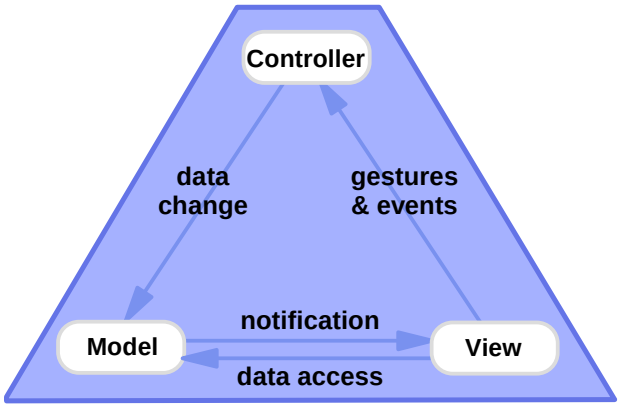
MVP: Model-View-Presenter
The Taligent Programming Model for C++ and Java

Mike Potel
VP & CTO
Taligent, Inc.

Taligent, a wholly-owned subsidiary of IBM, is developing a next generation programming model for the C++ and Java programming languages, called Model-View-Presenter or MVP, based on a generalization of the classic MVC programming model of Smalltalk. MVP provides a powerful yet easy to understand design methodology for a broad range of application and component development tasks. Taligent's framework-based implementation of these concepts adds great value to developer programs that employ MVP. MVP also is adaptable across multiple client/server and multi-tier application architectures. MVP will enable IBM to deliver a unified conceptual programming model across all its major object-oriented language environments.

Smalltalk Programming Model

The most familiar example of a programming model is the Model-View-Controller (MVC) model[1] of Smalltalk developed at Xerox PARC in the late 1970's. MVC is the fundamental design pattern used in Smalltalk for implementing graphical user interface (GUI) objects, and it has been reused and adopted to varying degrees in most other GUI class libraries and application frameworks since.



Smalltalk represents a GUI object such as a check box or text entry field using the three core abstractions of MVC. A model represents the data underlying the object, for example, the on-off state of a check box or the text string from a text entry field. The view accesses the data from the model and specifies how the data from the model is drawn on the screen, for example, an actual checked or unchecked check box, or a text entry box containing the string. The controller determines how user interactions with the view in the form of gestures and events cause the data in the model to change, for example, clicking on the check box toggles the on-off state, or typing in the text entry field changes the underlying string. The model closes the loop by notifying the view that its state has changed and the view needs to be redrawn.

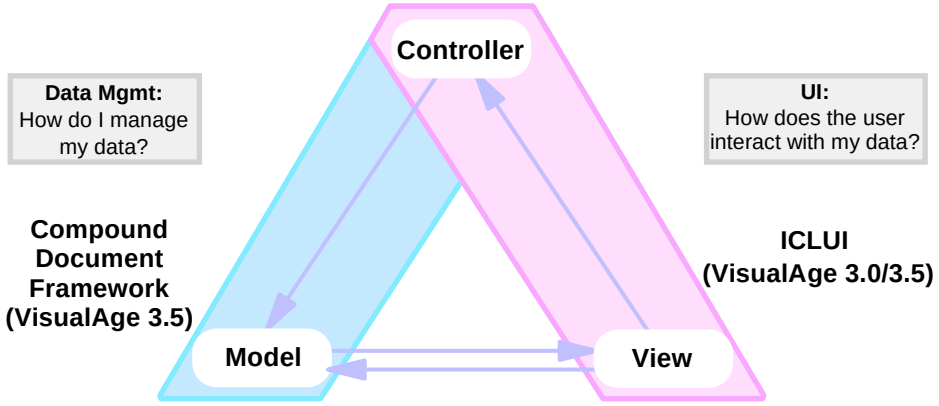
Programmers in Smalltalk create their GUI objects by subclassing and customizing base classes for the model, view, and controller abstractions provided in the Smalltalk class library. The Smalltalk

programmer benefits by inheriting from a tight relationship predefined among the three core MVC concepts within a single GUI object. More complex GUI objects such as a dialog box with multiple elements are then composed of multiple instances of different kinds GUI objects like those described above, all represented by MVC classes. Ultimately a whole interactive graphical application is built up using MVC.

Building the Taligent / Open Class Programming Model

In the Open Class[2] class libraries for IBM's VisualAge programming environments, Taligent has adapted and generalized the MVC programming model of Smalltalk to represent the overall structure of any interactive program. This evolution began in Taligent's original CommonPoint application system[3] and has continued through current and forthcoming versions of Open Class for VisualAge.

Taligent's overall approach has been to decompose the basic MVC concept into its constituent parts and further refine them to assist programmers in developing more complex applications[4]. The first step in this process has been to formalize the separation between the Model and the View-Controller, which we refer to as a Presentation. This represents the breaking down of a programming problem into two fundamental concepts: Data Management and User Interface. The two concepts embody two most basic design questions the programmer must address.



For Data Management, the question the programmer answers is "How do I manage my data?" This is more than just the underlying data representation within a model but also what data structures, access methods, change protocols, persistence, sharability, distributability, etc. are employeed.

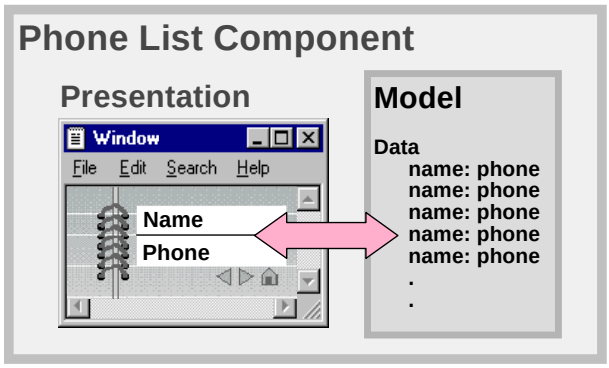
For User Interface, the question the programmer answers is "How does the user interact with my data?". This is more than just the drawing of the object on the screen and the mouse and keyboard events but also what semantic operations are enabled, what user actions are recognized, what gesture language is employed, and what feedback is given.

In the previous release of IBM's Open Class for VisualAge 3.0, IBM's ICLUI graphical user interface class library[5] provided tools which programmers use to represent the User Interface aspect of their program. With the 3.5 release, Taligent introduced into Open Class the first version of its Compound Document Framework to represent the Data Management aspect of a program. We will see how in the forthcoming 4.0 release of Open Class, both these aspects will be further generalized and refined.

Models enable encapsulation

There are several benefits to generalizing the model concept. The first is that it enables a clean separation of the Model and the Presentation. Suppose within my program I wish to implement a Phone List

Component. My model would encapsulate data for all my name and phone number pairs, as well as all the methods by which I could query and change them. My presentation could be an interactive phone book representation of the phone list data.

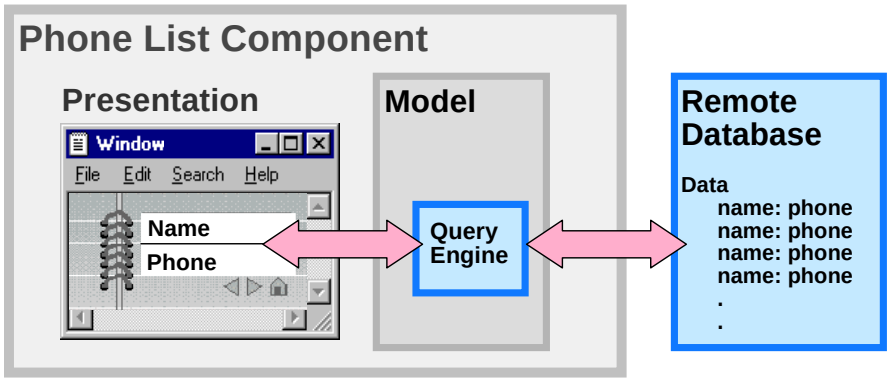


By having a clean separation and encapsulation, multiple benefits accrue. Over time I could change the underlying data structures in my model, for example, a linked list for more rapid insertion, a hashed or indexed table for better alphabetic searches, added fields such as mailing address for use by other programs, and so forth, and still reuse the same presentation code without modification. I could create multiple presentations without reimplementing the underlying model, for example, a simple phone book, a phone book with alphabetic tabs, a phone book with a search capability, a administrator's phone book that could add, delete, and change the information, or a phone book that could dial my phone. I could have different programmers work in parallel on models and on presentations and have them come together and work in different configurations.

These benefits may appear obvious, but surprising numbers of programs do not employ such clear decomposition or interfaces. Many a program is written such that if you change the underlying data structure, code has to be changed in numerous places where the data is retrieved, displayed, stored, transformed, etc. This isn't necessarily harmful in small programs and programs written by one person who can keep it all straight. But this style, often the way many programmers learn to program, does not produce programs that readily scale up, can be modified or enhanced, can be reused in whole or in part in other programs, or can be worked on by multiple programmers at a time or over time.

Models enable persistence

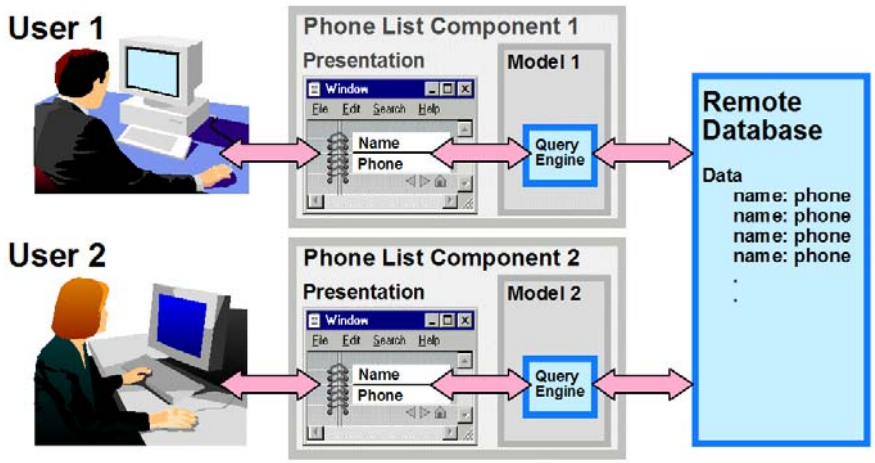
In the previous example, the data of the model was stored in the model itself, that is, a data structure was allocated within the model class for storing the data in memory during execution. This isn't always the case. How the model actually stores its data is up to the model. In the first example, the model might contain no data at all but rather a proxy for data accessed from a persistent data store. Or the model



could contain code for a query engine that goes to a remote database storing the name, phone number pairs. The model could implement access control and authentication (log-in), accounting or charging mechanisms, cache local copies for performance, use databases from different vendors, provide database replication and redundancy, etc. All this is achieved transparently to the presentation and can be changed over time or for different installations

Models enable sharing

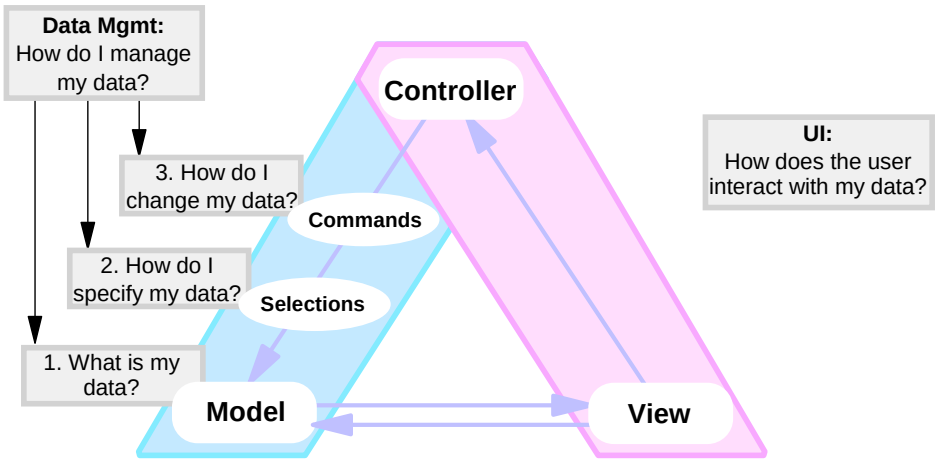
Abstracting out the model also permits greater flexibility of use among multiple users. Having different program models encapsulate the same remote data allows multiple users to share the same data, as with a company phone book. If a new phone book entry is made in the shared data model, everyone is always current, whereas in the case where models store their own data, everyone must be updated



individually. This approach also allows different users to collaborate simultaneously on the data and see the same information at the same time. Different users could have different presentations, such as users with a read-only phone book, an administrator with the ability to change a phone number, and a human resources manager able to access private information.

Three Data Management Questions

Based on broad experience developing programs with Data Management and User Interface (Model-Presentation) separation, Taligent has identified key additional refinements to the Data Management half of the problem. These refinements decompose the Data Management question into three subquestions.



The first question is "What is my data?". This is the model proper, equivalent to the basic Smalltalk model, with its encapsulated data, read and write access methods, etc.

The second question is "How do I specify my data?" The abstractions for specifying different subsets in my model's data we call selections.

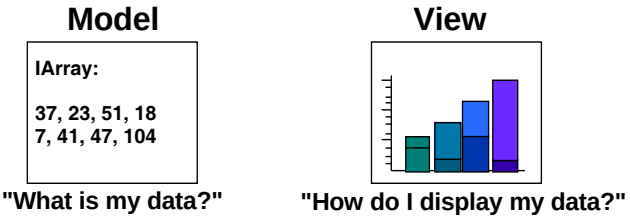
The third question is "How do I change my data?" The abstractions for representing the operations I can perform on selections in my model we call commands.

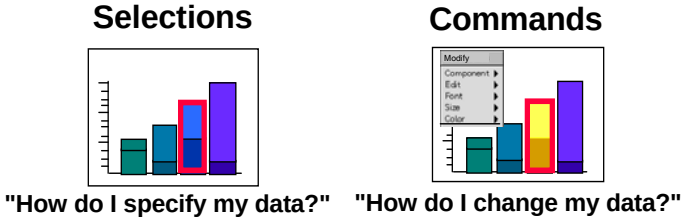
Model, View, Selections, Commands

Let's look at a simple example demonstrating these concepts.

Suppose our model is a two-dimensional array of integers.

A view for this model might be a stacked bar chart. There might be more than one kind of view of the same model, for example, a multi-line graph or a spreadsheet table would be other kinds of views of this model. Views don't need to be graphical, for example, other kinds of views could play notes on a piano keyboard, or specify valve settings in a factory. There might not be any view, for example, a server provides essentially a view-less model.





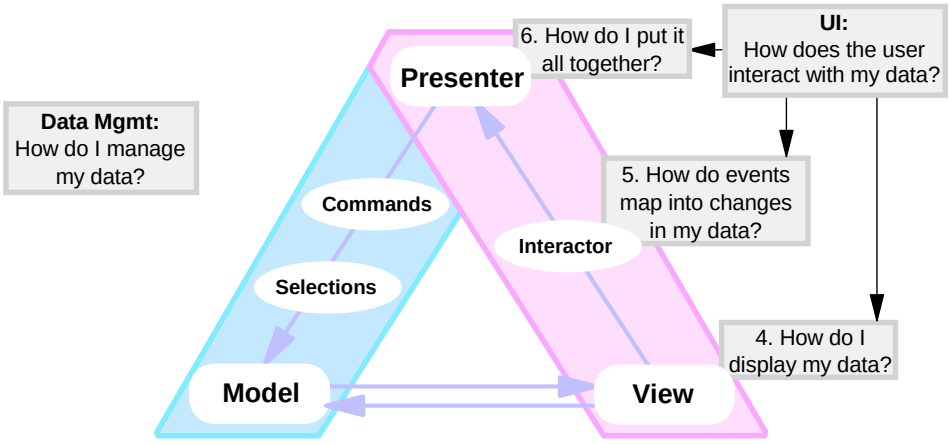
A selection in my model might be to pick a column of elements in my data. A row of data, a single element, or selecting all of my data would each be different kinds of selections. In a text model, a selection is what I get when I drag my mouse over some of my text. It might be a single character, a word, a sentence, or the entire body of text. A selection could also consist of multiple discontiguous segments. We even consider an insertion point to be a zero-length selection.

A command then defines an operation on a selection in my model. For example, I might change the color, pattern, or other appearance of my bar chart selection. Other commands might be to delete, move, copy, change the values, save, or print my selection. In a text model, there would be different commands to change the font, style, and size, or to cut, copy, and paste, or to check the spelling, etc. Each of my menu items and operations behind various buttons, gestures, and keyboard equivalents would be different commands, capable of operating on different selections (and perhaps only certain types of selections) in my model.

Intuitively, these abstractions can easily represent a very broad variety of interactive applications we all use every day.

Three User Interface Questions

Now let's focus on the interactive side of my application. How do I design the interactive user interface in my application? We break this into three more questions:



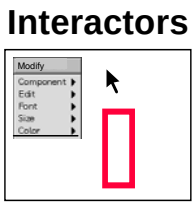
The first question is "How do I display my data?" This is the view as we've described it above. There may be multiple views and views don't have to be visual.

The second question is "How do events map into changes in my data?" We call these interactors. By interactor events, we mean the user-initiated actions like mouse movements and clicks, keyboard keystrokes, or operations like flipping a switch, changing a dial, or inserting a disk. In most interactive computers, there are a particularly rich set of mouse interactors like menu selections, drag and drop, button and check box clicks, and drawing operations, as well as extensions into pen, handwriting, and voice input.

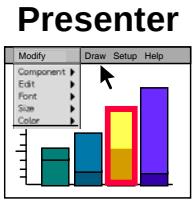
The third question is "How do I put it all together?" This represents the function of the classic Smalltalk controller, but elevated to an application level and taking into account the intermediate selection, command, and interactor concepts. To capture this distinction we refer to this kind of controller as a presenter. As a result, we refer to the overall resulting programming model as Model-View-Presenter or MVP, acknowledging its origins as a generalized form of MVC. The role of the presenter within MVP is to interpret the events and gestures initiated by the user and provide the business logic that maps them onto the appropriate commands for manipulating the model in the intended fashion.

Interactors and Presenters

Completing our previous bar-chart example, we see that the interactors account for the mouse tracking and events, specification of selections, menu picking, and keyboard equivalents.



"How do events map onto changes to my data?"



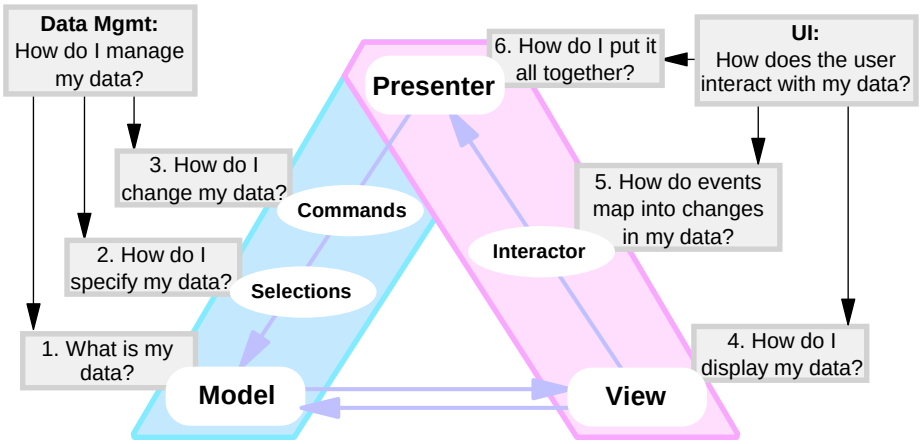
"How do I put it all together?"

The presenter then represents the traditional "main" or "event loop" part of the application, creating the appropriate models, selections, commands, views, and interactors, and providing the business logic that directs what happens when, like a traffic cop or orchestra conductor.

MVP: Model-View-Presenter

In summary, the three data management questions,

1) What is my data? (Model)
2) How do I specify my data? (Selections)
3) How do I change my data? (Commands)



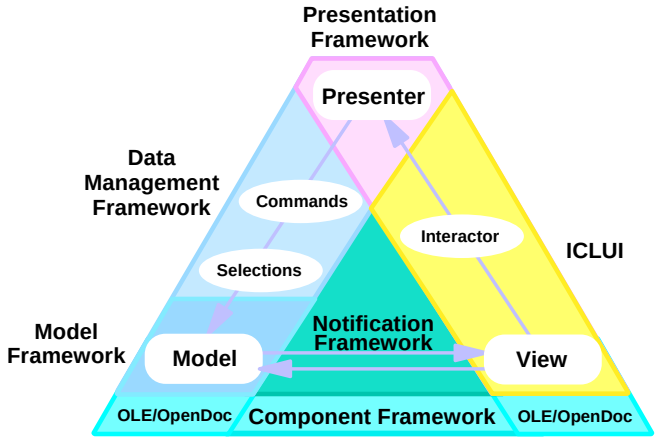
and the three user interface questions,

4) How do I display my data? (View)
5) How do events map into changes in my data? (Interactor)
6) How do I put it all together? (Presenter)

comprise the six questions a developer answers in creating a full MVP-based program.

Programming Model Frameworks

In Taligent's evolving development of the MVP programming model, designs and implementations of these abstractions are provided in object-oriented frameworks, to be subclassed and customized by developers in creating their particular versions of these elements for their application[4]. The Taligent frameworks may be used separately but are built to work together to provide an overall application programming model whose design and implementation are reusable by developers.



In the next release of Open Class for IBM VisualAge 4.0, Taligent will provide a Model Framework to aid developers in creating models. Taligent will also provide a Data Management Framework to aid developers in creating selections and commands. Selections and commands are often designed together, and at present having a single framework to help implement these abstractions appears to be the appropriate design choice.

While the MVP programming model can be used to create stand-alone applications, increasingly today's software involves creating components, or smaller program units that can be combined to work together to create full applications. In this case, the MVP model rests on top of a component architecture, so that the models, views, and events map to an underlying component runtime. Taligent provides a single Component Framework API, with implementations that map to OpenDoc[6] on OpenDoc platforms such as OS/2 or AIX, that map to OLE[6] on OLE platforms such as Windows, and will in the near future map to Java Beans[7], the emerging component standard for Java platforms. (In Taligent's original CommonPoint system[3], Taligent had its own component model to which this API was mapped. However, the CommonPoint component runtime has been abandoned and all versions of Taligent technology in Open Class now require and map to a native component model runtime such as OpenDoc or OLE.)

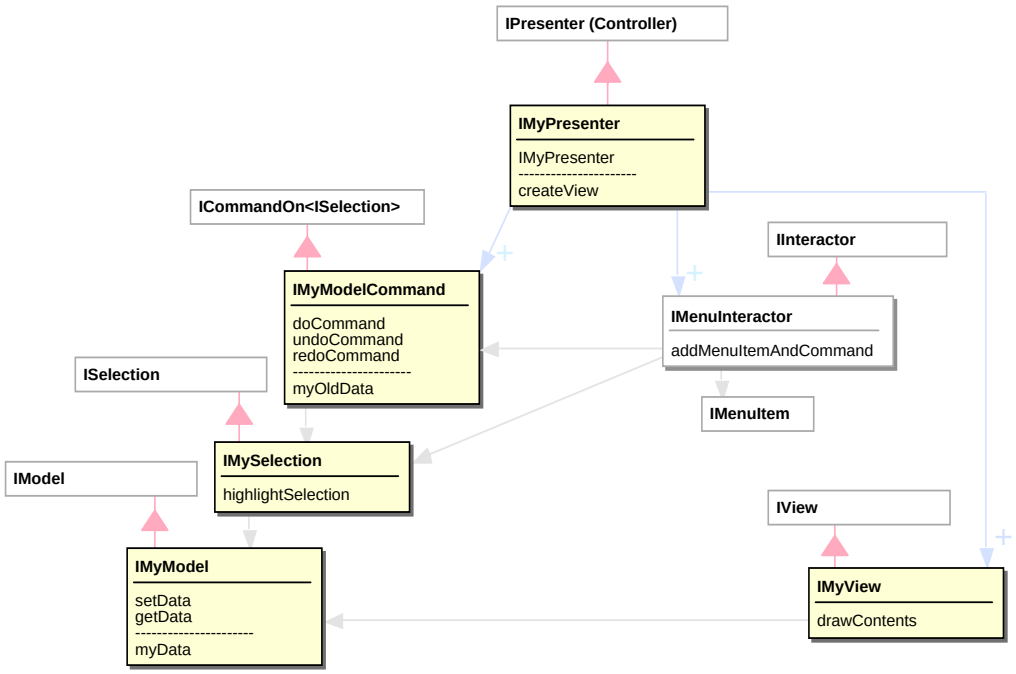
The Taligent Compound Document Framework that appeared in CommonPoint had the combined functionality of the Data Management Framework, the Model Framework, and the Component Framework. The Compound Document Framework that appeared in Open Class for VisualAge 3.5 had the functionality of the Model Framework and the Component Framework, but not the Data Management Framework, which will be reintroduced in Open Class for VisualAge 4.0.

The Taligent Presentation Framework that appeared in CommonPoint represented a combination of the presenter as well as the interactor and view abstractions, based on CommonPoint's own user interface library. Open Class for VisualAge 4.0 will use IBM's ICLUI user interface class libraries for the interactor and view abstractions, so that in Open Class for VisualAge 4.0 the Presentation Framework is distilled to just the presenter abstraction.

Finally, the event processing system underlying the MVP programming model is embodied in a Notification Framework, which provides for interest-based notification of events, senders and receivers of events, type checking of events, distribution of events, etc. to mediate all the interactions among the programming model abstractions.

Programming Model Classes

A class diagram for the MVP programming model reveals the direct correspondence of the class structure and the concepts which we have been discussing[4]. This diagram represents a simplified but illustrative overview of these relationships.



Notice that there are base classes for each of the MVP concepts: IModel, IView, ISelection, ICommand, IInteractor, and IPresenter. Developers create their own particular versions of these abstractions by subclassing these base classes and overriding their methods as appropriate to implement the desired functionality.

In its simplest form, IMyModel would define myData as a private data member and override IModel's getData and setData methods to read and write myData.

IMyView would override IView's drawContents method to implement the drawing of myData (obtained using a getData method call) on the screen.

IMySelection would define a simple selection of myData and provide an implementation of highlightSelection to cause the selection to be evident on the screen.

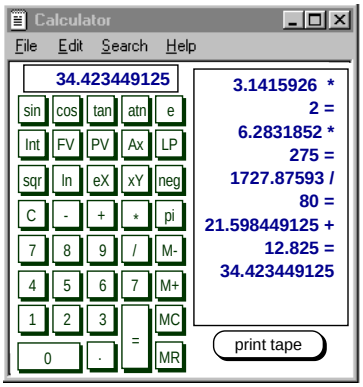
Separate subclasses of ICommand would be created for each of the desired commands, for example, cut, copy, and paste commands and/or clear and enter commands. In more complete implementations, each command has methods for doCommand, undoCommand, and redoCommand to allow the system to provide for multiple levels of undo and redo of commands.

Most ordinary applications do not need to create their own interactors. The Open Class class library comes with a rich set of predefined IInteractor subclasses for typical user interface elements such as an IMenuInteractor or an IButtonInteractor. Most applications simply instantiate an existing subclass like IMenuInteractor, instantiate IMenuItem objects in specific order with particular labels, and request that corresponding commands be invoked on the current selection. Most of these relationships won't even be set up programmatically, but rather with a graphical user interface building tool, which instantiates and defines the proper object.

Finally, developers subclass their IMyPresenter from IPresenter, which represents the top-level of an application framework comparable to that of MFC, OWL, or MacApp. The IMyPresenter creates the particular model, view, selections, commands, and interactors, wires up them all together, and thereby creates the functioning application.

Building An Application

The power of the MVP programming model is not only that it presents an intuitive set of abstractions, but also that it provides rich default implementations with added functionality around the developer's code. Let's use the simple example of a calculator application to illustrate these benefits. This example also illustrates that the MVP concepts need not all be learned at once. Rather one can start with a more traditional programming style and introduce MVP abstractions step-by-step as the application is evolved.



First, one can write a calculator application without using the presenter abstraction at all but rather by hand-coding a traditional main routine and event-loop. A certain number of hard-core programmers prefer to "roll their own" in this way. However, developers familiar with OO tools like MFC find it easier just to subclass a framework like the presenter and inherit a main and event-loop from it. A good presenter framework then implements for the developer the application launch and quit functions, basic menu bar and overlapping windows, perhaps a help capability, and basic mouse and keyboard events. It is possible to write the whole calculator application within the methods of the presenter, making raw graphics calls to draw the buttons, low-level mouse tracking to implement button pressing, and direct redrawing to implement calculator updates. All the other MVP abstractions could be ignored.

However, if the developer subclasses IView to write the graphics calls for redrawing the calculator, IView will take care of redrawing on window resizing and exposure, will provide for default printing, and will work with a graphical user interface builder to compose or change the particular calculator layout.

With just a presenter and a view, the developer would just store the raw calculator registers as data members within the view. These would have a lifetime of just the calculator's execution, and results would be lost when the calculator quit. If, however, the developer subclasses IModel to hold the calculator data, the model can be used to represent a calculator "document", similar to a calculator paper tape. This document could be created new, record the calculated values, be saved in a file and closed, then opened and read back in. In addition because of Taligent's tie between the Model Framework and the Component Framework, the whole calculator could be an OpenDoc or OLE component and be embedded in other OpenDoc or OLE documents, respectively.

So far, operations on the model are direct calls on its set and get methods. Suppose we now implement selections. With this addition, it becomes possible to highlight individual data elements in the paper tape. The Data Management Framework could now provide default cut, copy, and paste commands for the selected data. The selection (not just the whole document) could be embedded or linked via OpenDoc or OLE into another document, perhaps providing values to drive fields in a form. In addition, a selection for an insertion point could now enable embedding of OpenDoc or OLE components into the paper tape.

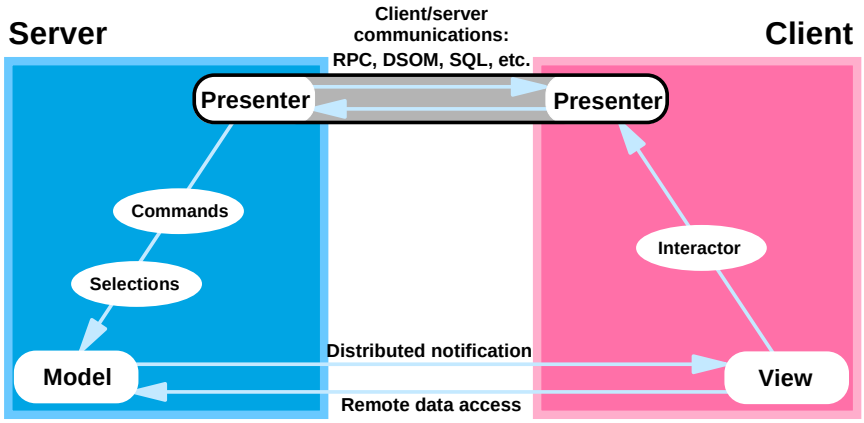
To this point, the actual operations of keying in numbers and arithmetic operators are entirely provided by the developer. If these were now implemented with commands, the calculator operations would be undoable and redoable, thus the paper tape could run backwards. The commands could be scripted, that is, stored away to be played back later, with pauses to allow for user input. Further, commands could be distributed across the network to another machine running the same calculator program, allowing two or more people to collaborate in real-time. Anyone could hit the keys and everyone would see the same results.

Finally, if the developer started instantiating or even creating more interactors, many more capabilities could be added. Keyboard equivalents could enable use of the keyboard's numeric keypad. Additional buttons and functions could implement scientific or financial calculators. Menu items could convert the calculator base from decimal to hexadecimal or octal. Other menu items could change the locale of the calculator, enabling different numeric or currency systems (using Taligent's International Frameworks[8], this could even be done with no additional coding for the number or currency input or display). Finally, by creating more specialized interactors, ambitious developers could support pen input and handwritten number entries.

All this might seem like overkill for a simple calculator, but imagine the power these kinds of capabilities would provide in creating a highly interactive, document-centric, embeddable, linkable, multilevel undoable, scriptable, real-time collaborating, internationalized version of your favorite business application. All these capabilities and more are possible by deriving your application from the MVP programming model abstractions.

Client/Server

We have one more topic to discuss, namely, how one creates client/server applications using the MVP model. Client/server versions of applications involve deciding which of the MVP abstractions are implemented in whole or in part on the client or the server[4]. The most traditional client/server split would be made by factoring the presenter. The model, selections, and commands represent typical server-side functionality. The view and interactor represent typical client-side functionality. The presenter then "straddles" the boundary between client and server, that is, code appears on both sides representing a single conceptual presenter. Most of the presenter code could be on the client side in a "fat client", talking SQL across the wire to a simple server-side presenter. Conversely, most of the presenter code could be on the server side, with only a "thin" GUI application on the client-side. The nature of the split is determined in part by what protocol the developer chooses to use across the boundary, examples being RPC, DCE, SQL, CORBA, etc.[6]



Notice that in more elaborate applications, we may have some or all the MVP abstractions on both sides. The client application may have a model "surrogate" or "proxy" talking to a remote data server, as in the "Models enable persistence" example. The remote data server, also implemented with MVP, would store the real data, or itself go off to a third tier. Similarly, the server application may be just a view-less application posting update events to the real view on the client side. The key idea is that any or all of MVP is available on both client and server for implementing the desired client/server architecture.

Multiple versions of client/server

Indeed, factoring the MVP abstractions in different places and placing the predominant functionality on either the client side, the server side, or split across seems to correspond to a broad range of popular client/server solutions.

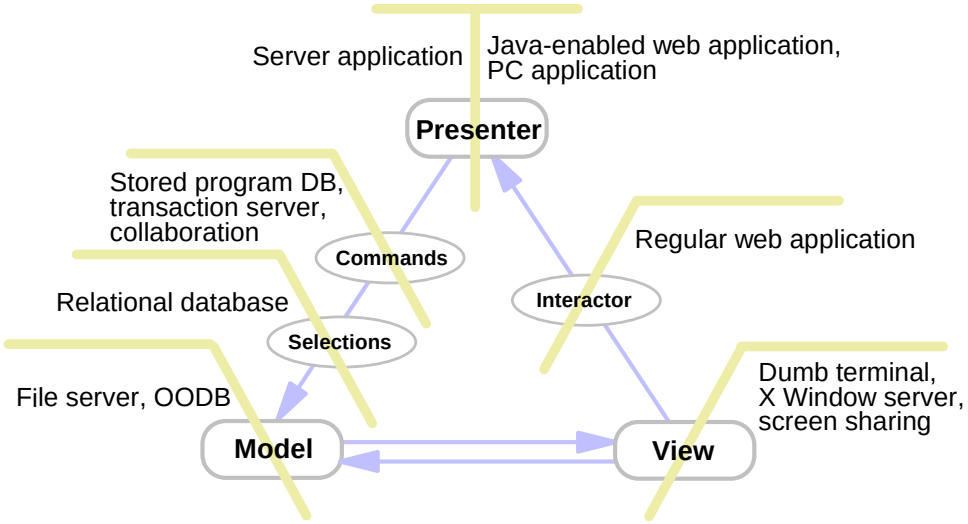
Suppose only the model is on the server and everything else is a "fat client". This corresponds to having a remote data store or file server, or similarly an OODB, which abstracts only the reading and writing of data (models).

Suppose the model and the selections are on the server. This corresponds to what a basic relational data base does. The "SELECT" command of SQL is, indeed, a central primitive of relational data bases, specifying what data is to be chosen for processing, but with the processing (commands) usually performed on the client side.

Suppose the model, selections, and commands are on the server. By moving the command processing to the server, we have modeled essentially a stored program data base, or at a different level, a transaction server. Now the client simply requests commands for processing the data that execute on the server.

Let's go to the other extreme. Suppose only the view is on the client. At its simplest, this is essentially what a dumb terminal does, or what an X Window server does in allowing remote viewing, or what simple screen sharing is. The client is only a view and everything else is happening on the server.

Suppose the view and the interactor are on the client, but all the data management and business logic (presenter) is on the server. This is essentially what today's ordinary web applications do. The view is the HTML page rendered in a browser, and the interactor is using the mouse to click on HTML links or form-based entry fields and buttons. But all execution, business logic, commands, and data are on the Web (HTTP) server.



A Java-enabled Web application, where some code is downloaded for execution on the client, is essentially the beginning of migrating business logic, hence some of the presenter, to the client side. If all the logic stays on the server side, we have the classic server-based application. If all the logic goes on the client side, we have the classic PC application. If it is split, we have the classic distributed application. If we split twice in the right places, we have the classic 3-tier application: thin GUI client, mid-tier application logic, backend database.

We now see that for programming distributed applications, the seventh and final design question is "How do I partition my application between client and server?"

As described above, all the analogies are relatively superficial, and serious client/server and multi-tier applications typically have much more sophisticated relationships between abstractions on the different levels. But it is clear that a broad range of popular and familiar client/server architectures can be modeled by simple variations within the MVP model. This serves to give us heightened confidence that the MVP model captures a fundamental and expressive set of abstractions common to most prevailing application architectures. For these reasons, we believe the MVP programming model is a fundamental design pattern for applications development.

Benefits of Abstractions

We have demonstrated that the MVP concepts are sufficient to model many kinds of applications. It is also worth examining whether all the distinctions are necessary and justified by the benefits they bring. To this end, for each pair of adjacent abstractions within the MVP triangle, we can ask "is there value in making this distinction?"

The Model/View distinction brings the benefit of View Independence. For example, different calculator layouts (different levels of function, different arrangements, different "brands" of calculators) can all be programmed as different views on the same underlying calculator engine and data representation.

The Selection/Model distinction provides the benefit of Model Independence. Model Independence allows developers to change and evolve data structures or file formats without changing how the data is displayed or processed in the rest of the program, as well as introduce persistence, remote data bases, and sharing.

The Command/Selection distinction provides the benefit of Command Reuse. Single commands can apply to selections of one, many, or all data elements in a model, rather than having a special case for each of these in the programmer's code. Once implemented, commands can be reused in multiple applications by

different presenters, for example, base or currency conversion commands can be applied to numbers in any document. In this way the CommonPoint system supported global tools[3], where any pen could write on any document, any font selection tool could change any text, any color picker could change any color, any spelling checker could check any text, etc.

The Interactor/View distinction provides the benefit of Input Generality. Without changing my application logic or the rendering of my data, I can support different menus, dialogs, keyboard equivalents, gestures, or handwriting/pen input.

Distinguishing the Presenter from Commands and Interactors provides the benefit of Reusable Logic. When abstracted from specific display or data management details, program logic and algorithms can be reused in different applications. For example, a scientific calculation like unit conversions or a financial calculation like compound interest can be reused in different contexts.

Finally, an overall benefit of using frameworks to implement the various MVP abstractions is that they facilitate portability across platforms, multiple standards (eg, OpenDoc, OLE, ...), distribution, and multi-tier partitioning.

Summary

Taligent and IBM are developing the Model-View-Presenter (MVP) programming model across multiple development environments. Taligent's implementation of Open Class for VisualAge 4.0 will support MVP for C++ on multiple OS platforms. Taligent is also implementing a very lightweight version of MVP which will run in any standard Java environment. MVP is essentially a generalization of the classic MVC programming model of Smalltalk. Thus, the development of MVP for IBM's primary development environments provides a unified conceptual programming model across today's most popular object-oriented programming languages.

References

1. Steve Burbeck (1992). "Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC). Available at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

2. IBM Corp. (1996). Open Class Library User's Guide, Version 3.5. IBM Canada Ltd. Laboratory, North York, Ontario, Canada.

3. Sean Cotter & Mike Potel (1995). Inside Taligent Technology. Addison-Wesley, Reading MA.

4. Christina Warren (1996). "Taligent Programming Model IPO." Taligent internal technical specification.

5. Kevin Leong, William Law, Bob Love, Hiroshi Tsuji, & Bruce Olson (1994). OS/2 C++ Class Library: Power GUI Programming with CSet++. Wiley, NY.

6. Robert Orfali, Dan Harkey, & Jeri Edwards (1996). The Essential Distributed Objects Survival Guide. Wiley, NY.

7. JavaSoft, Inc. (1996). "Java Beans: A component architecture for Java." Available at http://splash.javasoft.com/beans/WhitePaper.html

8. Mark Davis, Jack Grimes, & Deborah Knoles (1996). "Creating global software: Text handling and localization in Taligent's CommonPoint application system. IBM Systems Journal, vol 35, no 2.

The Model View Presenter ("MVP") software pattern originated in the early 1990's at Taligent, a joint venture of Apple, IBM, and HP, and was the underlying programming model for application development in Taligent's C++-based CommonPoint environment. The pattern was later migrated by Taligent to Java and popularized in a paper by Taligent CTO Mike Potel [[http://www.wildcrest.com/Potel/Portfolio/mvp.pdf "MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java." Mike Potel]] . MVP stands for Model View Presenter and was invented by Martin Fouler at Microsoft in the 1990â€™s (although there are some claims that this was actually invented even before; donâ€™t they always?). In the same way as MVC, MVP uses several components that are familiar to the Swift developers, to begin with the View. View (or actually) ViewController.Â The presenter is an abstract entity that contains the business logic of the application, becoming completely agnostic of the actual service implementation. The MVP pattern allows us to decouple the ViewController from the Presenter, and then again the presenter from the services. The ViewController will have to instance the services and the Presenter, and inject the service into the presenter. M (1996) MVP: Model-View-Presenterâ€”the Taligent programming model for C++ and Java. http://www.wildest.com/Potel/Portfolio/mvp.pdf. Desktop Framework Concepts (1995) http://www.cubik.org/mirrors. Jan 1995. Taligent Online.Â We have used Presenter First on projects ranging in size from several to a hundred MVP triads. This paper describes MVP creation, composition, scaling, and the tools and process we use. An example C# application illustrates the application of the Presenter First technique.Â Model-View-Presenter (MVP) is a pattern which aimed at providing a cleaner separation between the View, the Model and the Presenter. The paper advances an architecture model of MVP pattern on .NET platform and a formal method of how to implement it.