On User Interface Architectures and Implementation

Tero Kukola

University of Tampere
Department of Computer Sciences
Master’s Thesis
Supervisor: Roope Raisamo
25.5.2008
Abstract
The definition of MVC model has become distorted. Many MVC adaptations use a mediating controller between model and view layers, which is not part of the original MVC/80 model. While the separation of model and view has benefits, the mediating controller leads to excessive redundancy in code and should be avoided. Removing the mediating controller simplifies UI architectures. This simplification can be continued further by adopting dynamic features and ultimately by adopting dynamic languages.
# Table of Contents

1 Introduction .................................................. 1

2 MVC .......................................................... 3
   2.1 MVC/79 .................................................... 3
   2.2 MVC/80 ..................................................... 6
   2.3 MVC/80 Controller Is a Strategy for Input Handling .... 10
   2.4 MVC Is Unimplementable? .............................. 11

3 Mediating Controller .......................................... 13
   3.1 PAC ....................................................... 13
   3.2 MVC++ .................................................... 15
   3.3 Role of the Mediating Controller ....................... 18
   3.4 Mediating Controller Models a Process Recursively .... 19
   3.5 Problems of the Mediating Controller .................. 20
   3.6 Mediating Controller Should Be Avoided ............... 23

4 Model/View Separation ....................................... 24
   4.1 Encapsulation ............................................ 25
   4.2 Mapping Data ............................................ 26
   4.3 Direct Manipulation ..................................... 27
   4.4 Presentation Model ...................................... 28
   4.5 Visual Proxy ............................................. 29
   4.6 Naked Objects .......................................... 30
   4.7 Model/View Separation Is Sometimes Worth Doing .... 31

5 Dynamic Languages ........................................... 32
   5.1 Generic Messaging and Data Passing ..................... 32
   5.2 Using Multiple Languages ................................ 35
   5.3 Resistance to Adoption ................................... 36
   5.4 Choosing a Dynamic Language ............................ 38
   5.5 Design Patterns ......................................... 40
   5.6 Lexical Closure .......................................... 41
   5.7 Domain Specific Languages ............................... 43

6 Conclusion .................................................... 46

References ....................................................... 48
1 INTRODUCTION

Model-View-Controller (MVC) is the most well known model, or architectural pattern, for UI application design and implementation. The first widely known MVC definition is MVC/80 for the Smalltalk-80 environment [Krasner & Pope, 1988]. The main characteristic of MVC is the separation of the model from the view.

MVC is arguably also the most misunderstood UI model. Trying to learn MVC can lead confusion as many MVC adaptations provide conflicting definitions of MVC. Much of the misunderstanding can be traced back to the original MVC/80 as it is not directly applicable with modern UI toolkits. Namely, the MVC/80 controller is not relevant because its role is embedded into UI widgets.

Many MVC adaptations reinvent the controller and add a mediating controller between model and view layers. The MVC/80 controller is a strategy for input handling, not a mediator for decoupling models and views. In this sense the mediating MVC adaptations are wrongly named and are closer to models that use a mediating controller as part of their design, for example PAC and MVC++.

The use of mediating controller should be avoided, because the added implementation complexity and redundancy outweighs the benefits. Because a mediating controller is often a stateless router between just two objects it can be considered a misapplication of the mediator pattern.

Model/view separation has benefits, but similarly to the mediating controller, the separation increases implementation complexity. The model/view separation can be counter-productive for some application domains, including direct manipulated applications.

In addition to avoiding the mediating controller, UI implementations can be simplified further by using generic messaging and data passing. This allows removing observer interfaces and allows extending the messaging without needing to change class interfaces. Both features emulate dynamic language features and are trivially implemented with dynamic languages. Using dynamic languages allows simplifying or removing many design patterns present in C++ and Java implementations.

Discussion related to reusability is mostly avoided. While reusability is a desirable goal, it is rarely achieved in reality. Firstly, reuse is often hampered by architecture and interface mismatches [Garlan et al., 1995]. Secondly, the variables that chiefly influence reuse success are not technical [Morisio, 2002]. Thirdly, creating reusable components and frameworks increase the complexity
and cost of an implementation [Brooks, 1975]. Fourthly, perhaps there are not that many software components that can be reused [Glass, 1998].

Dynamically typed programming languages are often called scripting languages [Ousterhout, 1998]. Here the term dynamic language is preferred to scripting language, as it characterizes such languages better [Ascher, 2004]. Here system level programming languages are C, C++, Java, C#. Although Java and C# have gained and will be gaining dynamic language features, they are considered here system level programming languages.

The goal of cross-platform development is often raised when discussing UI applications written in system languages. Especially the model layer in applications is recommended to be written in a cross-platform manner. This approach allows porting the application to new platforms by rewriting the UI layer only. Here cross-platform related issues are mostly avoided. If cross-platform functionality is needed, using a cross-platform UI toolkit should be preferred.
2 MVC

Contrary to a common perception the Model-View-Controller is not one model, it actually refers to two different models and their adaptations. The models are:

- **MVC/79**, the original MVC proposed by Trygve Reenskaug ¹.

- **MVC/80**, the Smalltalk-80 implementation of MVC.

- **MVC adaptations**, the MVC/80 derived models used outside Smalltalk-80 context.

The differences between the models are large enough to treat them separately. Although there is variation between different MVC adaptations they are considered here a single class of MVC models.

2.1 MVC/79

MVC/79 was devised by Trygve Reenskaug in the late 1970s at Xerox PARC Smalltalk group. The model was based on Reenskaug’s experiences in Norwegian shipyard information systems. According to Reenskaug [2007], the purpose of MVC was to bridge the gap between the user’s mental model and the application domain model; the user should have the illusion of directly manipulating the domain model.

MVC/79 took its initial form as Thing-Model-View-Editor metaphor. Reenskaug [1979a] defines the parts of the model as follows:

**Thing** Something that is of interest to the user. It could be concrete, like a house or an integrated circuit. It could be abstract, like a new idea or opinions about a paper. It could be a whole, like a computer, or a part, like a circuit element.

**Model** A model is an active representation of an abstraction in the form of data in a computing system.

**View** To any given model there is attached one or more views. Each view is capable of showing one or more pictorial representations of the model on the screen and on hardcopy. A view is also able to perform such operations upon the model that are reasonably associated with that view.

¹ Reenskaug uses MVC/79 in his later works [2003] and the same naming is adopted here.
**Editor** An editor is an interface between a user and one or more views. It provides the user with a suitable command system. For example in the form of menus that may change dynamically according to the current context. It provides the views with the necessary coordination and command messages.

The model part can consist of multiple nesting models. A model is decoupled from the view and contains no presentation information. While the real-world application domain, the thing, is considered a part of the model, it is never a part of an implementation as it is abstracted by the model. Hence the Thing-Model-View-Editor could be reduced to Model-View-Editor.

The Thing-Model-View-Editor paradigm evolved to Model-View-Controller paradigm, which keeps the definition of the model and the view, removes the 'thing' and adds a controller. Reenskaug [1979b] describes the controller as follows:

“A controller is the link between a user and the system. It provides the user with input by arranging relevant views to present themselves in appropriate places on the screen. It provides means for user output ² by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and passes these messages to one or more of the views.”

Reenskaug further defines that a controller never supplements views. In other words, a controller does not participate in drawing of a view. All input handling is contained in controllers, views are not aware of user input.

The introduction of the controller redefines the editor [Reenskaug, 1979b]:

“A controller is connected to all of its views, which are called the parts of the controller. Some views provide a special controller, an editor, which permits the user to modify the information that is presented by the view. Such editors may be spliced into the path between the controller and its view, and will act as an extension to the controller. Once the editing process is completed, the editor is removed from the path and discarded.”

---

² Reenskaug’s viewpoint is from the user. Typically the viewpoint of an implementation is the reverse, user provides input and is given output.
Figure 2.1 The MVC/79 model [Reenskaug, 1979b]. Note that one controller can refer to multiple views and multiple views can refer to the same model.

Reenskaug further adds that an editor is closely associated with the view and acquired from the view.

Considering the above descriptions and definitions we can conclude the following:

- Models are not aware of controllers or views.
- Views are not aware of user input.
- Views edit the model.
- Controllers create and arrange the views.
- One controller can control multiple views.
- Controllers handle user input and send appropriate messages to the views.
- Editors are specialized controllers that provide a view specific command interface for model manipulation.

Figure 2.1 presents one possible interpretation of the MVC/79 model.

The description of MVC/79 leaves room for ambiguities. Firstly, the role of editors and views in model manipulation is not clear. Both model definitions above allow views to perform operations on models. In the foreword to [Pawson, 2004] Reenskaug states that the view was an editor that enabled the user to inspect and modify the model. Views are clearly allowed to manipulate the model. On the other hand, the editor definitions do not allow or deny model access. Considering that editors sit between views and the controller and provide
a command interface for the views, one could deduce that editors do not access model directly but rather apply commands via the view.

Secondly, the relationship between non-editor controllers and models is undefined. From the description of controller one might deduce that a controller communicates only with views. On the other hand, Reenskaug [2007] has developed the MVC/79 further by allowing a view to accept and handle user input relevant to it. This indicates the removal of separate editors. In addition, in [Reenskaug, 2007] the connection between controllers and models is explicitly allowed. The nature of this controller/model connection is still not defined.

Thirdly, MVC/79 does not consider recursive composition. The composition of views can be understood as the geometrical nesting of views or the formation of graphs of views. The role of controllers and editors in such composition is unclear as neither has the geometric properties which make view composition straightforward. In addition, as controller provides the means of user input, what is the nature of such means? If mouse is used, the means are most probably graphical, such as menus. Are such graphical means of input themselves modeled using the MVC/79 paradigm? According to Reenskaug's later work [2007], view and controller roles can be combined into a single object when they are tightly coupled.

Because MVC/79 was never implemented and documentation about it is scarce, the definition and details of MVC/79 remain vague. As a minor detail, in MVC/79 the order of components in the name Model-View-Controller makes sense. Some later MVC adaptations use a mediating controller between the model and the view. In the context of those models the order Model-Controller-View would be more appropriate. For some reason, the name Model-View-Controller tends to be used for all MVC adaptations regardless of the actual structure of the model.

2.2 MVC/80

The first MVC implementation was the MVC/80 for the Smalltalk-80 environment. In addition to the implementation of Model-View-Controller paradigm, MVC/80 refers to the Smalltalk-80 UI toolkit as a whole.

MVC/80 differs structurally from MVC/79, see figure 2.2. The main idea of MVC/80 is to separate input (controller), output (view) and information (model). MVC/80 model and view definitions are similar to the MVC/79 ones. In other words, the model is the domain model of the application and the view is a pre-
Figure 2.2 The MVC/80 model, reproduced from [Krasner & Pope, 1988]. Note that while the figure does not indicate a dependency from a view to its controller, such dependency exists as controllers are stored into the view as an instance variable.

In other words, controllers interface input devices and handle the user interaction towards models and views. A controller is an implementation of strategy pattern [Gamma et al., 1995] for input handling.

While MVC/80 can be considered a three part model, a triad, it is essentially two layer architecture, the model being the other layer and the UI being the
other. The UI layer is further refined by splitting it into two facets, the view for presentation and the controller for input handling.

In MVC/80 views and controllers come in pairs; there is one controller for each view and vice versa. In addition, one controller/view pair has exactly one model. One model can be interfaced by multiple controller/view pairs. Because multiple views can refer to the same model and models are not aware of views or controllers, a callback system to notify about model change was developed. The system is called *dependents* and it is similar to observer design pattern [Gamma et al., 1995]. For example, a view can enlist itself as model’s dependent and receive notification whenever the model changes. The dependent support is built into the Smalltalk root *Object*, hence any object can act as a model that supports dependents. The model can also be passive [Burbeck, 1992]. With passive models, the dependents mechanism is not used and the controller must synchronize the view when the model changes.

One MVC/80 user input cycle is as follows:

1. The user does something that produces input to the application.

2. A controller receives the input, determines its meaning and changes the model.

3. The model notifies about change to all connected dependents.

4. The dependent views receive the notification and refresh their state to match the model’s new state.

Views and controllers are usually developed in parallel. Views hold the controllers and typically create a default controller. Controllers are therefore installed to and queried from the views. While the connection from view to controller exists, the view is not expected to call its controller beside the view base class methods related to controllers.

MVC/80 provides base classes for models, views and controllers. Each view must have a controller; it is not optional. Controllers are an integral part of the Smalltalk-80 UI toolkit; sharing of input devices and scheduling of views are done by the hierarchy of controllers. Views that do not handle any input can use an instance of NoController, which is a controller that does nothing. As every view must have a controller the controllers form an implicit parallel hierarchy to the view hierarchy. In addition, the controllers must be correctly implemented,
because nonstandard flow of control can crash the Smalltalk-80 system [Burbeck, 1992]. From the perspective of modern desktop environments, such a fragile system seems unacceptable; a desktop environment and the underlying operating system should cope with ill-behaving applications.

MVC/80 seems to be biased towards widget implementation. That is, while the definition of MVC/80 [Krasner & Pope, 1988] defines the widget level structure, it neither defines the application level structure nor the composition of widgets in detail. This bias is also noted by Holub [1999a], although in Java context.

The following goals were behind the development of MVC/80 [Krasner & Pope, 1988]:

1. To create the special set of system components needed to support a highly interactive software development process.

2. To provide a general set of system components that make it possible for programmers to create portable interactive graphical applications easily.

The above can be interpreted as a desire to create a UI toolkit and considered as further proof of bias towards a widget level model.

The view composition examples by Krasner and Pope [1988] and by Burbeck [1992] use pluggable views. When a view is composed by using pluggable views as subviews, the subview input handling is not done by rewriting the subview’s controller. Rather, the pluggable view provides an interface for parameterizing actions related to the subview. Typically this allows setting blocks as action targets. For example, when a block is installed to a button, the block is executed whenever the button is pressed. Therefore, applications built from pluggable widgets do not implement custom controllers, but rather use the pluggable views’ default controllers. The composite view itself will use a default controller or a NoController as all input handling is being done in the blocks attached to the subviews. As a result, MVC/80 applications use two styles, Model-View-Controller for atomic widgets and the pluggable view style for composite views.

While MVC/80 is sometimes considered a pattern, it is best understood as an aggregate of different design patterns [C2c, 2008]. The first pattern is the separation of model and view; the second pattern is the observer implemented by dependents; the third pattern is the strategy implemented by controllers. The parameterization of pluggable views by attaching blocks can also be considered an implementation of strategy pattern.
2.3 MVC/80 Controller Is a Strategy for Input Handling

Application UIs are typically composed from existing widgets provided by a UI toolkit. The interaction with a certain widget tends to be fixed by the UI toolkit and the functionality corresponding to the MVC/80 controller is embedded into the widget.

Considering the above, what is the rationale behind the view/controller separation in MVC/80? Firstly, the purpose of a strategy pattern is to be able to change a run-time algorithm, possibly dynamically. The MVC controller follows the strategy pattern and allows changing the input handling of a view [Krasner & Pope, 1988]. It is also possible to use the same controller for different views. For example, a controller that provides a parameterizable menu can be used with multiple views.

Secondly, according to Lewis [1995] using a separate controller allows views and the controllers to inherit from different classes. This allows overcoming Smalltalk’s single-inheritance and allows structuring view and controller hierarchies differently.

Obviously, if there is no need for changing input handling, there is no need for separate controllers for input handling. In a typical mainstream UI toolkit:

- There is no separate controller, the functionality of MVC/80 view and controller are combined into the same widget.

- Input devices are not used directly.

- Scheduling and sharing of input is done behind the scenes by the UI toolkit.

In other words, widgets do not participate in the inner workings of the UI toolkit the same way MVC/80 controllers do.

As a result, the approach of modern UI toolkits is closer to MVC/80 pluggable views and the use of MVC/80 style controllers is not supported at toolkit level.

Custom UI widgets can usually handle input close to the input device level by overloading keyboard and mouse handlers. It is possible to use the MVC/80 controller as a strategy pattern for input handling at widget level; the underlying UI toolkit neither supports this approach directly nor prevents it. The input would come logically from the widget base class; a difference to MVC/80 where only controllers receive input from the input sensors.

Computer games and text editors are examples of applications that usually allow configuring input handling and might benefit from using a MVC/80 style
controller. Another example is direct manipulated applications, which typically have a canvas and a set of tools. When a tool is changed the canvas (view) remains but the input strategy (tool) is changed. The tool could be modeled as controller or as MVC/79 tool.

2.4 MVC Is Unimplementable?

Reenskaug’s [2007] original MVC/79 was a conceptual model. An implementation of a conceptual model requires interpretation. As interpretations can vary, the resulting implementations can become different while still sharing the same conceptual roots.

MVC/80 is closer to an implementation model. Even though it has undefined aspects it is more clearly defined than MVC/79. Despite this, there is a widespread confusion about MVC. The typical article, tutorial or discussion ends up presenting an MVC adaptation instead of the original MVC/80. The resulting confusion is visible in public discussion [C2b, 2008] and also noted by Fowler [2006].

The differences between MVC adaptations are typically related to the role of the controller [C2f, 2008]. The controller often becomes a mediator between the view and model layers and the model ends up being conceptually closer to PAC or MVC++, which will be presented later, than to MVC/80. One example of MVC adaptation using a mediating controller is Apple Cocoa framework [Apple, 2007].

Because modern UI toolkits combine MVC/80 controller and view into pluggable widgets, adaptation is needed to use MVC/80 outside its original context. In addition, because C++ and Java do not support Smalltalk blocks the MVC/80 pluggable view style is not possible without adaptation. As a result, many MVC adaptations reinvent the controller as a mediating controller, which becomes part of the application architecture instead of being a facet of the view layer as in MVC/80. In addition, the mediating controller decouples the view from the model whereas in MVC/80 the view interfaces the model directly.

MVC might be considered not implementable as noted in C2 Wiki discussion [C2d, 2008]. This is because there is no canonical implementation for MVC and because MVC adaptations provide conflicting definitions of MVC. It seems that the definition of MVC has become distorted beyond repair. Neither MVC/79 nor MVC/80 is defined with the required axiomatic precision, both have considerable omissions. For example, MVC/80 does not define how communication between
MVC triads is done. It neither defines how the model is adapted in situations where a full blown MVC is not desirable, e.g. menus which do not follow the MVC/80 pattern.

If someone wants to understand the MVC model, it is likely that an adapted version of MVC is learned first, after which reading about the original MVC/80 will be confusing. Reading more about MVC can only add to the confusion as more conflicting definitions are provided. Some readers might be lucky enough to run into Fowler’s [2006] take on MVC first and avoid further confusion.
3 Mediating Controller

Because many MVC adaptations end up using mediating controllers it is worth presenting models that are actually designed to use a mediating controller. Here two models are chosen as examples; PAC is a conceptual model developed independently of MVC, MVC++ is an implementation model for C++ derived from MVC.

3.1 PAC

Presentation-Abstraction-Control (PAC) is a UI model lesser known than MVC. In PAC a UI is formed by a tree of agents where each agent consists of three parts: Presentation, Abstraction and Control [Coutaz, 1987].

A presentation is the UI layer of an agent. It does not interface the model directly, such dependencies go through the control layer. A presentation implements the UI layer as fully as possible to minimize dependencies towards the control layer. A presentation is similar to a combined MVC/80 controller and view, excluding direct model access.

An abstraction contains the application’s domain model. It is not aware of controls or presentations. In other words, abstractions are similar to MVC/80 models.

The role of a control is to communicate with other agents and to express dependencies between abstraction and presentation layers. Because a control is a mediator, the abstraction layer does not communicate directly with the presentation layer and vice versa. A PAC control does not have a correspondent in MVC/80.

PAC triads are composed recursively. The result is a tree of PAC triads and conceptually an application is a PAC triad. Communication between the PAC triads is done via the control hierarchy.

PAC is connected to PAC-Amodeus, which is an agent-based architecture for modeling interactive systems. PAC-Amodeus takes a holistic approach on user interface architecture and considers aspects such as multimodalism, parallelism, task composition and decomposition. Because PAC-Amodeus is a conceptual model, it does not suggest any particular implementation style. PAC-Amodeus has been further developed to support groupware with PAC* [Coutaz, 1997].

Coutaz et al. [1993] acknowledge that in real world designs and implementations it is often not practical to follow conceptual models to the letter. They have
created the Slinky Metamodel which gives guidelines to adapting PAC-Amodeus
to the needs of individual designs and implementations. To compare PAC better
with other models presented here PAC-Amodeus will be adapted, yielding the
following metamodel instantiation:

1. Presentation Techniques Component is left out as cross-platform UI toolkits
already serve that purpose.

2. Interface with The Functional core is left out. It is assumed that the model
is local to a computer and in the same process with the UI.

3. Multimodalism and parallelism are not considered. Input is assumed to
come from keyboard and mouse. The UI is assumed to be single threaded.

As a result, only the Dialogue Controller and PAC triads remain. The PAC
properties are retained inside the Dialogue Controller which is formed by the tree
of PAC triads.

PAC does not require that a triad consists of three separate objects, one for
each facet [Coutaz et al., 1993]. It is enough that an implementation separates
the PAC facets. For example, a PAC triad could be implemented in one C module
[Coutaz, 1987]. Here object-oriented implementation is assumed, with each facet
of the triad implemented as a separate object. Depending on the need, a PAC
component can leave out the abstraction and presentation facets [Coutaz et al.,
1995].

Because PAC presentation contains all of the UI code PAC is directly usable
with modern UI toolkits. In other words, PAC does not have the conceptual and
implementation problems similar to the use of MVC/80 controller. Because many
MVC adaptations end up using mediating controllers, they are more similar to
PAC than to MVC/80. It would be more correct to call such adaptations PAC
adaptations rather than MVC/80 adaptations. Unfortunately, PAC is not as well
known as MVC.

Similar to MVC/79 the definition of PAC is conceptual and not formal. In ad-
dition, the PAC literature do not provide implementation examples. As a result,
it is not straightforward to apply the PAC model. The roles of the abstraction
and the presentation layers should be familiar, as the model/view separation is a

---

1 It should be noted that the terminology of PAC-Amodeus components has changed between
[Coutaz et al., 1993] and [Coutaz et al., 1995]. Our terminology is adopted from the latter
one.
recurring UI model pattern. Heuristic rules for devising PAC agents are provided by Coutaz et al. [1995]. While these rules give starting point for implementations, the role of the control in the application level architecture remains somewhat unclear.

3.2 MVC++

MVC++ model [Jaaksi, 1995] is a modified MVC model, a MVC adaptation in other words. MVC++ is intended to be used in C++ implementations, although the model is usable in other object-oriented languages.

MVC++ is a layered architecture where an application is divided to three layers: model, view and controller. According to Jaaksi [1995] the roles of the layers are as follows:

- The model part of an application represents "the real world". It is a collection of objects representing the concepts of the problem domain. The model has no UI elements.

- The view part is the outer layer that is visible to the end user. It is created by the controller. The objects of the view form the UI. There is a single controller for every view object.

- The controller part controls the interaction between the model and the view. There is a single view object for every controller object. On the other hand, one controller object may have relations with multiple model objects, and the same model object may be connected to many controller objects.

When compared to MVC/80 there are two main differences. Firstly, there is no direct connection between the view and the model. Any dependencies towards the model must go through the controller. This will decouple the view from the model. Secondly, the role of the controller has changed from the strategy pattern to the mediator pattern. In addition, the controller is no longer contained in the view and no longer the input handling facet of the view. The reasoning behind the mediating controller is to increase reusability by decoupling the model and the view. It can be argued that not much beside the model/view separation remains from the MVC/80, see figure 2.2.

MVC++ controllers and subcontrollers form a tree of controllers:
- One controller is the main controller. The main controller creates and controls the main view and other controllers, called subcontrollers.

- Dependencies between subcontrollers are handled via the main controller. The main controller decides the actions to take.

- A controller hierarchy can be recursive. That is, a subcontroller can contain a subcontroller and so forth. In such a case, the parent controller acts as the main controller to the subcontroller.

The tree structure allows adding new subcontrollers without modifying the existing subcontrollers.

MVC++ decouples the view from the controller. The MVC++ controller makes all the application specific decisions. As the view cannot have a direct dependency to the controller but still must be able to delegate actions to the controller, a view specific interface is needed between the view and the controller. This interface is called an abstract partner and it is implemented by the controller via inheritance. An abstract partner is essentially a single-seated specialization of the observer design pattern [Gamma et al., 1995].

One MVC++ interaction cycle is as follows:

1. A user does something that causes input. For example, a menu command is invoked.

2. The command arrives to a view. The view delegates decision making to a controller by calling the abstract partner interface.

3. The controller applies the command to the model.
4. The controller asks the view to update itself. If needed the controller may inform the parent controller.

The model is typically a passive part of an application [Jaaksi, 1995]. Although Jaaksi does not promote the use of abstract partners for models, adding such support would be straightforward by implementing support for multiple observers into the model. This would make the model active and basically bring the MVC/80 dependents system to MVC++.

According to Jaaksi MVC++ has the following benefits over MVC:

- Advantages of the original MVC are obtained in C++ environment.

- Application structure becomes clear and stable.

- Reusability and readability of classes increases.

Because MVC++ does not have MVC/80 style controller for input handling and does not promote active models by using dependents, the model/view separation can be taken as the only advantage obtained from MVC/80.

Application structure becomes clear and stable in the sense that the application is constructed recursively by MVC++ triads. Because the view delegates actions to controller, the interaction between the controller and the view can become complex and tightly coupled. Interaction sequences can become unclear as they are scattered between the controller and the view. In addition, if a view and a controller become tightly coupled, changes are propagated over the abstract partner interface and the implementation stability decreases.

Reusability considers views and models only, controllers are application specific and are not reusable. Reusability is achieved by making the model independent of the UI, decoupling views from controllers by using abstract partners, by carefully considering the responsibilities of different layers, and by minimizing the dependencies between the layers [Jaaksi, 1995]. MVC++ takes a forced approach to reusability; decoupling is applied to views that are not really reusable resulting in unnecessary complexity. For example, in complex cases a view and a controller can become tightly coupled, despite the abstract partner, and any reusability is unlikely. The price of implementation complexity and redundancy is still paid. From this viewpoint PAC provides a more balanced model as it allows combining control to presentation. On the other hand, nothing prevents from adapting the MVC++ model by allowing combining controllers and views.
The goal of MVC++ is to make each part in MVC++ triad as independent as possible. This is achieved by supporting a strict division of labor and implementing a minimal set of connections [Jaaksi, 1995]. On the other hand, a complex view is likely to require a complex controller; while the set of connections might be minimal, it will not be small and the implementation complexity is not minimal.  

Jaaksi [1995] argues that one benefit of using MVC++ is that the underlying windowing system can be changed only by modifying the classes in the view layer. Because the UI logic is implemented in controllers, this is possible at some level. However, changing the UI toolkit is not trivial as different UI toolkits have different implementation styles and changes can propagate to the rest of the system. For example, some UI toolkits require that a parent widget is provided when a widget is constructed. This can lead to subtle implementation problems if the previous toolkit allowed the creation of widgets without knowing a parent object at construction time. It is better to try avoiding the need for changing UI toolkit by using a cross-platform UI toolkit.

When compared to PAC, a MVC++ controller is partially in the presentation and control layers. While MVC++ controllers mediate between models and views and communicate with other MVC++ triads, the MVC++ controller is too involved in the view coordination in order to be a PAC Control. In PAC the control does not implement presentation logic, its purpose from presentation perspective is only to fulfill the dependencies towards a model and towards other PAC triads.

3.3 Role of the Mediating Controller

The MVC/80 framework requires a controller. If a MVC/80 controller is missing or implemented incorrectly, the application will not function properly [Burbeck, 1992]. In modern UI toolkits the sharing of the input devices is handled by the UI toolkit internally and the functionality of the MVC/80 controller is embedded in the widgets.

When adapting MVC/80 the mediating controller is an optional object; it is not required by the underlying framework. In addition, the role of the mediating controller is different from the MVC/80 controller. Instead of being a facet of a view, an instantiation of strategy pattern for input device handling, the mediating controller sits between the model and the view decoupling them and mediating the interaction between the two. A mediating controller handles the following tasks:
• Event routing between a model and a view.

• Mapping between the model and the view data structures.

• Communication with other controllers.

Event routing between the model and the view is the purpose of the mediator pattern. Mapping between the data structures is less clear. Model/view separation mandates that the model is unaware of the UI. If the mediator is supposed to completely decouple model and view, then the view should be insulated from model data types. The controller will have to convert model data to view data and vice versa. On the other hand, the mediator could be used for decoupling the view from specific model data while still allowing access to some model data. In other words, the model and the view could be sharing data structures to reduce the need for data conversions. Communication with other controllers assumes that the controllers form the hierarchy driving the application. Views form a hierarchy too, but the views do not have explicit dependencies between them.

A mediator can reduce complexity when there are a number of interdependent classes [Gamma et al., 1995]. The explicit dependencies between such classes can be removed and centralized into the mediator. Since a mediating controller often mediates between just two objects, a model and a view, the addition of the mediator can be questionable; adding a mediator between two objects can only increase complexity. PAC advocates combining a control to its presentation if there is no strong reason for the separation.

By comparing PAC and MVC++ it is clear that there can be lots of variation inside the mediating controller beside the tasks mentioned above. A PAC control is a thin layer, whereas a MVC++ controller is more complex containing much of the UI logic.

3.4 Mediating Controller Models a Process Recursively

Holub [1999a] does not consider MVC model object-oriented, because view objects need to know too much about model objects, and because there is too much data flowing in the system to make it maintainable. In other words, MVC works against the object-oriented principle of encapsulation, where objects should provide services rather than expose their internal structure and implementation details. Not everyone [C2e, 2008] agrees with Holub.
Holub's critique is related to model/view separation rather than MVC per se. If lots of data is flowing in UI models based on model/view separation, the addition of a mediating controller can only increase the trend of violating data encapsulation. From Holub's perspective, the result would be even less object-oriented.

Mediating controllers are universal in the sense that all UI applications have the following elements:

- Domain logic that contains the actual application functionality.
- User interface that presents the domain to the user.
- Glue functionality that combines domain logic with the UI. This includes mapping model data to UI data and vice versa.

The above elements are not necessarily separate objects, the elements exist in code regardless of the form of the implementation. For example, in architectures based on model/view separation, the data mapping between model and view data types is done by the views. A UI model based on the mediating controller makes the data flow steps explicit by modeling the data flow as three objects, the model, the controller and the view.

The existence of this process is caused by the conceptual decoupling of model and view and by the difference of data types between the two. The model concepts must be presented as UI concepts and the data must be mapped between the model and UI data structures. The gap is filled by the controller. It is generally not possible to remove the process because of the model/view separation; either the model should adopt the view data types or vice versa.

Using a mediating controller allows using the same Model-Controller-View pattern recursively through the whole application. This is visible in PAC and MVC++.

3.5 Problems of the Mediating Controller

A mediating controller sits between a model and a view. As a result any explicit dependency between the model and the view becomes implicit. When the view wants to display some data it queries that data from the controller. Because the controller works as an insulation layer between the two, there should be more freedom to change the model and the view as the controller can absorb changes.
Unfortunately, decoupling and indirection do not remove dependencies, they only make them implicit. A certain view is typically optimized as the presentation of a certain model, it cannot display whatever model. A certain model has to satisfy the requirements of a certain view to be displayed. For example, if the model has structure it must provide access to the structure to allow presenting the structure, whether or not a mediating controller is used. During application implementation the interfaces tend to change enough to propagate over controllers defeating their purpose.

Pawson [2004] compared two implementations of automotive dealership applications, CarServ1 and CarServ2. CarServ1 used a mediating controller and two level model layer. CarServ2 did not use a mediating controller, but rather Naked Objects model where only the model layer is explicitly implemented. According to Pawson [2004], the modifications to CarServ1 were scattered and involved all four layers, resulting in a larger modification effort when compared to CarServ2. Pawson's findings show the effect of the implicit dependencies; despite decoupling, changes are not contained and tend to propagate over layers.

If the goal of the mediator is to reduce dependencies, then it should be remembered that classes already provide interfaces and encapsulation. That is, the class implementation is separated from the class interface and the implementation can be changed to some extent without affecting the class interface. If binding the view to a specific model is not desirable it is always possible to use an abstract model interface and bind the view to a certain class of models instead of a specific model. This applies to reusability concerns as well. This is what Krasner and Pope [1988] proposed by MVC/80 transformer objects. This approach is also known as Model-Model-View-Controller [C2a, 2007].

If the purpose of the mediating controller is to completely decouple the model and the view, then the controller must map the model data into an intermediate data structure or to a UI data structure in order to maintain the decoupling. When the presented model has complex and large structure, this mapping causes too much work compared to the benefits. The typical solution is to break the decoupling and pass model objects to the view via the controller. This, of course, partially defeats the purpose of the mediating controller and the view becomes directly dependent on the model. If model objects are passed directly to the view, it will have direct access to manipulate the model objects. The result is perhaps the worst combination: explicit dependencies combined with a mediator.

Interdependencies can often removed by refactoring. For example, consider class A that has functionality that is needed by class B. Unfortunately, A contains
lots of other functionality that should really not be exposed to B; B should not really depend on A. Adding a mediator or indirection between A and B certainly decouples the two, but B would still be indirectly dependent on A. The solution is to isolate the functionality shared by A and B to a third class C. This breaks the dependency between A and B completely. The same elementary design principle can be applied to UI designs as well. Instead of adding a mediating controller to decouple UI and model layers the model layer should be factored correctly.

Using a mediating controller does not scale. In small examples the redundancy of using a mediating controller is tolerable, but in large projects where considerable part of the codebase is in controllers, the extra work needed for implementation and maintenance becomes significant. Controllers that mediate between just two objects increase complexity without achieving the benefits of mediators.

Adding a mediating controller means more work for achieving the same features as without a mediating controller; more files, more types and interfaces, more boiler plate code. In case of MVC++, additional interfaces come from abstract partners. While the impact of additional files, types and inheritance may seem unimportant, given large enough code bases compilation speed can become a serious problem [Lakos, 1996], for example. Especially excessive use of inheritance can slow down C++ compilation greatly.

When a feature requires changes to the related model and view, also the controller must be changed; implicit dependencies result in explicit changes. The result is that controllers are under constant change and become a shearing layer [Foote & Yoder, 1999]. The shearing effect is amplified during the early development phase as both model and view layers tend to change rapidly. If the architecture has excessive layers of code, then changes that affect all layers lead to excessive changes. In such cases, the layering does not prevent changes, it amplifies them.

It is natural to think that the application structure is defined by the mediating controller hierarchy as the controllers are coordinating the application. On the other hand, the views form another but not necessarily identical hierarchy parallel to the controller hierarchy. Two dissimilar but parallel hierarchies can lead to subtle situations where the construction of the view hierarchy, for example, can constrain the construction of the controller hierarchy. In MVC/80 the controller hierarchy is implicit, it is a part of the view hierarchy.

A composite view is typically constructed by compositing existing widgets. The composite view does not have input handling or drawing functionality, it
serves as a container for the child widgets. In addition, it will typically intercept child widget actions and forward them to the mediating controller. Considering the above, a composite view is essentially a controller subclassed from a widget. Using a mediating controller with composite views is essentially adding a second controller layer.

3.6 Mediating Controller Should Be Avoided

When comparing PAC and MVC++, PAC provides more balanced approach as it allows combining elements of the PAC triad and minimizes the role of control by implementing the UI fully in the presentation. MVC++ is prone to excessive implementation redundancies caused by the division of labor between the controller and the view.

Whichever mediating model is used, the mediating controller is likely to become a stateless router between the model and the view. Its repetitive code could be removed from the implementation without taking away any functionality. The result would be simplified application architecture and reduced glue code.

By default the mediating controller should be avoided. It should be added only if it is explicitly needed. Mediating between only two objects is not such a need.
4 Model/View Separation

The UI models presented in Chapters 1 and 2 agree on one aspect: the model is separated from the UI. From the viewpoint of model/view separation, there are no differences between MVC/79, MVC/80, MVC adaptations, PAC and MVC++.

The differences between them consider the structuring of the UI layer, namely how the UI layer is broken into separate controllers and views. While the model/view separation is a recurring pattern, is it really worth making?

One way to motivate the model/view separation is that models and views are in different problem domains. A model does not involve human-computer interaction, whereas the purpose of a view is to act as an interpreter between the user and the model. Separating the model from the view could therefore be the result of any design which leads to a separation of concerns.

The model/view separation can increase maintainability and allow reusability [Krasner & Pope, 1988]. In addition, the separation has the following benefits:

1. Multiple views can refer to the same model.

2. The model and UI layers can be changed and developed relatively independently, though the interface to the model must be agreed. This is the benefit of encapsulation in general.

3. As the model has no dependencies to the UI, it can be used in other platforms, applications and UIs; the model can become reusable.

4. Making the model independent of the UI removes a possible source of problems, especially if the UI or UI toolkit is under constant change.

5. Model can be unit, integration and regression tested independent of the UI. In addition, the testing can be done on other platforms beside the UI platform.

While model/view separation adds complexity, it is often a natural separation of concern in real-world applications. For example, when clients are distributed and the model is a database accessed over a network, the application domain itself forces the model/view separation.

The model part of an application is often considered to contain most of the domain logic of an application. This holds for model/data driven applications. On the other hand, there is a class of applications for which the domain logic is the
UI itself. For example, image manipulation applications and web browsers are principally UIs. Adopting model/view separation might be counter-productive for applications where the majority of the code is in UI or the model contains visual data. The situation is also noted by Soegaard [2008].

In addition, Pawson [2004] suggests that the separation of the user interface from the domain model leads to the separation of procedure and data in object-oriented designs. This is against the idea of object-orientedness and is visible in implementations that use the model as a data structure and implement the behavior of the model partially in the view layer.

4.1 Encapsulation

Model/view separation leads to increased implementation complexity, as two layers of code and encapsulation are introduced. PAC and MVC++ increase the number of layers to three, as the UI layer is broken into controller and view layers, which will further increase the implementation complexity.

While the model is not explicitly dependent on the view, there will be implicit dependencies, as the view might require features which make no sense from the model perspective but do make sense from the view perspective. These implicit dependencies can break the model encapsulation in several ways, all of which lead to exposing the model's internal implementation.

Firstly, if the model is structurally complex and this structure should be presented in the view, there is a temptation to make shortcuts, which includes exposing the model structure by returning pointers or by using public variables. Such temptation can often be avoided by abstracting the access to the model structure by using visitor, iterator, tree walker or similar patterns. Functional programming styles can also be used, including map and inject.

Secondly, if the model is relatively flat and contains a large number of data elements, perhaps of the same type, there is a temptation to skip writing set and get methods and expose the variables in the public part of the class. In such cases it is worth considering whether the data should be class members at all, but rather values in a data structure such as hash table. On the other hand, according to Holub [1999a], set and get methods are an elaborate way to make private data public and should not be considered object-oriented.

Thirdly, for performance reasons it might be tempting to expose as much of

---

1 Also known as map, hash, dictionary, finite map, lookup table.
the model as needed. As access abstractions and encapsulation decrease performance, there might be no way to resist the temptation, e.g. in graphics intensive applications.

In stand-alone applications encapsulation violations are not fatal, because such violations can be fixed or simply ignored. On the other hand, if the model/view separation is implemented in a public API available to third parties, encapsulation should be taken seriously. In such APIs, if source and binary compatibility are promised, reversing from design and implementation defects can become impossible. Doing such changes would break third party applications, if the third party implementations had started to rely on the exposed features, which they invariably will do. Working around such defects will quickly make any design and implementation an abomination.

### 4.2 Mapping Data

Core language libraries tend not to contain definitions for graphics types, including types such as point, rectangle, color and bitmap. These types are typically defined by the UI toolkit. As the model is independent of the UI it cannot share the graphics types with the UI. If a model contains graphics data and model/view separation must be preserved, the model must define its own set of graphics types. This leads to a situation where the model and the UI end up with different and possibly incompatible definitions for similar types.

There are several ways to achieve mapping between the model types and UI types:

1. The most common graphics types are in core libraries. The model and view share the core definitions.

2. The UI can be parameterized with the model types.

3. Model types are implemented identically, in data and structure, to the UI types. This allows direct type casting.

4. Provide mapping functions between the types.

The first one is not possible, as UI toolkits implement their own types and core language libraries do not contain UI specific types. Java Swing is almost an exception, as Swing UI components use abstract models. For example, a list UI
component has pluggable list model. Any data object which implements the abstract list model interface can be plugged to a list UI component. Unfortunately, Swing’s abstract models are in the java.swing package. Using Swing’s abstract model interfaces in model will therefore create a dependency to the UI package. And further, not all Java editions contain the Swing toolkit.

The second one is rarely available. Rare examples include OpenGL and Direct3D, which allow defining the source data layout for vertices. In Direct3D this feature is called the Flexible Vertex Format. As a result, it is possible to use custom vertex data in the model and use it directly in the rendering code.

The third one can be considered bad style, although it is unlikely that the UI toolkit types will change. On the other hand, requirement for implementing the model types identical to the UI types might not suit the model’s needs.

The fourth is the least efficient and potentially the most verbose one. Unfortunately, it is the most likely one. For every model type instance the data must be converted to UI type instance. Either this conversion is done by hand or by using helper functions. In C++ it is possible to reduce verbosity by implementing an assignment operator for these types. Such operator would make type conversions implicit, i.e. the compiler would take care of the conversion whenever needed.

The incompatibility of types is not constrained to graphics types. For example, C++ UI toolkits tend to use their own string implementations. A C++ application using the Qt UI toolkit most probably ends up using three string types: C string, std::string, and QString. Another problem related to strings is Unicode versus ASCII strings. For example, C and C++ strings are ASCII, but Symbian OS APIs use 16-bit Unicode strings.

4.3 Direct Manipulation

Direct manipulated applications break the requirement for decoupling model and view. If the model contains inherently visual information, such as vertex mesh or bitmap, the model has direct visual presentation; the view is the model and the model is the view. In such situation, decoupling the model and the view can become counter-productive. For example, mapping between a model specific bitmap and UI specific bitmap can be inefficient. It is likely that the model bitmap is stored directly to a UI specific data structure and the model is no longer decoupled from the UI.

Direct manipulation usually involves primitive level graphics and input operations. As a result, the role of the UI library is to work as a drawing canvas
and as a source of raw input data. Computer games are an extreme example as they tend to be implemented by using only the most primitive drawing and input functions. The typical productivity applications are hybrids combining direct manipulation with widget based views.

Input handling for direct manipulation must be written separately, it cannot be provided by a UI toolkit. Input handling is explicitly dependent on the visualization of the model. As a result, the code that handles the input must also know how the model was rendered. Often the code which renders the model also handles the input.

Direct manipulated applications can use the event driven approach available in most UI platforms. Unfortunately, this can lead to nonresponsive UI, if keyboard input and mouse input events are generated too slowly. To get as smooth input feedback as possible, input devices should be polled in a message loop, for example. This turns the input handling on its head, instead of handling incoming input events the input devices must be polled directly.

4.4 Presentation Model

In some situations the model/view separation is too coarse. An example: a domain model contains non-visual data. In the view each piece of data is drawn as text and highlighted by red if certain conditions are true. Multiple different views use the same visualization approach. If the coloring functionality is implemented in the views, the views end up containing model behavior (the highlight condition) and could end up with duplicate data (the highlight color). Implementing the coloring functionality in the domain model would introduce functionality that is outside the model domain, which would be against the very idea of a domain model.

The situation can be solved by creating a UI specific model, which will contain the highlight condition logic and color information. This UI specific model is known as presentation model [Fowler, 2004]. The introduction of presentation model creates two layers of models, one layer for the domain model and another layer for the UI model. The presentation model is considered to be part of the presentation layer.

According to Fowler [2004], the presentation model should not be a facade to a model, but rather an abstraction of the view. One benefit of this approach is that the state of the UI is not in the widgets, but rather, the widgets reflect the state of the presentation model.
4.5 Visual Proxy

Holub’s [1999b] Visual Proxy is a UI model which abandons the model/view separation completely. While the model and view objects are still separate objects, they are implemented in the scope of a single class. The model and view layers are tightly coupled, not decoupled. Visual Proxy is intended to be used in Java implementations, although nothing prevents using it with other object-oriented languages.

According to Holub the purpose of objects is to provide services. The purpose of a model is to provide services to a view. If the model is decoupled from the view it cannot provide all services for the view. For example, the model cannot draw its objects for the view. Holub’s approach with Visual Proxy is to make the model objects able to provide all services, including viewing them. This requires that models are not decoupled from views.

According to Holub it is more common to display object’s attributes and their combinations than whole objects. In addition, a single attribute can almost always be displayed using the same visualization; it is even desirable in a consistent UI.

A visual proxy is a view tightly coupled with model objects and its purpose is to provide a UI presentation for a model object or some of its attributes. Complete UIs are created by compositing visual proxies.

Model objects provide visual proxies, which are typically implemented as inner classes. As a result, visual proxies can refer to the model instance variables directly. Model objects communicate to visual proxies via an observer interfaces. There are no dependencies between visual proxies. Multiple visual proxies can refer to a single model object. From the MVC/80 point of view, a visual proxy combines model, view and controller into the same class.

Because visual proxies are implemented as part of model objects, the model becomes dependent on the used UI library. According to Holub this does not present a problem because Java’s AWT and Swing UI toolkits are already abstract and platform independent.

The approach of visual proxy may seem unorthodox when viewed from the model/view separation perspective. On the other hand, as noted by Holub, visual proxies actually support model/view separation because visual proxies are separate objects; UIs constructed from visual proxies are not constructed from model objects but from visual proxies that are queried from the model. In other words, while visual proxy separates model and view objects, it does not decouple
them. The traditional model/view separation also decouples the model from the view.

The virtue of visual proxy is that it allows minimizing controller code and also minimizes the need for data mapping between the view and the model. As a result, using Visual Proxy could lead to simplified implementations for a specific class of applications, namely forms based applications.

4.6 Naked Objects

Naked Objects is a UI architecture proposed by Pawson [2004]. Similar to Visual Proxy, the approach taken by Naked Objects is somewhat radical compared to the other UI architectures presented here. In Naked Objects the UI is automatically generated from the model by analyzing the model interface by using Java’s reflection features. As a result, the model must contain all of the application domain logic.

Naked Objects is a rare example of UI framework that is parameterized by the model. Visual Proxy attempts to minimize the work caused by the data mapping process by moving the object presentation close to the object implementation but cannot remove it completely. Naked Objects manages to make the process implicit from developer’s point of view; mapping between model data and UI data still happens, but it is done by the generated UI layer.

In Naked Objects the dependencies between model and view are explicit, but changes are implicit because the UI is generated. With a mediating controller dependencies between model and view are implicit because of the controller, but changes are explicit as they need manual programming to all layers.

Reenskaug’s foreword to Pawson’s [2004] thesis and Kay’s [2003] recent comments suggest that the motivation behind MVC/79 was closer to Naked Objects than MVC/80, which was the first implemented MVC model.

Naked Objects is suitable to a certain class of applications where the UI layer can be constructed from a predefined set of widgets. Because the UI is generated it will end up similar between different applications using Naked Objects. It is clear that Naked Objects are not for building direct manipulated editors, for example. On the other hand, the Naked Objects approach could be used for displaying objects for direct manipulation; the rendering code could analyze the model object interfaces and render the model objects automatically on the canvas.
4.7 Model/View Separation Is Sometimes Worth Doing

The examples provided by Visual Proxy and Naked Objects show that while model/view separation is a recurring pattern in UI designs and implementations, the alternatives are worth considering. Both Visual Proxy and Naked Objects can lead to more simplified architectures when compared to traditional model/view separation.

Both Visual Proxy and Naked Objects are constrained by language requirements. Visual Proxy relies on Java’s inner classes, Naked Objects relies on Java’s reflection to analyze class interfaces. Implementing either model with C++ would be significantly more complex. In addition, using Naked Objects requires a separate framework for generating the UI. Its approach is not possible with Java Swing only, for example.

While model/view separation does provide benefits, it may be counter-productive in some domains, including direct manipulated views. One of the strongest benefits of a separate model is that it can be tested without the UI layer. Personal experience from large, real-world application and platform implementations show that testing is often an overlooked aspect of design and implementation.

A natural application for model/view separation is a centralized model which is accessed by multiple distributed UI clients. On the other hand, both Visual Proxy and Naked Objects can handle this particular problem domain equally well.
5 Dynamic Languages

5.1 Generic Messaging and Data Passing

Avoiding the mediating controller simplifies application architecture and implementation. Personal experience in large C++ projects shows that the typical UI implementation can be simplified further by adopting dynamic features. One such area is messaging between the components. The lack of focus on messaging in object-oriented programming and research is noted by Alan Kay et al. [2006].

In MVC/80 dependents allow the use of any object as a model and offer a generic way for receiving messages from the model. Although it is possible to mimic the generic dependents style with C++ and Java, messaging between components is often implemented by using interfaces and the observer design pattern. Especially Java Swing UI widgets follow the observer style closely. In this style, the observed component provides an interface for adding observers and requires that the observer implements a certain interface. While the observer is simple to implement and allows breaking explicit dependencies from observed to observer, it is not without trade-offs.

The problem with observers is verbosity, namespace pollution and ironically dependencies. Verbosity happens because every observed component requires implementing an observer interface. That is, to receive even the simplest message a class must be inherited and its methods implemented and typically every observed component requires a specific observer interface.

Namespace pollution happens at two levels. Firstly, each separate observer interface introduces a new class to the system. Secondly, namespace collisions can happen when a single object inherits multiple different observer interfaces that have similar function names. Java’s inner classes help avoiding this problem.

Dependency problems happen when the observer interface is changed. A change in observer interface propagates to all observing components and requires interface changes. This can lead to binary compatibility breaks. Observers are also monolithic, which contributes to the dependency problem. Implementing a subset of an observer interface requires adapters ¹.

The above problems can be avoided by using a generic messaging system. It is also makes plugging components together more straightforward, at least from interface point of view. Generic messaging is nothing new. For example, the original Windows API was based on generic messages. Modern UI toolkits

¹ In the Java meaning of an adapter.
typically use message loops internally while the widgets provide an object-oriented interface, for example Trolltech Qt.

```cpp
class IMsgReceiver
{
public:
    virtual void HandleMsg( const IMsgSrc& aSrc,
                            uint aMsgId,
                            const IMsgData* aData ) = 0;
};
```

Listing 5.1: Generic message interface in C++. The observer interface `IMsgReceiver` must be implemented by any object that wishes to receive generic messages. Parameter `aSrc` defines the source object of the incoming message, parameter `aMsgId` contains the message identifier, parameter `aData` contains the message data. `IMsgData` could be, for example, an interface to hash table or array. Note that `aData` can be `NULL`; this is to allow messages without data.

Listing 5.1 provides an example of generic message observer interface. The example is similar to Java’s rarely used class `java.util.Observable` and interface `java.util.Observer`. While these generic observer interfaces exist in Java, no Swing UI component is using them as all Swing observer interfaces are subclassed from `java.util.EventListener`.

Generic messaging requires generic data passing. This can be done by using a data collection, such as an array, list or hash table. Support for dynamic typing can be added to the data collection. For example, the collection might contain functions for getting values for the keys as integers or strings. If an integer accessor was used for a string, a dynamic type error would be generated.

When compared to the observer interface style, generic messaging has the following benefits:

- Less types and interfaces, class interfaces are simplified.
- Adding new messages and new message data is straightforward, no interface or interface changes needed.
- It is possible to avoid binary compatibility breaks when adding or changing messaging.

It is worth noting that binary compatibility is an often overlooked design and implementation aspect. Often there is a need to fix defects or change an
existing implementation to add new features. Doing such changes in a stand-alone application is straightforward as its internals can be changed freely. Doing such changes to a library with external dependencies is not straightforward, as either the backward compatibility must be preserved or all dependent binaries must be changed and rebuild. Rebuilding binaries is often not feasible, especially if the binaries have been installed to hundreds or thousands of machines. Unless breaking backward compatibility is the only option, the scales tend to tilt towards preserving backward compatibility. As a result, it is possible that new features or error corrections become impossible because of binary compatibility requirements. Implementation strategies can be adopted to reduce the possibility of binary compatibility problems. Generic messaging and generic data passing allow adding new messages without changing the messaging interface signature.

Generic messaging does not come without drawbacks:

- Message handling routines can become large switch/cases or if/else/if/else chains.
- Compile time type checks are lost.
- Message ID maintenance must be done.
- Accessing the message data is harder when compared to plain function parameters.
- UI toolkits do not support custom generic messaging.

Large switch/cases can be messy, but the switch/case can act as a dispatcher which forwards the messages to corresponding handler functions. Loss of compile time type checking can be offset by unit testing and defensive programming (e.g. invariants, asserts). ID maintenance is straightforward, the message ID range must be large enough and the ID space could be divided into ranges. Getting the message data from a collection is always more complex than using function parameters directly, though it will not be harder than using a hash table or an array.

Allocating the message data from the heap is not always desirable. It is possible to implement the generic parameter collection as a templated array where the template parameter defines the number of parameters in the array. This way message data allocation can be done from the stack and the programmer needs not to consider memory allocation and deletion.
The generic messaging system described above is rudimentary and could be extended to filter unwanted or wanted messages, to allow extensions by third parties, to allow defining message propagation rules, to use message queuing and so forth.

5.2 Using Multiple Languages

While generic messaging and generic data passing make sense from the C++ or Java perspective, both are essentially dynamic features and implementing them will probably be inferior to what is already available. In other words, taking the C++ or Java perspective is too narrow. Dynamic languages, such as Python and Ruby, already provide the needed dynamic features to build generic messaging. In fact, implementation of the rudimentary messaging interface described previously is trivial with dynamic languages.

Beside generic messaging and data passing, supporting generic properties would provide similar benefits; the property interface would be small and adding new properties could be done without affecting class interfaces explicitly. On the other hand, implementing support for generic properties in C++ or Java would make the classes similar to hash tables, which would be yet another step towards dynamic language emulation.

Ironically, going forwards and starting using dynamic languages is actually going backwards in time. For example, Smalltalk and Lisp are both dynamic programming languages which existed well before C++ or Java.

Most UI applications use a single programming language implementation, typically C++, Java or C#. Applications that use mixed language strategy, by combining system level programming language and dynamic programming language in the same implementation, include GNU Emacs, AutoCAD, Gimp and Adobe Photoshop CS. Using multiple languages is not a new idea. For example, in mid-1970s it was common to write programs in C and glue them together by using shell scripting [Raymond, 2003].

Ideally, in a mixed language implementation, the system language and the dynamic language complement each other. System languages are designed for data structures and algorithms whereas dynamic languages are often designed for connecting components together [Ousterhout, 1998]. UI programming is typically done by compositing existing widgets. Because dynamic languages provide features for gluing, they should allow making UI programming simpler. For example, simplifying the construction of UI applications was one of the motivators
for TCL [Ousterhout, 2008].

Dynamic languages sacrifice speed and static typing for strong language features and flexibility. A system language is used for interfacing the underlying platform, including operating system and hardware, and for performance critical code. In other words, the application would be implemented fully with the dynamic language and the system language would be used only when needed. The mechanism for using a system language is typically extending the dynamic language with modules written in C.

Visual Proxy and Naked Objects both take steps towards using dynamic language features. Visual Proxy relies on Java’s inner classes and Naked Objects relies on Java’s reflection. Both are inferior to what is available in dynamic languages in the form of lexical closures and meta-class programming. MVC/80 was implemented in Smalltalk-80, which is a dynamic language. The lack of dynamic features, such as blocks, partially explains why MVC adaptations differ from MVC/80.

5.3 Resistance to Adoption

Modern PCs are powerful enough for using dynamic languages. As evidence, mixed language implementations have started to appear in mainstream applications. For example, Adobe Photoshop CS is using LUA as an application macro language. Still, dynamic languages face resistance to adoption. Two common objections to dynamic languages are slow speed compared to system languages and loss of compile time type checking.

Performance is often the very first objection against using dynamic languages. It is true that when compared to system level languages, such as C, dynamic languages will, on the average, execute slower and use more memory. It is also true that programmer skills matter; a dynamic language implementation can outperform system language implementation [Prechelt, 2000].

Much of UI code is not performance critical. For example, event handlers are executed only when the user invokes an action. If the performance impact of using a dynamic language is not visible to the user, it does not make sense to speed optimize something that does not affect usability.

Performance is not being sacrificed for nothing, it is traded for dynamic language features. Application domain dictates how much one can trade performance for features before it becomes a bottleneck. Dynamic features can and often will be implemented in system level languages. Unfortunately, the average quality of
such an effort has inspired Greenspun's tenth rule of programming:\footnote{While the rule is a well known programming language anecdote, even Philip Greenspun himself cannot remember where he originally wrote it, see http://philip.greenspun.com/research/}:

"Any sufficiently complicated C or FORTRAN program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp."

The usual advice is that one should implement an application first without considering performance too much, because premature and instruction level optimizations can be counter-productive by making later modifications to the code harder. If performance becomes an issue, the performance critical code can be implemented in system level language.

Dynamic typing loses compile time type checking and causes new kinds of run-time errors (e.g. a function can be missing). On the other hand, compile time type checking does not guarantee functional code as code that compiles can have run-time errors. A way to verify that interfaces and functions are working correctly is to run unit and module tests. Unit testing as run-time testing is equally effective for statically and dynamically typed languages. Hence, if such tests are required for statically typed code to ensure quality, then the same quality should be reachable with the same tests and dynamic language. In other words, the presence of unit tests reduces the importance of compile time type checking.

Dynamic languages face a potential resistance to adoption. The best example of this is Lisp, which is typically resisted because of its syntax, which is unfamiliar to those accustomed to the 'bracket syntax' of Java, C++ and C. A somewhat famous example of resistance to Lisp is Viaweb, a web-based application for building online stores. Graham [2001a] attributes much of the Viaweb’s success to Common Lisp. Viaweb was sold to Yahoo and they decided to rewrite Viaweb with C++ and Perl. According to Graham [2003b] this happened because:

(a) The reason they rewrote it was entirely that the current engineers did not understand Lisp and were too afraid to learn it.

(b) The resulting program is a new world's record case of Greenspun's Tenth Rule. The Yahoo Store Editor called compile at runtime on s-expressions made on the fly. To translate this into C++ they literally had to write a Lisp interpreter.
(c) Even then, they had to drop some features (involving advanced uses of closures)."

Another Lisp rejection example involves Gnome Desktop Environment which decided to switch its window manager from Lisp-based Sawfish to Metacity which is written in C [Schaller, 2007]. Yet another example is Reddit, a social bookmarking site, which decided to switch from Lisp to Python [Huffman, 2005].

Using two languages in the same implementation requires more skills than single language implementation [Raymond, 2003]. The added skill requirements might contribute to the resistance. For example, if the boundary between the languages is in wrong place, then the result can be overly complex and arduous, as the division of labor is not optimal.

Resistance to adoption will probably never go away as a new language requires learning new syntax, new concepts and new libraries. What makes a language popular has sparked discussion [Graham, 2001b; 2003a; Yegge, 2005].

5.4 Choosing a Dynamic Language

Choosing the right dynamic language is hard as there are many alternatives. Although not UI specific, the following aspects must be considered:

- Many languages have an application domain bias. For example, some dynamic languages are biased to being an embeddable application macro language, while others are biased to being general purpose scripting languages.

- Some dynamic languages are experimental. Core language features and even syntax may change between versions.

- Language run-time availability on target and development platform.

- Depending on the developed application, the language run-time license should be checked carefully. Most dynamic languages are community projects and are open source or free software. If the languages will be embedded into an application the license must be quite liberal, for example BSD license. In such case, any GPL license is out of question for proprietary applications.

For UI development, the language should have the following characteristics:
• Either a language native UI library or UI bindings to system level or cross platform UI libraries must be available. If this feature is not part of the core language libraries, installing the needed module should be straightforward.

• The language itself should be mature and actively maintained. In addition, there should be an established user base. In other words, the language should be actively used and somewhat popular.

• The language should be extensible by system level languages.

UI toolkits native to dynamic languages are rare, one example is the TK library for TCL. System native UI toolkits have traditionally been implemented in C or C++. As a result, UI programming with dynamic languages is often done by using bindings to the native UI toolkits. While this approach is a lightweight way to gain access to UI toolkit it has several problems.

Firstly, the UI binding modules tend to be experimental. The bindings often expose only a part of a toolkit, are maintained only occasionally, lag behind the newest toolkit version or simply have been abandoned.

Secondly, the bindings are usually not part of the standard language libraries. That is, the bindings neither come with the language environment nor are available via package installation tools, such as Perl CPAN or Ruby Gems. Because of this, the installation of the bindings must often be done manually or by a separate installer. This increases the difficulty of distributing and maintaining the UI application.

Thirdly, even if the bindings were of high quality and readily available, interfacing a foreign language implementation can leak the foreign language programming model and constraints to the dynamic language. This can lead to unintuitive use and can prevent using desirable dynamic language features. For example, using TK in Python feels awkward compared to native Python libraries.

For a general purpose dynamic language, the popularity criteria immediately excludes experimental, research only and inactively developed and maintained languages. In addition, because of the UI bias, pure scripting languages such as shell scripts and Perl are excluded. Python fulfills the set criteria and has usable UI bindings, including bindings to wxWidgets, GTK and Trolltech Qt. Lisp is another good choice. It has a long history and large applications are using it, including AutoCAD and Emacs. Unfortunately, Lisp implementations are many and incompatible with each other. In addition, support for UI bindings is bad compared to Python.
Python and Common Lisp are biased towards being the main application language to which extensions written in C can be added. Another approach is to embed the dynamic language interpreter to a system language implementation. For embeddable dynamic languages good candidates are LUA and Scheme. Both of them are designed to be minimal and embeddable. LUA and many Scheme implementations have liberal licenses, hence they can be used in commercial applications without licensing restrictions.

A straightforward choice for system language is C; most dynamic languages are implemented in C. This will make C their native extension and embedding language. Using C++ is harder, as C++ has too many implicit language constructs, such as vtables, and C++ object model must be mapped to dynamic language object model (if any).

Whatever the choice of dynamic and system level languages is, they should serve different domains. For example, mixing Java and C# in a single implementation does not make sense. Using middle-of-the-road languages such as Java, C# and D in mixed language implementations provides fewer benefits as they already provide some dynamic features.

5.5 Design Patterns

Many design patterns are related to working around language related constraints. While design patterns can solve some problems elegantly, using them can lead to repeating boilerplate code. According to Norvig [1998] 12 of 23 design patterns presented by Gamma et al. [1995] have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern. Some design patterns are not needed at all in some dynamic languages. For example, the iterator pattern is not needed in Lisp [Baker, 1993].

The problem with design patterns in C++ and Java is that they are recurring, one has to repeat them over and over again when implementing large UI applications. If metaprogramming is available some design patterns can be factored out. The design patterns would still exist but would not be recurring and requiring constant repeating manually.

Lisp macros are perhaps the ultimate metaprogramming facility. Lisp macros allow changing Lisp syntax. With macros the repetitive pattern is abstracted into a macro and expanded when the macro is applied instead of repeating the pattern manually. While Ruby does not provide Lisp style macros it allows class metaprogramming. For example, it is possible to add implicit pre processing and
post processing steps to Ruby method calls.

5.6 Lexical Closure

Observer is one of the most frequent design patterns in UI programming. In object-oriented languages, such as C++ and Java, observers have subtle implementation problems.

Consider classes Model, ModelObserver and View. When Model is changed it sends a notification to all connected ModelObservers. In order to observe the Model the View implements the ModelObserver interface and connects itself to the Model. This approach works in Java and C++.

In C++ function pointers could be used instead of a ModelObserver. When the Model is changed it calls all connected functions. In order to observe the Model the View implements a callback function with the required signature. Unfortunately, calling a function cannot target an object, the View in this case. This can be solved by storing a reference to the View to a global variable, which the function could refer. Unfortunately, this is not practical if there are multiple Views and callback functions. Another approach is to pass a reference to the View as parameter. In this case the Model must store the reference in order to pass the View to observer function call. In C++ method pointers could be used, but similarly to plain functions, a separate reference to the View is needed.

Lexical closure is a function object that remembers the context where it was created. Closure is supported by many dynamic languages, for example Lisp and Ruby. Lexical closures allow implementing the observer pattern with functions only. When observer is implemented with closures, the View would provide a function with the required signature to the Model. The Model would send a notification to all connected observers by calling the function objects. The function can refer to the View because it carries a reference to the View in its context.

Closure allows implementing observer in object-oriented way by using function objects only. This can be viewed as improvement over the Java or C++ object model, because no separate observer interface class is needed for implementing the observer pattern (see listing 5.2). As a result, the number of types is reduced and the observer implementation needs not to be bound to objects. In addition, no placeholder objects are needed because the function will carry its context. For example, if the Model supports only one observer a closure can be used for dispatching the event to multiple observers, no separate object is needed for storing the extra observers.
class Widget
  def initialize()
    @observers = []
  end

  def add_observer(o)
    @observers.push(o)
  end

  def change()
    @observers.each { |fn| fn.call() }
  end
end

class Counter
  def initialize( widget )
    @count = 0
    fn = lambda { @count += 1; printf("count %d\n", @count) }
    widget.add_observer(fn)
  end
end

w = Widget.new()
c = Counter.new(w)
w.add_observer( lambda { printf("changed\n") } )
w.change()

Listing 5.2: Ruby closure example. Class Widget has a list of observer functions. Calling Widget’s method change (line 10) will call the attached observers (line 11). Class Counter observes changes in the Widget and increments its count member. For this the Counter creates an anonymous function (line 18) that closes in the member variable count; a lexical closure is created. From lines 18 and 25 it is visible that function call and method call are similar.

Java’s inner classes are a limited version of closures. An inner class closes in the outer class and can refer to its variables. Java does not have proper closures, to get closures Java should add function objects. Closures are not possible with C or C++.
Using the MVC/80 pluggable views approach is a good alternative to a controller object. Instead of a discrete controller object the equivalent functionality is distributed to closures that are bound to the view. This is in line with the observation by Yegge [2004] that strategy objects are typically stateless, making them first order functions in disguise.

Smalltalk has blocks which are equivalent to lexical closures. Pluggable views allow binding blocks to view actions when the view is constructed. In other words, the pluggable view functionality can be augmented by hanging code in the form of closures. Typically pluggable views in C++ and Java are implemented by using the observer pattern. This might be the reason why many MVC adaptations for Java and C++ include the controller as a separate object even though it cannot be used in the original MVC/80 sense.

5.7 Domain Specific Languages

As seen in previous sections, dynamic languages can simplify UI implementations when compared to system languages. Beside all the simple patterns usable with C++ and Java, the 'Gang of Four' [Gamma et al., 1995] also presented the more complex interpreter pattern, which allows defining Domain Specific Languages. With dynamic languages it is possible to get the interpreter pattern for free.

General purpose languages, such as Java and C++, can be used for describing UIs. Unfortunately, their fixed syntax cannot be extended to suit other purposes, they do not scale beyond their natural syntactic elements. A Domain Specific Language (DSL) is a language that is designed to support the vocabulary, structure and semantics of a certain domain. DSLs are the opposite of general purpose programming languages. For example, computer game Unreal Tournament uses domain specific language called UnrealScript that is designed for third person shooters. Another example on DSL is Adobe Flash, which is optimized for UIs and multimedia.

A straightforward way to implement a DSL is to define a custom XML document format. On the other hand, XML is perhaps too verbose and baroque to some tastes. Implementing a DSL from scratch might seem a daunting task from first sight. Fortunately, parser generators exist and ease the task, for example ANTRL, CodeWorker, Flex and Bison. Parser generators typically require the definition of the language with BNF notation, for example, and implementation for the emitted language atoms.

User Interface Definition Languages (UIDL) are designed for describing UIs.
Many UIDs use XML, examples include the file formats used by Qt Designer and Glade. While a UIDL can be used for describing the structure of the UI, external programming language code is still needed to make the UI functional.

The border between UIDL and a programming language is hard to draw. A declarative UIDL without any code makes the separation clear, the UIDL defines the structure of the UI and the code defines the actions, but it is perhaps too limited as even the simplest of actions must be handled in code. If some event to action mapping or code is allowed in the UIDL, it will end up with language concepts and constructs. The problem is to define the scope; if some programming constructs will be implemented in the UIDL, where to draw the line as the very goal of a DSL is not to be a general purpose programming language?

Perhaps the easiest way to define a DSL is to use Lisp. Because Lisp uses uniform syntax for all of the language constructs and allows new syntactic constructs by using macros, all implementations using Lisp essentially implement DSLs. Because Lisp allows using the same syntax for code and data, there is no need to draw line between a declarative DSL, such as UIDL, and a generic programming language.

Lisp code can be naturally mixed in DSLs, provided that the DSLs are in Lisp; the problem of separation between UIDL and programming language does not exist in Lisp. In addition, a DSL implemented in Lisp does not need code generators, separate loaders or parsers. The DSL can be used with the existing Lisp interpreter.

```
<widget width="100" height="100">
  <shape type="circle" width="100" height="100">
    <fill pattern="solid" color="red"/>
  </shape>
  <action input="mouse.press" target="handle_mouse"/>
</widget>
```

```
(widget :width 100 :height 100
  (shape :type 'CIRCLE :width 100 :height 100
    (fill :pattern 'SOLID :color 'RED))
  (action :input #'mouse-press :target #'handle-mouse))
```

Listing 5.3: XML vs. Common Lisp. UI widget drawing a filled circle and routing mouse button press events to a handler function.

Lisp s-expressions are similar to XML expressions. Consider the listing 5.3,
which shows that XML and Lisp are isomorphic; Lisp and XML syntax are structurally equivalent. In this sense, using Lisp for a DSL is not far fetched. Conversely, programming constructs can be added to XML the same way they are in Lisp. Unfortunately, complex XML syntax quickly becomes unreadable. Rivest [1997] submitted an IETF draft for s-expression based file format, which was unfortunately rejected.

DSLs make embedded interpreters interesting. For example, often applications use configuration files to configure the application during startup. The Windows INI file format is popular for this purpose. Although INI files are straightforward to parse, a more powerful approach would be to implement a DSL for configuration files and use an embedded interpreter to parse the file. Most dynamic languages provide an eval feature which allows executing arbitrary language expressions.

As Lisp provides the eval feature and makes no difference between data and code, it would be trivial to add code into the configuration file, something which is not possible with INI files, for example. For configuration files, plug-ins and extensions an embeddable dynamic language is a good choice, for example LUA and Scheme.
6 CONCLUSION

While MVC/80 is not applicable as an application level model with modern UI toolkits, it is still relevant for widget level implementation. Many MVC adaptations deviate from the MVC/80 controller style by using a mediating controller. Such adaptations are closer to PAC or MVC++ than the MVC/80.

Conflicting MVC definitions lead to confusion. Learning about MVC/79 and MVC/80 after learning mediating MVC models first can only add to the confusion. Developers aspiring to understand MVC will hopefully find Fowler's [2006] take on UI architectures first. The complete history of MVC would make an interesting read. On the other hand, researching and writing such a book might not be a pleasant experience, as MVC has probably been distorted beyond repair.

The mediating controller should be avoided as it is typically a misapplication of the mediator pattern. Mediating between two objects adds unnecessary complexity and redundancy compared to the benefits. Removing the mediating controller simplifies application architecture and implementation without removing application functionality.

With model/view separation the benefits and drawbacks are more balanced. The separation is natural for some application domains, including centralized models which are accessed by multiple distributed UI clients. The separation can be counter-productive; in direct manipulated applications it is hard to separate model from view as the model contains view data. Visual Proxy and Naked Objects show that alternatives to model/view separation exists and should be considered, especially for implementing forms based applications.

Visual Proxy uses Java's inner classes and Naked Objects uses Java's reflection. Both Java features are inferior to what is available in dynamic languages. Inner classes are a limited form of lexical closure and Java's reflection is weak when compared to, for example, Ruby's metaclass programming. It should be possible to develop Visual Proxy and Naked Objects further by using dynamic languages.

Avoiding mediating controller and, in some cases, model/view separation can reduce complexity. Dynamic features can be used for reducing complexity further. For example, generic messaging and generic data passing simplifies class interfaces and reduces the work needed for messaging changes. Instead of emulating dynamic features in system languages, using dynamic languages should be considered as they provide such features by default.

Analyzing UI models by using the object-oriented model provided by Java and
C++ leads to diminishing returns. The result is the cataloguing of seemingly random collections design patterns. In the light of dynamic languages, continuing such analysis does not make sense as many Java and C++ patterns can be simplified or removed by using dynamic languages. Dynamic languages also allow going beyond simple design patterns, for example, by easing the use of Domain Specific Languages.

While using higher level languages allows simplifying implementations, the UI models do not magically become better. UI models seem to be collections of design patterns. Analyzing them and reasoning about them is hard because of omissions and ambiguities. Trying to develop a bottom-up axiomatic system for defining UI models could be a more fruitful approach.
REFERENCES


