

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

Artem Syromiatnikov

Department of Computer Science  
Linnaeus University  
Växjö, Sweden  
artem.syromiatnikov@gmail.com

Danny Weyns

Department of Computer Science  
Linnaeus University  
Växjö, Sweden  
danny.weyns@lnu.se

**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 paper 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.

## I. 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 understandability and maintainability of the program code [10], [21]. 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 Sauer-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 II, we briefly explain the methodology we used in this study. Section III gives an overview of the pattern families. In section IV, 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 V illustrates the selection of a pattern in the project with Sauer-Danfoss AB. Finally, we draw conclusions in VI.

## II. 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 [2], [14], 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 Sauer-Danfoss AB. Finally, we classified the different MV\* design patterns and documented the results of the study in this paper.

## III. 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.

### A. 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.

Fig. 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 Fig. 1.

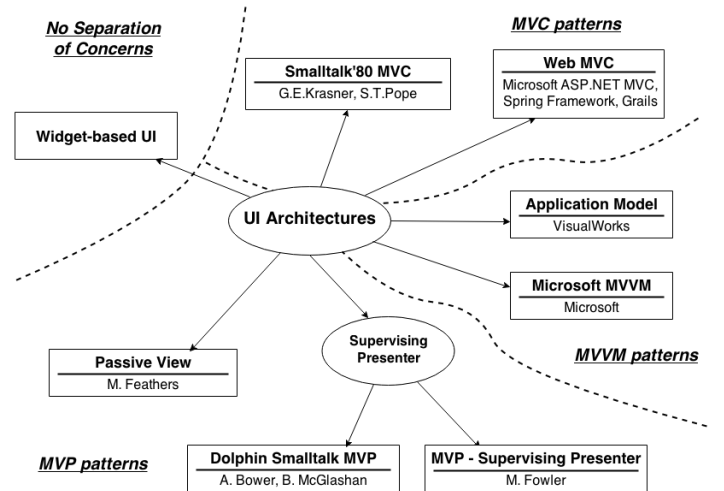


Fig. 1. The Land of MV\* Design Patterns

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.

### B. 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 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, life-cycle 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.

## IV. 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.

### A. 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.

### B. 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 [10], [14]. 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.

#### 1) *Smalltalk'80 MVC:*

a) *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.

*b) 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].

*c) 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.

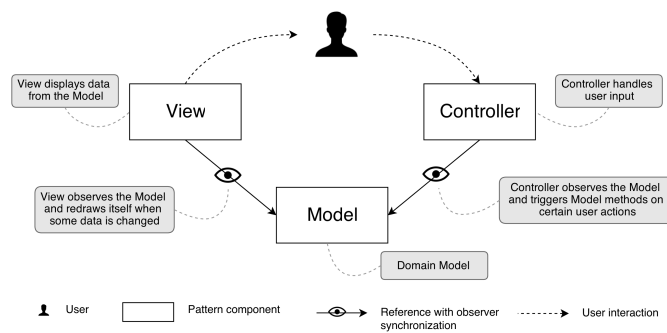


Fig. 2. Smalltalk'80 MVC pattern

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.

*d) 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.

*e) 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.

## 2) 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<sup>1</sup> and Knockout MVVM<sup>2</sup>). However, in this section, we discuss the original server-side Web MVC pattern.

*a) 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.

*b) 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.

*c) 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,

<sup>1</sup><http://documentcloud.github.io/backbone/#FAQ-mvc>

<sup>2</sup><http://knockoutjs.com/documentation/observables.html>

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.

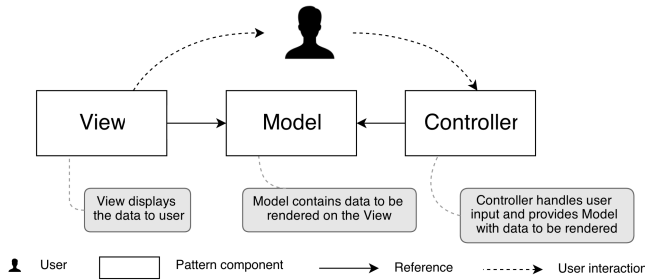


Fig. 3. Web MVC pattern

*d) 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 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.

*e) 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.

*3) 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.

### C. 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.

#### 1) Application Model:

*a) 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.

*b) 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.

*c) 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<sup>3</sup> do not directly interact with Model. Instead, the pattern includes an intermediate component, the Application Model (i.e., the Presentation Model in this pattern), that handles View state and provide the means to process user input before submitting it to the Model.

*d) 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

<sup>3</sup>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.

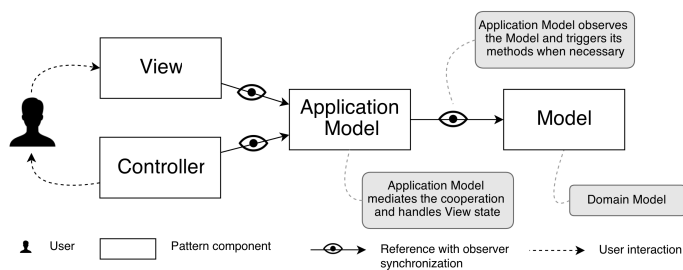


Fig. 4. Application Model pattern

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 ApplicationModel properties and updates the representation of the value when it changes.

*e) Consequences:* Application Model deals with some of the shortcomings of the original Smalltalk80 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 adaptors. 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.

## 2) Microsoft MVVM:

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

*b) 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].

*c) 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 and 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.

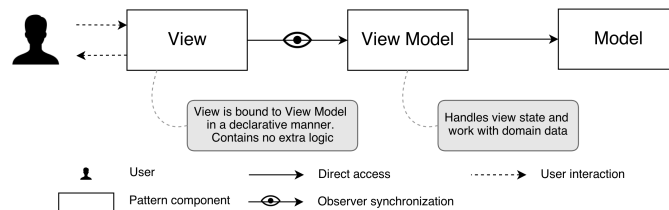


Fig. 5. Microsoft MVVM pattern

*d) 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, 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.

*e) 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.

3) *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.

#### D. 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.

##### 1) Dolphin Smalltalk MVP:

a) *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).

b) *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.

c) *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.

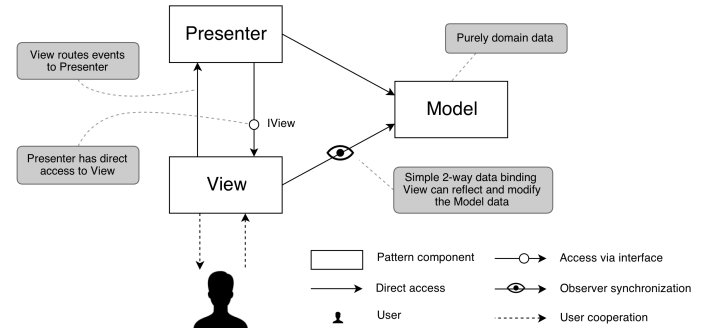


Fig. 6. Dolphin Smalltalk MVP pattern

d) *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 routes 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.

e) *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.

##### 2) Supervising Presenter:

a) *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).

*b) 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.

*c) 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.

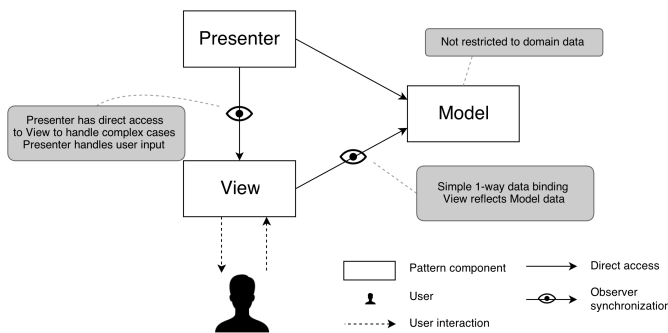


Fig. 7. Supervising Controller pattern

*d) 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.

*e) 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.

### 3) Passive View:

*a) 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).

*b) 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!

*c) 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.

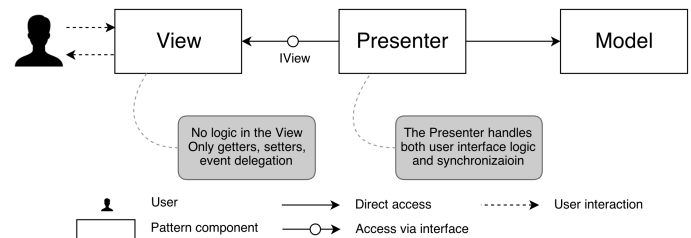


Fig. 8. Passive View pattern

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.

*d) 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.

4) *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.

## V. 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 Sauer-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.

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.

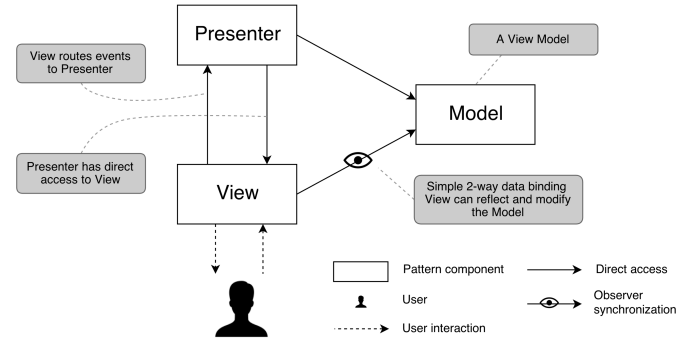


Fig. 9. Our realization of the MVP pattern

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. Fig. 10 shows a screenshot of the user interface of the application.

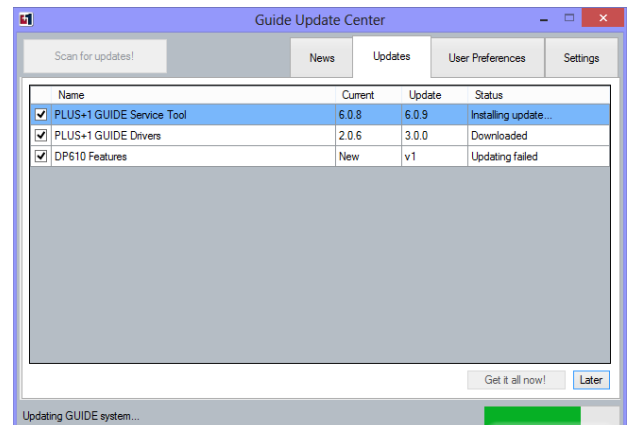


Fig. 10. Screen shot of the user interface of the Sauer-Danfoss application

## VI. CONCLUSIONS

MV\* design patterns provide reusable solutions to the recurring problem of synchronizing user interfaces with domain data. In this paper, we have provided an 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 Fig. 11.

Patterns	Pros	Cons
MVC	<ul style="list-style-type: none"> <li>- Original pattern with separation of concerns.</li> <li>- Good choice for web applications as View and Controller roles are naturally separated in this context.</li> </ul>	<ul style="list-style-type: none"> <li>- Poor handling of view state logic.</li> <li>- Assumes decoupled View and Controller (in practice UI widgets usually combine these responsibilities).</li> </ul>
MVVM	<ul style="list-style-type: none"> <li>- Supports multiple views for the same data.</li> <li>- Declarative specification and automatic synchronization of View and View Model.</li> <li>- Strong separation of concerns.</li> </ul>	<ul style="list-style-type: none"> <li>- Extensive use of observer synchronization may affect performance.</li> <li>- Relies on underlying technology.</li> </ul>
MVP	<ul style="list-style-type: none"> <li>- Flexibility to allocate responsibilities in different ways.</li> <li>- Can be adjusted to a wide range of application scenarios.</li> </ul>	<ul style="list-style-type: none"> <li>- Not very strict regarding separation of concerns. May increase the complexity of the code and hamper maintainability.</li> </ul>

Fig. 11. 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.

## 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/eaDev/uiArchs.html>, 2006, [Online; accessed 06-October-2013].
- [9] —, "Supervising controller," <http://www.martinfowler.com/eaDev/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 (EuroPLOT)*, 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](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/8bxy49h.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](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>