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 common software engineering problems 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. However, when applying these patterns in practice, problems arise, which are mostly addressed to an insufficient extent in the existing literature (if at all). So, the developers have to find their own solutions.

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. We then identify four typical problems that usually arise when developing GUI applications and discuss how they can be addressed by each of the patterns, based on our own experience and investigation of the available literature.

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

1 Introduction

Developing a large and complex graphical user interface (GUI) application for displaying, working with, and managing complex business data and processes is a rather difficult task. 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 user input or track all the relationships between the GUI and the business logic components. Without proper system design, GUI code and business logic may get deeply intertwined with each other, which leads to another important issue: The *code organization*. A general requirement is that the source code be understandable, maintainable, and reusable. Clear code organization and coding principles are essential as regards communication among the developers within the whole system lifecycle.

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

With time, a number of patterns for GUI applications have been proposed by the software community. The well-known Model-View-Controller (MVC) with its variants and, to a much lesser extent, the Presentation-Abstraction-Control (PAC) patterns provide a good starting point for the developers. However, when applying these patterns in practice, a number of problems arise both at the design and the implementation levels. For instance, an essential constraint is the GUI platform (or toolkit) used. It 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 be taken into account when applying a particular pattern.

In large and complex GUI applications other issues become apparent as well, such as how the application's GUI part is constructed, how user input is handled, or how user-system dialog (interaction) is managed. Most of these problems are addressed to an insufficient extent (if at all) in the existing literature.

In this paper we revisit three well-known patterns for GUI applications – MVC, HMVC (Hierarchical MVC) and PAC – by identifying four typical problems in the development of GUI applications and discussing how they are, or could be, addressed by these patterns.

The paper is organized as follows. In the Background section we give a brief description of the MVC, HMVC, and PAC patterns, as well as present the four problems, which are then discussed one by one in the subsequent sections. The discussion is illustrated with numerous examples. Some implementation specifics for Java programming language and Java Swing toolkit are provided as well. We conclude our paper with a brief summary. Throughout the paper, we assume that the object-oriented approach is used.

2 Background

Patterns are usually applied within a systematic approach to system development, where the system architecture is first defined given the system requirements. Through the system design, following the separation of concerns principle, it is then refined into a set of classes, representing the detailed system structure. Each class is assigned specific responsibilities. At runtime, the system can be seen as a network of interacting objects.

Let's consider a typical usage scenario of a GUI application: 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 processed by the application, and results are displayed to the user.

Implementing such a scenario, we get a number of interacting objects with different functions. For instance, presentation (or GUI) objects comprise the GUI, display data to the user, and process user input. Other objects could be responsible for transforming between GUI- and application-specific data types. The input data is passed to the application logic objects and processed there. The results are then returned to the GUI part, and the visual state of the presentation objects is updated.

In a large application, with dozens of usage scenarios, the system design may contain hundreds of classes with intensive interaction among objects at runtime. The more features a GUI application provides, the tighter the relationships among objects are.

This makes the developing of GUI applications a rather difficult task. Typical questions would be:

- How to identify classes? How to assign specific functionalities to each class and design their interaction (as this can be done in different ways)?
- How to map classes onto implementation-level constructs (such as GUI toolkit widgets or Java classes)?
- How to initialize the resulting object network at runtime, instantiating particular objects and establishing relationships among them, i.e., initializing object references (known also as the *visibility problem* [Marinilli06])?

Apparently, 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).

2.1 Patterns for GUI Applications

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 controller handles user input and forwards it to the model by calling the corresponding function(s). The model processes input and changes the application 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 state changes, the model notifies all the registered components, which then retrieve the required data from the model.

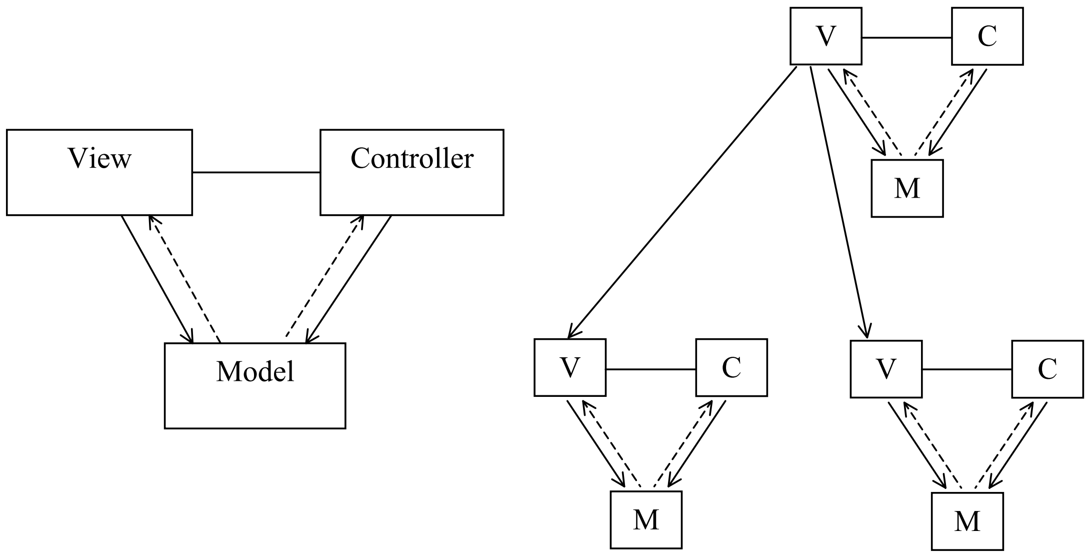

Figure 1. MVC and a hierarchy of MVC-triads

This is basic MVC. The whole application is then built as a hierarchy of MVC-triads organized around the view components, i.e., this hierarchy reflects the hierarchy of visual components (such as main window, menu bar, menus, 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 a kind of the dialog control logic.

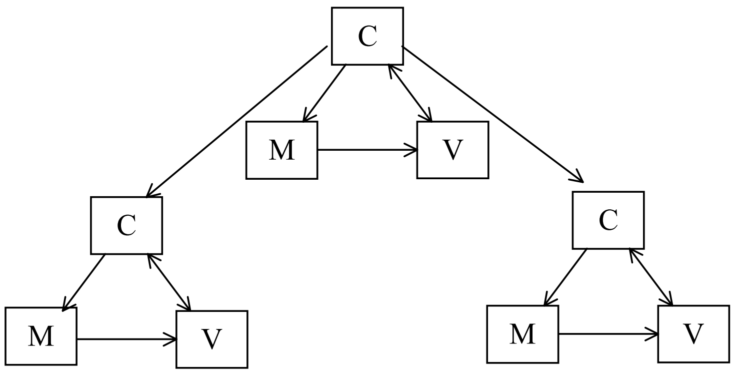

Figure 2. HMVC

Presentation-Abstraction-Control

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

An agent (usually, it is a 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, which are then implemented as lower-level agents. 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 elements, such as menus or tool bar. It also coordinates all other agents.

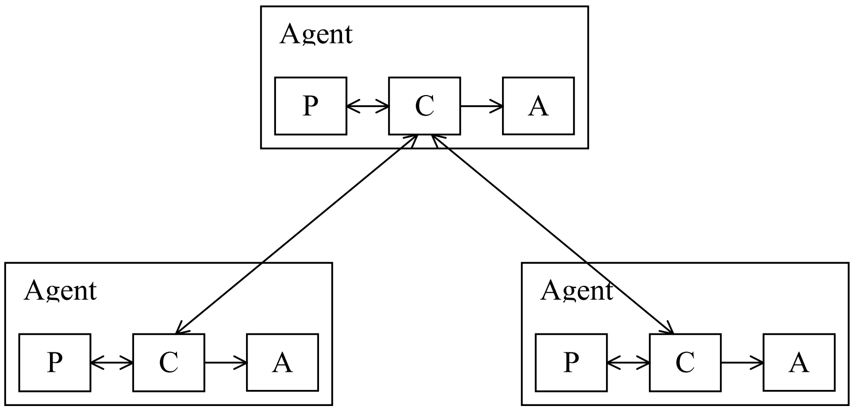

Figure 3. PAC

2.2 Typical Problems in GUI Development

The patterns presented above are similar to each other in that they all identify basic application components and assign certain functionalities to these components, such as displaying data to the user, handling user input, and processing the data in the application logic. These are, however, only some of common functionalities inherent to each GUI application. A number of other typical problems usually arise both at the design and implementation level, which becomes especially apparent when developing large and complex GUI applications.

In this paper we consider the following four of these problems:

- **Creating and assembling GUI (Content)**: The application's GUI part is comprised of visual components, or widgets, which are usually organized in a hierarchical manner into panels, menus and/or other containers within the 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 whole GUI (this is especially inherent to large GUIs).

- **Handling user input**: The GUI provides various input means to the user. Primary user input handling is usually implemented through callback methods, which are called by the GUI platform when the corresponding event occurs. In large and complex GUIs, with dozens of widgets, the code that handles user input may become rather large. Yet user events must be forwarded to the business logic for application-specific processing, which implies additional design-level issues.
- **Dialog control**: Dialog control is about managing user-system interaction scenarios, which may be rather complex involving several interaction steps. 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 design viewpoint, this involves interaction among a number of objects from GUI and business logic, and the problem is to design and control such interactions.
- **Integrating GUI and Business Logic**: Eventually, the GUI must be coupled with the business logic. The way the business logic is designed is important with respect to how this coupling could be implemented.

The first two problems presented above are rather implementation-level ones. In this regard, an essential constraint is the GUI platform (toolkit) used, which assumes a specific way of how the application's GUI part should be developed (*a path of least resistance*). This should be taken into account when applying a particular pattern. The last two problems are more or less independent from the GUI platform.

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 they 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.

2.3 The Smart UI Approach

The pattern-based approach to system development is not the only way the applications can be developed. Here we present the Smart UI approach, which will be used in the subsequent sections to illustrate the GUI development problems we discuss in this paper.

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 as one single class, which contains the GUI and user event handling logic, as well as mathematical operations.

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

The main characteristic of the Smart UI, in general, is mixing business and presentation 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” [Evans03].

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 (GUI Builders), which let the developers design the GUI with a kind of visual editor and automatically generate the source code, including empty callback methods to handle user input. 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.

However, 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]).

3 Creating and Assembling GUI (Content)

A GUI typically consists of a main window, and a number of dependent secondary windows (basically, for output purposes) and/or dialogs (for user input). Windows and dialogs are composed of various visual components (widgets) that are organized into visual containers in a hierarchical manner. Hence, a window or a dialog can be seen as *a hierarchy of visual components*. A typical 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)
- Menu bar
- Tool bar
- Status line

As it could be seen, GUI may have a rather complex hierarchical structure. In this respect, we identify the following problems, which are especially inherent to large GUIs:

- How and where is each particular widget created (i.e., which object is responsible for creating and initializing a given widget)?
- How and where is the whole GUI assembled from particular widgets (i.e., which object (or objects) establishes relationships among widgets within the hierarchical GUI structure)?

From the implementation viewpoint, GUI platform (or toolkit) is the major constraint, since it provides widgets from which the GUI is constructed.

Regarding the GUI patterns, GUI toolkit implies another important issue. Each GUI pattern has elements (classes) that are responsible for the GUI (presentation) concerns. These elements have eventually to be implemented with the given GUI toolkit. As a consequence, this may require that we extend standard toolkit widgets to provide the pattern-related functionality. The widgets are then used in a specific way, which differs from the standard way intended by the toolkit, as expected by the developers (*path of least resistance*). This, in its turn, implies additional implementation effort.

This section is organized as follows. We start with the Smart UI approach to illustrate typical problems when creating and assembling the GUI. We then consider how MVC, HMVC and PAC address the identified problems.

3.1 Creating and Assembling GUI in the Smart UI Approach

Within the Smart UI approach, it is typical to implement the main window as a single class that instantiates all the widgets, initializes and composes them into visual containers, and eventually assembles the whole GUI. GUI builders – popular tools for GUI programming – encourage this way of creating GUIs, which is therefore often used in practice, especially by non-experienced GUI developers.

Let's see what the GUI creation code within the main window class includes. First, we would have a class instance variable (field) for each widget (both simple widgets and visual containers). 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. Listing 1 illustrates possible implementation in Java Swing.

Listing 1: Initializing a menu item in Java Swing

```
// 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, menu items must be added to their menus, which themselves must then be added to the menu bar.

Imagine now how the main window class would look like just with a hundred of menu items. It gets quickly very large, even if we take into account just the GUI creation and assembly code (without event handling or business logic, which the main window class would most likely contain in the Smart UI approach). And as it seems, the menu bar with all its menus and the tool bar (to a lesser degree) are the main contributors in this respect.

The code size and code organization problems 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 becomes large, too, we can extract methods from it, which create particular menus within the menu bar, as it is illustrated in the Listing 2.

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
}
```

The code size problem can be solved then by applying the Extract Class [Fowler99] or Extract Panel [Marinilli06] refactorings. With this technique, we extract all the GUI code for, say, menu bar into a separate class. The same can be done for tool bar, a specific panel or other containers within the main window.

3.2 Creating and Assembling GUI in MVC and HMVC

In MVC, GUI (presentation) concerns are handled by the view component. Logically, it is responsible for presenting application (business) data to the user. Eventually, each view has to be implemented with the given toolkit. Several designs are possible here.

Widget-level MVC

In the widget-level MVC approach, a view is a visual component (widget) from the toolkit. More precisely, each toolkit widget in the GUI is considered as a view component (which results in an MVC-based framework [POSA I]). Each widget has its model and controller components. Together, they form a simple MVC triad. The application is then created by instantiating such MVC triads for each GUI widget and assembling them together.

This approach originates from the Smalltalk environment, which 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 controller classes.

Widget-level MVC seems to work rather well, if an MVC-based toolkit is used and the application is simple. Let’s consider potential problems that might arise when developing large and complex GUIs.

First, as in the Smart UI approach, it is the large main window class problem, if the whole GUI creation and assembly work is done in one class. Again, a solution would be to apply the Extract Method and Extract Class refactorings (see also the discussion in the previous section). For instance, with the Extract Class refactoring, we implement certain view components in separate classes. Each such view class instantiates all its child MVC triads and assembles its GUI part from the child view components.

Another problem concerns the model component associated with each widget. In MVC, the model is responsible for business logic and uses its own data type(s) to represent the business data to be displayed to the user. On the other hand, the view component uses, in general, another (usually toolkit-specific) data type(s) for the data it displays. So, we need to provide data type transformation. Let’s consider this on the Java Swing example.

In Java Swing, each widget has its standard Swing model class, which maintains data displayed by the widget. However, this data is kept in Swing-specific format, which 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, the `JTree` component requires a class that implements the Swing-specific

`TreeModel` interface as the associated model, which represents a hierarchical structure. 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 then connect a widget to the corresponding business data?

Extend Widget
One way is to extend the widget class associating it with our own model class that represents business data in its internal 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).

Extend Model
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 implement the required data type transformation as well.

As we can see, in both alternatives the developers have to extend the standard Java Swing functionality.

Consider now the situation when the toolkit does not support MVC, and we are going to apply MVC at the widget level, as described above. In this case, we would have to extend all the toolkit widgets we use 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. As a result, we would 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 the developers are familiar with (as suggested by the toolkit – path of least resistance). 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 cases 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 (especially for the toolkits that do not support MVC). Another problem is that applying MVC for each widget is a rather low-level and fine-grained approach with too many pattern-specific details (imagine having a separate MVC triad for each menu item!). This makes things unnecessarily complex and may discourage the developers, especially if a large GUI application is developed (see also the MVC liabilities section in [POSA I]).

Container-level MVC. HMVC Approach

This approach tries to avoid the problems of widget-level MVC we have discussed above. The key idea here is to move from the level of particular widgets to higher levels in the visual component hierarchy representing the application's main window (or any other window). Namely, MVC is applied at the level of visual containers – parts within GUI that group together other (usually related) visual components. Typical containers are panels, dialogs, menu bar and its menus, and tool bar, which are considered now as MVC views.

Each container-level view is implemented as a separate class through extending the corresponding container widget from the toolkit (recall again the Extract Class refactoring technique!) and adding the required functionality. In particular, it is responsible for handling its content, such as creating, initializing, and assembling together the visual components it contains. The key point here is that all the GUI-specific work for all the child widgets within a container-level view is done in standard toolkit way (Listing 3). We have no more to treat each child widget in the MVC way and create (a large number of) new classes for this purpose. Thus we avoid all the low-level details and complexity of the widget-level MVC. This is particularly important for toolkits that do not support MVC. And the developers can now work with the toolkit as they used to. Only the container classes must be adapted to MVC.

Listing 3: Container-level MVC

```
public class MyXYZPanelView extends JPanel {
   // Child widgets are used in standard Java Swing way
   private JTextField  myTextField;
   private JTextField  myAnotherTextField;
   private JLabel      myLabel;
   private JLabel      myAnotherLabel;
   private JButton     myButton;
   ...
   private JCheckBox   myCheckBox;
   ...
   public MyXYZPanelView (){
      ...
      this.myTextField = new JTextField(10);
      this.myAnotherTextField = new JTextField(10);
      this.myLabel = new JLabel("My Label");
      this.myAnotherLabel = new JLabel("My Another Label");
      this.myButton = new JButton("My Button");
      ...
      this.myCheckBox = new JCheckBox();
      ...
      // other initialization code
    }
   ...
   // other code
}
```

This approach is used in HMVC [CKP00], where main GUI parts, such as menu bar, 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 MVC approach is the granularity level, i.e., which containers should be modeled as MVC views. 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, the menu bar (container) is usually comprised of several menus (containers), where each menu consists of either menu items (leaf nodes) or other submenus (containers), and so on. Complex forms are often composed 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 particular 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 a container.

So, we can put a more general question: Which visual components within GUI should be considered as MVC views?

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

- If a child container within a given container has complex structure (contains many widgets), then implement this child container as a separate view.
- (Optional) If a widget (visual component) within a container must provide some specific functionality, then implement this widget as a separate view.

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 the two extremes: having a single monster class that implements everything or having too many classed with low-level details and unnecessarily complex design.

The following examples illustrate the idea. 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 5-7 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.

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 from the toolkit. Another view would be the navigation pane itself.

As a counter example, consider a pane comprised of, say, 20 child panes, each having few (e.g., 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 of its child panes is rather simple to be considered as a separate view.

3.3 Creating and Assembling GUI in PAC

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

First, an application is organized as a hierarchy of agents, which represent 'self-contained semantic concepts' (see also PAC description in Section 2.2). Each agent has the presentation component, which handles 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 to note here is that the design of the whole application and its GUI part in particular is driven by agents. Regarding GUI issues, this means that we take, roughly speaking, 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 strongly influenced by the visual design of the GUI.

More insight on what is hidden behind the agent's presentation component could be deduced 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. Such a class could be derived from the toolkit widget class suitable to handle agent's visualization. For instance, for an agent representing some value label or text field widgets could be used. As it can be seen, this approach is very similar to applying MVC at the widget level. Therefore, if we have a large number of simple agents, we'll face similar problems. Namely, we have to extend standard toolkit widgets to provide PAC-specific functionality and we get unnecessarily complex design with too many low-level details. These issues are also discussed in the PAC liabilities section 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 agents to control their number. With this, the concepts from the business domain implemented by simple agents are handled within more complex agents. Note that this approach is similar to the container-level MVC and HMVC, where container-level views handle all their 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. Regarding the graphical editor example [POSA I], we would have an agent for the graphical editor only, but not for each graphical object. The editor agent then handles all the graphical objects internally.

In such complex agents, the presentation part gets also more complex. As mentioned above, it may include menus, window, dialogs, etc., which could be implemented as a set of (interacting) classes. For the interaction with the outside world through the agent's control component, the Façade pattern could be used [POSA I]. In this case, the Façade object hides the internals of the presentation part from the control component and is the only PAC-aware object of the agent's presentation. An essential consequence of this design solution is that it allows the developers to implement the visual components within the agent's presentation part in the 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 the containers-level MVC and HMVC!).

To illustrate these ideas, consider the 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 could be implemented in a separate class (probably with a number of additional helper classes). We would also have a 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.

The next problem may arise in the top-level agent. As stated above, its presentation part is responsible for the 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. However, these new agents would imply additional interaction issues, which are discussed below (Section 5.3).

So far, we have considered how and where GUI parts are created in PAC. Now we need to assemble them. PAC does not address this issue explicitly (as given in [POSA I] or [Coutaz87]). In general, agents should be extended to provide functionality for GUI assembly. 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 therefore be initialized 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 to the tool palette agent, its child. All this just increases coupling among agents and their complexity.

4 Handling User Input

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

After a callback method has intercepted a user event, it must be processed in an application-specific way. A typical way, which is often used in practice (Smart UI approach!), is to implement application-specific event processing logic in the callback methods. This works rather well in simple cases. However, application-specific event processing may be rather complex and involve components from business and/or presentation logic. So, we believe that a better design would be to just intercept the user events in the callback methods (we call this *primary event handling*) and forward them to other application components for application-specific processing.

In large GUIs, comprised of dozens of widgets and providing rich interaction features to the user, primary event handling code implemented in the callback methods may become rather large. A proper organization of this code is therefore an important task.

Let's look at how the patterns address this issue.

4.1 Handling User Input in MVC and HMVC

In MVC, user input is handled by the controller component, which translates user input 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 handles events on the corresponding widget (the view). We first create the widget. We then create its controller and associate the controller with the widget. From the implementation viewpoint, we have to provide 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. If you work with an IDE or a visual programming tool, you can get the event handling code (empty callback methods) generated automatically for you by the IDE. However, modifying this code to get pure MVC design would be a bad idea.

As for the GUI creation and assembly, we get similar problems with event handling. 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 methods. This unnecessarily complicates the design. The main source of such controller classes would be menu items, tool bar buttons, and widgets of complex dialogs.

Therefore, we find it more reasonable to shift to the layer of visual containers, as it is done for the GUI creation and assembly problem (container-level MVC). There, certain visual containers within the GUI are implemented as container-level views. For each such view, we have to define the controller component. Such a controller handles all the events that occur on the visual container itself, as well as on all the child widgets of the container. Thus, we avoid having a separate controller class for each particular widget.

This idea is also used 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 presentation from user input handling, resulting in two different logical components (the view and the controller), while HMVC handles both issues in one component (the view).

As an illustration, let's consider how a container-level controller could be implemented in Java Swing. A common way to handle a user input event on a widget is to provide a class that implements the corresponding `Listener` interface, where event handling callback methods are declared. Such a listener class is then registered 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, say, a text field and a couple of buttons among other widgets.

Consider first pure MVC approach. Here, we have a separate controller. The first implementation alternative is when the controller class implements the required `Listener` interfaces. In our example these would be `ActionListener` and `KeyListener` (if we want to handle each particular keyboard input event on the text field), as shown in Listing 4.

Listing 4: Controller class implementing Listener interfaces

```
public 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
   }
  ...
}
```

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 (the view), which registers the controller as event listener on all its widgets (Listing 5).

Listing 5: Registering controller on widgets

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

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 add the corresponding getter methods to 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 field in the example above fire `ActionEvents`). 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 then compared to each widget of interest to find out the event source. With this, we get conditional if-else-…-else blocks within callback methods, which may get rather large for complex views (imagine, for instance, such an if-then-else construct for a menu bar with all its content!). But it also means that the controller class must have access to the widgets of the dialog class!

The second alternative is to implement the MVC controller with Java anonymous inner classes. Using this feature of Java language, the controller does not need to implement `Listener` interfaces anymore, but needs to have access to the view's widgets. Listing 6 illustrates this approach (we assume that `myButton` and `myTextField` class members are defined with package access in the `MyDialog` class, see also Listing 5). Thus, we do not have anymore to work with cumbersome if-else-…-else constructs. All the event handling code for a particular widget is now localized 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.

Listing 6: Implementing controller with anonymous inner classes

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

      this.dialog.myAnotherButton.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
  }
  ...
}
```

We can also use 'normal' inner classes as event handlers for particular widgets inside the controller class. 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. First, we avoid the visibility problem inherent to MVC where the controller is implemented as a separate 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]. The view registers itself as a listener with each of its child widgets, which results again in if-else-…-else blocks within the callback methods to find out the source of an event.

The second alternative is to have anonymous inner classes within the view class that implement callback methods (see also Listing 6). We find this solution more elegant, as we do not have to do with the if-else-…else blocks.

In general, the view in HMVC is more complex than in MVC, since it is responsible for GUI creation and assembly, as well as for user input handling.

4.2 Handling User Input in PAC

In PAC, user input handling, along with the GUI concerns, is assigned to the agent's presentation component. As for the GUI creation and assembly problem, the details of how event handling could be implemented are left open. In general, it would depend on how the GUI is implemented, which has been discussed in Section 3.3. So, the implementation is completely up to the developers, and we just give some general ideas.

If agent's visualization is simple, we implement the agent's presentation component as a single class. Event handling is provided in this class 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. Each complex visual component is implemented in a separate class, probably with some additional helper classes (see also Section 3.3). So, we need to provide event handling for these components. As PAC does not specify how this should be done, we can, for example, apply the solutions for MVC and HMVC discussed in the previous section. 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. The callback methods process the events and send them to the agent's control component, or to the Façade object that could be used to hide the inner structure of the agent's presentation from the agent's control component.

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 discussed in the following sections.

5 Dialog Control

The problems discussed so far pertain to the presentation logic and its main responsibilities – user input and output. The application's presentation, or GUI, part has to be coupled with the business logic, where the user input is processed and the output is provided that is then displayed to the user. 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 create UML diagrams. A typical usage scenario within such an editor would be to change some properties of a UML element. This can be done in the following manner: The user selects the required UML element in the navigation tree, gets a popup menu by clicking the right mouse button, and then selects the Properties menu item in the popup menu. Upon this action, the Properties dialog is displayed, where the user can edit various data related to the selected element. When the user clicks the OK 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 on 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 first the popup menu and then the Properties dialog, which must be populated with the data (state) of the selected UML element. After the input has been processed, we must communicate state changes to several parts within the GUI.

How could such an interaction be designed and implemented? And where should the corresponding logic reside, i.e., how do we assign responsibilities among our components? For instance, an interesting question in the above scenario is who (which object) creates and displays the popup menu and the Properties dialog?

The logic responsible for user-system interaction is often put into the event handling callback methods (the Smart UI approach). This works rather well in small applications with simple interaction scenarios (such as the calculator example from Section 2.3).

However, for complex GUIs we'll face certain difficulties. Consider again the UML editor example and implementation of the popup menu. We may create and display the Properties dialog in the callback method for the Properties menu item. The problem is that we must populate this dialog with the data of the selected UML element. As this data is obtained from the business logic, we have to access the business logic from the callback method. Another solution would be to provide the Properties dialog with a kind of identifier of the selected element and let the dialog obtain itself the required data from the business logic. The dialog also needs access to the business logic to send the user input when the OK button is pressed. After the input has been processed, the state changes to the navigation tree and the drawing pane. This could be done either by the dialog or by the business logic. In both cases, we get additional dependencies among different parts of the application, which need to be initialized and maintained (the visibility problem [Marinilli06]).

Another problem is to provide data type transformation, since in general, the data types used in the GUI toolkit widgets and the business logic are different.
To address these issues, the dialog control logic is provided, which controls and manages complex user-system interactions and performs data type transformations. The dialog control decouples, and glues together, the presentation and the business logic. In this section, we discuss how the patterns address the dialog control problem.

5.1 Dialog Control in MVC

In the 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 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., zooming).

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) and controllers, 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 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 more complex interactions, where several GUI and business logic elements are involved. As a result, the developers have to find their own solutions.

Let's take the above scenario with the Properties dialog in the UML 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 the tree-like navigation area MVC, the drawing pane MVC, the Properties dialog MVC, and the popup menu MVC (in the latter case, the corresponding model component 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 as follows:

1. 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 or at least some specific menu items within the 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 and associated with the popup menu here as well (alternatively, the popup menu could create its controller itself).

2. When the user selects the Properties menu item, the event is handled in the controller of the popup menu. The controller's event handling callback method must create and display the Properties dialog (the view) and associate it with the corresponding controller and model, which is the same business object as for the popup menu.
3. As the Properties dialog is displayed, the user performs the required changes and presses the OK button. This event is handled in the controller of the Properties dialog MVC, which gathers user input and sends it to its associated business object (the model) by calling the corresponding method on it.
4. Upon this, the business object validates the user input and changes the object's state. The next step is to communicate this change to the views. These are the dialog itself, the drawing pane, and the navigation area. Moreover, if the model state has been saved prior to this state change, the File menu and the tool bar should be notified as well to enable the Save menu item and the Save button, respectively.

In order to implement the state change notification in Step 4, we need to register all the GUI components (views) being notified with the model component of the Properties dialog MVC. And this should be done in the controller (namely, in the corresponding callback method) of the popup menu MVC, which creates and initializes the MVC-triad representing the Properties dialog (Step 1). This implies 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 components (the visibility problem!).

Another problem is that registering the drawing pane view on the model of the Properties dialog MVC would violate the MVC design, since we get 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. This 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 is discussed in section 6.1.

In general, the problems arise in scenarios where interaction involves several MVC-triads. In complex GUIs 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.

5.2 Dialog Control in HMVC

HMVC tries to address the dialog control issues inherent to 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 as the root (see also Section 3.2). The triads are connected through the controller components. The resulting controller hierarchy is responsible for the communication among different MVC-triads, 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. In particular, it performs data type transformation. Hence, each control component has two basic roles – it provides internal interaction between agent's parts and external interaction with other agents [POSA I].

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

For complex presentation parts, we have to provide interaction among visual components that comprise the agent's presentation. Recall the graphical editor example from Section 3.3. The presentation part of the drawing pane agent includes the drawing pane, the tool palette, maybe several dialogs and popup menus, which interact with each other. For instance, upon selecting a menu item in the popup menu, certain dialog has to be displayed, while when the user submits this dialog, the drawing pane might need to be updated. A straightforward way would be to implement the interaction functionality within the visual components themselves. However, this may result in a tight coupling among them, if the interaction is intensive. The Mediator pattern could be applied to decouple the visual components from each other. As an option, the mediation role could be assigned to the Façade object, which is used to hide the visual components from other components of the agent (as discussed earlier in Section 3.3). A potential problem here is that the Façade object may get rather complex. Basically, it is again the agents' granularity issue, where the developers have to find balance between the agent number and agent complexity.

Interaction among agents is more complex. In general, it is about exchanging command, data, and state change events. For example, upon selecting a menu item, the top-level agent, which is responsible for menus, must send the command message to the corresponding lower-level agent(s). Regarding data exchange, the entire application model in PAC is implemented in the abstraction component of the top-level agent [POSA I]. 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. As a consequence, when the user provides some input to a bottom-level agent, this input is first sent up to the top-level agent, which processes the input and changes the system state in its abstraction component. The state 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.

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

- **Composite Message Pattern**. Agents provide two methods to send and receive messages, containing the information about what is sent, like message type and content. For incoming messages, the developer must implement the logic that analyses the content of a 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.
- **Agent-specific Interfaces**. Each agent implements its specific interface used by other agents to interact with it. With this, we can avoid having complex if-then-else constructs inherent to the Composite Message pattern, but the agents become tightly coupled to, and dependent on, each other.

The drawbacks of both approaches are tolerable per se. However, they may become a real nightmare when applied in PAC. The reason is that PAC agents do not interact with each other directly, but through the agent hierarchy organized around the control components. For instance, the bottom-level agent in the data exchange scenario does not interact with the top-level agent, but rather 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 a Composite Message approach, we'll get large if-then-else decision-making block within the message receive method. When using the 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.

We find both approaches completely unsuitable for large and complex GUIs. Again, the reason lies, to a greater extent, in the hierarchy-based agent interaction. It is a huge effort to implement such an interaction mechanism, which is also difficult to adapt and extend. For instance, if you have to extend the interaction logic between two agents, all the intermediate agents must be adapted to provide it, either through changing the message forwarding code in the if-then-else constructs, or through extending the agents' interfaces.

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 lying between two interacting agents in the agent hierarchy be aware of, and participate in, the interaction between these two agents? Why can't these agents interact directly with each other? One possible answer is that direct interaction among agents would break the tree-like agent hierarchy.

Given these problems, other approaches have been proposed. [POSA I] specifies also an agent interaction mechanism based on the Publisher-Subscriber [POSA I] or Observer [GoF] patterns. All the agents (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 a notification, the dependent agents update their states by retrieving the data from the publisher agent. Therefore, agents interact directly with each other. The Composite Message pattern or agent-specific interfaces approach could be applied here. The only problem is the registration, as the interacting 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. Publishers provide events of certain type for a specific channel. Subscribers register themselves with this event channel for events they want to receive. So, the subscribers must know only the event type.

[HO07] proposes a Hierarchical Service Locator mechanism. First, each agent (using PAC terms) implements the Service Locator interface and registers its interface for interaction with other agents. When an agent needs to communicate with another agent, it performs lookup. 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. After that, the agents interact directly with each other through interfaces, which results in tight coupling among agents. Another issue is that the agents have to implement the Service Locator functionality.

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

As the dialog control logic glues together the application's presentation and the business logic, it is important to know how the latter is designed, and how it can be accessed. . For example, Domain Model or Transaction Script patterns [Fowler02] could be applied when implementing the business logic. In this section we discuss how it is treated by the patterns for GUI applications.

6.1 Business Logic in MVC

Business logic is handled in MVC by the model component(s). In section 5.1 on the dialog control issues in MVC, we have outlined the problem of what the model actually is. Here, we discuss this issue in details.

In small applications with simple business logic, we may have just one model component. View(s) and controller(s) work with this single model.

In large applications, the underlying business logic is usually rather complex, and involves various business objects and relationships among them. With the Domain Model pattern [Fowler02], a business object is implemented as a separate class or a set of related classes. The following questions arise in this regard:

- What are the MVC model components in applications with complex business logic?
- How do the business objects relate (or could be mapped to) the model components?

A straightforward way of applying MVC is to consider each business object that is displayed to the user as an MVC model and associate it with the corresponding view(s) and controller(s). The main problem with this approach is that we have to design business logic in the MVC way. In particular, we have to put the change notification mechanism (using the Observer pattern) into the business objects, thus mixing the concerns. In some case, business logic may already exist, so it could be highly expensive or even not feasible to add the MVC-specific functionality. We might also have views that display a number of business objects (e.g., trees or lists) implying that these views depend on several models (see also section 5.1). Technically, we must extend the change notification mechanism for such views, so that they are able to find out which of the associated models has fired a change notification event.

In order to avoid having MVC specifics being implemented in the business logic, we can add a layer of indirection. For instance, we can have a separate model component for each business object. In this case, the model holds the data to be displayed, while the business object implements the corresponding business logic. 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 logic from the presentation and (part of) dialog control issues.

Business logic is often hidden behind Façade objects, where each Façade is responsible for one or several (usually related) use cases. In such a case, we can consider Façades as MVC models. In particular, they have to implement the change notification functionality. As Façades are more coarse-grained that particular business objects, each one will have more views (and controllers) associated with it. So, we might need to introduce event types and let the views subscribe only for events they are interested in. Otherwise, a Façade would notify all its associated views upon each single state change.

6.2 Business Logic in HMVC

HMVC does not assume business logic be implemented in the model components (in contrast to pure MVC). In general, the developers are free in how they implement the models. For instance, the model within an MVC triad can keep the data to be displayed and retrieve data from an application server or a database. Hence, the models decouple the presentation from the business logic (acting as Façades). As an important consequence, the developers are not forced to incorporate HMVC specifics into the pure business logic, which could be implemented the way the developers like.

6.3 Business Logic in PAC

In PAC, the application's business logic 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' [POSA I], that is, business objects from the business logic. The state of the business object is then kept 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 section 5.3 on the dialog control problem in PAC).

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 the container-level MVC approach 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 on 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 et al. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[Fowler02] M. Fowler et al. Patterns of Enterprise Application Architecture. Addison Wesley, 2002.

[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).