all existing UI logic and update it accordingly.

For example, at some point we have got a new feature request: allow users to perform login. This would provide them updated User Preference tree with some content that is not available to unauthorized users. The authorization is optional for user, so corresponding buttons were placed on the Settings tab. Let us see what consequence it brought.

- For each scenario when some server interaction is involved, we should disable Log In and Log Out buttons. We should enable them back when the action is done.

- When user performs log in or log out operations, we should disable buttons that trigger Scan for updates, update installation and virtual file system retrieving (and enable them back again when operation is done)
- After user logs in/out, the virtual file system should be reloaded, as it depends on user authorization.
- If user logs out when Updates tab is in Scan Results mode, it should be reset back to Idle: Scan results could potentially include some packages that become unavailable for user when he logs out.

The problem is: with each new implemented feature the complexity of UI scenarios grows exponentially. In a short time we found that keeping the UI consistent takes more efforts than the feature development itself. We realized that the application architecture must be redesigned in order to use structured, logical approach for maintaining UI consistent and responsive. That led us to study of the user interface patterns.

4 The Study

The following chapter is a direct copy of a paper we presented on WICSA 2014 conference [22] and therefore it does not follow the writing style of the rest of report.

4.1 Introduction

Every software program that requires at least a bit of interactivity with users requires a user interface. This makes the integration of the user interface with the application domain a recurring engineering problem. In this paper, we focus on design patterns to solve this integration problem. Design patterns provide generic solution schemes for recurring design problems, offering reference materials that give engineers access to the field's systematic knowledge [3].

Model-View-Controller (MVC) is a widely known design pattern to integrate a user interface with the application domain. MVC was first introduced in Smalltalk'80 by Krasner and Pope [14]. Central to MVC is the separation of the representation of the application domain (the model) from the display of the application's state (the view) and the user interaction processing (the controller). However, studying the literature reveals that a variety of other related patterns exists, which we denote with Model-View-* (MV*) design patterns.

Since the late 1980s when MVC was documented, numerous new MV* design patterns emerged that aimed to eliminate the drawbacks of their predecessors. These patterns include for example Model-View-Presenter [17] and Model-View-View Model (or Presentation Model) [8]. With the evolution of programming languages and software technology, the MV* design patterns were changing as well, and different families of patterns emerged. As a result, the designer today is confronted with a variety of patterns with subtle but important differences. To effectively select a concrete pattern, the designer has to study several patterns and versions in order to understand their purposes, areas of usage, strengths and weaknesses, and their applicability to the problem at hand. The selection of design patterns is known to be important for understand-ability and maintainability of the program code [21, 10]. Using a wrong pattern (or no pattern at all) for integrating the user interface with the application domain may lead to complex code, where a minor change leads to significant overhead to maintain consistency between the user interface and the application domain.

We faced the complexity of integrating a user interface with the application domain in a recent R&D project between Danfoss AB and the Computer

Science Department of Linnaeus University. In this project, we studied how to notify users dynamically when software updates become available, and how to support automatic downloading and installation of the updates. In the initial phase of the project we used a standard Windows Forms approach for user interaction. However, during the course of the project, a variety of new user interface requirements emerged that compelled us to look for a systematic approach to guarantee consistency between the user interface and the underlying domain state, while keeping the solution understandable and manageable. To that end, we studied different existing patterns for integrating the user interface with the application domain. From this study, we learned that the patterns differ in subtle ways, which makes the selection of a pattern a difficult task.

A number of authors have compared patterns for integrating a user interface with the application domain state, see e.g., [8, 13]. While these studies offer good overviews of the general concepts of MV* patterns as well as specific properties of different pattern families, they lack a deeper analysis of the differences between pattern families and between patterns within families.

This paper contributes with an overview of existing families of MV* design patterns with a particular emphasis on versions of patterns in the three main families: MVC, MVVM and MVP. Our study takes a practitioners' point of view putting the patterns in an evolution perspective driven by development of programming languages and technology. We discuss the essentials of the patterns, their tradeoffs and particular differences of usage. The goal of the paper is to bring more clarity to the area of MV* patterns and help practitioners to make better-grounded decision when selecting patterns.

The remainder of the paper is structured as follows. In section 4.2, we briefly explain the methodology we used in this study. Section 4.3 gives an overview of the pattern families. In section 4.4, the heart of the paper, we zoom in on the patterns. We discuss a selection of patterns of the main families in detail and clarify the differences between the patterns. Section 4.5 illustrates the selection of a pattern in the project with Danfoss AB. Finally, we draw conclusions in 4.6.

4.2 Methodology

The goal of our study is to identify different families of MV* design patterns and closely examine the differences between families as well as differences between patterns within the families. It is not our aim to study all MV* design patterns that have been documented, which would require a systematic survey of the literature. Instead, we have selected a representative set of patterns in each MV* family for detailed analysis and comparison.

As a baseline for our study, we have selected material that describes influential patterns, including Smalltalk'80 MVC papers [14, 2], VisualWorks Application Model documentation [11], and MVP articles of the Taligent Programming model [17] and Dolphin Smalltalk [1]. As our study emphasizes a practitioners' point of view, we also assessed the evolution the MV* design patterns and their use in current practice. To that end, we studied the use of MV* design patterns in a number modern frameworks, including Microsoft's Windows Forms [16], WPF and Silverlight for desktop development [19], Spring [20], ASP.NET [15], and Grails [18].

During the study, the material was analyzed in detail and data from the articles was extracted to describe the different patterns. We have written sample applications with patterns of every MV* family to get a better insight of their use in practice (this includes Windows Forms Widget-based UI, ASP.NET MVC and Grails frameworks, Silverlight MVVM and few versions of MVP pattern on Windows Forms technology). To compare the patterns, we derived component diagrams for each pattern, identified the role of each component, and specified the types of interactions between the components. Based on the material, we then identified the similarities among the patterns and the differences. The insights derived from this study were then used to re-engineer the integration of the user interface with the application domain in the project with Danfoss AB. Finally, we classified the different MV* design patterns and documented the results of the study in this paper.

4.3 The Land of MV* Design Patterns

We now give an overview of the different families of MV* design patterns. We also discuss *flow* versus *observer* synchronization, a key principle of each MV* design pattern.

4.3.1 Families of MV* Design Patterns

The problem of how to integrate a user interface with the application domain became particularly relevant when graphical user interfaces emerged in the 1980s. Around the time MVC was documented (1988), Coutaz introduced PAC [4], which structures an interactive application in three parts: Presentation (defines the syntax of the application), Abstraction (defines the semantics), and Control (maintains the consistency between the domain entities and the presentation to the user). Later, in the 1990s, new patterns for integrating the user interface with the application domain emerged driven by the needs of new technology (e.g., touchscreens and voice input, web and mobile applications, etc.) and the evolution of programming languages that offered new coding concepts (e.g., events, generics, lambda-expressions, etc.). This evolution forced software developers to reconsider existing practices for integrating the user interface with the application domain. Eventually, entire families of patterns emerged.

All MV* patterns are based on the idea of *separation of concerns* [5]. Separation of concerns in user interface design refers to a separation of the application domain model and the user interface in two layers, where the domain model is unaware of the user interface. Following this principle leads to clear design that is easy to maintain; it also allows to effectively distribute work on different layers between developers, simplifies testing, etc.

Figure 4.1 gives an overview of the land of the design patterns. Patterns in the area at the top left, Widget-based User Interfaces, do not separate the domain logic from the presentation logic. Patterns in all other areas realize separation of concerns in one or another way. Our main focus in this paper is on the latter types of patterns, denoted as MV* design patterns.

At a coarse grained level, the MV* design patterns can be divided in a three families: MVC [14], MVP [17] and MVVM [8]. However, as argued by Karagkasidis [12], it is incorrect to state that there are only three general patterns for handling the synchronization between the application domain and the user interface. Although the patterns in each family share a general principle, there are different ways to concretely realize the patterns. In our study, we identified representative variants of each family, as shown in Figure 4.1.

In particular, the MV* design patterns differ in the way they are structured and handle the synchronization between the user interface and the application domain state. In general, MV* design patterns typically comprise three components: M, V, and a third variable component denoted with *. M refers to Model, which represents the application domain; V refers to View, which represents the presentation to the user. The third component binds M with V, that is, this component defines how the M and V components communicate with each other and with the user. In the following section, we explain how patterns differ in the way they allocate responsibilities to the different components and how the components interact with one another and the user.

4.3.2 Flow Versus Observer Synchronization

Before we discuss the patterns in detail, we first explain the essential difference between flow and observer synchronization [7]. Synchronization in this context refers to the mechanism to realize consistency between the application state and the user interface that represents it. Flow and observer

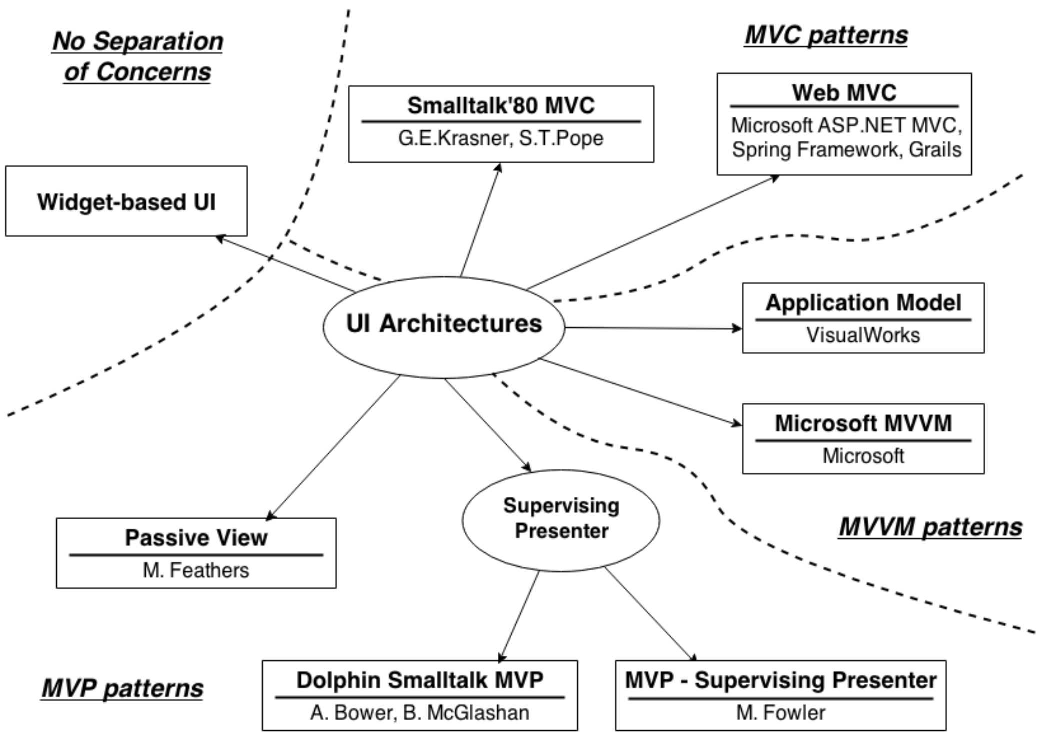


Figure 4.1: The Land of MV* Design Patterns

synchronization take a different perspective on this mechanism, and each MV* design pattern is essentially based on one of them.

Flow synchronization is based on sequential command execution: e.g., read user input from text box A, processing it with method B, and write the result to text label C. Flow synchronization uses direct calls between user interface components and domain components. For small applications with relatively simple user interfaces, flow synchronization results in clear and easily understandable code. For more complicated programs and more sophisticated user interfaces, the approach can result in code that is difficult to maintain due to the lack of separation of domain and user interface concerns.

Observer synchronization structures the domain logic and interface logic in separate layers. The domain layer must implement a notification mechanism to which components of the user interface layer can subscribe. This allows the user interface components to update the state of the user interface when relevant changes in the domain occur. Observer synchronization is particularly useful when the user interface involves multiple views on the same domain data. Furthermore, the clear separation of concerns supports the distribution of domain development tasks and user interface tasks to different developers or development teams. The main disadvantage of observer synchronization is implicitness. It might be difficult to oversee the potential impact of changes in domain components as potentially any observer may be affected by the changes. Performance could also be an issue if the number of observers increases. To that end, more fine-grained notification logic may be used, however, this comes with increased complexity. Finally, lifecycle management requires attention. In case views are no longer interested in particular notifications, they must unsubscribe from the domain events, otherwise they could become ghost objects and may cause memory leaks.

4.4 Patterns

To document the MV* design patterns, we use a simplified version of The Gang of Four's template [10]. In particular, we use the sections intent, motivation, structure, collaboration, and consequences. Other information relevant to this study, such as usage of the patterns, applicability, and known uses are integrated in the pattern descriptions. We now discuss representative patterns of the different families in detail.

4.4.1 Widget-based User Interfaces (or Forms and Controls)

Widget-based User Interfaces (also called Forms and Controls) are a common way of building applications with graphical elements as provided by many integrated development environments. The idea is straightforward: the developer arranges a set of predefined user interface widgets on a form (window), and writes code that handles all the logic in the Form class. Data displaying, user input handling, calculations, domain model handling, etc. are performed in the same class. The developer has unlimited access to both user interface widgets and domain data. Domain and user interface logic are mixed, there is no separation between the domain and user interface concerns, which may hamper maintainability. Still, this approach has several benefits, which makes it a popular way of building user interfaces in many modern tools:

- Simplicity: widget-based user interfaces are very simple to understand: the widgets on the form become fields of a Form class, so the developer can access the widgets just as any other field.

- Consistency: the approach employs flow synchronization. Synchronization is handled explicitly: target views/widgets are modified by direct calls, so the code is easy to understand.

- Efficiency: creating a sophisticated multi-tier architecture for applications that do not require rich user interfaces would be overkill. Keeping the design simple in such cases reduces development time and increases maintainability.

Widget-based UI is a perfect choice for small applications or applications with simple user interfaces.

4.4.2 MVC: Model-View-Controller

Model-View-Controller (MVC) is the most influential family of design patterns for synchronizing a user interface with the state of the application domain. The approach was introduced in the 1980s, even before widget-based user interfaces were used [14, 10]. Initially, MVC was used for designing and building desktop applications with rich graphical user interfaces. Over time, the original MVC pattern evolved and variants emerged driven by technological evolutions and new needs. Nowadays, MVC is used for integrating interface logic with domain logic in development of various domains, such as Web applications and mobile systems.

We discuss two representative patterns of the MVC family. These patterns provide a good basis for expressing the core idea of the pattern family: separating responsibilities related to domain state, displaying domain state, and handling user interaction. The selected patterns illustrate how this core idea can be realized in different ways. To get a deeper understanding of a pattern family, it is important to study a set of concrete realizations.

Smalltalk'80 MVC

Intent Separate the concerns of the application domain and its representation in three modules, each handling a specific task: store and manage data, display data, and handle user input.

Motivation Support the design and development of highly maintainable applications with rich user interfaces by maintaining a strict separation between domain logic and presentation. The application logic manages the domain data, and the presentation reflects the data. Separating the domain from the user interface concerns makes it easier for the designer to understand and modify each particular unit, without having to know everything about the other units [14].

Structure The three key components of the Smalltalk'80 MVC pattern are Model, View and Controller. The Model component is responsible for the domain data and logic. This component has no reference to the other components of the triad. As such, the application logic does not depend on the presentation of domain data. The View component is responsible for displaying model data. The last one, Controller is responsible for handling user input.

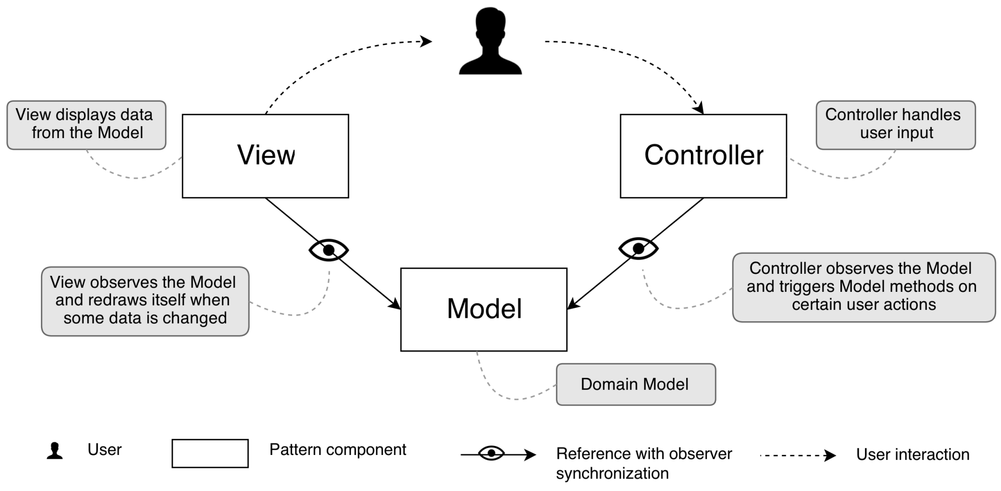

View displays data from the Model
View
Controller
Controller handles user input
View observes the Model and redraws itself when some data is changed
Model
Controller observes the Model and triggers Model methods on certain user actions
Domain Model
User
Pattern component
Reference with observer synchronization
User interaction

Figure 4.2: Smalltalk'80 MVC pattern

View and Controller work as a pair allowing the user to interact with the user interface. For example, the user interface may provide a text box allowing the user to enter a user name. The View is responsible for rendering the text box. The user can change the text and press keys (e.g., the Enter key) — such events are handled by the Controller. The Model maintains the domain data. Often, the application has one Model and a set of View-Controller pairs working with it. Although View and Controller work in pairs, they are considered as two separate entities with minimal coupling. That is, displaying data and handling user input are treated as distinct activities enhancing separation of concerns.

Collaborations The cooperation between Model, View and Controller relies on observer synchronization. As mentioned above, Model is not aware of the other components. However, the View and Controller maintain direct links to the Model in order to observe, read and modify it.

The Model provides a notification mechanism to which the other components can subscribe. Every change of data generates an event to notify the subscribers. The View has a passive role in the triad. It listens to Model events and reflects the data changes. The Controller handles user input and modifies Model data when appropriate.

As an example, suppose that a user changes a value in a text box, which is referred by the Model's property `name`. The Controller will call the `setName()` method of the Model and pass the new value. When the value is updated, the Model will send a notification to all subscribers informing them that the property `name` was updated. On retrieving the notification, all interested Views will read the Model's new name and update their representation accordingly.

Consequences The division of responsibilities of the Smalltalk'80 MVC pattern has proven to be very effective. As a result of the strict separation of concerns, developers are able to change a domain model without affecting the presentation logic and vice versa; the pattern allows to create several types of user interfaces (e.g., a command line and a graphical user interface) without affecting the application logic.

However, with the evolution and new demands of user interfaces, some weak spots of the Smalltalk'80 MVC pattern were revealed. We illustrate this with an example. Suppose that a text field of a financial report should be colored red if the value is negative, and black otherwise. The text color is purely a user interface property and therefore should not be part of the model. On the other hand, the standard text label view draws text black by default, and is not aware of red negative numbers. Smalltalk developers found ways to handle such cases, e.g., by developing custom Views that implement the required logic. However, these solutions did not solve the underlying problem: Smalltalk'80 MVC provides a good solution for displaying the model data itself; however, it provides no explicit means to deal with the presentation of state that is not part of the model but that makes a user interface more convenient for usage. This issue triggered further research and led to new MV* design patterns, such as Application Model and Model-View-Presenter.

Web MVC

Patterns to integrate a user interface with the application domain are widely used in web development. Originally, server-side MVC patterns dominated (as in ASP.NET MVC, Spring, Grails and other frameworks). Recently a number of JavaScript frameworks emerged based on MV* patterns(e.g. Backbone MVC[1] and Knockout MVVM[2]). However, in this section, we discuss the

[1]http://documentcloud.github.io/backbone/#FAQ-mvc
[2]http://knockoutjs.com/documentation/observables.html

original server-side Web MVC pattern.

Intent Separate the domain logic from the presentation logic for Web applications in three components with distinct responsibilities: Model for storing data, Controller for handling user actions, and View to generate HTML layout.

Motivation Due to specifics of the way the Web works, it matches well with the principles of the MVC design pattern. In particular, the Web inherently supports the separation between View and Controller responsibilities: data is displayed on the client-side as HTML pages, while the logic that handles user input is invoked on the server-side. HTML pages provide the inputs allowing users to interact with the system, that is, through hyperlinks, input forms, buttons. The provided data is processed on the server-side.

Structure The general principles of the MVC family apply to the Web MVC pattern: Model contains data to be displayed, View defines the presentation layout, and Controller handles user input. However, what changed compared to the Smalltalk'80 MVC pattern are the responsibilities of Model and Controller, and the way they work together. In Web MVC, the application logic is triggered by the Controller, while the Model only stores data that needs to be displayed on a certain View. Model can be a domain entity, but it can also be an entity not connected to the domain. For example, Model may store the data for rendering the Pagination control. Model is responsible for providing correct data for rendering the final layout. The Controller in Web MVC is responsible for handling user input and create the Model for further rendering. Controller has access to the application domain logic, it is able to read and modify domain data, run custom calculations, etc.

Collaborations Smalltalk'80 MVC uses observer synchronization to directly synchronize the user interface with the domain data. However, the Web is stateless and operates as a set of requests and responses, so there is no need of strong synchronization. Consequently, Web MVC can use flow synchronization.

As an example, consider a user that wants to visit a website to find data about books and enters a web address in a browser. The server receives the request and invokes an appropriate controller associated with given address. The Controller invokes business domain operations to obtain the list of books. The Controller then instantiates a model and initializes it with the list of books. Subsequently, the Model is passed to the View component

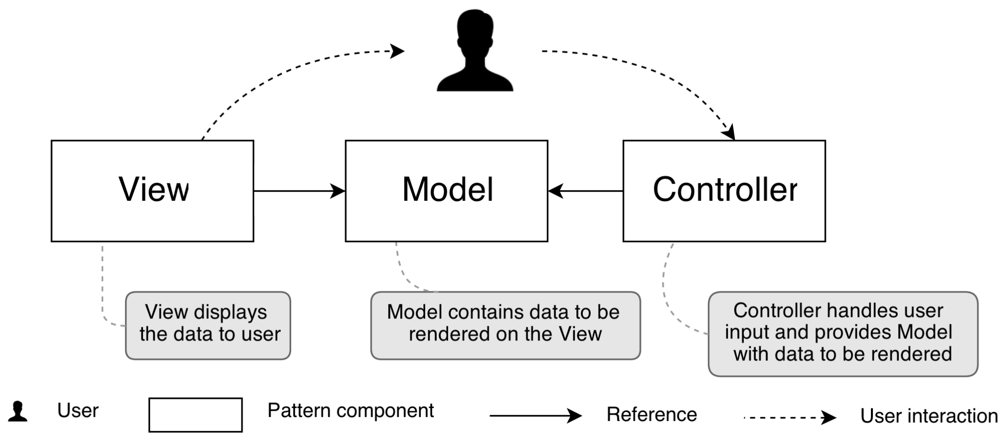

View
Model
Controller
View displays the data to user
Model contains data to be rendered on the View
Controller handles user input and provides Model with data to be rendered
User
Pattern component
Reference
User interaction

Figure 4.3: Web MVC pattern

for rendering. The View generates the layout of the page, which includes book titles, authors, covers and hyperlinks to page with book details, which is shown to the user.

Consequences The Web MVC design pattern supports clear separation of responsibilities of web application logic, which leads to better-organized code that is easy to understand and maintain. Therefore, the Web MVC has become a popular pattern in the domain of Web development. A growing number of frameworks employ the Web MVC design pattern (ASP.NET MVC, Spring, Ruby on Rails, Grails, etc.). This success is based on the inherent decoupling of the logic to display data from the logic of handling user input on the Web; the View works on the client-side, while the Controller works on the server-side.

MVC Summary

The fundamental feature of MVC patterns is strict separation between View and Controller responsibilities. The View is responsible for displaying data, the Controller for handling user input. This feature makes the pattern particularly convenient for Web applications, but less evident for desktop application built with many modern desktop development frameworks, where user interface widgets usually combine rendering and basic input handling capabilities. The Model in MVC contains data for View rendering, and Controller can operate on it. However, Model is not necessarily a domain model. Depending of the context of use, it could be a domain entity and/or just a set of properties for correct View rendering.

4.4.3 MVVM: Model-View-View Model

As we explained in the consequences of Smalltalk'80 MVC, one of the problems with the pattern is managing view state that is not part of the domain model. This issue resulted in a new family of MV* patterns called Model-View-View Model, also referred to as Model-View Presentation Model.

Presentation Model (or View Model) is a wrapper for the Domain Model (Model in the diagram). The Domain Model maintains domain state and the Presentation Model maintains View state [21]. The Presentation Model also handles the logic that is not part of the Domain Model.

The View in MVVM observes and operates with the Presentation Model, without direct reference to Domain Model. Furthermore, the View-Controller pair (as in MVC) is not considered as two distinct components, but merged in a single View component. This evolution was motivated by common practice of developers to embed both presentation logic and basic user input handling in user interface widgets.

We discuss two patterns of the MVVM family. The first is Application Model that emerged from the VisualWorks implementation of Smalltalk. The second is Microsoft MVVM that is used in technologies such as WPF and Silverlight.

Application Model

Intent Separate the domain logic from the presentation logic in four modules with distinct responsibilities: store and manage data, display data, handle user input and handle view state. Domain data and view state handling should be treated separately.

Motivation The key driver behind the Application Model design pattern is to handle MVC's inability to deal with view state and provide the ability to process user input before submitting it to the model.

Structure Application Model builds upon the Smalltalk'80 MVC pattern and inherits its basic structure. Model contains the domain data, View is responsible for displaying data, and Controller handles user input. However, View and Controller[3] do not directly interact with Model. Instead, the pattern includes an intermediate component, the Application Model (i.e., the

[3]There is still separation between View and Controller components, although the framework provides widgets; i.e., reusable UI elements that combine View and Controller functionality. In later patterns responsibilities of these two components are merged together, referred to as View.

Presentation Model in this pattern), that handles View state and provide the means to process user input before submitting it to the Model.

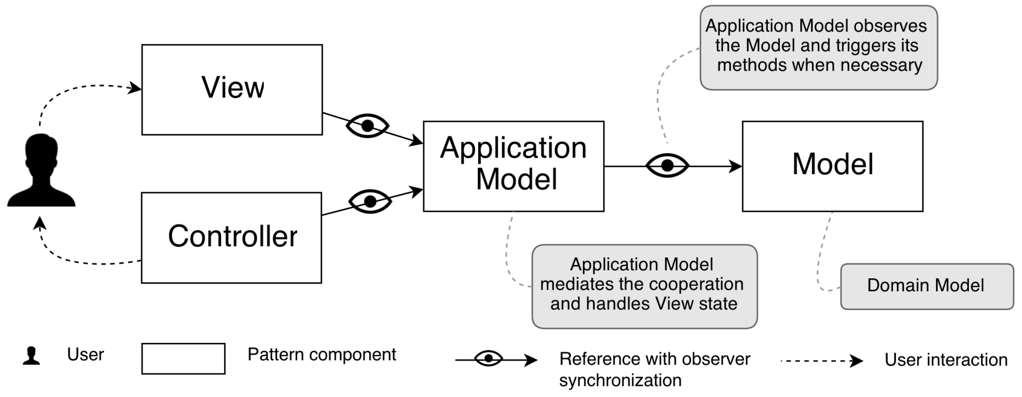

Application Model observes the Model and triggers its methods when necessary
View
Application Model
Model
Controller
Application Model mediates the cooperation and handles View state
Domain Model
User
Pattern component
Reference with observer synchronization
User interaction

Figure 4.4: Application Model pattern

Collaborations As Application Model extends MVC, the principle collaborations remains the same. View and Controller work with Application Model (that wraps Model with domain data), in a similar way as it worked with a regular Model. The main difference is the collaboration between Application Model and Model. In particular, Application Model observes the Model in order to notify Views when some data is changed. Application Model is also able to modify Model data and call its methods when necessary.

As an example, let us take the aforementioned case with a colored text field in a financial report. With the Application Model pattern, the developer defines two Application Model properties: `resultValue` that returns a number, and `resultColor` that returns a color. Both refer to the same property of a Model, the `resultValue`. The Application model observes the Model and updates the properties when the `resultValue` changes. View in turn observes the Application Model properties and updates the representation of the value when it changes.

Consequences Application Model deals with some of the shortcomings of the original Smalltalk'80 MVC pattern. Application Model simplifies handling view state and provides extra logic that deals with user input before passing it to Model. Nevertheless, in order to implement complex logic (such as the example with text color), the developer needs to write custom widgets and adapters. As this may be time consuming activities, developers will search for workarounds, such as direct modifications of widgets from the Application Model code. However, such solutions violate the separation of concerns principle, and consequently may complicate later maintenance.

Microsoft MVVM

Intent Separate the domain logic from the presentation logic in three modules with distinct responsibilities: handle the domain data (Model), view state and user interaction (View Model), and rendering of the visual user interface (View). Furthermore, bind View and View Model declaratively by leveraging on observer synchronization.

Motivation Microsoft MVVM allows every View to have its own View Model, while each View Model can have several Views. This allows displaying the same data in different ways, possibly simultaneously. Microsoft's realization of the Model-View-View Model works very efficiently with WPF and Silverlight, which were designed with MVVM in mind [19].

Structure The pattern has a linear structure. The View is responsible for rendering the user interface; it can observe the View Model, trigger its methods an modify its properties when needed. View maintains a one-way reference to the View Model. When a View Model property is changed, View is notified by observer synchronization. On the other hand, when a user interacts with the View, View Model properties are directly modified. View Model is responsible for handling view state and user interaction; it has access to the domain Model, so it could work with domain data and invoke business logic. View Model is unaware of View. Model is responsible for handling domain data and is unaware of View Model. This approach allows creation of several different views for the same data, and observer synchronization makes these views work simultaneously.

Collaboration View Model has a direct reference to Model in order to work with the domain data. View Model can use Model to invoke various kinds of actions, such as triggering service calls, accessing a database, etc.

The cooperation between View and View Model is particularly interesting. In order to achieve simple data synchronization, Microsoft introduced Data Binding. Data Binding allows a developer to bind user interface widget properties to View Model data in a declarative manner, without the need for writing explicit code in the View component. Every time the user interface widget changes, the corresponding field value of the View Model is updated,

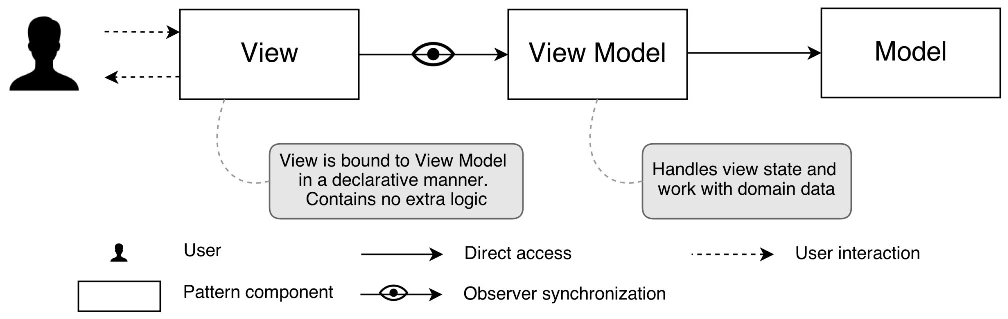

View
View Model
Model
View is bound to View Model in a declarative manner. Contains no extra logic
Handles view state and work with domain data
User
Direct access
User interaction
Pattern component
Observer synchronization

Figure 4.5: Microsoft MVVM pattern

and vice versa. The synchronization of View and View Model is completely handled by the WPF and Silverlight frameworks.

As an example, let us extend the earlier case of the financial report. The new goal is to provide a user two views of the report: a table and a pie chart. When data is changed in the Model, the View Model is notified. The View Model then calls the Model to get the required data (e.g. the list of objects that describe the subject and an amount) and puts the data in a property `Data`. Two Views (one for the table and one for the pie chart) observe the View Model and update their representations when the Data is changed. Each View has own logic how to present the data. The first view renders a table with columns subject and amount; the second uses the amount to draw a pie chart, and the subject to provide a legend.

Consequences Microsoft's MVVM pattern offers developers rich functionality for handling non-trivial synchronization of user interfaces with domain data. Declarative Data Binding between View and View Model keeps their logic separate. As such, the developer does not need to know how the View internally works; the only tasks are to handle the Model that deals with the domain data and logic, and the View Model that deals with view state and logic.

The power of Microsoft's MVVM pattern relies on the powerful mechanism of declarative Data Binding and automatic handling of the binding at runtime. Data Binding (and, therefore, the Microsoft MVVM pattern) is realized in all Microsoft User Interface frameworks. However, some realizations do not provide the required level of support for Data Binding; e.g., Windows Forms has a number of widgets (like Tree View), that do not have the required level of Data Binding support. This somehow limits the applicability of the pattern in practice.

MVVM Summary

The key feature of the MVVM patterns is a Presentation Model (Application Model and View Model respectively in the example patterns) that extends the Domain Model functionality. The Presentation Model offers support for handling view state by providing extra properties and logic. The patterns rely on observer synchronization. The View observes the Presentation Model, reflects upon its properties, changes them as the user updates the user interface, and calls appropriate methods when needed. As far as observer synchronization is essential part of MVVM, it relies on the support of the underlying technology.

4.4.4 MVP: Model-View-Presenter

The MVP pattern family builds upon the other MV* patterns aiming to improve them. In particular, the family uses MVC as a starting point, but its component roles and cooperation rules were adapted in order to achieve higher flexibility and deal with some shortcomings of the predecessors. MVP represents the biggest family of MV* design patterns. The first MVP pattern was introduced by IBM and its subsidiary Taligent and is described in Potel's paper in the mid 1990s[17]. The idea was later popularized by the Dolphin Smalltalk pattern realization.

Most concrete patterns of the family use observer synchronization and have a Presenter component that oversees the View, handles user events and, if necessary, modifies the View via direct calls. We discuss two examples of this group of patterns: Dolphin Smalltalk MVP and Supervising Presenter. Other members of the MVP family use flow synchronization. We discuss one pattern from this group, called Passive View. Passive View aims to combine the best of both worlds, i.e., the explicitness of flow synchronization and the clarity of separation of concerns.

Dolphin Smalltalk MVP

Intent Separate the domain logic from the presentation logic in three modules with distinct tasks: handle the domain data (Model), handle basic user input functionality and user interface rendering (View), and supervise the synchronization between View and Model by direct access to the components (Presenter).

Motivation The Dolphin Smalltalk developers spent a great effort on studying existing MV* patterns when facing the limitations of the MVC and

MVVM families [1]. To deal with the problems, they adapted the responsibilities of the pattern components. In MVP, the core of the application behavior is located in Presenter, and not in Model as in earlier patterns. Alongside that, the View component (which combines the View and Controller responsibilities) is kept as simple as possible. To that end, the Presenter has direct access to the View to supervise it and modify widgets when necessary.

Structure The Model component represents pure domain data. As in earlier patterns, Model is unaware of the presentation logic, but it still provides notification when the data is changed.

The role of View in MVP remained almost unchanged. View is responsible for displaying the data on the user interface. It also supports basic handling of user input: it delegates user actions to the Presenter by direct calls.

The Presenter is responsible for keeping the application synchronized. Presenter handles user input, invokes domain methods, keeps the Model in consistent state and provides extra logic to update the View when necessary.

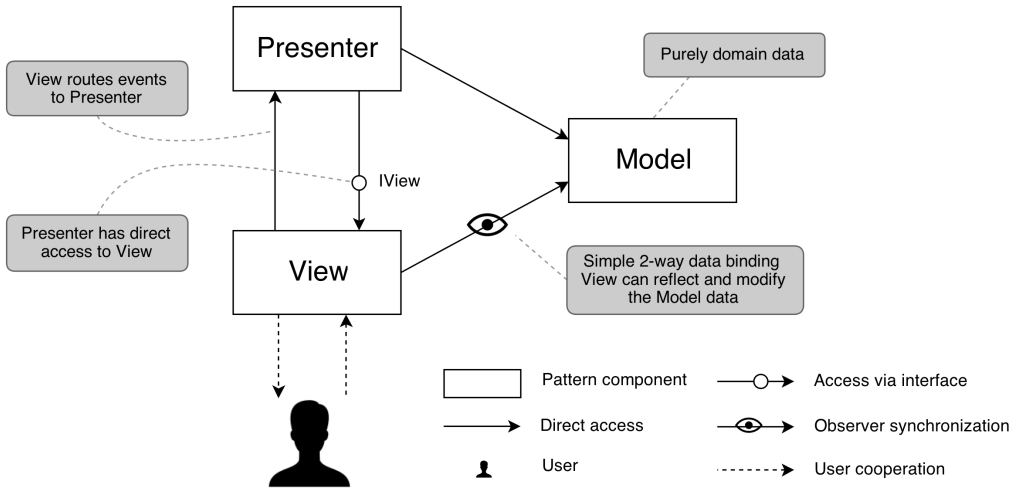


Figure 4.6: Dolphin Smalltalk MVP pattern

Collaboration The View observes the Model and represents its state in the user interface. Two-way data binding is supported. For simple cases, the Presenter has direct access to the View (e.g., updating a text box value will immediately update the model field). For cases that are more complex, the View routs events (by a direct call) to the Presenter. The Model data is displayed on the View automatically based on observer synchronization. If the representation of the View requires extra features (e.g., text coloring), the Presenter will handle this by invoking direct calls on the View.

Let us consider as an example a scenario of a user who edits the age of a contact person in an address book. Suppose the age should be number between 0 to 120, other values are considered as invalid. The contact data is a domain entity maintained in the Model. The View observes the contact properties and renders fields for their editing. When the user updates the `age` field, the corresponding Model property should be updated. However, as the number needs to be checked, validation logic needs to be executed, so the View triggers the Presenter to perform this check. The Presenter analyses the `age` value and commands the View to update the age if the value is valid, or alternatively show an error message and disable the Save button in case an invalid age was provided.

Consequences The great flexibility of the Dolphin Smalltalk MVP pattern comes from the decision that the Presenter can directly access the View. This design decision is the key difference compared to previously discussed patterns. Furthermore, the developer has the possibility of introducing an interface for the View, and hence make the Presenter work with different View realizations—as long as they implement the required interface.

Supervising Presenter

Intent Separate the domain logic from the presentation logic by creating three modules with distinct tasks: handle the domain data and possibly the presentation state (Model), handle simple mapping between the user interface and the data of the model (View), and handle input response and complex view logic (Presenter).

Motivation The Supervising Presenter pattern provides a step away from MVC towards the Widget-based User Interfaces approach aiming to make the pattern behavior more clear and flexible. Direct interaction between the Presenter and the View allows effective access to View widgets.

Structure According to M. Fowler [9], the Model does not have to be limited to contain domain data. The Model can also contain data required to render correctly the View state (similar to a Presentation model). The Model supports a notification mechanism to be observed by the View.

The View is responsible for data presentation and basic user-input handling. In contrast with the previously discussed patterns, the View has no direct access to the Presenter, which reduces coupling and improves separation of concerns.

The role of Presenter, as before, is to make the View and Model components work together. Presenter handles user input, updates the Model and View and invokes domain logic when needed.

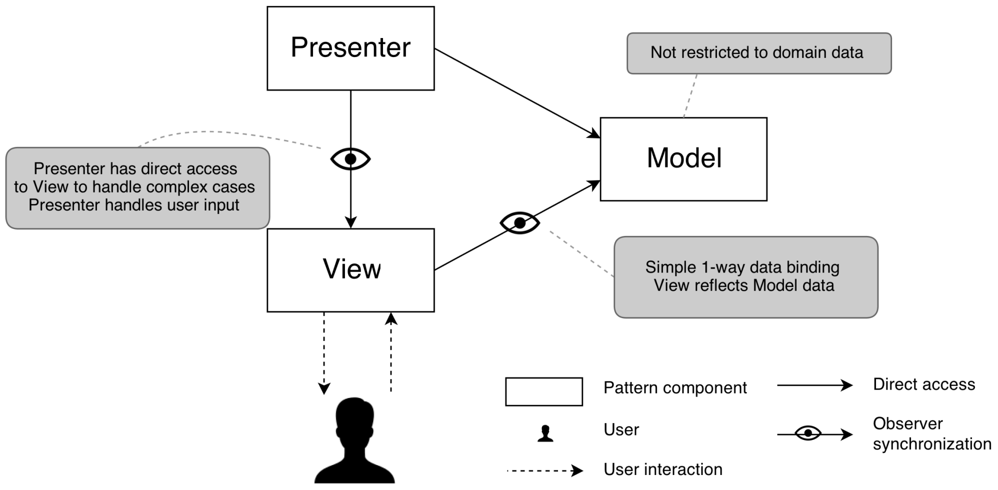

Presenter
Not restricted to domain data
Model
Presenter has direct access to View to handle complex cases Presenter handles user input
View
Simple 1-way data binding View reflects Model data
Pattern component
Direct access
User
Observer synchronization
User interaction

Figure 4.7: Supervising Presenter pattern

Collaboration The View reflects the Model data. However, the View does not modify the Model data; this responsibility is delegated to the Presenter. The View has no direct reference to the Presenter, but provides a notification mechanism, so it can be observed.

The Presenter has both direct access to the View and it can observe the View. The Presenter is able to modify the View directly, but this only applies to complex cases, when data requires processing before it can be used for updating the user interface.

Let us now see how the example of editing the age of a contact person in an address book would work with the Supervising Presenter pattern. With this pattern, the Model is not restricted to domain data and may contain fields with contact information (e.g. `age`), as well as flags that indicate if values are valid (e.g. `ageValid`). The View observes the values and renders fields for their editing. The Presenter observes the View, and invokes validation logic when the `age` field is updated on the View. The validation result is put to the `ageValid` field of the Model. The View reacts to the change by showing or hiding a corresponding error message and enabling/disabling the Save button.

Consequences The Supervising Presenter pattern maintains good separation of concerns and removes logic from the View component, improving understandability. By not being restricted to domain data, the Model can provide capabilities for managing View state. On the other hand, Presenter has significant knowledge about the View component. This may lead to difficulties when several Views for the same data are required. It may also negatively affect testability.

Passive View

Intent Separate the domain logic from the presentation logic by creating three modules with distinct tasks: handle the domain data (Model), provide a representation of the data (View), and handle user synchronization and View state management (Presenter).

Motivation The foremost driver of the Passive View pattern is testability. Automated testing proved to be powerful tool, but graphical user interfaces are often considered as the most difficult area for unit testing. Therefore, the idea of Passive View [6] is simple: if it is hard to test the view, make it so simple that no real testing is needed!

Structure The structure of the Passive View pattern closely resembles with the Microsoft's MVVM pattern, but conceptually there is a huge difference. While MVVM fully relies on observer synchronization, Passive View utilizes flow synchronization.

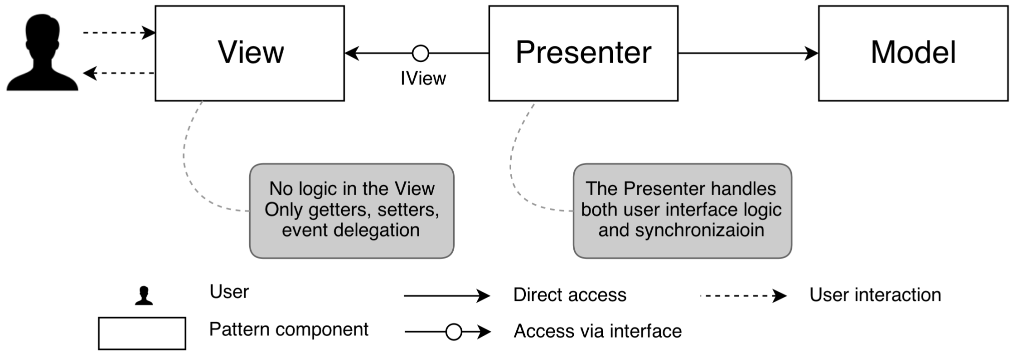


Figure 4.8: Passive View pattern

The Model refers to the domain, providing business logic and data. Model has no access to the other components of the pattern and does not implement a notification mechanism.

The View component is kept as simple as possible. It might have only getters and setters, and event delegation logic. This way, the View is just a lightweight shell with no real logic, which makes testing trivial.

The Presenter component does all the work. It handles user input, invokes business logic and updates the View state with new data. All the synchronization logic resides in the Presenter, and it uses explicit invocations.

Consequences Passive View may lead to the same problems as Widget-based User Interfaces: when the user interface becomes rich the synchronization logic complexity grows, which may eventually result in code that is very difficult to maintain. Nevertheless, Passive View provides better separation of concerns than widget-based user interfaces, and provides as such much better support for testability.

Let us now see how the example with the address book works with the Passive View pattern. With this pattern the Model is restricted to the domain. When the user opens a contact, the Presenter retrieves the contact entry from the Model and updates the text fields on the View with contact data. When the user updates the `age` field value, the Presenter is triggered. The Presenter then analyses the data and updates the View via direct calls: it commands View to show or hide the error message, and sets the correct state of the Save button. The domain entity would be updated only when user clicks the Save button to finish the editing.

MVP Summary

The central component of the MVP patterns is the Presenter that directly accesses the View and Model and coordinates their interaction. The Presenter handles synchronization, invokes domain method calls and handles user input delegated from the View.

The View and Model components can use observer synchronization for handling simple cases. This approach allows to reduce the complexity of the Presenter, as the synchronization is handled automatically, and Presenter only needs to handle complex cases. Alternatively, the View and Model may have no explicit access to the other components, making the Presenter fully responsible for synchronization. In between, the View may be aware of the Presenter and invoke its methods by direct calls. These different variants make MVP a very flexible pattern that can be effectively adopted to many use case scenarios.

4.5 Case Study

As explained in the introduction of the paper, we faced the manageability problems of synchronizing a user interface with the domain data in a recent R&D project between Danfoss AB and the Department of Computer Science at Linnaeus University. In this project, we studied a Windows Forms desktop application (developed on top of .NET) to automatically notify users when software updates become available, as well as semi-automatically support dynamic downloading and installation of the updates.

We started with using a standard Windows Forms approach for user interaction. However, due to new requirements and increasing size and complexity of the software, we were forced to find a systematic solution. Some of the user interface requirements we had to deal with are:

- The user interface must be responsive at any point of time.
- The user interface must prevent the user from performing actions that are not possible for any given situation.
- The Windows task bar notifications should notify the user when new updates are available.
- A Tree View allows the user to select content of interest.
- The user interface must dynamically display the updating progress, including listing of updating items, their status (waiting, downloading, installing), and download progress in bytes.

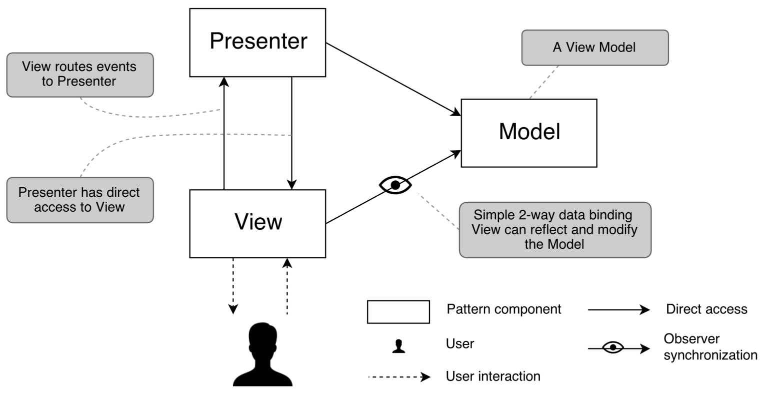
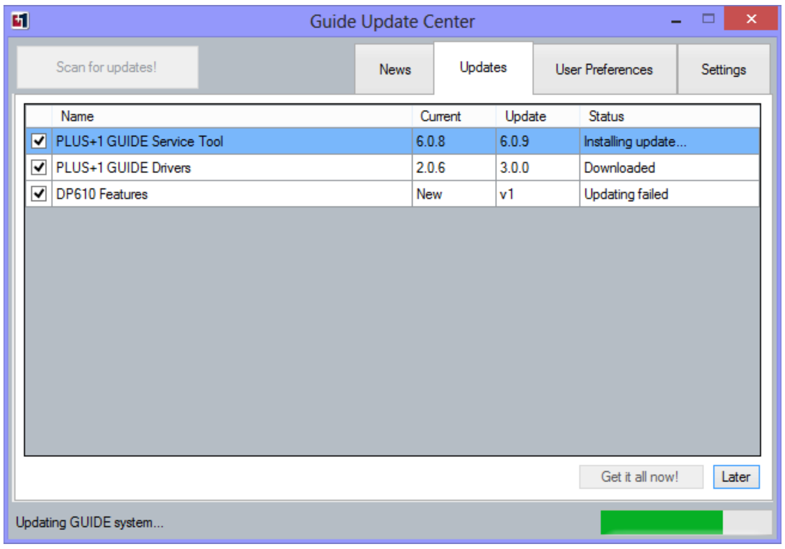

Presenter
View routes events to Presenter
A View Model
Model
Presenter has direct access to View
View
Simple 2-way data binding View can reflect and modify the Model
Pattern component
Direct access
User
Observer synchronization
User interaction

Figure 4.9: Our realization of the MVP pattern

After studying the different patterns and evaluating their pros and cons, we decided to select Dolphin Smalltalk MVP. The MVC patterns turned out impractical for the Windows Forms application, as user interface widgets already combine View and Controller functionality. MVVM appeared as a better solution, as Windows Forms has some built-in support for this pattern. Unfortunately, some of the user interface widgets we used (e.g. Tree View) did not provide decent data binding support. We also required direct access to the View in order to implement features like Windows task bar notifications.

Our realization of the Dolphin Smalltalk MVP pattern differs from the original pattern as the Model component is not a pure domain entity, but includes some view state handling. View and the Model are connected via two-way data binding that handles the major part of synchronization; cases that are more complex (e.g., Windows task bar notifications) are handled through the Presenter.

The introduction of the pattern significantly improved the application structure and maintainability. By allocating distinct responsibilities to Model, View, and Presenter we realize better separation of concerns which improved significantly understandability and extensibility. The use of Data Binding reduced the size of the user interface synchronization code. The synchronization happens mostly automatically, except for the complex cases (with Tree View), which are managed by the Presenter. Figure 4.10 shows a screenshot of the user interface of the application.



	Name	Current	Update	Status
✓	PLUS+1 GUIDE Service Tool	6.0.8	6.0.9	Installing update...
✓	PLUS+1 GUIDE Drivers	2.0.6	3.0.0	Downloaded
✓	DP610 Features	New	v1	Updating failed

Updating GUIDE system...

Figure 4.10: Screen shot of the desktop application UI

4.6 Conclusions

MV* design patterns provide reusable solutions to the recurring problem of synchronizing user interfaces with domain data. In this paper, we have provided a overview of the major MV* pattern families and discussed concrete patterns of each family. The study shows that there is no single dominating leader, as each pattern family has its particular pros and cons, as summarized in Table 4.1.

Patterns	Pros	Cons
MVC	- Original pattern with separation of concerns. - Good choice for web applications as View and Controller roles are naturally separated in this context.	- Poor handling of view state logic. - Assumes decoupled View and Controller (in practice UI widgets usually combine these responsobilities).
MVVM	- Supports multiple views for the same data. - Declarative specification and automatic synchronization of View and View Model. - Strong separation of concerns.	- Extensive use of observer synchronization may affect performance. - Relies on underlying technology.
MVP	- Flexibility to allocate responsibilities in different ways. - Can be adjusted to a wide range of application scenarios.	- Not very strict regarding separation of concerns. May increase the complexity of the code and hamper maintainability.

Table 4.1: Summary of pros and cons of MV* pattern families

MVC patterns are the pioneering patterns for synchronizing user interfaces with domain data and an excellent choice for Web-based applications as the Web structure naturally supports the division of responsibilities of the components for MVC patterns. However, these patterns suffer from poor handling of view state logic, and assume decoupled View and Controller which does not match with many state of the frameworks in practice.

MVVM patterns support simultaneous representing of multiple views on the same data. State of the art frameworks that support MVVM provide support for declarative specification of parts of the synchronization and its automatic execution. MVVM emphasizes separation of concerns, which support understandability and maintainability. On the other hand, extensive use of observer synchronization combined with multiple views can have a negative effect on system performance.

MVP patterns provide flexibility for the designer who can allocate responsibilities in different ways, so the patterns can be adjusted to a wide range of application scenarios. On the counter side, MVP patterns are not very strict regarding separation of concerns, which may increase the complexity of the code and hamper maintainability.

This evaluation summary brings us to the conclusion that the developer must be careful when selecting a pattern for a problem at hand. The decision must be grounded on the use cases at hand, the primary quality requirements, and the underlying technology that is chosen. On the other hand, as illustrated in the case study, patterns provide flexibility and can be tailored to the problem at hand, or alternatively concepts of different patterns can be combined to provide a proper solution.

Interesting dimensions for future research on MV* patterns are models pollution (data in Model may range from pure domain data, representational data, to domain-related operations), developer effort (to develop the different components of patterns), components distribution (study how patterns differ in how components are distributed over different machines). To provide deeper insight in the world of MV* design patterns a systematic literature review is required. Meanwhile, we hope that the study result presented in this paper, will bring more clarity in the variety of MV* patterns and help practitioners to make better grounded solutions when selecting patterns.

5 The Solution

In this chapter we discuss the pattern we have built for GUIDE Update Center and how it helped to fix the problem.

5.1 Pattern Selection

After studying the variety of patterns we have got a solid understanding of discussed pattern families, and got means to make a reasonable choice of UI architecture.

MVC pattern was rejected immediately as it proved unpractical for Windows Forms applications. It considers UI rendering and user input handling as two separate components that know almost nothing about each other. This concept does not fit Windows Forms technology, as it provides a set of UI controls that already combine these two responsibilities. Forcing them to be separate again would barely be justified. Besides, MVC patterns have problems with handling view state, which is another reason for finding a better alternative.

MVVM patterns appear to be much better decision. They provide strong separation of concerns and reach capabilities for handling view state. Windows Forms supports Data Binding, a concept that allows to bind UI and underlying data in declarative manner, without writing executable code. This approach could reduce efforts for UI integration significantly, that is why we are willing to use it in our solution.

Unfortunately, MVVM has some drawbacks as well. Unlike Microsoft WPF UI framework (where MVVM is supported out of the box), Windows Forms has lower level of Data Binding support. That makes it less convenient to adapt MVVM for this technology. For example, if does not support command objects—a way to bind user commands (like clicking a button or giving focus to a control) to View Model methods; handling such cases would require developer to write delegation logic explicitly. Some UI widgets have insufficient level of Data Binding support, e.g. Tree View control cannot be effectively bound to data. Finally, our application logic requires us to display Windows task bar notifications. In Windows Forms it would require us to call View method explicitly. It cannot be done in MVVM as View Model maintains no reference to the View.

MVP is the most flexible family of patterns. It gives much control to developer, allowing us to decide on every aspect of synchronization we need. MVP allows us to use Data Binding where it fits us; it also allows us to handle synchronization manually in cases where Data Binding capabilities are not enough to reach our goals.

5.2 Pattern Overview

The final version of a patterns we have created is mostly based on Dolphin Smalltalk MVP, with some ideas borrowed from Martin Fowler's Supervising Presenter. This way we have got a way to combine flexibility of MVP with expressive power of Data Binding and high level of handing view state.

5.2.1 Structure

As in M. Fowler's vision of MVP, the model is a View Model. It provides domain data as well as view state related data in an appropriate format for binding to the View component.

The View is responsible for rendering data. It also supports basic user input handling by delegating user events to the Presenter.

Presenter is a core of the pattern. It maintains references to both View and Model and is capable of modifying them directly. It handles user input, invokes business rules, and handles complex synchronization cases in order to keep UI consistent.

5.2.2 Collaboration

The View and Model are bound by two-way Data Binding. View observes Model and redraws itself each time when Model state is changed. Simple user actions (like modifying textbox value or changing checkbox state) are automatically sent to Model.

User actions that cannot not be handled in that way (e.g. button click), are delegated to Presenter that takes care of their proper handling.

Presenter has direct access to both Model and the View. Presenter could read Model properties in order to know user preferences and settings. It updates Model when some data is retrieved from server (and then that data is automatically reflected on the View).

Presenter can manipulate View directly, which allows us to trigger Windows task bar notifications.

5.3 Consequences

The pattern solves the problems we have faced so far, though it still could be improved. There is a strong coupling between View and Presenter components; it hampers separation of concerns. It could negatively affect testability and maintainability of application in long perspective. In order to fix the issue, we should reduce dependency between View and Presenter. Supervising Presenter MVP shows how to do that: View should not reference Presenter, the latter should observe the former. All user events must be delegated to Presenter by rising notifications, not by invoking direct calls.

Another possible issue is a common problem of all MVP patterns we have observed. Presenter references the View; such approach makes it hard to support several views for the same data—a task that was trivial for MVVM. A way to fix it in MVP is adding intermediate layer between View and Presenter, it could be called a View Adapter. It would reference a list of Views implementing certain interface. Presenter would work with View Adapter which in turn would relay these commands to all his Views. On the other hand, this layer will add some complexity to the architecture, so it should be implemented only if there is a necessity of supporting several views of the same data.

5.4 How Does the Pattern Solve the Problem

The chosen pattern takes benefit of declarative Data Binding. The state of UI controls is defined by set of View Model properties they are bound to. For example, we could create `CanScanForUpdates` property in View Model and bind it to `button.isReadOnly` property for all UI buttons that could start scanning for updates. Consequently, maintaining consistent UI means maintaining a consistent set of properties in View Model.

First step to achieve this was adding a *State* property indicating current state of the application (Table 5.1).

The State property is a main (but not only) property that allows us make a decision about UI control states. Below we provide a partial scheme of our View Model (Figure 5.1) to show how UI-related properties are calculated, and their mapping to View widgets.

1. **File System.** This property represents a tree to be rendered in User Preferences tab.

2. **Update Suggestions.** This property contains a list of found updates to be displayed on Scan Results view.

3. **State.** The property indicating application state.

4. **Is Global Idle.** Boolean property that indicates that no activity is currently running in application.

5. **Is Global In Progress.** Boolean property that is opposite to previous one. Indicates that some activity is running in background. Equals true when application state is `ScanInProgress`, `UpdateInProgress` or `RetrievingFileSystemInProgress`.

Status	**Description**
Idle	The application is in its default state. No activity is running. *Updates* tab is in Idle mode (Figure 3.2). Buttons are enabled, progress bars hidden.
ScanInProgress	The scan for updates is in progress. Buttons should be disabled, progress bars should be shown, etc.
ShowScanResults	The scan is complete and there are some updates found. Updates tab should turn to Scan Results mode (Figure 3.3). Buttons are enabled, progress bars hidden.
UpdateInProgress	The update scenario is running. Buttons are disabled, progress bars shown, scan results table is read-only.
ShowUpdateResults	Updating is complete. Buttons are enabled, progress bars hidden. Scan results table is read-only, user can review success status of each package.
RetrievingFileSystem InProgress	Application is retrieving content catalog for tree view on User Preferences tab. Buttons are disabled, progress bars shown, tree view is read-only.

Table 5.1: Application States

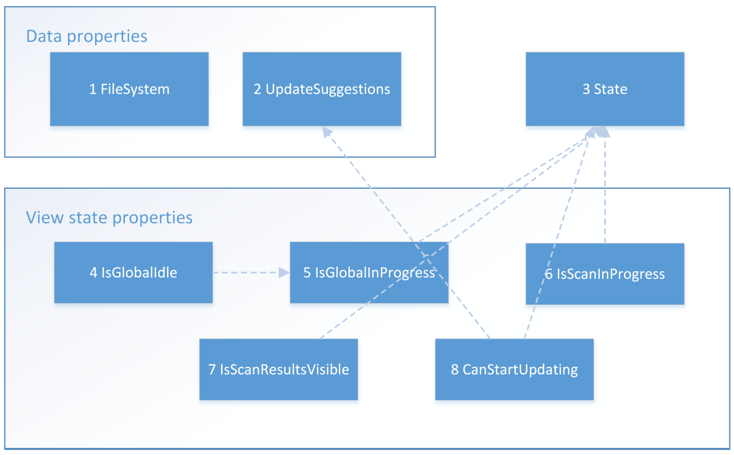


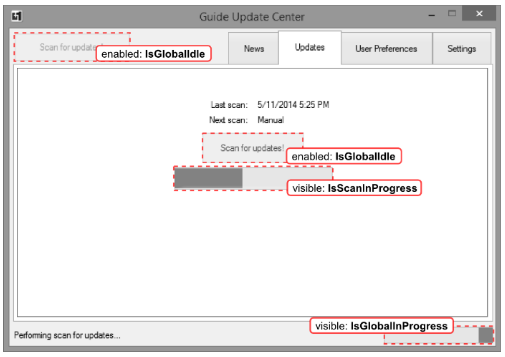
Figure 5.1: ViewModel scheme

6. **Is Scan In Progress.** This flag indicates that scanning for updates is currently running.

7. **Is Scan Results Visible.** This flag indicates that Update tab should be turned to `Scan Results` mode, as system detected some updates that could be downloaded and installed.

8. **Can Start Updating.** This flag indicates if application can start updating scenario. It equals true when application is in `ShowScanResults` state and at least one item in Update Suggestions is selected.

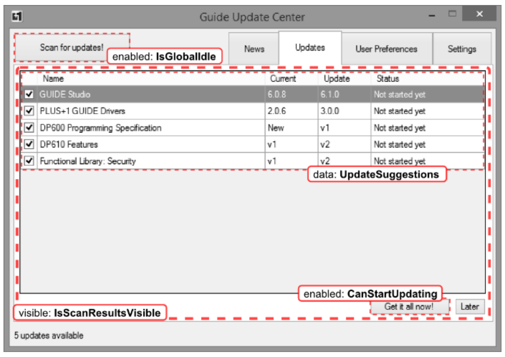
Figures 5.2 and 5.3 represent how UI widgets are bound to aforementioned View Model properties.



Last scan: 5/11/2014 5:25 PM
Next scan: Manual

Performing scan for updates...

Figure 5.2: VM Mapping: Updates tab while scanning



	Name	Current	Update	Status
✓	GUIDE Studio	6.0.8	6.1.0	Not started yet
✓	PLUS+1 GUIDE Drivers	2.0.6	3.0.0	Not started yet
✓	DP600 Programming Specification	New	v1	Not started yet
✓	DP610 Features	v1	v2	Not started yet
✓	Functional Library: Security	v1	v2	Not started yet

5 updates available

Figure 5.3: VM Mapping: Updates tab in Scan Results mode

As we see on the scheme, most view state properties depend on State. When State is changed, its dependencies are automatically recalculated, which, in turn, forces View to update. This way, we are released from necessity of writing dozens lines of code (as we discussed in chapter 3.2) for updating UI. The only thing we need is to change the State property.

Below I provide pseudo code sample of Presenter method that runs scanning for updates.

```
ScanForUpdates() {
    ViewModel.State = ScanInProgress

    ViewModel.PackageUpdates = businessLogic.ScanForUpdates()

    if (ViewModel.PackageUpdates.Count > 0) {
        ViewModel.State = ShowScanResults
        RaiseWindowsTaskBarNotification()
    } else {
        ViewModel.State = Idle
    }
}
```

6 Conclusions

The goal of this master degree project is to study software design patterns for building user interfaces, to provide an overview of existing practices and give some guidance how to choose proper pattern for custom needs. In this thesis work we studied three major families of UI architectures—MVC, MVP and MVVM, taking in consideration several patterns from each family, including both original and modern implementations. We performed analysis and found which features are essential for each pattern and which are optional, identified their area of use, benefits and shortcomings.

As expected, the study showed that there is no single leader pattern. Each pattern has situations when it performs better that others. Furthermore, pattern are deliberately left underspecified, which means that they could and often should be tuned for particular problem at hand. This way they grant software architects power to adapt familiar UI integration concepts for a range of problems.

The practical problem that inspired the study gives an illustration of how patterns could be tailored for specific application. We combined two MVP patterns: first one, Dolphin Smalltalk, gave us a general scheme of a pattern and collaboration rules (two-way Model-View binding), the second one, Supervising Presenter, provided a way for better View State management (the Model is actually a View Model). Such combination of pattern resulted in an effective UI integration, clean code that is easy to maintain and expand.

The study brings us to conclusion that developer should be extra careful when selecting an UI architecture for a project. The decision must be grounded on requested use cases and system requirements, capabilities of chosen technology and other available project knowledge. However, developer should not blindly apply one of existing patterns. Software design patterns provide flexibility and can be tailored to the problem at hand, or alternatively concepts of different patterns can be combined to provide a proper solution.

There are two main directions of a future work. First, as it was noted earlier, the study takes pragmatic approach as we analyze limited set of patterns. Performing systematic literature review on UI architectures would increase the value of the study significantly. Overview of Javascript UI frameworks represents an interesting direction as well, as far as they were wrongly omitted in our report.

Another direction of further work, suggested by one of reviewers of original paper, is widening study scope. We could study such ideas as model pollution (the role of Model component in each pattern and its responsibilities), components distribution within pattern, and assess amount of efforts required for development.

References

[1] A. Bower and B. McGlashan, “Twisting the triad,” *Tutorial Paper for European Smalltalk User Group (ESUP)*, 2000.

[2] S. Burbeck, “Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc),” *Smalltalk-80 v2. 5. ParcPlace*, 1992.

[3] P. Clements and M. Shaw, “The Golden Age of Software Architecture: Revisited,” *IEEE Software*, vol. 26, no. 4, pp. 70–72, 2009.

[4] J. Coutaz, “PAC, an Object-Oriented Model for Dialog Design,” *Human-Computer Interaction*, vol. Interact, pp. 431–436, 1987.

[5] E. Dijkstra, *Selected writings on computing : a personal perspective.* New York: Springer-Verlag, 1982.

[6] M. Feathers, “The humble dialog box,” *Object Mentor*, 2002.

[7] M. Fowler, *Patterns of Enterprise Application Architecture.* Addison Wesley, 2003.

[8] ——, “GUI architectures,” http://martinfowler.com/eaaDev/uiArchs.html, 2006, [Online; accessed 06-October-2013].

[9] ——, “Supervising controller,” http://www.martinfowler.com/eaaDev/SupervisingPresenter.html, Tech. Rep., 2006., Tech. Rep., 2006, [Online; accessed 06-October-2013].

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] T. Hopkins and B. Horan, *Smalltalk: an introduction to application development using VisualWorks.* London New York: Prentice Hall International (UK) Ltd., 1995.

[12] A. Karagkasidis, “Developing gui applications: Architectural patterns revisited.” in *European Conference on Pattern Languages of Programs (EuroPLOP)*, 2008.

[13] S. Koirala, “Comparison of Architecture patterns MVP(SC), MVP(PV), PM, MVVM and MVC,” http://www.codeproject.com/Articles/66585/Comparison-of-Architecture-presentation-patterns-M, 2010, [Online; accessed 06-October-2013].

[14] G. E. Krasner and S. T. Pope, “A cookbook for using the model-view controller user interface paradigm in smalltalk-80,” *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759

[15] Microsoft, “ASP.NET MVC Overview,” http://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx, [Online; accessed 06-October-2013].

[16] ——, “Windows Forms Overview,” http://msdn.microsoft.com/en-us/library/8bxxy49h.aspx, [Online; accessed 06-October-2013].

[17] M. Potel, “MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java,” *Taligent Inc*, 1996.

[18] G. Rocher, P. Ledbrook, M. Palmer, J. Brown, L. Daley, and B. Beckwith, “The Grails Framework - Reference Documentation,” http://grails.org/doc/latest/, [Online; accessed 06-October-2013].

[19] J. Smith, “Wpf apps with the model-view-view model design pattern,” *Microsoft Developer Network magazine*, no. 2009, 2009.

[20] Spring, “Web MVC framework,” http://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx, [Online; accessed 06-October-2013].

[21] P. Sukaviriya, J. D. Foley, and T. Griffith, “A second generation user interface design environment: the model and the runtime architecture,” ser. INTERACT and CHI, 1993. [Online]. Available: http://doi.acm.org/10.1145/169059.169299

[22] A. Syromiatnikov and D. Weyns, “A journey through the land of model-view-* design patterns.” International Working Conference on Software Architecture (WICSA).