Developing GUI Applications: Architectural Patterns Revisited

A Survey on MVC, HMVC, and PAC Patterns

Alexandros Karagkasidis

karagkasidis@gmail.com

Abstract. Developing large and complex GUI applications is a rather difficult task. Developers have to address various general software engineering and GUI-specific issues. To help the developers, a number of patterns have been proposed by the software community. At the architecture and higher design level, the Model-View-Controller (with its variants) and the Presentation-Abstraction-Control are two well-known patterns, which specify the structure of a GUI application. When applying these patterns in practice, a number of implementation-level issues arise, which are considered, however, mostly to an insufficient extent in the existing literature (if at all). So, the developers have to find their own solutions to these issues.

In this paper, we revisit the Model-View-Controller, Hierarchical Model-View-Controller, and Presentation-Abstraction-Control patterns. We first set up a general context in which these patterns are applied, thus identifying their place and role in the overall GUI development process. We identify then the implementation-level problems that usually arise when developing GUI applications and discuss how they can be addressed by these patterns, based on our own experience and investigation of the available literature.

We hope that this paper will help GUI developers to address the implementation issues, when applying these patterns.

1 Introduction

Large and complex graphical user interfaces (GUIs) intended for presenting, working with, and managing complex business data and processes are rather difficult to develop. A common problem is tackling system complexity. For instance, one could really get lost in large number of visual components comprising the GUI, not to mention the need to handle the user input and to track all the relationships between the GUI and the business logic components. As a consequence, the GUI code and the business logic may get deeply intertwined with each other.

Another important issue is code organization. A general requirement is that the source code be understandable, maintainable, and reusable. Several teams could be involved into the development of a large system, each responsible for its part. The teams need to communicate with each other when integrating the developed system parts. Even within a team, the developers must understand the code written by other team members. Next, many systems evolve over time, so the code should be easy to maintain and modify. Clear code organization and coding principles are essential in this regard.

Given these challenges, a (now) common way in software engineering is to start with defining a system architecture. Iterative refinement leads then to the lower-level design, providing the detailed structure of the system. Separation of concerns is a basic software engineering principle underlying this process.

With time, a number of patterns for GUI development have been proposed by the software community. The well-known Model-View-Controller (MVC) and, to a lesser extent, the Presentation-Abstraction-Control (PAC) patterns are usually applied at the higher design level (though, as we discuss below, they could actually be applied at different layers of abstraction). At the lower design level, many patterns from [GoF] can be used (actually, most of them stem from GUI applications or GUI toolkits, as it could be seen from numerous examples provided in the book). So, the developers do not have to start from scratch, as these patterns provide a sound basis when developing GUI applications.

MVC and PAC specify the structure of GUI applications by identifying the components that comprise a system and the way they interact with each other. However, when applying these patterns in practice, a number of issues arise at the implementation level. For instance, an essential constraint in this regard is the GUI platform (or toolkit) used. The GUI platform often assumes a specific way of how GUI applications are, or should be, developed (a *path of least resistance*), which may not comply with the pattern used. This should therefore be taken into account when applying a particular pattern.

In large and complex GUI applications other issues become apparent, such as how the GUI is constructed, how the user input is handled, and how the user-system dialog is controlled. These problems, however, are mostly not addressed (or addressed to an insufficient extent) in the existing literature on the patterns for GUI applications.

In this paper we revisit the well-known patterns - MVC, HMVC (Hierarchical MVC), and PAC – for GUI applications. In the first part of the paper, we set up a context in which these patterns are

applied, thus identifying their place and role in the overall GUI development process. We start with an overview of architectural approaches for GUI applications. We proceed with a brief description of the MVC, HMVC, and PAC patterns. After that, we identify common implementation-level problems in the GUI development the developers usually must address. In the second, main, part of the paper we discuss how these problems are (or could be) solved within each of the patterns. The discussion is illustrated with numerous examples .We also give some implementation specifics when using Java programming language and Java Swing toolkit. We conclude then with a brief summary.

Throughout the paper, we assume that the object-oriented approach is used.

2 Background

Figure 1 schematically illustrates (a part of) a general top-down GUI development process. There are two basic alternatives. The first one – the Smart UI approach – is not to have any system architecture at all and proceed immediately to the system implementation. Another alternative assumes a more systematic approach. Given system requirements, a system architecture is chosen first by system architects. During the design phase, the architecture is refined iteratively, and the detailed system structure is obtained. It is mapped then onto the implementation-level platform-specific constructs. The platform the system will be built upon is often a strategic decision. Hence, it is typically known at the beginning of a project.

The MVC, HMVC, and PAC patterns we discuss in this paper relate rather to the high-level design. So, these patterns are actually applied in a certain context, which is determined by the high-level architecture and, most likely, the implementation platform.

In this section we set up this context, thus identifying the place and the role of the patterns in the top-down approach to GUI development.

2.1 Architectures for GUI Applications

Smart UI Approach

Consider a calculator GUI application having about 20-25 buttons, a text field, a few menus, and some other widgets. We can implement it just in one single class, which contains all the logic related to GUI, user event handling, and performing mathematical operations.

In such simple applications developers can easily go without paying much attention to the system architecture and design. The main point here is that there is no need to complicate things unnecessarily. Such a one-class solution is a simple case of what is known as the Smart UI approach [Evans03].

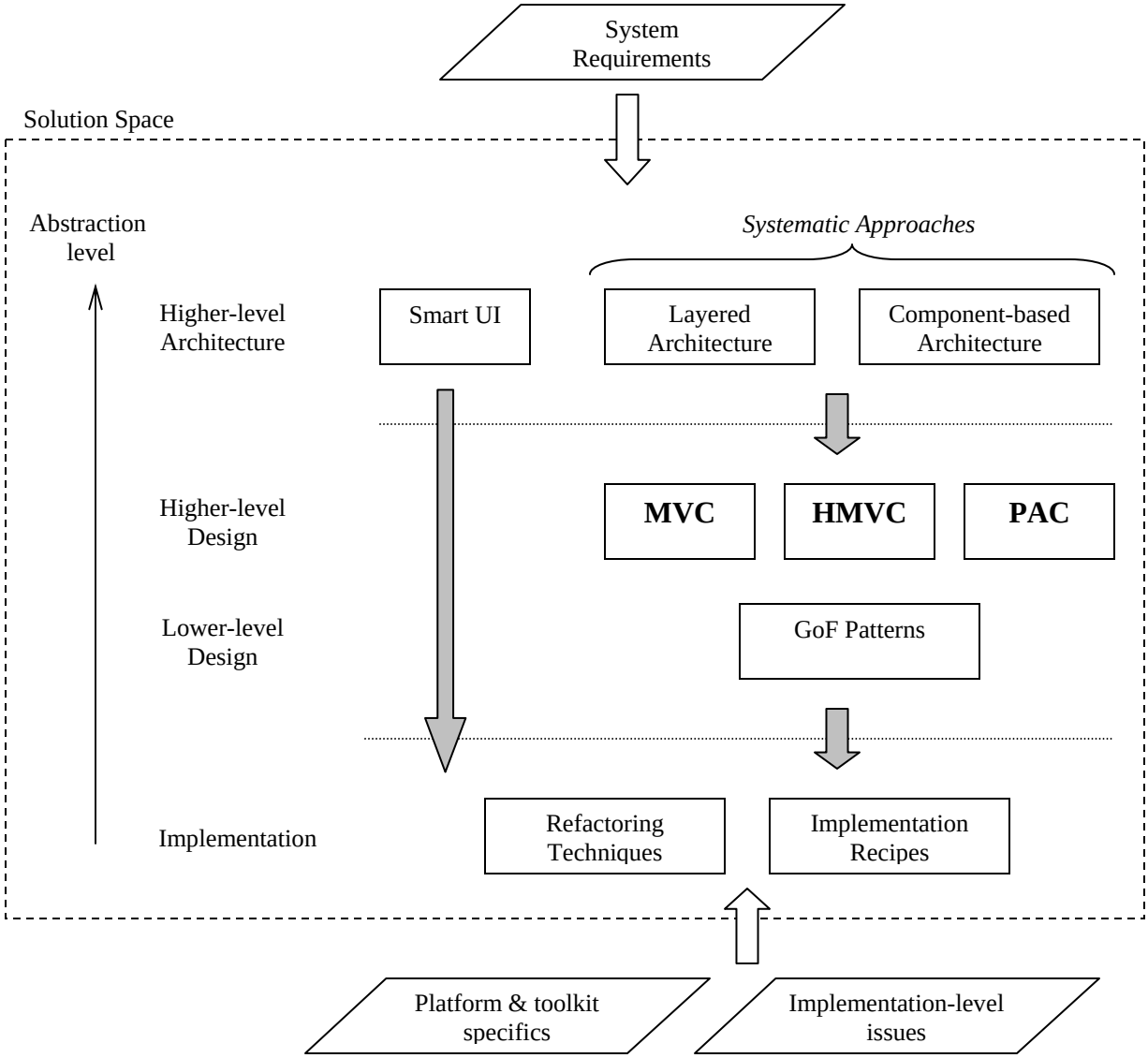

Figure 1. Top-down approach to GUI development. A Road-map.

The main characteristic of the Smart UI is mixing the business logic with the presentation (GUI) logic, like GUI creation and event handling. As Evans points out, “this happens, because it is the easiest way to makes things work, in the short run”.

The Smart UI is a straightforward way to implement GUIs, which is very simple to understand. It is widely applied by developers, especially when working with popular visual programming tools, which let the developers design the GUI with a kind of visual editor and automatically generate the source code, including the empty callback methods to handle the input events. The

developers often directly implement the required business logic in these callback methods. Such a programming style can also be found in many books and articles on GUI programming.

As an application gets more complex, a solution based on the Smart UI approach degrades quickly – in the absence of any clear system architecture and design, with intertwined business and GUI logic, the system and the code become difficult to understand, maintain, reuse, or modify (turning the Smart UI into the Smart UI 'anti-pattern' [Evans03]).

To make some improvements at the design level, several refactoring techniques could be applied within the Smart UI (the long gray arrow in the Figure 1), e.g., *Extract Method*, *Extract Panel*, or *Separate Domain from Presentation* ([Fowler99], [Marinilli06]). However, such a bottom-up approach to structure a system will also not work, to our opinion, for complex cases – it just postpones system degradation.

Separation of Concerns and Layered Architectures for GUI Applications

To overcome the limitations of the Smart UI approach, *separation of concerns* (or *functional decomposition*) is a common software design principle usually applied. At higher abstraction level this results in a layered system architecture, where each layer implements a set of related functionalities ([POSA I], [Evans03], [Marinilli06]).

The intention is to provide high cohesion within each layer, while making them loosely coupled with each other, so that the layers can be developed and evolve (more or less) independently from each other. A key aspect is that the business logic is completely separated from the GUI logic.

Several variations of the layered architecture for GUI applications can be found in the literature, which usually differ in the number of layers and their responsibilities ([Evans03], [Marinilli06], [Arch92]). We name here three key layers (from the GUI development viewpoint):

- **Presentation (Content)**: This layer is responsible for the visible part of an application, the GUI, comprised of a number of various visual components (widgets). It also implements primary (low-level) user input handling. (In some architectures, the event handling functionality is assigned to the dialog control layer)
- **Dialog Control (Interaction and Control)**: The definition of this layer varies among different approaches. In general, it manages the user-system interaction and glues together the presentation and the business logic layers. This includes forwarding of user events to the application, returning results to the GUI, and transforming between data types.
- **Business (Application) Logic**: This layer implements the business domain (business data and processes).

Figure 2. Key layers in a layered architecture for GUI applications

Additional layers may include various system services that do not belong to the business domain, like persistence, security, remote communication, and database access. (For the purposes of this paper, we assume that these services are hidden behind the business logic.)

In the literature, the dialog control layer is often omitted. However, it plays an important role separating the presentation from the business logic, which therefore could be completely decoupled and implemented separately from each other.

Inside Layers

Such a high-level architecture based on functional decomposition is very fine as the first step to start with, but the layers need to be refined further. We need to get the detailed (low-level) design of each layer, as well as how the layers interact with each other. For instance, the presentation (GUI) part is comprised of a large number of widgets that need to be assembled in windows and dialogs. Business data displayed in the GUI must be obtained from the business logic, while user input must be forwarded to the business logic.

And here is where the difficulties start…

Consider a typical scenario. A user wants to accomplish some (business) task. He/she selects the corresponding menu item, and an input dialog is displayed. The user types some data in and submits the dialog. The data is forwarded to the business logic, processed there, and the results are displayed to the user.

An implementation of such a scenario involves elements (objects) from each layer. Presentation layer objects display the dialog and process the user input. Upon submitting the dialog, the data the user has typed in is passed to a dialog control object, which transforms the data into a format specific to the business logic. The transformed data is passed then as method call parameters to the business object(s) responsible for the business task the user has run. The results are returned to the dialog control object, which updates the visual state of the related presentation objects, thus reflecting the results to the user. And all this for a single scenario only! What if you have dozens or even hundreds of such scenarios involving objects from different layers?

As a result, we’ll get an intensive inter-layer interaction. The layers get tightly coupled with each other and, therefore, more difficult to develop.

Components and Component-based Architecture

Another architectural approach is based on the observation that user-system interaction scenarios often represent various operations on business objects. From this viewpoint, the basic idea is to have a separate component (or module, unit, subsystem, etc.) for each business object that implements all the use cases related to that business object. Therefore, such a component is responsible not only for the implementation of the business object itself (business data and business operations), but also for all the issues within the related use cases, that is, presentation, user input handling, dialog control, etc.

The components are used then as building blocks to construct the whole application. They are integrated within an (global) application framework (yet to be developed), which usually provides application-level functions, such as menus and toolbar, and services, as well as enables interaction among the components.

Figure 3 illustrates an example of such an architecture for a fictitious university information system. Each particular component implements the corresponding business object and the related scenarios (use cases), including all the presentation aspects, such as GUI components and user input handling. The Lecture component, for instance, handles the Lecture business object and provides a number of dialogs to create, edit, search for, or delete lectures.

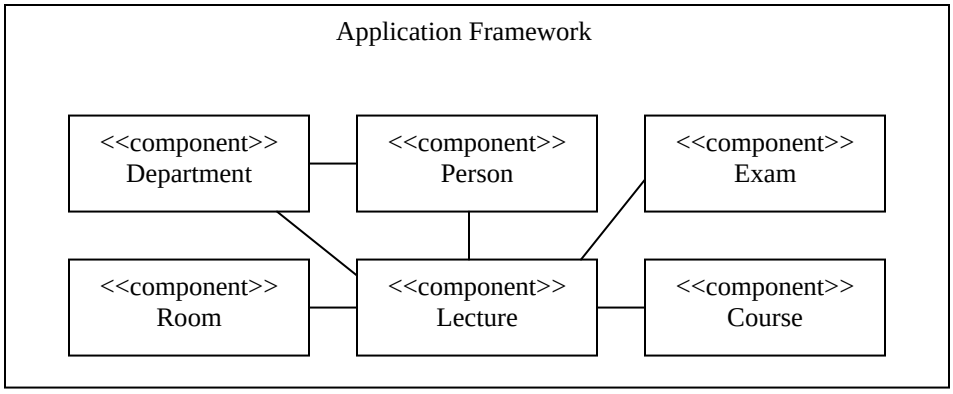

Figure 3. Component-based approach to GUI development:
The university information system example.

[Marinilli06] uses the notion of *composable units* to denote the components from which a GUI application is built. They can be viewed as autonomous entities (‘mini GUIs’) that handle all the related aspects – presentation, dialog control, and business logic – with respect to the concepts from business domain.

[Holub99] goes further with these ideas and proposes the *visual proxy* architecture, which is based on pure OOP approach assuming that the functionality be assigned to the object that possesses the knowledge how to perform it. Namely, it is stated that since a business object needs to be displayed, it and only it knows how to do that and, therefore, it should be responsible

for displaying itself. Hence, we put the business and presentation logic into one place (the object), which seems to be contradicting the separation of concerns principle.

A kind of component-based architecture for GUI applications is proposed in [HO07]. Here, the components are various dialogs within the user-system interaction, which are managed by the Dialog Framework (Container) - a run-time environment for dialogs.

As with the layered architecture, the low-level design for each component should be provided then. Here, we can apply the separation of concerns principle as well, but now at the component level. Figure 4 illustrates a possible component structure.

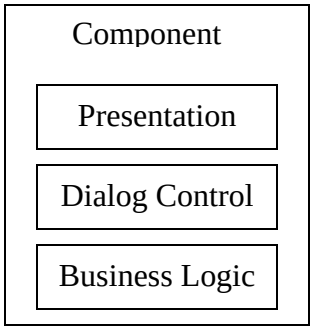

Figure 4. Applying the separation of concerns principle at the component level. Component inner structure

The major problem with this approach (especially regarding large and complex GUIs) is that in reality the components (i.e., the application building blocks) are not fully autonomous, but rather interact with each other. State changes in one component may imply changes in state or behaviour of other components. Therefore, a communication mechanism should be designed within the application framework that enables component interaction.

2.2 Patterns for GUI Applications

For both high-level architectural approaches, we'll get a network of interacting objects when coming down to the low-level design (see Figures 5a and 5b). In a large application, such a network would be comprised of probably hundreds of objects.

As mentioned above, implementation of each use case implies tight cooperation among a number of presentation, dialog control, and business objects. The more features a GUI application provides, the tighter the relationships among objects are. For instance, features such as drag and drop or enabling/disabling of various GUI components upon certain actions or changes in the application state require additional implementation effort as regards object interaction.

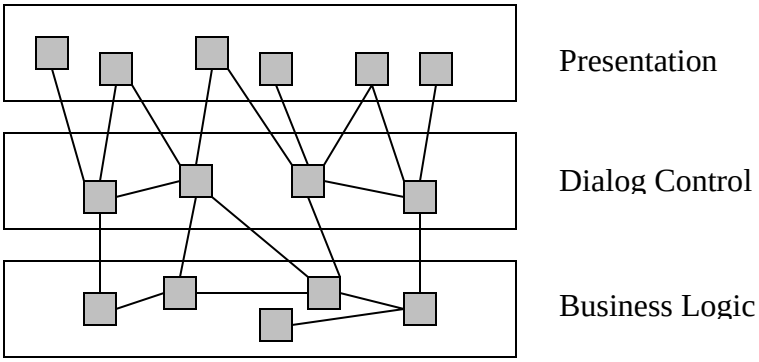

Figure 5a. Object network in the layered architecture

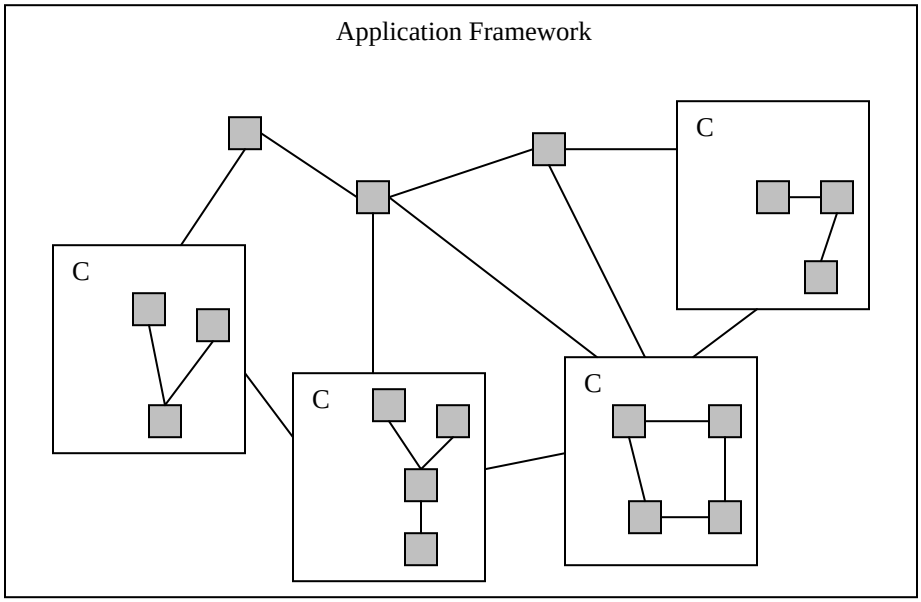

Figure 5b. Object network in the component-based architecture

We consider this as a fundamental characteristic of GUI applications from the software design point of view, making the implementation of large and complex GUIs rather a difficult task.

Typical questions in this regard would be:

- How to get such a network of objects? How to assign specific functionalities to each object and design their interaction?
- How to map objects onto implementation-level constructs (such as GUI toolkit widgets or Java classes)?
- How to initialize the object network, instantiating particular objects and establishing relationships among them, i.e., initializing references (this is known as the *visibility problem* [Marinilli06])?

However, we do not have to start from scratch when tackling these issues, since these seem to be in the realm of the patterns for GUI applications, such as Model-View-Controller or Presentation-Abstraction-Control (and their variations).

Here, we briefly review the patterns that are discussed in this paper. A detailed description could be found, e.g., in [POSA I] (see also the References section).

Model-View-Controller

Model-View-Controller (MVC) is probably the most popular and frequently cited pattern stemming from the Smalltalk environment ([POSA I], [KP88], [Burbeck92]). In MVC, a GUI application is built from components (objects) of three types: models, views, and controllers. The model represents the application logic – functions and data. The view displays the model data to the user (there can be several views connected to the same model component). The control handles user input and forwards it to the model by calling the corresponding function(s). The model processes the input and changes the state. Such changes need to be communicated to the views depending on the model, as well as to the controllers, since the way they handle user input may also depend on the system state (e.g., enabling/disabling buttons or menu items in certain states). For this purpose, the Observer pattern is used. The views and controllers register themselves with the model. Upon change, the model notifies all the registered components, which then retrieve the required data from the model.

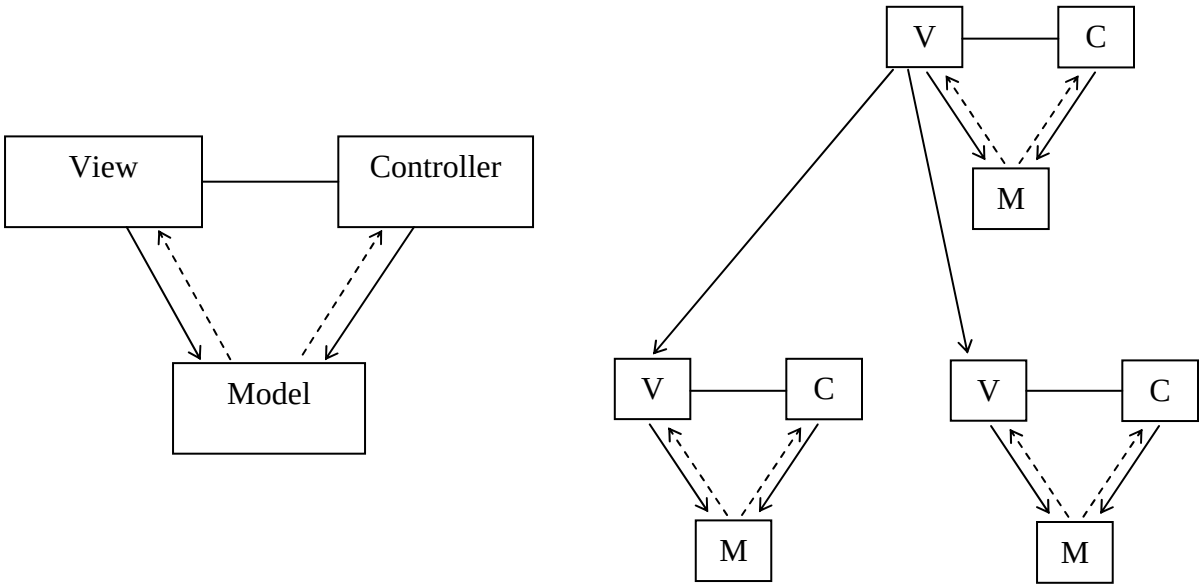

Figure 6. MVC and a hierarchy of MVC-triads

As it could be seen, views and controllers belong to the presentation layer, while models – to the business logic. The dialog control layer is not present explicitly in MVC.

This is basic MVC. The whole application is built as a hierarchy of MVC-triads organized around the view components, i.e., this hierarchy reflects the hierarchy of visual components (main window, menu bar, panels, widgets) within GUI [Burbeck92].

Hierarchical Model-View-Controller

Hierarchical Model-View-Controller (HMVC) is an extension of basic MVC [CKP00]. A GUI application is modeled as a hierarchy of MVC-triads connected through the controllers.

There are some essential differences as compared to MVC. First, user input is handled now in the view component, which forwards it to its controller. The controller, in its turn, transforms user events into method calls on the model. Upon a state change, the model updates the view supplying it with the new state to be displayed (recall that in MVC the model only notifies its views upon state change, and the views retrieve then the data).

Another role of the controller is to enable communication with other MVC-triads. Such a communication is achieved through a hierarchy of controllers, thus providing the dialog control logic.

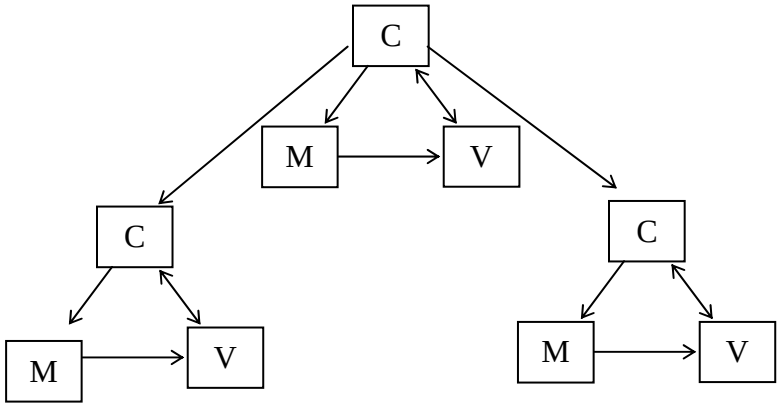

Figure 7. HMVC

Presentation-Abstraction-Control

Presentation-Abstraction-Control (PAC) organizes a GUI application as a hierarchy of cooperating agents ([POSA I], [Coutaz87]).

A typical (usually, bottom-level) agent represents a ‘self-contained semantic concept’ [POSA I] and is comprised of three components responsible for various aspects of the agent (that is, of the corresponding semantic concept). The presentation component is responsible for interaction with the user: It displays data and handles user input. The abstraction component represents agent-specific data. The control component connects the agent’s presentation and abstraction parts with each other and is responsible for communication with other agents. When an agent wants to send an event to another agent, it forwards it its parent (intermediate-level) agent. The parent agent either sends the event to one of its other children, or it forwards the event to its parent, if it does not know what to do with it, and so on.

Intermediate-level agents serve two purposes. First, it is composition of lower-level agents. This is needed when we have a complex semantic concept represented by other concepts. The latter are implemented then as lower-level agents, while the former – as an intermediate-level agent. Second, it is communication among agents, as described above.

The top-level agent implements the business logic and is responsible for the application-level GUI parts, such as menus or tool bar. It also coordinates all other agents.

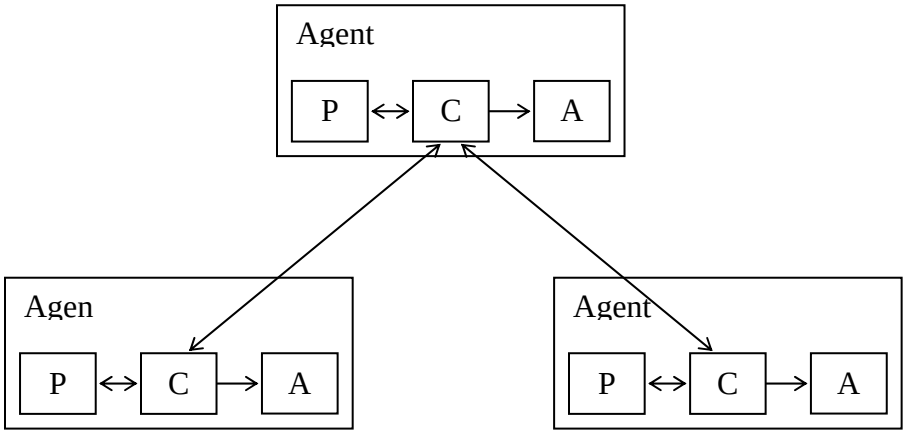

Figure 8. PAC

2.3 Common Implementation-level Problems in GUI Development

The patterns specify the structure of a GUI application. They identify the components that comprise the system, their responsibilities and how the components interact with each other. However, when applying these patterns in practice, a number of issues arise at the implementation level, especially when developing large and complex applications. In this regard, an essential constraint at the implementation level is the GUI platform or toolkit used, which assumes a specific way of how a GUI application is developed. This should be taken into account when applying a particular pattern.

Here, we give a brief description of these problems:

- **GUI (Content) creation and assembly**: A GUI is comprised of a number of visual components, or widgets, which are usually organized in a hierarchical manner into panels, menus and/or other containers within the GUI main window or dialogs. Main problems here are how widgets are instantiated (in particular, regarding their relationships with other objects) and how they are assembled into the GUI (this especially relates to large GUIs).
- **User input handling**: The GUI provides to the user various means for input. Primary user input handling is usually implemented by means of callback methods, which are called when the corresponding event occurs. In large and complex GUIs, with dozens of widgets, event handling code may become rather large. Yet user events must be forwarded to the business logic for application-specific processing, which implies additional design issues.
- **Dialog control**: Dialog control is about managing user-system interaction scenarios, which may be rather complex. Roughly speaking, at each step the user provides some input (through the GUI), which is processed in the business logic, and the results are displayed (again, through the GUI). From the implementation viewpoint, this involves interaction among a number of GUI and business objects, and the problem is to design and control such interactions.
- **Business Logic**: Eventually, the GUI must be coupled (through the dialog control layer) with the business logic. The way the business logic is designed is important with respect to how this coupling could be implemented.

The available literature on the patterns for GUI applications does not provide enough information (if any) on these problems leaving many questions open. So, the developers have to find their own solutions.

In the second part of the paper, we give a detailed consideration of these problems and discuss how these problems are (or could be) addressed by the MVC, HMVC, and PAC patterns, respectively. This work is based on our own experience and on the analysis of the existing literature on the patterns for GUI applications and GUI development.

3 GUI Creation and Assembly

First, we give some basic ideas on the essence of GUI, which we use in the further discussion.

Typically, the GUI part of an application is represented by a main window, and a number of dependent secondary windows (basically, for output purposes) and dialogs (for user input).

Windows and dialogs are comprised of various visual components (widgets) that are organized into visual containers, such as panels, menus, tool bars, etc.

In this respect, each window or dialog can be seen as a hierarchy of visual components, which suggests applying the Composite design pattern [GoF].

In a typical GUI, the main window would contain the following areas ([CKP00], [Marinilli06]):

- Main working area (e.g., a drawing pane)
- Navigation (or Selection) area (e.g., a tree-based browser of some entities)
- Menu bar
- Tool bar
- Status bar

Visual components are usually standard widgets provided by GUI toolkit, which determines the toolkit-specific way of how they could be used, such as how the widgets are created and assembled then into visual containers and, eventually, into the whole GUI.

Toolkit specifics are the major implementation-level constraint, which has certain implications.

For instance, standard widget set of a particular toolkit is sometimes not enough to solve some specific problem. This means that we have either to extend standard widget functionality (this is known as *widget customization*) or to implement new visual components (*ad-hoc* components) [Marinilli06].

Another issues will be discussed below.

We start our consideration with the Smart UI approach. This would help us to identify some issues that are relevant to other patterns as well. We proceed then with the discussion of how the well-known GUI patterns address the problem of GUI creation.

3.1 GUI Creation in the Smart UI Approach

A typical way (especially for newcomers) is to implement the main window in a single class (one-class solution), that instantiates all the widgets of the main window, initializes them, composes into containers, and eventually assembles the whole GUI from that containers.

With this, the main window class gets very quickly a real monster, even if we take into account only the GUI creation code (without event handling or business logic, which the main window class would inevitably contain with the Smart UI approach used)

Let's see what the GUI creation code within the main window class consists of. First, we would have a class instance variable for each widget. Then, each widget must be instantiated and its properties must be set up. For instance, a menu item could be given a name, an accelerator key, a mnemonic key, an icon, and a tip text, and its state could be set to enabled or disabled. All this requires several lines of code (of course, setting up widget properties could be done often through various constructors, but to make code more readable it is better to set up each property explicitly, e.g., through specific setter methods). The following code illustrates possible implementation in Java Swing. It is rather straightforward, and the property values would most likely be obtained from the configuration file, but it is not the point here.

Listing 1: Initializing a menu item

```
// JMenuItem saveMenuItem;
...
saveMenuItem = new JMenuItem("Save");
saveMenuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                                   ActionEvent.CTRL_MASK));

saveMenuItem.setMnemonic(KeyEvent.VK_S);
saveMenuItem.setIcon(new ImageIcon("resources/images/save.gif"));
saveMenuItem.setToolTipText("Save");
saveMenuItem.setEnabled(false);
...
```

At last, the main window class has to establish all the relationships among widgets within the visual component hierarchy, i.e., assemble the GUI. For instance, all the menu items must be added to their menus, which themselves must then be added to the menu bar.

Imagine now how the code of the main window class would look like just with a hundred of menu items.

So, the size of the GUI creation code depends directly from the number of widgets within the main window. And as it seems, the menu bar and the tool bar (to a lesser degree) are the main contributors with this respect.

The code size and code organization problem can be solved by a number of well-known refactoring techniques ([Fowler99], [Marinilli06]).

With the Extract Method refactoring technique, we can organize the code within the main window class, so that, e.g., the menu bar and all its content are completely created within a separate method. If this method also becomes large, we can extract from it methods that create particular menus within the menu bar.

Listing 2: Extract Method

```
public class MyMainWindow extends JFrame {
 ...
   private void createMenuBar(){
      ...
      this.createFileMenu();
      ...// create other menus
    }

   private void createFileMenu(){
      // code that creates the File Menu
    }
 ... // methods creating other menus
}
```

Such code organization improves the readability and understandability of the source code. However, the code size is not reduced. And besides the GUI creation code, we have other logic in the main window as well, such as event handling and business logic, which may work with the widgets. If you change something related to a specific widget, like its name, you need to rut through the whole code and find all places where this widget is used.

These problems can be solved then by applying the Extract Class [Fowler99] or Extract Panel [Marinilli06] refactorings. With this technique, we extract into a separate class all the code related, e.g., to the menu bar, to the tool bar, or to a specific panel or other container within the main window.

3.2 Typical Problems

The above discussion leads us to the following general questions that must be answered when discussing how the patterns address the GUI creation problem:

- How each particular widget is created and who creates it?
- Who assembles the whole GUI, i.e., who establishes the relationships among widgets within the visual hierarchy?

A GUI toolkit as the main implementation-level constraint implies another important issue.

The point here is that each pattern describes a solution to a specific problem in terms of interacting objects, each having certain responsibilities within the solution. In the GUI patterns, we have objects that are responsible for the GUI concerns, in particular, for GUI creation. But eventually we have to implement these objects with the GUI toolkit used. Consequently, this might require that we extend the standard functionality provided by the toolkit with the pattern-specific functionality. Technically, this means that we might need to extend the standard toolkit widget classes. In this case, extended widgets would be used in a pattern-specific way, but not in standard way as intended by the toolkit and as expected by the developers.

Given these considerations, we start our discussion.

3.3 GUI Creation in MVC and HMVC

In MVC, GUI (presentation) issues are handled in the view component. Logically, it is responsible for presenting application (business) data to the user.

From the implementation point of view, however, the things are not so clear.
The main question is: How should a specific view be implemented with a given toolkit?

Widget-level MVC

The first approach originates from the Smalltalk environment and is used now in other toolkits, such as Java Swing.

Here, a view is a visual component (widget) from the toolkit. More precisely, each toolkit widget is considered as a view component. I.e., MVC is applied at the widget level resulting in an MVC-based framework [POSA I]. Each widget has then its corresponding model and controller components.

The Smalltalk environment provides predefined view, model, and controller classes for standard GUI widgets, like buttons, menu items, text fields, etc. ([KP88], [Burbeck92]). Java Swing is also an MVC-based toolkit: Each widget has its standard model and a controller classes. Together, they form a simple MVC triad.

The application is created then by instantiating such MVC triads for each widget in the GUI and assembling them together. This approach seems to work rather well, if an MVC-based toolkit is used. We consider here potential problems that might arise when developing large and complex GUIs.

As in the Smart UI case, it is the single main window class problem, if the whole GUI creation work is done in one class. Again, a solution would be to apply the Extract Method and Extract Class refactorings (see also the discussion above).

Another problem concerns the model component associated with each widget. We consider this on the Java Swing example.

In MVC, business data displayed to the user is handled in the model component, which uses its own data type(s) to represent this data internally. On the other hand, the view uses in general another (usually toolkit-specific) data type(s) to represent the data it displays. So, we need a data type transformation.

In Swing, each widget has its corresponding standard Swing model class, which maintains data displayed by the widget, i.e., the widget's state. However, this data is kept in a Swing-specific format. This implies the following design problem. What if you want to implement your business data using other data types? In that case, the standard model class provided by Swing cannot be used directly.

For instance, `JTree` component in Java Swing requires a class that implements the `TreeModel` interface as the associated model. The model class represents thus a hierarchical structure, which is, however, Swing-specific. And you'd most likely not want to model your business objects and relationships between them (which you want to display with the `JTree`) by implementing the `TreeModel` interface, but rather choose your own data structures.

So, how should we connect then the widget to the corresponding business data?

One way is to extend the widget class to associate it with our own model class that represents business data in its own toolkit-independent format. The standard model class provided by Swing is no needed then. The extended widget retrieves the data to be displayed from the model and transforms it into its own format. The model has still to implement change notification mechanism (through the Observer pattern).

Another alternative would be to extend the Swing model class to connect it to our specific business object class. In this case, the extended model class can also implement the required data type transformation.

In both cases, the developers have to extend the standard Swing toolkit functionality.

Consider now the situation when the toolkit does not support MVC, and we are going to apply the MVC pattern at the widget level, as described above.

In this case, we would have to extend toolkit widget classes to provide the MVC-specific functionality, such as connection to the controller and model components (see also the MVC liabilities section in [POSA I]).

This has some important implications for large and complex GUIs. First, for each widget in the GUI, we would have a separate class extended from the corresponding toolkit widget class. Eventually, we get a large number of new classes in our application (even for simple widgets, such as buttons or menu items), which need to be maintained. Second, the developers have to use the extended widgets in the MVC-specific way, which may differ from that suggested by the toolkit and the developers are familiar with. This means for the developers a new way of thinking about the widgets.

The main window assembly problem is present here as well.

Concluding, we can say that for both alternatives we face the problem of being forced to extend standard toolkit functionality through implementing new classes in order to provide 'true' MVC nature of our application. This applies especially to the toolkits that do not support MVC. Another problem is that applying the MVC pattern for each widget is rather a low-level and fine-grained approach, with too many pattern-specific details. This makes things unnecessarily complex (see also the MVC liabilities section in [POSA I]) and might discourage the developers (especially if a large GUI application is developed).

Container-level MVC. HMVC Approach

Another approach tries to avoid these problems. The basic idea here is to move from the level of particular widgets to higher levels in the visual component hierarchy representing the main window or a dialog. I.e., MVC is applied at the level of visual containers – parts within GUI that group together other (usually related) visual components.

Thus, the views are now specific containers within GUI, such as menu bar, tool bar, panels, and dialogs. Each such container-level view is implemented now in a separate class (recall the Extract Class refactoring technique!). It is responsible for handling its content, such as creating, initializing, and assembling together the visual components it contains, as well as retrieving business data from the model, which is displayed by its child visual components. Hence, the container-level view acts as a Façade for its child visual components.

A container-level view can be implemented in a straightforward way through extending standard class of the corresponding toolkit container widget and adding the required functionality. And the key point here is that all the GUI-specific work within the widget-level view is done in a standard toolkit way. Thus we avoid all the low-level details and complexity of applying MVC for each widget. This is particularly important for toolkits that do not support MVC. We have no more to treat each particular widget in the MVC way and create (a large number of) new classes for this purpose. The developers may work now with the toolkit as they used to. Only the toolkit container classes must be adapted to MVC.

Basic ideas of this approach are given in [CKP00], where main GUI parts, such as menu pane, navigation pane, main content pane (working area), and status pane, are modeled as views. These views are assembled then within the main window, which is the root view component.

The main design issue with the container-level views approach is the granularity level, i.e., which container should be modeled as a separate view. In the hierarchy of visual components representing the structure of the main window there could be several layers of intermediate containers, depending on the GUI complexity. For instance, menu bar (container) is usually comprised of several menus (containers), where each menu consists of either menu items (leaves) or other submenus (containers), and so on. Complex forms are often built from a number of panes (e.g., tab panes), where each pane may have its own sub-panes.

Another aspect is that sometimes we have to extend the functionality of certain widgets to provide some specific behaviour not supported by the toolkit used. Naturally, we would have a separate class for such a widget. We can then regard it as a separate view component and provide the corresponding model and controller components. Alternatively, we can treat it as a new “standard” widget just as other child widgets within its container.

So, we can put a more general question: Which visual component in the hierarchy should be modeled as a separate view?

The following two rules of thumb could be suggested in this regard:

- If a child container within a given container is complex (contains many widgets), then provide a separate view for that child container.
- If a widget (visual component) within a container must provide some specific, usually domain-related, functionality, then provide a separate view for this widget.

We are not going to give concrete numbers, but just illustrate the idea on a couple of examples.

Consider first a menu bar. If is has two or three menus, each consisting of a couple of menu items, then we could model the whole menu bar as a single view component. If, however, some menu has, say, more than 7 menu items, it would be a candidate for a separate view. The same applies then to each particular menu. Until a menu remains simple, it is modeled as one view. When it becomes more complex, with large submenus added, then it is time to think about extracting submenus into separate views.

The key point here is to find a balance between the one-class solution and applying MVC for each particular widget. With this, we are trying to avoid having a single monster class that implements everything (one extreme) and too many low-level details and unnecessary complex design for simple things (the other extreme).

Consider now a navigation pane with two navigation trees. It could be implemented as a tabbed pane, with two tabs each containing a tree component. Though we have few widgets here, we would rather have a separate view for each tree, because each implements a specific behaviour, and we must customize the tree widget provided by toolkit. Another view would be the navigation pane itself.

The first rule-of-thumb does not work always. Consider a pane comprised of, say, 20 child panes, each having few (3-5) simple widgets like buttons and text fields. For instance, this could be a pane representing various parameters of some physical process, with a couple of controls to regulate each parameter. So, what to do in this case? The whole pane having dozens of widgets is rather large to put all its content into just one view (i.e., class), but each child pane is very simple to be assigned a separate view.

3.4 GUI Creation in PAC

The way the GUI creation problem is addressed in PAC is logically quite different from, and more complicated as, that of MVC. So, we provide first some details on PAC from [POSA I], before we discuss the implementation issues.

First, an application is organized as a hierarchy of agents. Each agent has the presentation component, which handles, roughly speaking, agent's GUI aspects (input and output) and interacts with the agent's control component.

What the presentation component actually is depends on the agent's type.

The presentation of the top-level agent '*includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars…*' [POSA I]. The presentation of the bottom-level agents '*presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it*' [POSA I]. There are also intermediate-level agents. Some of them represent complex abstractions (concepts) from the business domain, which are comprised of other abstractions. In this case, the presentation of an intermediate-level agent handles the visual aspects of the corresponding compound abstraction.

Well, a little bit complicated as compared to MVC…

An important aspect here is that the design of the whole application, and its GUI part in particular, is driven by agents. Roughly speaking, we take a business object from the business model and then devise how it should be displayed to the user. This differs from MVC and HMVC where the view components represent widgets or containers within the main window, i.e., the application design is driven rather by the GUI.

More insight on what is hidden behind the presentation component could be from the following statement: '*… All components that provide the user interface of an agent, such as graphical images presented to the user, presentation-specific data like screen coordinates, or menus, windows, and dialogs form the presentation part*' [POSA I].

This helps in reasoning about the implementation issues of the presentation component. First, for a simple agent, its presentation part could be implemented as a single class, which handles output, user input, and interaction with the agent's control component. This class could be derived from toolkit widget class, if that widget is suitable to handle agent's visualization.

This is very similar to applying MVC at the widget level. Hence, if we have a large number of simple agents, we could get the same problems, as in MVC – unnecessary complex design with too many low-level details. This problem is discussed in the liabilities section of the PAC pattern in [POSA I], and a good example is provided (a graphical editor where each graphical object is implemented as an agent).

A potential solution would be to choose the appropriate granularity level of the agents to control their number. With this, the concepts of the business domain implemented by simple agents are handled within more complex agents. (For the [POSA I] example, we would have an agent for the graphical editor only, but not for each graphical object. The editor agent handles then all the graphical objects internally.)

Note that this solution is similar to the container-level MVC and HMVC, where a container-view handles all its child widgets, instead of having an MVC triad for each particular widget. The difference is that in MVC and HMVC we deal with the hierarchy of visual components, while in PAC it is the hierarchy of agents.

For a complex agent, the presentation part, as mentioned above, may contain menus, window, dialogs, etc., which could be implemented as a set of (interacting) visual components. For the interaction with the outside world through the agent's control component, the Façade pattern

would be a nice solution [POSA I]. The Façade object hides then the internals of the presentation part from the control.

An essential consequence of this design solution is that it allows the developers to implement the visual components in a typical toolkit-specific way, using standard toolkit widgets, which might need to be customized to meet some domain-specific requirements. But we do not have to implement any PAC-related functionality in the visual components (recall how containers-level views in MVC and HMVC are implemented!).

As so, to implement the visual components that comprise agent's presentation we can use the ideas discussed above. In particular, a complex visual component (container) is implemented in separate class, while its children are used 'as is', i.e., as standard toolkit widgets. More complex components might require several classes. For instance, in Java Swing, a customized rendering of tree nodes in a `JTree` component would require a separate class implementing the `TreeCellRenderer` interface, while the data displayed by the tree is held in a class that implements the `TreeModel` interface.

To illustrate these ideas, consider a graphical editor example with an agent representing the editor's drawing pane. Suppose that the agent's presentation part includes the drawing pane itself, various dialogs and popup menus, and a tool palette. Each of these visual components would be implemented then in a separate class (probably with a number of additional helper classes). We would also have the Façade class. Another alternative would be to have only the drawing pane in the presentation, while the tool palette, dialogs, and popup menus are implemented as child agents of the drawing pane agent. Note that none of the visual components implements any really PAC-specific functionality.

Another potential problem may arise in the top-level agent. As stated above, its presentation part is responsible for menu bar. Again, if menus comprising the menu bar have many items, this would result in a very large presentation component. A potential solution is to implement menus as separate agents, which is equivalent to the Extract Class refactoring technique discussed above. However, additional agents imply additional interaction issues, which are discussed below.

As it can be seen, PAC specifies how and where GUI parts are created. Now we need to assemble them. This issue is not addressed by PAC as given in [POSA I]. In general, agents should be extended to provide this functionality. For instance, a parent agent may require its children for visual components from their presentation parts to construct its part of GUI.

The major problem here is that the PAC agent hierarchy does not fully match the hierarchy of visual components. Consider again the drawing editor example. Suppose the tool palette is implemented as a child agent of the drawing pane agent, but is presented visually as a tool bar, which is a child visual component of the main window and should be initialized therefore by the top-level agent. For this purpose, the top-level agent must require the drawing pane agent for the tool palette. The drawing pane agent, in its turn, forwards the request further to the tool palette agent, its child. All this just increases coupling among agents and their complexity.

4 User Input Handling

The user interacts with the system through the GUI using various input devices. Events from these devices pass through the operating system and are delivered to the application. Typically, user input events are handled in the corresponding callback methods provided by the application. The details of how this is done are toolkit-specific.

After a callback method has intercepted a user event, it must be processed in application-specific way. The corresponding logic will most likely involve the business and/or the presentation layer components and can be rather complex, and keeping it in the callback methods (the Smart UI approach) would violate the separation of concerns principle. So, the basic role of the callback methods is to intercept user events and forward them to other parts of application for further processing.

In large GUIs, with dozens of widgets providing to the user rich interaction features, the event handling code in callback methods can become rather large, even if its only task is to intercept and forward events. Moreover, we need to couple this code with those parts of the application that provide application-specific processing of events. A proper design and organization of the user input handling code is therefore an important task.

Let’s see how the patterns address this issue.

4.1 User Input Handling in MVC and HMVC

In MVC, user input is handled by the controller component, which translates user input events into the method calls on the model or view components.

As with the GUI creation and assembly problem, concrete implementation depends on at what level MVC is applied.

Consider first the widget level MVC approach. The controller component would handle then the events on the corresponding widget (the view). We create first the widget, then its controller, and associate the controller with the widget. From the implementation viewpoint, we have to implement event handling callback methods in the controller class and register these event handlers with the widget. This is rather straightforward to implement in many toolkits and platforms (like Java Swing and SWT, or .NET Windows Forms in C#, or Qt), if you write the code yourself and do not work with an IDE or a visual programming tool, where you’ll get the event handling code automatically generated. Modifying this code to get pure MVC design would be a bad idea.

As for the GUI creation, we get similar problems when applying MVC at the widget level. The main issue is that having a separate class for each controller results in a large number of small classes each providing probably only one or a couple of callback method. This complicates the design unnecessarily. Again, the main source of such controller classes would be menu items, tool bar buttons, and complex dialogs comprised of a large number of simple widgets.

Therefore, we find it more reasonable to shift again to the layer of visual containers, as it is done for the GUI creation problem. There, specific visual containers within the GUI are modeled as container-level views. Now we define the controller component for each such view. Such a controller handles all the events that occur on the visual container itself, as well as on all its child widgets. Thus, we avoid having a separate controller class for each particular widget.

This approach is used also in HMVC [CKP00]. The difference is that in HMVC user input handling is assigned to the view component (presumably stemming from the PAC pattern), while the controller component of an MVC-triad is responsible for communication with other MVC-triads.

Thus, we get two slightly different approaches. Pure MVC separates the presentation from the user input handling, resulting in two different logical components (the view and the controller), while HMVC handles both issues in one component (the view).

Let's consider now how a container-level controller could be implemented in Java Swing. A common way in Swing to handle a user input event on a widget is to provide a class that implements the corresponding `Listener` interface, where the event handling callback methods are declared. This class is registered then as an event handler with the widget by calling the corresponding `addXXXListener(XXXListener l)` method on that widget.

Suppose we are implementing event handling for a dialog with some text fields and buttons.

Consider first pure MVC approach. We get, therefore, a separate controller. The first implementation alternative is when the controller class implements the required `Listener` interfaces, which would be in our case `ActionListener` and `KeyListener` (if we want to handle each particular keyboard input event on text fields), as shown in Listing 3.

The next step is to register the controller as listener on all the widgets of the dialog class. Here, we have two options, depending on who performs the registering. An easier way is to pass an object of the controller class to the dialog, which then registers that controller object as event listener on all its widgets (Listing 4).

The second option is to perform the registration in the controller class, which implies the controller to have access to the dialog's widgets. As the widgets are class members in the dialog class, at least package access must be provided. If we want to keep them private, we should provide then the corresponding getter methods in the dialog class. So, the first alternative looks more elegant.

Note that the controller is registered as listener on several widgets, which may produce the same types of events (e.g., both buttons and text fields in the example above fire `ActionEvent`s). Hence, the controller must be able to find out in its callback methods on which widget a particular event has occurred. This could be done with the `getSource()` method called on the event passed as a parameter to the callback method. The `getSource()` method returns an instance of the Object class, which is compared then to each widget of interest to find out the

event source. With this, we get conditional if-else-…-else blocks within callback methods, which is not a very nice way of programming. But it also means that the controller class still needs to have access to the widgets of the dialog class!

Listing 3: Controller class implementing Listener interfaces

```
class MyDialogController implements ActionListener, KeyListener {
 ...

  // ActionListener interface
  public void actionPerformed(ActionEvent ae){
     // event processing code
     // Note! We must identify here on which widget the event has occurred!
   }

  // KeyListener interface
  public void keyPressed(KeyEvent ke){
     // event processing code
   }

  public void keyReleased(KeyEvent ke){
     // event processing code
   }

  public void keyTyped(KeyEvent ke){
     // event processing code
   }
 ...
}
```

Listing 4: Registering controller on widgets

```
class MyDialog extends JDialog {
 ...
  public void registerController(MyDialogController controller){
    ...
     this.myButton.addActionListener(controller);
     this.myOtherButton.addActionListener(controller);
     this.myTextField.addActionListener(controller);
     this.myTextField.addKeyListener(controller);
    ... // and so on
  }
 ...
}
```

Another alternative to implement the controller with Java anonymous inner classes. Using this feature of Java language, the controller does not need to implement `Listener` interfaces anymore, but still needs to have access to the dialog class widgets. Listing 5 illustrates this approach (we assume that `myButton` and `myTextField` class members are defined with package access in the `MyDialog` class).

Listing 5: Implementing controller with anonymous inner classes

```
class MyDialogController {
  MyDialog dialog;
...
  public void registerControllers(){
       this.dialog.myButton.addActionListener(
          new ActionListener(){
              public void actionPerformed(ActionEvent ae){
                    // event processing code
              }
           }
       );

       this.dialog.myTextField.addActionListener(
          new ActionListener(){
              public void actionPerformed(ActionEvent ae){
                    // event processing code
              }
           }
       );

       this.dialog.myTextField.addKeyListener(
          new KeyListener(){
                public void keyPressed(KeyEvent ke){
                    // event processing code
                 }

                public void keyReleased(KeyEvent ke){
                    // event processing code
                 }

                public void keyTyped(KeyEvent ke){
                    // event processing code
                 }
           }
       );
      ... // and so on
 }  // registerControllers()
 ...
} // MyDialogController
```

Thus, we do not have anymore to work with cumbersome if-else-…-else constructs. All the event handling code for a particular widget is localized now within few (usually one) inner classes. If we need to change something in that code, we don't have to run through all the if-else-…-else blocks looking for the code to be changed.

'Normal' inner classes could be used inside the controller class as event handlers for particular widgets as well. They would have to implement then the corresponding `Listener` interfaces.

Consider now the HMVC approach. In this case, we implement event handling in the view class. The two alternative solutions described above can be applied here as well. In the first, the view class must implement all the required `Listener` interfaces. This solution is used in [CKP00]

(as well as in many other papers on Java Swing). The view registers itself as a listener with each of its child widgets. This will result again in if-else-…-else blocks within callback methods to find out the source of the event. Note also that we avoid the visibility problem inherent to MVC where the controller is implemented as a separate class.

The second alternative is to have anonymous inner classes within the view class that implement callback methods, as shown above. We find this solution more elegant, as we do not have to do with the if-else-…else blocks.

In general, we have in HMVC more complex view class, since it is responsible now for both the GUI creation and the user input handling.

4.2 User Input Handling in PAC

In PAC the agent's presentation part is responsible for user input handling, along with the GUI issues. As for the GUI creation problem, the details of how event handling could be implemented are left open. So, the implementation is completely up to the developers.

Here, we give some general ideas.

If agent's visualization is simple, we implement agent's presentation component as a single class, which also provides input handling through callback methods.

In more complex cases, the presentation may include several visual components, like windows, dialogs, menus, etc., which are responsible for various visual aspects of an agent. A Façade object hides this inner structure of the presentation from the outside world. Each complex visual component is implemented in a separate class, probably with some additional helper classes (see also the GUI creation in PAC section).

We need to provide now event handling for these components. As PAC does not specify how this should be done, we can apply, e.g., the solutions for MVC and HMVC described above. Namely, we can either put event handling logic (callback methods) into the visual component class itself (HMVC approach), or implement it in a separate class (MVC approach). Using Java Swing, this could be done either by implementing the corresponding `Listener` interfaces or by using anonymous inner class, as shown above.

4.3 Application-specific Processing of User Input

So far, we have discussed the design and implementation of the primary user input handling logic, which only intercepts user events in callback methods. Application-specific event processing logic is provided in other parts of the application (separation of concerns). Thus, we have to couple our callback methods with that logic.

In MVC, the controller transforms user events into the calls on the model. In HMVC, the view forwards user input to the controller, which decides then what to do with it. The same is done in PAC.

Details are given in the following section.

5 Dialog Control

The problems discussed so far relate to the presentation layer and its main responsibilities – user input and output.

The next step is to couple the presentation layer to the business logic, which processes user input from, and provides output to, the presentation layer.

A number of issues may arise in this regard.

Consider a UML graphical editor. It would most likely have a tree-like navigation area representing UML diagrams and their elements, and a drawing pane to draw UML models. A typical scenario within such an editor would be to select a UML model element in the navigation tree, get a popup menu by clicking the right mouse button, and select the properties menu item in the popup menu. The Properties dialog is displayed, where the user can edit various data related to the selected element. Upon clicking OK or Apply button, the input is forwarded to the business logic and processed there. After the input has been processed, the GUI is updated to reflect the changes made upon the selected element. In particular, these changes are visible both in the navigation tree and in the drawing pane.

Note how many things are done before user input is sent to the business logic: We need to display the popup menu and then display the Properties dialog, which must be populated with the data of the selected element. After the input has been processed, we must communicate state changes to several parts within the GUI.

How could such an interaction be implemented? And where should the corresponding logic reside, i.e., who is responsible for these issues? For instance, in the above scenario, an interesting question is who creates and displays the popup menu and the Properties dialog?

Often, such kind of logic is put into the event handling callback methods (the Smart UI approach). However, in complex cases, we might face some difficulties.

Consider the event handling code for the popup menu in our example. It is not a big problem to create and display the Properties dialog in the callback method for the Properties menu item. The problem is that we must populate that dialog with the data of the selected element. How could this information be obtained within the callback method? This implies that our popup menu should have access to the business logic. Another solution would be to provide the Properties dialog with a kind of identifier of the selected element and let the dialog itself obtain the required

data from the business logic. For this, it must have access to the business logic, which is also needed to send the user input when the OK button is pressed. After the input has been processed, the dialog needs to communicate the state changes to the navigation tree and the drawing pane. Alternatively, this should be done by the business logic, which makes it aware of the presentation layer.

Another issue is data types used. The dialog must transform its input into the data types used in the business logic, becoming dependent on those data types. Alternatively, data type transformation can be performed in the business logic, making it dependent on the presentation layer.

As a result, implementing complex scenarios in the event handling callback methods might be rather intricate leading to tight coupling among the presentation and the business logic.

To address these problems, a dialog control layer is provided, which controls and manages complex user-system interactions and performs data type transformations. This layer decouples, and glues together, the presentation and the business logic layers.

In this section, we discuss how the patterns address the dialog control problem.

5.1 Dialog Control in MVC

In MVC literature (e.g., in [POSA I]), simple (basic input-process-output) interaction scenarios are mainly considered, where user input is first handled by the controller, which transforms it into the method call(s) on the model. The model changes the system state and notifies all the dependent views and controllers upon that change (using the Observer pattern). The views retrieve then the data from the model and display it to the user. Alternatively, the controller forwards user input directly to the view, if it is about changing the GUI visual state only and doesn't relate to the business logic (e.g., clicking a zoom button).

The dialog control functionality is therefore distributed between the controller, model, and view components (in addition to their main responsibilities as defined in MVC):

- The controller transforms low-level, toolkit-specific user input events into application-specific method calls.
- The model implements notification of the dependent view(s), while its basic concern is business logic.
- The view retrieves data from the model and performs data type transformation, while its main responsibility is displaying data to the user.

This implies that the separation of concerns is not completely achieved.

As a result, the view is tightly coupled to the model and its data types [POSA I]. So, any change in the model implies changes in the dependent view(s).

Despite these shortcomings, such an interaction mechanism works rather well in simple cases, where interaction is confined within a model and its dependent view(s) and controller(s).

However, MVC does not address complex interaction scenarios. What if other GUI parts are involved in a scenario? How could we provide this? The developers have to find their own solutions.

Let's take the above scenario in the graphical editor, which is rather complex. How could we implement it with MVC?

First, consider what GUI parts we have in this scenario, which we would model as MVC-triads. Most likely, these are a tree-like navigation area MVC, a drawing pane MVC, a Properties dialog MVC, and a popup menu MVC (the corresponding model component in this case could be the business object implementing the UML model element, which is represented by the selected element in the navigation area).

The scenario could be implemented then as follows. The controller of the navigation area MVC handles mouse events. Upon clicking the right mouse button, the controller should first identify the element in the navigation area to which the right click relates, and then create and show the popup menu for this type of elements (there are usually several types of elements, each having specific popup menu). The popup menu (the view) must be associated with the corresponding business object (the model). The controller for the popup menu is created here and associated with the popup menu as well (alternatively, the popup menu could create its controller itself). When the user selects the Properties menu item, the event is handled in the popup menu controller. The controller's event handling callback method must create and display now the Properties dialog (the view) and its corresponding controller and model, which is the same business object as for the popup menu.

The user changes some data and presses OK button. This event is handled in the controller of the Properties dialog, which gathers user input and sends it to its associated business object (the model) by calling the corresponding method on it.

Upon this, the business object validates the user input data and updates the object's state. The next step is to communicate this change to the presentation objects. These are the dialog itself, the drawing pane, and the navigation area, but also, e.g., the Undo menu item in the Edit menu. Further, if the model state has been saved, the File menu and the tool bar should be notified as well to enable the Save menu item and the Save button, respectively.

For all this to be done, we need to register all these GUI components the model component of the Properties dialog. And this should be done in the controller (namely, in the corresponding callback method) of the popup menu, which creates the Properties dialog. This implies then that the popup menu controller should keep all these logical relationships, which is actually not its concern, and be also initialized with the references to these GUI parts (the visibility problem)! And so on.

And that's not all. Actually speaking, registering, e.g., the drawing pane on the model of the Properties dialog would violate the MVC design, since we get then the drawing pane view associated with two models – one is the drawing pane's own model and the other is the model of the properties dialog.

So, the things get really complicated. In a complex GUI, this leads to tight coupling among objects and to cumbersome and involved code, which is hard to write and understand, because it is very difficult to follow all the relationships among objects and provide reference initialization.

In general, the problems arise in scenarios where interaction involves view components having different models, because according to MVC each view is dependent only on one model, and in complex cases views become dependent on several models playing the key role in the dialog control.

This issue raises a more general question of what the models in MVC actually are. In particular, how many model objects should we have in our application? Should we have a separate model component for each business object or just one model that hides all the business logic? This is an important design issue which we consider in the next section. Here, we discuss some implications regarding the dialog control.

For instance, in the above example, the model of the Properties dialog was the corresponding business object (an element of the UML model). This object, however, is part of the compound business object representing the entire UML model, which is the model component of the navigation area.

One possible design alternative would be then have this model as the only model component in our editor application, with all the views depending on this single model.

However, for large GUIs we get problems in this case as well. The main one is that the single model component has to implement now more sophisticated change notification mechanism. For example, we have to introduce event types and let the views in our application subscribe only for certain event types (otherwise we get all the views being notified upon each single change in the model). This solution has some significant drawbacks. First, if a view is interested in several event types, its notification method will contain not much nice if-then-else blocks to find out which event has occurred. Second, consider the case of adding a new view with a new event type. We have to find out all the views that depend on this new event type and change their notification methods accordingly. Another problem is that one single model component will have a large number of getter methods for the views to retrieve the data to be displayed.

5.2 Dialog Control in HMVC

HMVC tries to address these issues with the dialog control in MVC by providing a means for communication among different GUI parts.

MVC triads in HMVC are organized hierarchically according to the hierarchy of visual components, with the main window in the root (see also the GUI creation section above). The triads are connected through the controller components. The resulting controller hierarchy is responsible in HMVC for the communication among different GUI parts, thus providing a solution to the dialog control problem.

Presumably, this mechanism has been adapted from the PAC pattern and is discussed below.

5.3 Dialog Control in PAC

PAC explicitly addresses the dialog control problem.

The application is modeled as a hierarchy of interacting agents. Within each agent, the control component is responsible for the agent's interaction with other agents. The control also mediates the agent's presentation and abstraction components, performing, e.g., data type transformation. Hence, each control component has general two roles – it provides internal interaction between agent's parts and external interaction with other agents [POSA I].

Interaction among agent's components is rather simple. The presentation part forwards user input events to the control and receives from it the data to be displayed (already in presentation-specific format). This data is obtained from the abstraction part, where it is kept in the application-specific (business logic) format.

In case of complex presentation parts, we might need to provide some interaction among visual components that comprise the agent's presentation. Recall the graphical editor example from the section on GUI creation in PAC. The presentation part of the drawing pane agent consists of the drawing pane, the tool palette, dialogs, and popup menus. Suppose we have also a zoom button. Clicking this button needs to be communicated to the drawing pane visual component.

The interaction functionality within the presentation part should be assigned then to the visual components. If it is intensive, this may result in a tight coupling among them. The Mediator pattern could be applied in this regard to decouple the visual components from each other. As discussed in the section on GUI creation in PAC, such visual components could be hidden from the rest of the agent by a kind of Façade object. As an option, the Façade object can play the Mediator role as well. A potential problem here is that the Façade object may get rather complex. Basically, it is again the agents' granularity issue and finding balance between the number of agents and their complexity.

Interaction among agents is more complex and involves several aspects. First, it is exchanging of data. In PAC, the entire application model is implemented in the abstraction component of the top-level agent. Abstraction components of other (basically, bottom-level) agents keep agent-specific data, which is part of the application model. Therefore, this data is obtained from the top-level agent. So, in general, the agent's local data must be kept synchronized with the application data in the top-level agent. This has the following consequence. When the user provides some input to a bottom-level agent, this input is sent first up to the top-level agent,

which processes the input and changes the system state in its abstraction component. This change is notified back to the bottom-level agent (and all other agents depending on this data). And only then retrieves the bottom-level agent the data from the top-level agent and updates its own abstraction and (most likely) presentation components. Another type of interaction is exchanging of various events among agents, like changes of the presentation state (e.g., pressing a zoom button).

[POSA I] specifies two basic mechanisms for inter-agent interaction. We give here brief descriptions thereof.

- **Composite Message Pattern**. The agent provides two methods to send and receive messages, containing the information that is passed. For incoming messages, the developer must provide logic that analyses the content of the message and decides what to do with it – process locally by sending it to the agent's presentation or abstraction components, or forward the message to another agent. From the implementation point of view, this may result in large and complex if-then-else blocks within the message processing methods. Another issue is identifying appropriate event types and introducing new ones. Recall also the problems we have discussed above when considering the dialog control in MVC with single model component.
- **Agent-specific Interfaces**. Each agent implements its specific interface used by other agents to interact with it. With this, we can avoid the complexity of if-then-else constructs of the Composite Message pattern, but the agents become more tightly coupled to, and dependent on, each other.

The drawbacks of both approaches are tolerable per se. However, in the context of PAC they may become a real nightmare. The reason is that the agents in PAC do not interact with each other directly, but through the agent hierarchy (actually, through the hierarchy of the control components). For instance, in the data retrieving scenario, a bottom-level agent does not interact directly with the top-level agent, but with its parent agent, which, in its turn, forwards the request to its parent, and so on, until the request arrives the top-level agent. This means, however, that all the intermediate agents participate in this interaction and each of them must provide the corresponding logic, which is basically the request forwarding logic. For example, when using agent-specific interfaces approach, each intermediate agent must expose interfaces of all the agents below it (the child agents) to the agents above it (in particular, to the top-level agent), and vice versa.

Imagine now what would be, if you change the interface of only one single agent. This would result in changes of all the intermediate-level agents providing interaction with this agent.

Well, it still could work for small applications (regarding the code understandability and maintainability issues as well). For large GUIs, however, we find both approaches utterly inappropriate. Again, the reason lies, to a greater extent, in the hierarchy-based agent interaction style mechanism.

It is also not a very nice solution from the software design in general, and separation of concerns in particular, viewpoint. Indeed, why should we make the intermediate-level agents be aware of (and implement!) interaction among other agents? Why can't those agents interact directly with

each other? One possible answer is that direct interaction among agents would break the tree-like agent hierarchy.

Given these issues, other approaches have been proposed.

[POSA I] specifies also an agent interaction mechanism based on the Publisher-Subscriber [POSA I] or Observer [GoF] patterns. In this approach, all the agents (which are then subscribers, or observers) that depend on data or events of a specific agent (publisher, or subject) register themselves on that agent, which notifies them when changes occur. Upon notification, the dependent agents update their states, retrieving data from the publisher agent. Agents interact therefore directly with each other. This could be done then by applying the Composite Message pattern or agent-specific interfaces as described above. The only problem is the registration, as the agents may reside in different parts of the hierarchy and we have to make them aware of each other.

[Wellhausen] provides a solution that solves this problem. It uses the Event Channel variant of the Publisher-Subscriber pattern, which decouples publishers and subscribers. Event channels are given as a system service. The publisher provides events of certain type for a specific channel. The subscribers register for events of that type with this event channel. So, the subscribers must know only the event type.

Another solution is described in [HO07], where a Hierarchical Service Locator mechanism is used. Each agent implements the Service Locator interface. The agents (that are used by other agents,) register their interfaces in the Service Locator. When an agent needs to communicate with another agent, it performs lookup with the Service Locator. If the required agent interface is not found locally, the lookup request is forwarded to the parent agent (which also implements the Service Locator interface), and so on, until the required interface is found somewhere in the hierarchy. This solution allows direct agent interaction resulting in tight coupling among agents.

A simple Service Locator mechanism could be used as well. In this case, it should be provided as global system service to all agents.

6 Business Logic

The presentation must eventually be coupled (through the dialog control layer) with the business logic. The way the business logic is designed and how it can be accessed from other layers is important to know.

Here, we consider how the patterns treat the business logic.

6.1 Business Logic in MVC

In MVC, the business logic is handled by the model component.

In the previous section, when discussing the dialog control in MVC, we have outlined the problem of what the model actually is. Here, we discuss this issue in more details.

In simple applications, there could be one model component hiding the entire business logic, which is implemented in a single class. The views and controllers work then with this single model.

However, in large applications, the underlying business logic is usually rather complex, involving various business objects and relationships among them. This implies more complex design, where each business object is implemented in a separate class or a set of related classes. The following questions arise in this regard:

- What is the model in MVC?
- How do the business objects relate (or could be mapped to) the model components in MVC?

The first design alternative is to consider each business object that is displayed to the user as a model, and associate it with the corresponding view(s) and controller(s). It is a natural way of applying MVC. This approach implies that we put the change notification logic (the Observer pattern functionality) into the business object, thus mixing the concerns. Moreover, as discussed in the section on the dialog control, in complex interaction scenarios we may need to provide communication among different views, which results in the views depending on several models.

The second approach is to have a single model component hiding the entire business logic. In this case, the model component acts as a Façade to the business logic and implements change notification mechanism. Thus, we do not have to put this functionality into the business objects, as in the previous approach. The drawback is that the model component as Façade gets itself large and complex. Another problem is implementation of the change notification mechanism, which has been discussed in the section on dialog control.

Yet another alternative is to have a separate model component for each business object. In this case, the model holds only the data to be displayed, while the business logic itself resides in the business object. The model forwards user input to the business object. It also performs data type transformations between the presentation and business logic and implements change notification mechanism. Thus, we decouple the business objects from the dialog control logic. Still, we have the problem of views depending on several models, which arises in complex scenarios. This problem could be solved by making the model components interact with each other. In this case, when a state change has to be communicated to other views, the model component notifies the model components of those views, which then notify their views in a standard manner. As such interactions may result in many-to-many relationships among model components, the Mediator pattern [GoF] could be applied. As it could be seen, the model components form a layer between

the views and controllers (the presentation layer) and the business logic. This is the dialog control layer.

6.2 Business Logic in PAC

As mentioned above, the application's business logic in PAC is implemented in the abstraction component of the top-level agent, which provides an interface to retrieve and change business data. Hence, we can see it as a Façade to the business logic. On the other hand, bottom-level agents usually represent 'self-contained semantic concepts', i.e., business objects from the business logic. The state of the business object is kept then in the agent's abstraction component and is a duplication of the business object's state from the business logic implemented in the top-level agent. Therefore, the abstraction component of the bottom-level agent just holds the data, but does not provide any business operations on the corresponding business object. It must therefore be kept synchronized with the abstraction of top-level agent (see also the dialog control section).

The design of the business logic itself is left to the developers.

7 Summary

The complexity of GUI development is sometimes underestimated in software community, as compared to the server-side programming. In this paper, we have tried to specify common problems when developing GUIs, and how these problems are, or could be, addressed by existing patterns for GUI development. We have also tried to identify the problems when applying these patterns for large and complex GUI applications and to illustrate all this with various examples. A detailed discussion of really complex (and therefore interesting) examples deserves, however, a particular paper.

We do not want to give here any conclusions regarding each particular pattern we have discussed or compare the patterns with each other, but rather end this paper with some general observations.

First, separation of concerns is the main principle underlying each pattern. Second, each pattern works rather well in simple cases. On the other hand, pattern descriptions give only basic considerations, without going much into details. The more complex the GUI application is, the more questions will arise, which are not answered by the patterns, and the more design alternatives the developers will have. Identifying visual component classes in container-level MVC and designing complex presentation parts in PAC are examples of the problems the developers have to solve themselves. Another example is designing the dialog control to provide interaction among different application parts.

For us, the main result of our work on applying and analyzing patterns for GUI development is that there is always place for creative work – to go beyond the pattern descriptions and investigate new design alternatives. And this is in patterns' nature – they do not (and probably must not) provide ready solutions, but rather give an advice what to start with.

8 References

[Arch92] A Metamodel for the Runtime Architecture of an Interactive System. The UIMS Developers Workshop, SIGCHI Bulletin 24 (1), 1992.

[Burbeck92] S. Burbeck. Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC), 1992.

[CKP00] J. Cai, R. Kapila, G. Pal. HMVC: The Layered Pattern for Developing Strong Client Tiers. www.javaworld.com, 2000.

[Coutaz87] J. Coutaz. PAC: An Object Oriented Model for Implementing User Interfaces. SIGCHI Bulletin, 19 (2), pp. 37-41, 1987.

[Evans03] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[Fowler99] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[GoF] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software. Addison Wesley, 1994.

[HO07] M. Haft, B. Olleck. Komponentenbasierte Client-Architektur. Informatik Spektrum, 30 (3), 2007 (In German).

[Holub99] A. Holub. Building User Interfaces for Object Oriented Systems. www.javaworld.com, 1999.

[KP88] G. Krasner, S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalt-80 System. Journal of Object Oriented Programming, 1 (3), pp. 26-49, 1988.

[Marinilli06] M. Marinilli. Professional Java User Interfaces. John Wiley & Sons, 2006.

[POSA I] F. Buschmann et al. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley and Sons, 1996.

[Wellhausen2005] T. Wellhausen. Ein Client-Framework für Swing. JavaSPEKTRUM, 2005 (In German).