User Interface Patterns

Dorin Sandu [1]
sandu@acm.org

August 4, 2001

[1]School of Computer Science – Carleton University, 5302 Herzberg Laboratories, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6

Contents

CHAPTER 1

User Interfaces

Application user interfaces should be easy to build, modify, and maintain. The obvious way to meet this goal is to have all developers consistently use certain implementation patterns such as the *Model View Controller* [2], *Model View Presenter* [4], *Observer* [3], and the pattern language for developing form style windows [1]. After building user interfaces for awhile now, we found that more details are required to have the application interact with its user interface in a consistent way. In this chapter we introduce these new patterns along with a set of patterns that promote user interface reuse.

1.1 Introduction

Modern graphical user interfaces are hierarchical collections of visual components built around the *Composite* pattern [3]. Top-level root components, such as windows and dialogs, manage primitive and composite components, where the latter may contain additional nested primitive and composite components. All components interact with the application via some glue code that tells each component what aspect of the application model to operate on. In this chapter we will refer to the user interface object as the object that initializes the root visual component along with its nested visual components collection, visually positions them inside their parent containers, and implements the glue code between the application and the components.

User interface code, like with any other code for that matter, should be easy to understand, modify, and extend. This is hard to achieve if programmers use unique programming styles. While each of them may understand their own code, it takes a lot more time than desired to understand, modify, and extend somebody else's code. Therefore, you need an approach enabling all programmers, not just the original one, to work on an interface implementation whithout spending too much time understanding somebody else's code.

The obvious way to meet this goal is to use an architecture that standardizes the design. Patterns and pattern languages are excellent vehicles for expressing common approaches to designs and architectures. For example, patterns such as *Model View Controller* [2], *Model View Presenter* [4], *Observer* [3], and the pattern language for developing form style windows [1] already provide a partial set of patterns for building user interfaces:

- The *Model View Controller* pattern divides the user interface into three parts: the model, the view, and the controller. The model represents a domain level object, for example, it holds application data and provides behavior to access and modify it. The view represents a visual component that renders the model onto the screen and the controller changes the model in response to user input. The relationship between the model, view, and controller is depicted in Figure 1.1.

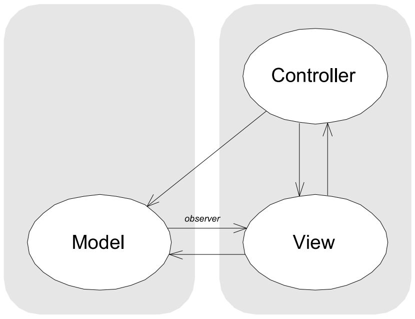


Figure 1.1: Model View Controller.

- The *Model View Presenter* pattern is a variation of *Model View Controller* that also divides the user interface into three parts: the model, the view, and the presenter. As with the *Model View Controller* pattern, the model represents a domain level object and the view renders the model to the screen. Additionally, the view handles low-level user input events which it forwards to the presenter. The presenter matches a model with a view object and modifies the model in response to the low-level events it receives from the view. This relationship is depicted in Figure 1.2.
- The *Observer* pattern manages the notification and updating between dependent objects called observers and subjects. It describes how to notify observer objects when a subject changes state, without forcing the subject to know the specific classes of the observers. *Model View Controller* and *Model View Presenter* use *Observer* to decouple the model from the view so, when the model changes, it notifies all its views to update. This way, there can be many views operating on the same model at the same time, whithout the model explicitly knowing about all of them.

You can use *Model View Controller* and *Model View Presenter* and still develop hard to maintain code. Assuming you will implement the user interface using one of

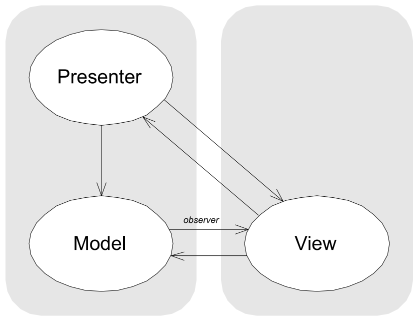


Figure 1.2: Model View Presenter.

these patterns, and the visual components in your user interface already implement one of these patterns, you will have to set dependencies, via *Observer*, between the user interface model and the visual components. In other words, each visual component will have to observe some aspect of the user interface model, and update itself when that model aspect chages. You will find the resulting dependency graph is hard to understand and maintain.

However, there are some higher level design decisions you can apply along with these patterns to improve user interface design and implementation. We incorporated these decisions into the patterns *Event Handler*, *Complete Update*, and *Multiple Update* and used these patterns to refactor *Subform*, *Alternative Subforms*, *Subform Selection*, *Subform Match*, and *Subform Mismatch*. Use these patterns in one of the following contexts:

- *You are developing a new user interface.* Windows and dialogs are top-level visual containers that contain a title, border, menu bar, and other visual components such as buttons, lists, and forms. Your user interface is responsible for the layout of other visual components. It is also responsible for mapping messages generated by the visual components, normally initiated from user actions in the user interface, to the appropriate actions in the models. This task ensures the user interface and its visual components accurately reflect the state of the models, and that it can handle its own events such as window or dialog closing.

- *You are developing a new form or subform.* A form or a subform is just a simple container class. It provides space in which to layout other visual component, including other forms but not a window or a dialog. A form is similar to a window in behavior, but it does not have the responsibilities associated with maintaining or manipulating a title or menu bar. Your new form is to be used eventually as a component of other windows or forms, or both. Therefore, your windows and dialogs can contain forms, and your forms can contain forms.

Additionally, use these patterns when:

- *The visual components contained in your user interface need to be loosely coupled.* The visual components depend on the models and must reflect their state.

However, when you update a visual component, this does not force the update of another and another, and so on. Rather, a user action in a visual component, requires a change in the models, which in turn requires a single update in the visual components, but that's where the process ends. For example, you are not building a spreadsheet view where a change in a cell may trigger a large number of other cells to update their contents.

- *The visual components use the Observer pattern to notify the views of changes.* Visual components could invoke specific methods in the user interface but this would increase the coupling between them. Since many windowing systems, including Java, already support this form of notification, we find that more details are required to handle the event notification from the visual components and for updating the visual components.

Within the global contexts, our three synchronization patterns, *Event Handler*, *Complete Update*, and *Multiple Update* along with the five subform patterns, *Subform*, *Alternative Subforms*, *Subform Selection*, *Subform Match*, and *Subform Mismatch*, attempt to resolve the following global forces:

- *User interface functionality changes often.* As an application evolves from early prototypes to a shipped product, practical experience suggests that its overall look-and-feel will change much more than the underlying business model. Such changes can occur as a result of changing business requirements, ports to different platforms, the desire to construct a more user-friendly version of an interface or from clients requesting a custom user interface. Examples of changes include laying the components out differently, moving its components to another user interface, and changing its behavior. All types of changes must be easy to make but not require a rethinking of the overall user interface design.

- *A significant period may elapse between the time a developer last worked on a user interface and the time he is making changes to it.* As a developer works on different parts of an application, weeks or even months can go by before he revisits a view that he developed. It should not take a long time for the developer to remember and understand his view's implementation.

- *The original developer of a user interface is not always the same person that maintains or extends it.* Developers often change groups or companies during the time required to develop a single application. Therefore, the original developer may be unavailable or unwilling to discuss their previous design with a developer assigned to support the old view. As with the original developer, it should be easy for the new one to understand the previous design and implementation and for the same reasons. However, the easier a user interface's design is to understand the easier it is to change it. New developers often feel the need to make changes to suit their styles. These changes come at a time when the user interface should be stabilizing, not changing.

- *Tightly coupled user interface components are difficult to merge or split.* It should be possible to merge or split user interfaces with minimal effort. Ideally, when making changes, it is desirable to refactor the interface rather than throw it away and implement it from scratch. The cost of refactoring decreases significantly if parts or entire user interfaces can be merged or split, for example, one should be able to add and remove visual components with minimum disturbance to the existing ones or to the user interface itself. However, initial refactoring increases development times and costs.

- *Visual containers that have many constraints between their visual components, such as tree views and grids, are difficult and time consuming to update.* Updating one component at a time is simple to implement and understand. However, due to the constraints, cyclical dependencies, and changes to other components, a visual component may be updated several times before a full update completes. This approach is inefficient and produces a flickering effect on the window.
- *A large percentage of time is required for user interface development.* One often hears of view construction requiring 80% of the development effort. Therefore, it is important to try to minimize the cost required to develop and maintain user interfaces, keeping your effort level low at this part of development. However, your views are what end-users see and interact with, and you must take the time to make them visually appealing, easy to use, and ensure the user makes a strong connection between them and your application.
- *Ad hoc user interface designs are difficult to support.* User interfaces implemented in a consistent way are much easier to understand and maintain because everyone involved with the task works from the same design principle. Consequently, no developer is required to figure out what the other has done. On the other hand, this can make user interfaces too similar, constraining novelty and innovation.
- *User interfaces must be user-friendly.* A user interface that is slow to react to events or update itself in real-time, will frustrate any user. Users expect responsive user interfaces that operate in real-time. However, this balance between performance and the complexity of the underlying framework is hard to achieve.

1.1.1 Pattern Summary

Table 1.1 summarizes the problems addressed by the patterns in our pattern language and their solutions. In the following sections, we describe the three synchronization patterns *Event Handler*, *Complete Update*, and *Multiple Update* along with the five subform patterns *Subform*, *Alternative Subforms*, *Subform Selection*, *Subform Match*, and *Subform Mismatch*. None of them stand on their own, their dependencies on each other are shown in Figure 1.3.

Pattern	Problem	Solution
Model View Controller	How do you architect an interactive application?	Divide interactive application into three parts: model, view, and controller.

Pattern	Problem	Solution
Event Handler	How should a view, such as a form, handle an event notification message from its observable visual components?	In your user interface, create and register a handler method for each event message from your visual component.
Complete Update	How to you implement behavior in the user interface to update the visual components from the model?	Assume all visual components are out-of-date and update everything.
Multiple Updates	How do you implement a subform such that, when it changes its model, other parts of the main user interface depending on this model, update their contents to reflect the latest changes?	Have each subform notify its parent when it changes the model. The parent should react to changes in the subform via *Event Handler* and update its children components via *Complete Update*.
Subform	How do you design parts of user interfaces to operate on some model aspect?	Group the components that operate on the same model aspect into subforms.
Alternative Subforms	How do you design parts of user interfaces to operate on key variations of some model aspect?	Group the components that operate on every key variation of some model aspect into subforms. Select the appropriate subform at runtime using the *Subform Selection* pattern.
Subform Selection	How do you choose from a collection of subforms based on a given key variation of some model aspect?	Have the parent keep track of subforms in a collection and hide all except the subform that matches the given key variation of the model aspect. Select the subform using either *Subform Match* or *Subform Mismatch* patterns.
Subform Match	How do you determine that a subform can operate on a given key variation of some model aspect based on changes caused by the parent?	Implement a method for each subform that decides whether the subform can operate on the given key variation of some model aspect.

Pattern	Problem	Solution
Subform Mismatch	How do you determine that a subform cannot operate on a given key variation of some model aspect based on changes caused by the subform?	Have each subform notify its parent when it can no longer operate on the given key variation of data.

Table 1.1: Pattern Problems and Solutions.

The *Model View Controller* pattern represents the core of the pattern language. It provides the overall architecture for connecting models, views and controllers, and is used by the major user interface frameworks today. It is worth noting that Java makes use of a simpler version of the *Model View Controller*, denoted as the *model-delegate*. This design merges the view and controller into a single object that interacts with both the model and the user. In this chapter, we refer to the user interface object as the object that constructs the delegate by combining already implemented components like buttons and lists, and the glue code between the delegate and the model.

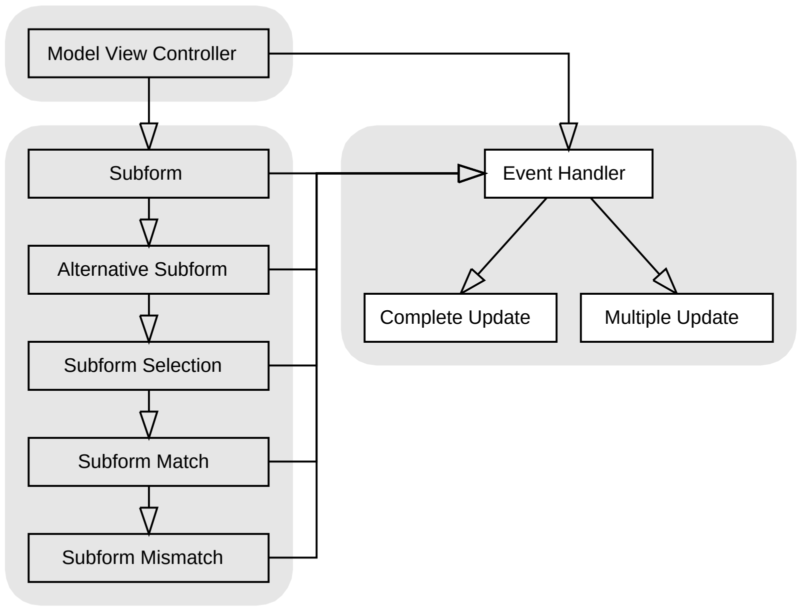


Figure 1.3: Pattern language dependencies.

The rest of the chapter is structured as follows:

- In Section 1.2, we describe how user interfaces using *Event Handler* handle event notifications from their visual components such that the update process between the model and delegate becomes independent from the visual components generating the events and from the models that have changed state.

- In Section 1.3, we describe how *Complete Update* can be used to structure the update code for a user interface such that you or any other future developer can quickly and easily understand the logic of the update.
- In Section 1.4, we describe how *Multiple Update* can be used to manage the update of visual components arranged in multiple subforms.
- In Section 1.5, we describe how *Subform* can be used to divide a user interface into smaller interfaces that operate on a certain aspect of the model, such that these smaller user interfaces reduce the complexity of the code and can be reused in different other parts of the system.
- In Section 1.6, we describe how *Alternative Subforms* can be used to create different subforms to edit key variations of the same model aspects.
- In Section 1.7, we describe how *Subform Selection* can be used to manage the different subforms needed to edit different variations of some model aspect at runtime.
- In Section 1.8, we describe how *Subform Match* is used to choose among a collection of given subforms to operate on a given key variation of some model aspect.
- In Section 1.9, we describe how *Subform Mismatch* can be used to, once the model aspect being edited changed, prompt the user interface to pick another subform to operate on the new variation of the model aspect.
- In Section 1.10, we implement a possible user interface for the bank account example presented in the chapter on constructing objects earlier in this book. The interface consists of a main window, a dialog, and a couple of subforms whose implementation is an example of applying the above patterns.

1.2 Event Handler

You have incorporated the *Observer* pattern into the design of your user interface. This may have been a deliberate decision or the result of using another pattern, such as *Model View Controller* or *Model View Presenter*, that uses the *Observer* pattern as part of its structure. Your user interface object is interested in receiving event messages from its observable visual components when specific events occur in them. The user interface needs to collaborate with one or more model objects in order to handle these notification messages and then update its visual components in response. You want your user interface to update its visual components in an organized, linear manner, so that components in corresponding regions of the user interface update together rather than an apparent random manner.

How should a user interface handle event notifications from its observable visual components such that it does not create any updating dependencies between the methods and the number of updates to the visual components is minimized?

Consider the following Java implementation of a TodoList window, as depicted in Figure 1.4. The application model consists of a collection of todo items, where each item is a `String` object. The user can add new items to the list by typing one item at the time in the text field and clicking the Add button. The user can remove items from the list by selecting an item and clicking the Remove button.

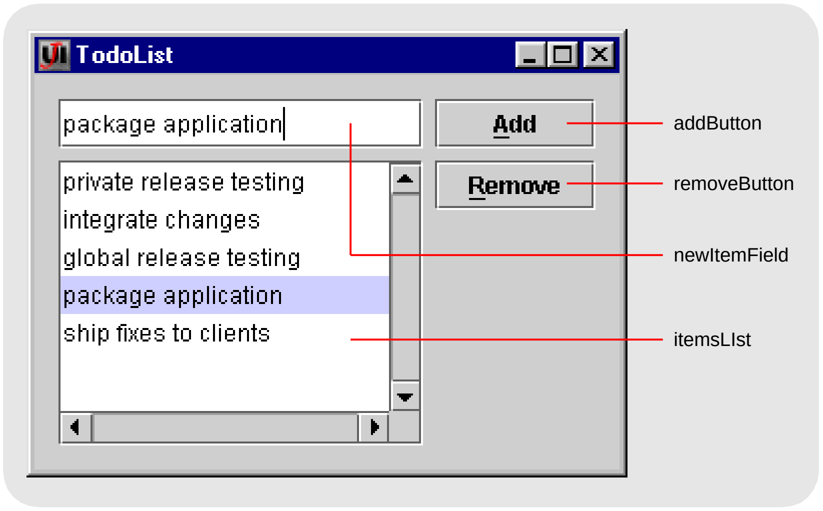


Figure 1.4: TodoList example.

The underlying model for the TodoList window is a set of three objects, the todo item collection, the todo item selection, and the todo item being edited. You encode these in class `TodoListWindow` as instance variables `items`, `selectedItem`, and `newItem`. Additionally, `TodoListWindow` creates, initializes, and manages each visual component along with a list of event listeners:

```
public class TodoListWindow {
  private String newItem, selectedItem;
  private ArrayList items;
  ...
}
```

The user can type in the text field, select an item in the list or click the Add and Remove buttons. Handler methods need to be written for each of these events. In Java, you need to add listeners to the appropriate visual components (code not shown in the interest of brevity) in order for the view to receive event notification messages from them. You can do this by having the user interface itself implement the appropriate interfaces. For example, `TodoListWindow` handles events from the two buttons via the `ActionListener` interface:

```
public class TodoListWindow implements ActionListener {
  ...
  public void actionPerformed(ActionEvent event) {
      JButton button = (JButton)event.getSource();
      if(button.getName().equals("AddButton")) {
        addTodoItem();
      }
      if(button.getName().equals("RemoveButton")) {
        removeTodoItem();
      }
    }
    ...
  }
```

Here, you have used conditional statements to determine which component sent the event message and how to process it in a single handler method. However, this is inefficient as now it becomes difficult to add, remove, or change the visual components. To do so, you have to edit every event handler method to either add or remove the code for the given visual component. Moreover, if you change the source of the event, you must always recompile the method. Replaceable, or plug-and-play, components will also not be an option.

Many observable visual components can generate identical event messages. If you handle each message with a different method, it is easy to add and remove visual objects as you have to add or remove one handler method. Therefore:

In your user interface, create and register a handler method for each event message from your visual components.

Do not register the visual components directly with the user interface model. Rather than letting the visual components operate on different aspects of the user interface model, as suggested by *Model View Controller*, have the user interface object coordinate this process. So, implement the event handler methods to pull information from the appropriate visual components and invoke corresponding services on the model. Then simply update the visual components to reflect the changes you just performed on the model. Your handler methods should not do any updating of the visual components themselves. Instead, have your handlers use a single update method that is responsible for updating all visual components.

Going back to the TodoList example, the `DocumentListener` for the text field sends the `TodoListFrame` the message `changedNewItemField()` every time the user types in the text field. The `changedNewItemField()` handler method reads the contents of the text field into the `newItem` instance variable:

```
private void changedNewItemField() {
  if(getNewItemField().getText().length() == 0) {
    setNewItem(null);
  } else {
    setNewItem(getNewItemField().getText());
  }
  update();
}
```

The `ListSelectionListener` for the list sends the `TodoListFrame` the message `changedItemsList()` when the user selects an item in the list. The handler method reads the selected item from the list into the instance variables `selecte-dItem` and `newItem`:

```
private void changedItemsList() {
  String value = (String)itemsView.getSelectedValue();
  setSelectedItem(value);
  setNewItem(value);
  update();
}
```

The Add and Remove buttons use an `ActionListener` to send the `TodoList-Frame` the messages `addTodoItem()` and `removeTodoItem()` when the user clicks on them, respectively. The `addTodoItem()` handler method adds the new item to the collection of existing items, whereas the `removeTodoItem()` handler method removes the currently selected item from the same collection:

```
private void addTodoItem() {
  getItems().add(getNewItem());
  setSelectedItem(getNewItem());
  update();
}
private void removeTodoItem() {
  getItems().remove(getSelectedItem());
  setSelectedItem(null);
  setNewItem(null);
  update();
}
```

All handler methods invoke a single update method just before they complete. The user interface model is modified as a result of handling user events, after which is loaded back into the visual components, during updating, to force the view to reflect the changes in the model.

Since only a single update is required, the entire user interface can be refreshed in a predictable, coordinate manner and redundant updates are avoided. After receiving an event message, the user interface object takes the required information from the appropriate visual components and invokes the corresponding services in the models. These actions change the states of the models and, consequently, the visual components are now in need of update to reflect the latest state of the model. All visual components, all events, and all updating is handled the same way in all views.

As a result, the user interface has become easier to maintain. To add the capability to handle a new event message from a visual component, add a new handler method. To remove the processing of an event, delete the corresponding handler method. To modify the way an event message is processed, change the corresponding handler method. Even more important, there is always one and only one method to look at for the addition, deletion or modification to the way an event message is handled, which makes it easy for you or future developers to understand. Moreover, no special flags or case statements are required to process events.

On the downside, *Event Handler* may make updating more complex. For very large user interfaces, a single update method may become too complex. To some extent, this can be solved having the main update method coordinate several smaller updates.

1.3 Complete Update

You have implemented your user interface with a clear separation between its model and its views, for example, the visual components do not operate directly on aspects of the model and do not automatically update themselves when the model changes. You have also used *Event Handler* to capture user events from the visual components. You handle these events in the user interface object and, after you finish modifying the model, you realize the visual components display an out-of-date version of the model.

How do you implement behavior in the user interface to update the visual components from the model?

Changes in the model objects may be the result of a user action or the action of another object the user interface does not know about. However, you want to handle both cases at once and avoid redundancy. The problem is you do not always know what has changed in the models or which visual components need updating.

You are not always the one who will maintain your views so the view's logic must be easily understood. You can often improve performance by using tricky implementation techniques. However, these tricks often complicate your code by making it less easy to read and comprehend. Therefore:

Assume all visual components are out-of-date and update everything.

In most cases, there will be superfluous updates which may cause the display to flicker, especially on systems that do not double-buffer their visual components' graphics. However, if the user interface is not too complex, most computers are fast enough to do the updates without recognizable flicker. Write a specialized update method for every visual components and subforms in your user interface. Each specialized update method is responsible for updating its corresponding visual components from information in the models. Permit each specialized update method to load model information directly into the visual components or compute the contents on the fly. Write a single, main update method that invokes all of the specialized update methods.

Continuing with the TodoList example, update methods must be written for all visual components. The update method for the text field sets the contents of text field to the string in `newItem` variable. The update method for the selection list sets the contents of the selection list to the collection of `items` and the selection to the `selectedItem` string:

```
private void updateNewItemField() {
  getNewItemView().setText(getNewItem());
}
private void updateItemsList() {
  getItemsList().setListData(getItems());
  getItemsList().setSelectedValue(getSelectedItem(), true);
}
```

The update methods for the Add and Remove buttons only take care of enabling or disabling the buttons depending on whether their events are permitted. The Add button cannot be clicked if there is nothing typed in the text field (in other words `newItem` is null). The Remove button cannot be clicked if there is no selection made in the list (in other words `selectedItem` is null):

```
private void updateAddButton() {
  getAddButton().setEnabled(getNewItem() != null);
}
private void updateRemoveButton() {
  getRemoveButton().setEnabled(getSelectedItem() != null);
}
```

The coordinating update method, which all handler methods use, invokes the specialized ones ensuring all visual components reflect the current state of the models:

```
private void update() {
   updateNewItemField();
   updateItemsList();
   updateAddButton();
   updateRemoveButton();
}
```

As a result, the user interface main coordinating update method ensures that its visual components reflect the current state of the models. It is very easy to determine how the visual components are updated as there is only one corresponding update method to add, remove, or change. This method does not make use of any special boolean flags to remember which visual components to update. On the downside, this approach may cause infinite loops and, in the case of complex user interfaces, may be too slow.

1.3.1 Variants

Infinite Loops

Complete Update can cause infinite loops when messages sent to one of the visual components in the update method trigger a new event message from the visual components, causing the subsequent execution of a handler method, which results in another invocation of the update method, and so on. To prevent this situation from occurring, you can disable the visual components from generating new event messages while the user interface is handling an event message. In our example, this involves removing the listeners from the visual components. The `disableListeners` method removes the Java listeners from the corresponding components, and `enableListeners` adds them back. The net result is you update all visual components while the listeners are disabled:

```
private void update() {
  disableListeners();
  updateNewItemField();
  updateItemsList();
  updateAddButton();
  updateRemoveButton();
  enableListeners();
}
private void disableListeners() {
    getNewItemField().getDocument().removeDocumentListener(newItemDocumentListener);
    getItemsList().removeListSelectionListener(itemsListSelectionListener);
    getAddButton().removeActionListener(addButtonActionListener);
    getRemoveButton().removeActionListener(removeButtonActionListener);
  }
  private void enableListeners() {
    getNewItemField().getDocument().addDocumentListener(newItemDocumentListener);
    getItemsList().addListSelectionListener(itemsListSelectionListener);
    getAddButton().addActionListener(addButtonActionListener);
    getRemoveButton().addActionListener(removeButtonActionListener);
  }
```

The user interface creates the listeners when it builds the window, associates them with the appropriate component, and caches them in private instance variable. You do not create new listeners (observers) or visual components (subjects) when disabling and enabling event messages as this would be inefficient. You use these references to enable and disable the corresponding listner objects. Taking this approach, the methods for disabling and enabling are simple to understand and modify. However, this does open another place to modify whenever a component is added or deleted.

Stopping Recursive Updates Without Disabling/Enabling Listeners

As mentioned before, the updating of visual components by the user interface may generate additional event messages. Since the handling of these event messages always invokes an update which may cause more event messages, and so on, the result can be an endless loop of update message, event message, and handler responses.

Rather than disabling and enabling the objects responsible for generating the new event messages, which may be computationally expensive, another solution is to discard any new event messages while already processing one. To include this ability, you can add a new private instance variable to your view called `updating` and initialize it to `false` when creating your view. In our example, the class definition for the `TodoListWindow` is modified as follows:

```
  public class TodoListFrame extends JFrame {
    private boolean updating  = false;

    private JTextField newItemField;
    private JList itemsList;
    private JButton addButton, removeButton;

    private String newItem, selectedItem;
    private ArrayList items;
    ...
  }
```

You also need to provide the following methods:

```
  private boolean isUpdating() {
    return updating;
  }
  private void beginUpdating() {
    updating = true;
```

```
}
private void endUpdating() {
  updating = false;
}
```

Next, you need to change all handler methods to the following format:

```
private void changedItemsList() {
  if(isUpdating()) return;
  String value = (String)getItemsList().getSelectedValue();
  setSelectedItem(value);
  setNewItem(value);
  update();
}
```

Finally, you need to remove the `enableListener()` and `disableListener()` methods for the main update method and replace them by `beginUpdating()` and `endUpdating()`:

```
private void update() {
  beginUpdating();
  updateNewItemField();
  updateItemsList();
  updateAddButton();
  updateRemoveButton();
  endUpdating();
}
```

Now, no handler methods can execute until the current one has finished, including its update. In cases where different threads send and handle different event messages, the handler and update methods must be synchronized.

Partial Updates

As the user interface implementation stabilizes, and refactoring becomes less and less frequent, one can improve the approach's performance by implementing partial updates. In this case, you can modify the update methods to update the visual components only when the information in them does not reflect what is in the models, rather than always. This variant may also be required to stop any noticeable flickering of the visual compoenents content on slower machines. For example, the update method for the text field can be written as follows:

```
private void updateNewItemField() {
  if(!getNewItemField().getText().equals(getNewItem())) {
    getNewItemView().setText(getNewItem());
  }
}
```

Another way to improve performance is to only update the visual components that need updating. To do this, you have to figure out the dependencies between visual components and, for every event handler, invoke only the update methods of its dependents.

1.4 Multiple Update

You are implementing a subform using the *Subform* pattern. You have used *Event Handler* and *Complete Update* to implement a number of handler and update methods to keep the visual components inside the subform up-to-date with the subform model. You are using the subform as a component inside a larger user interface, and the subform is related to the other components in one of three ways:

- One of the components may be the subform parent.
- Some of the components may be children of the subform.
- Some of the components may be siblings of the subform.

The root of a subform is usually a window, dialog or, in Java, an applet. This root object can contain additional subforms which, in turn, contain other subforms and visual components. Although an update may start anywhere in this hierarchy, any or all others participant components and subforms in the main user interface may need to go through their own update process.

How do you implement a subform such that, when it changes its model, other parts of the main user interface depending on this model, update their contents to reflect the latest changes?

We use subforms as plug-and-play components so they must be easy to install or remove from the user interface. For example, if you have an address view that works with a Canadian address, you may need to change this to a US address view when working with a US address object. The switching between subforms should not demand a large coding effort and should not affect the logic of the other components in the user interface.

We also tend to use subforms in dynamic user interfaces. In some applications, often involving notebooks or tab controls, parts of the user interface are swapped dynamically. This requires that user interface objects and their update mechanisms must work for all update situations, not just situations involving specific subforms. It would be wise then not to hardcode the updating between views.

Consider the following Java implementation of a multiple TodoList window, as depicted in Figure 1.5. This time we implement the todo list as a subform, the corresponding buttons, list and text field are assembled in `TodoListPanel`, a subclass of `JPanel`. The example consists of a single window with two child panels. The window and panels share common models: a collection of todo `items`, one `selectedItem`, and the `newItem` to be added. The functionality of the window is the same as in the original TodoList. However, you would like to have user actions reflect in both panels, for example, if an item is removed from one list, the changes should immediately reflect in the other list as well.

```
public class TodoListPanel extends JPanel {
  TodoListPanel sibling;
  ...
```

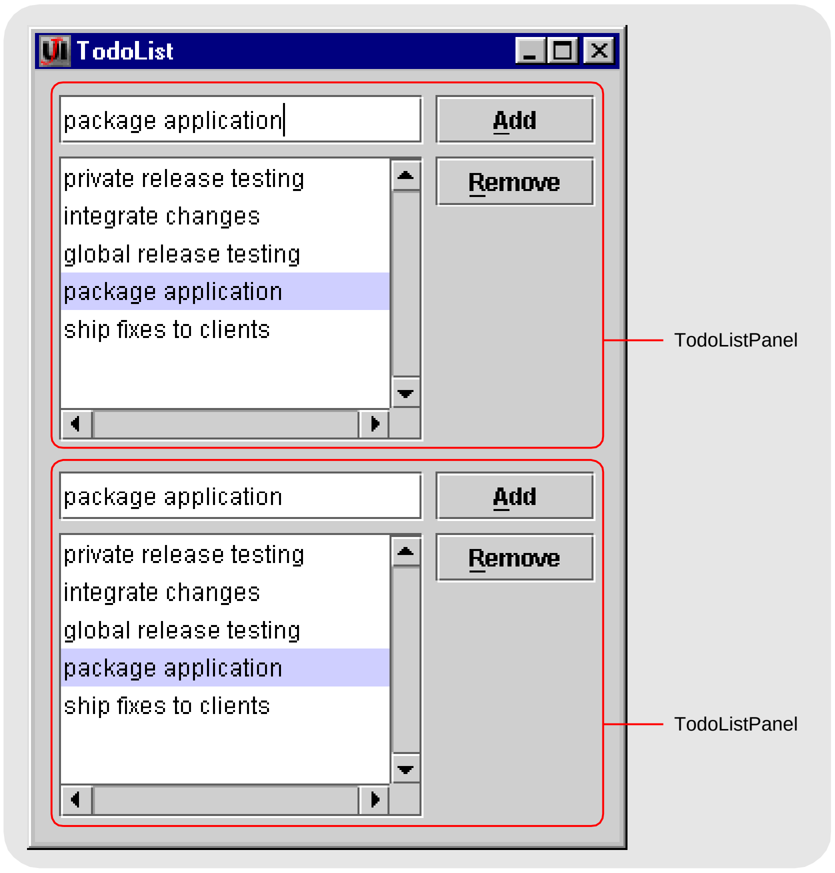


Figure 1.5: Multiple TodoList example.

```
    private void removeTodoItem() {
      getItems().remove(getSelectedItem());
      setSelectedItem(null);
      setNewItem(null);
      getSibling().update();
      update();
    }
  }
```

It is not scaleable for every subform to keep track of other subforms it needs to update. The two TodoList panels in the window can keep track of each other and, when one changes the model, tell the other one to update itself from the new model. Alternatively, you can use the *Observer* pattern with the two panels playing both the parts of observers and subjects, allowing each panel to be notified when the other changes. However, both approaches create dependencies between the panels when the dependencies should really be between the subforms and the models. Therefore:

Have each subform notify its parent when it changes the model. The parent should react to changes in the subform via *Event Handler* and update its children components via *Complete Update*.

We leave out the full implementation of the window and panels in the interest of brevity and focus on how updating occurs. In short, the handler and update methods for each panel are almost identical to the examples given before. However, in this situation, you must handle the fact that multiple panels may require updating. The update method for each of the panels invokes the update methods for their visual components, as before:

```
public void update() {
  beginUpdating();
  updateNewItemField();
  updateItemsList();
  updateAddButton();
  updateRemoveButton();
  endUpdating();
}
```

The event handlers however, trigger a change event by invoking `stateChanged()` on the listener object:

```
public void TodoListPanel extends JPanel {
  private ChangeListener listener;
  ...
  public ChangeListener getChangeListener() {
    return listener;
  }
  public void setChangeListener(ChangeListener listener) {
    this.listener = listener;
  }
  ...
  private void changedNewItemField() {
    if(getNewItemField().getText().length() == 0) {
      setNewItem(null);
    } else {
      setNewItem(getNewItemField().getText());
    }
    getChangeListener().stateChanged(new ChangeEvent(this));
    update();
  }
  private void changedItemsList() {
    String value = (String)itemsView.getSelectedValue();
    setSelectedItem(value);
    setNewItem(value);
    getChangeListener().stateChanged(new ChangeEvent(this));
    update();
  }
  private void addTodoItem() {
    getItems().add(getNewItem());
    setSelectedItem(getNewItem());
    getChangeListener().stateChanged(new ChangeEvent(this));
    update();
  }
```

```
    private void removeTodoItem() {
      getItems().remove(getSelectedItem());
      setSelectedItem(null);
      setNewItem(null);
      getChangeListener().stateChanged(new ChangeEvent(this));
      update();
    }
  }
```

The listener instance variable is used to keep track of the listener object provided by the parent window at construction time. This object will be notified every time some change occurs in the TodoList panel. Class `TodoListWindow` sets the listener to a inner class instance that, when notified, invokes the update method for the entire window:

```
  public class TodoListWindow {
    TodoListPanel panel1, panel2;
    ...
    private void initComponents() {
      ...
      getPanel1().setChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent event) {
          update();
        }
      }
      getPanel2().setChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent event) {
          update();
        }
      }
    }
    ...
    private void update() {
      beginUpdating();
      updatePanel1();
      updatePanel2();
      endUpdating();
    }
  }
```

As a result, from the single initiation point, update requests traverse the main user interface component hierarchy. Eventually, all subforms execute their updates and the components will reflect an accurate display of the underlying models. No magic flags are required by the approach and subforms and components can be changed, added or removed without having to make major modifications to existing subforms and components. The approach is so simple that any developer, not just the original one, can work on a view that uses it without knowing much about what coding was done in the past or in other views.

Unlike the *Model View Controller* that initiates synchronization of the visual components from the models, resulting in an unordered updating of them, this pattern synchronizes updating of the visual components after all models have completed their change of state, leading to a more uniform and ordered updating of the visual components.

Also, unlike the *Model View Controller*, which forces the setup and maintenance of dependencies between all models, views and controllers, in the *Multiple Update* pattern, the dependencies are only between the parent and children, which are usually maintained by the windowing system.

1.5 Subform

You are implementing a graphical user interface, possibly using the *Model View Controller* [2] pattern alone or in combination with the *Event Handler*, *Complete Update*, or *Multiple Update* patterns. Each view is responsible for editing some aspect of the model, and some of the views are related to each other as they operate on the same aspect of the model. For example, the interface for a bank account may allow users to edit owner information along with account information. The owner information is one aspect of the bank account model that can be operated on by text field visual components for first and last names along with a combo box widget for the salutation. In a similar way, the account information is another aspect of the bank account that can be operated on by text fields for balance and interest rate.

How do you design parts of user interfaces to operate on some model aspect?

For the bank account example, consider the implementation of a dialog to edit a bank account. Bank accounts have some integer id, a balance, and owners, represented by objects of type `Person` with `lastName` and `firstName` attributes. The definition for the classes involved looks as follows:

```
public class AccountDialog {
  private Account account;
  private JDialog frame;
  ...
  public Account show() {
    getFrame().show();
    return getAccount();
  }
  public JDialog getFrame() {
    // construct dialog here
    return frame;
  }
}

public class Account implements Cloneable {
  private Person owner;
  private int id;
  private float balance;
  ...
}

public class Person implements Cloneable {
  private String firstName;
  private String lastName;
  ...
}
```

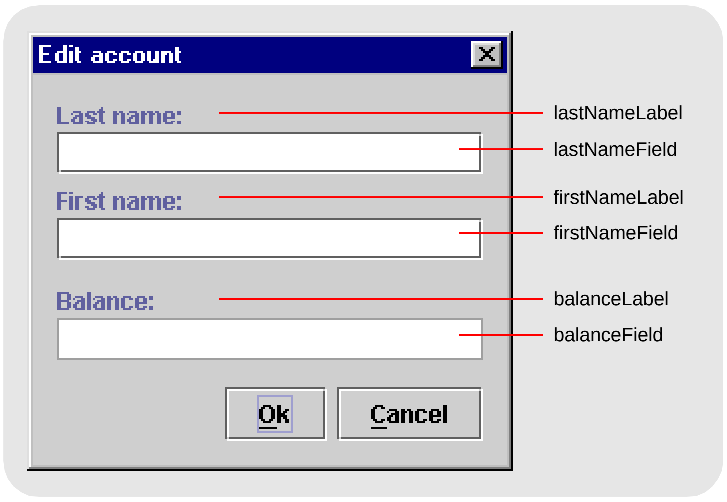


Figure 1.6: Account dialog.

To implement the dialog shown in Figure 1.6, you can have the dialog frame manage all the visual components required to edit the account. You can add the text fields for last name, first name, balance, and the corresponding labels to the dialog frame in the method `getFrame()` as follows:

```
public JDialog getFrame() {
  if(frame != null) return frame;
  frame = new JDialog(parent, true);
  frame.setTitle("Edit account");
  ...layout code not shown...
  frame.getContentPane().add(getLastNameLabel(), constraints);
  frame.getContentPane().add(getLastNameField(), constraints);
  frame.getContentPane().add(getFirstNameLabel(), constraints);
  frame.getContentPane().add(getFirstNameField(), constraints);
  frame.getContentPane().add(getBalanceLabel(), constraints);
  frame.getContentPane().add(getBalanceField(), constraints);
  update();
  return frame;
}
```

Each of the methods `getLastNameLabel()`, `getFirstNameLabel()`, `get-BalanceLabel()` returns an instance of `JLabel` instantiated with the appropriate text. The methods `getLastNameLabel()`, `getFirstNameLabel()`, and `get-BalanceLabel()` return instances of `JTextField` constructed with the appropriate listeners that invoke methods to edit aspects of the account object. The `update()` method at the end will invoke the update methods for every text field to load the current account values for last name, first name, and balance into the text fields when the dialog is created. The interface works as described in the *Event Handler*, *Complete Update*, and *Multiple Update* patterns.

The major disadvantage of this implementation is that you are creating extra dependencies between `AccountDialog` and `Person` by having the dialog itself know how to edit the owner of the account. In other words, you are violating the Law of Demeter which states that, in a given method, one should only access the objects created in the method, the instance variables of the receiver, the arguments of the method, or global variables. Consider the implementation of the methods for the last name field:

```
private void updateLastNameField() {
  getLastNameField().setText(getAccount().getOwner().getLastName());
}

private void changedLastName() {
  if(isUpdating()) return;
  if(getLastNameField().getText().length() == 0) {
    getAccount().getOwner().setLastName(null);
  } else {
    getAccount().getOwner().setLastName(getLastNameField().getText());
  }
  update();
}
```

As you can see from this code, the last name is accessed via:

```
getAccount().getOwner().getLastName()
getAccount().getOwner().setLastName()
```

This makes the account dialog dependent on person objects. As a result, a change in the person protocol that involves these two methods will affect the dialog implementation too. The same logic applies for the first name field. The obvious fix to this problem is to implement extra methods in the `Account` class:

```
public String getLastName() {
  return getOwner().getLastName();
}

public void setLastName(String lastName) {
  getOwner().setLastName(lastName);
}

public String getFirstName() {
  return getOwner().getFirstName();
}

public void setFirstName(String firstName) {
  getOwner().setFirstName(firstName);
}
```

Technically, this solves the problem, as now you can refer to the last and first names of the owner in the dialog with:

```
getAccount().getLastName()
getAccount().getFirstName()
getAccount().setLastName()
getAccount().setFirstName()
```

However, you note that the interface of the `Account` class has increased by four methods that do not really belong there, for example, it doesn't really make any sense to ask an account for its last or first name. A much better approach is to forget about modifying `Account` and have some other user interface object operate on the owner of the account. This new object that edits `Person` objects, should group toghether all the visual components editing person objects previously managed by the account dialog. Therefore:

Group the components that operate on the same model aspect into subforms.

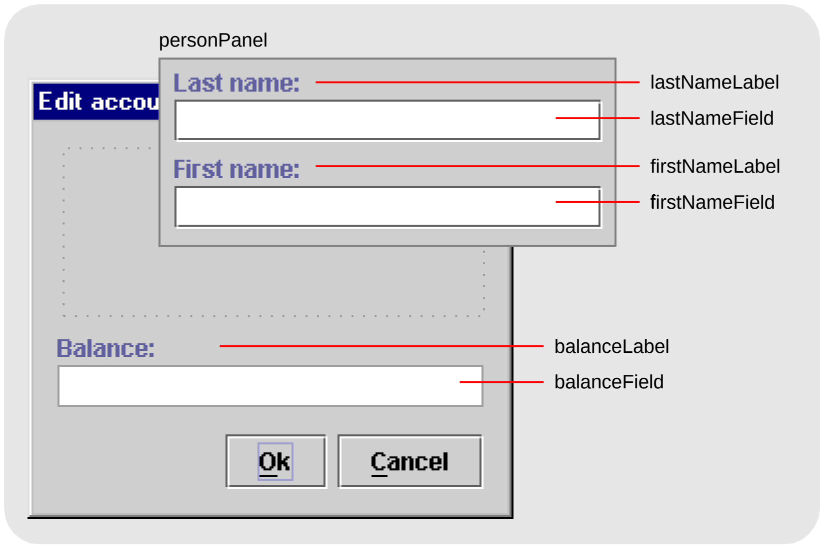


Figure 1.7: Account dialog with owner subform.

Forms are implemented in Java as subclasses of `JPanel` class, a generic lightweight container. Like other containers, a panel can manage a collection of Java visual components, and can itself be managed by some other container. The form to edit `Person` objects is implemented as follows:

```
public class PersonPanel extends JPanel {
  private Person person;
  ...
  public PersonPanel(Person person) {
    this.person = person;
    initComponents();
  }
  public Person getPerson() {
    return person;
  }
  public void setPerson(Person person) {
    this.person = person() {
    update();
  }
  ...
}
```

As you may have noticed, the public interface of `PersonPanel` is simple. The constructor accepts a person object as argument and constructs the visual components necessary to edit the last and first name in the private method `initComponents()`:

```
private void initComponents() {
  ...layout code not shown...
  add(getLastNameLabel(), constraints);
  add(getLastNameField(), constraints);
  add(getFirstNameLabel(), constraints);
  add(getFirstNameField(), constraints);
}
```

Other methods include the accessors for the `person` instance variable. The set method invokes the private method `update()` to update the form to reflect the state of the new person object. The form also contains methods to edit the last and first name of the person object:

```
private void updateLastNameField() {
  getLastNameField().setText(getPerson().getLastName());
}
private void updateFirstNameField() {
  getFirstNameField().setText(getPerson().getFirstName());
}
private void changedLastName() {
  if(isUpdating()) return;
  if(getLastNameField().getText().length() == 0) {
    getPerson().setLastName(null);
  } else {
    getPerson().setLastName(getLastNameField().getText());
  }
  update();
}
private void changedFirstName() {
  if(isUpdating()) return;
  if(getFirstNameField().getText().length() == 0) {
    getPerson().setFirstName(null);
  } else {
    getPerson().setFirstName(getFirstNameField().getText());
  }
  update();
}
```

These methods were previously implemented in `AccountDialog` and you moved them over to `PersonPanel` with minor modifications, for example, now you use `getPerson()` instead of `getAccount().getOwner()`. This is in accordance to Demeter's Law. Additionally, the implementation for the account dialog is now much simpler, as `AccountDialog` now only has to deal with one form to edit the account owner, as opposed to two labels and two text fields as before. The dialog is now created as follows:

```
public JDialog getFrame() {
  if(frame != null) return frame;
  frame = new JDialog(parent, true);
  frame.setTitle("Edit account");
```

```
    ...layout code not shown...
    frame.getContentPane().add(getPersonPanel(), constraints);
    frame.getContentPane().add(getBalanceLabel(), constraints);
    frame.getContentPane().add(getBalanceField(), constraints);
    update();
    return frame;
  }
  private PersonPanel getPersonPanel() {
    if(personPanel != null) return personPanel;
    personPanel = new PersonPanel();
    return personPanel;
  }
  private void updatePersonPanel() {
    getPersonPanel().setPerson(getAccount().getOwner());
  }
```

By creating a separate subform to edit `Person` objects, the implementation for `AccountDialog` has been greatly simplified and its dependencies to person types reduced. As an added bonus, the subform to edit persons can now be reused in other user interfaces. As a matter or fact, to make the code more modular, you should implement a subform for the account too.

This pattern has been previously proposed under a different form by Bradac and Fletcher [1].

1.6 Alternative Subforms

You are implementing a graphical user interface and have used the *Subform* pattern to group the visual components into subforms that operate on different aspects of the model. You realize that some model aspects change at runtime and these changes may modify the widget composition of the associated subform. For example, the visual components in the bank account information subform may have to change depending on the type of the account being edited.

How do you design parts of user interfaces to operate on key variations of some model aspect?

Consider extending the account dialog introduced for the *Subform* pattern to edit different accounts, in this case, savings and checking accounts. In addition to the balance, savings accounts provide a monthly interest rate, and checking accounts keep track of the number of checks issued to the account owner along with the serial number of the last processed check:

```
public class Account implements Cloneable {
  private Person owner;
  private int id;
  private float balance;
  ...
}
public class SavingsAccount extends Account implements Cloneable {
  private float interestRate;
  ...
}
public class CheckingAccount extends Account implements Cloneable {
  private String checkNumber;
  private int checkCounter;
  ...
}
```

Because you have already used *Subform* and implemented owner and account subforms, the creation method for `AccountDialog` looks like:

```
public JDialog getFrame() {
  if(frame != null) return frame;
  frame = new JDialog(parent, true);
  frame.setTitle("Edit account");
  ...layout code not shown...
  frame.getContentPane().add(getPersonPanel(), constraints);
  frame.getContentPane().add(getAccountPanel(), constraints);
  update();
  return frame;
}
private AccountPanel getAccountPanel() {
  if(accountPanel != null) return AccountPanel;
  accountPanel = new AccountPanel();
  return accountPanel;
}
private void updateAccountPanel() {
  getAccountPanel().setAccount(getAccount());
}
```

The `accountPanel` instance variable is declared of type `Account` and can be an instance of either `SavingsAccount` or `CheckingAccount`. Class `AccountPanel` has to decide how to edit a given account and, at first glance, should provide visual components to edit both types of accounts:

```
private void initComponents() {
  ...layout code not shown...
  add(getBalanceLabel(), constraints);
  add(getBalanceField(), constraints);
  add(getInterestRateLabel(), constraints);
  add(getInterestRateField(), constraints);
  add(getCheckNumberLabel(), constraints);
  add(getCheckNumberField(), constraints);
  add(getCheckCounterLabel(), constraints);
  add(getCheckCounterField(), constraints);
}
```

The balance label and field is used by both accounts. The interest rate label and field components are for savings accounts, and the check number and counter labels and fields for checking accounts. You have to implement code to interact with the components in such a way that account information gets displayed in the appropriate fields. For example:

```
private void updateInterestRateField() {
  if(getAccount().isSavingsAccount()) {
    SavingsAccount savingsAccount = (SavingsAccount)getAccount();
```

```
      getInterestRateField().setText(savingsAccount.getInterestRate());
    } else {
      getInterestRateField().setText(null);
    }
  }
  private void changedInterestRate() {
    if(isUpdating()) return;
    SavingsAccount savingsAccount = (SavingsAccount)getAccount();
    if(getInterestRateField().getText().length() == 0) {
      savginsAccount.setInterestRate(0.0f);
    } else {
      Float interestRate = new Float(getInterestRateField().getText());
      savingsAccount.setInterestRate(interestRate.floatValue());
    }
    update();
  }
```

Similar code has to be implemented for the checking account. As you may have noted, for every method that updates a component, you have to check the type of the account with `isSavingsAccount()` and `isCheckingAccount()` to decide whether the account has the right information. This is cumbersome to implement for every widget and it is not the only shortcoming of this method. Another disadvantage is that `AccountPanel` has to manage all the visual components for all the types of accounts, which will bloat the code as more account types are created. One way to improve the situation is to separate the components into different groups and move the code that checks for the right type of the account away from the update methods to some other object. Therefore:

Group the components that operate on every key variation of some model aspect into subforms. Select the appropriate subform at runtime using the *Subform Selection* pattern.

You implement one subform to edit savings accounts and one subform to edit checcing accounts. Both are subclasses of `JPanel`, and are implemented as outlined in the *Subform* pattern. So, rather than managing the visual components necessary to edit both accounts, `AccountDialog` manages the two forms. The method `getFrame()` lazily creates the dialog as follows:

```
  public JDialog getFrame() {
    if(frame != null) return frame;
    frame = new JDialog(parent, true);
    frame.setTitle("Edit account");
    ...layout code not shown...
    frame.getContentPane().add(getPersonPanel(), constraints);
    frame.getContentPane().add(getSavingsAccountPanel(), constraints);
    frame.getContentPane().add(getCheckingAccountPanel(), constraints);
    update();
    return frame;
  }
  private SavingsAccountPanel getSavingsAccountPanel() {
    if(savingsAccountPanel != null) return savingsAccountPanel;
    savingsAccountPanel = new SavingsAccountPanel();
    return savingsAccountPanel;
```

```
}
private CheckingAccountPanel getCheckingAccountPanel() {
  if(checkingAccountPanel != null) return checkingAccountPanel;
  checkingAccountPanel = new CheckingAccountPanel();
  return checkingAccountPanel;
}
```

This simplifies things a lot. Now, instead of having one account subform knowing how to edit all accounts, you have many account subforms specialized to edit only one type of account each. Should you need to create new types of accounts, you simply implement the new account class and the corresponding subform to operate on it. Should an account type become obsolete, you only have to remove the account class and the associated subform.

This pattern has been previously proposed under a different form by Bradac and Fletcher [1].

1.7 Subform Selection

You are implementing a graphical user interface and have used the *Alternative Subforms* pattern to implement subforms to handle variations of some model aspect. You now have to decide how to manage all these subforms such that the user interacts with the model through the appropriate subform. For example, although there are different subforms for different accounts, only one subform should operate on the account at any one time.

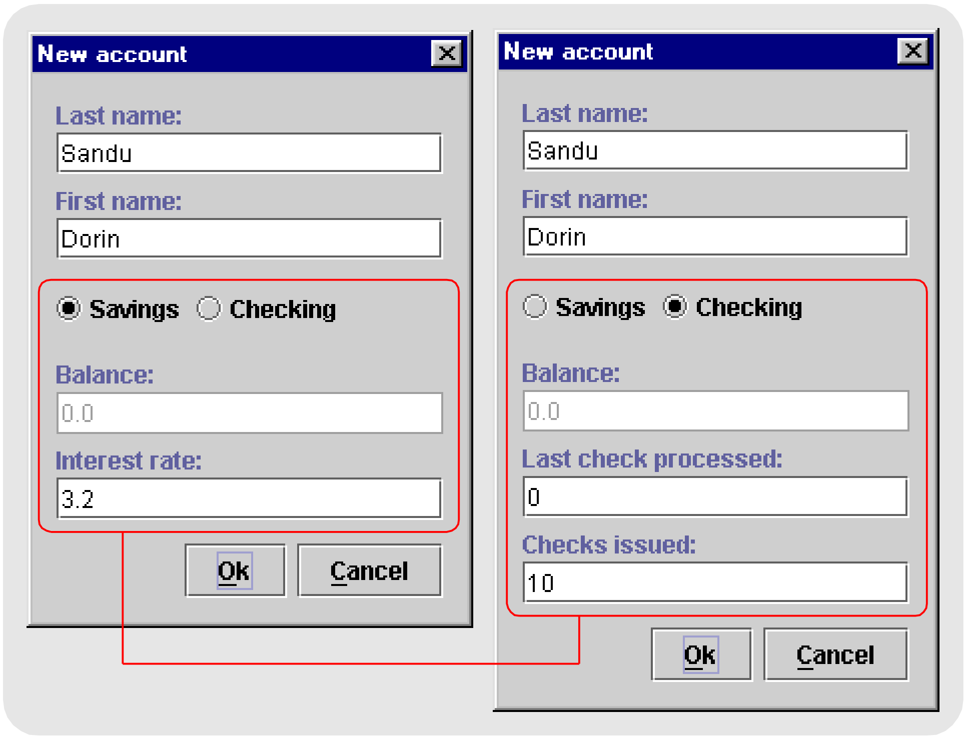
How do you choose from a collection of subforms based on a given key variation of some model aspect?

For the account example, you have to choose between the savings and checking subforms to edit a given account. At this point however, you are more concerned with how the two subforms are managed as opposed to how one is chosen over the other. This is important because you may need a lot of different accounts in the future and the cost of maintaining them should be minimized.

Once you decide on which account subform to use, you can instantiate and add it to the dialog frame. This is alright when the account type does not change at runtime and the user interface is static. Consider the dialog shown in Figure 1.8 where the account model changes from savings to checking type as the bank manager selects the appropriate radio button. To achieve this behavior at runtime, you have to remove the current subform from the dialog frame, instantiate the other one and add it to the frame at exactly the same position as the previous one.

This has the slight disadvantage that you have to create a new subform instance every time the account type changes, which can get expensive if the subform has to, in turn, create a large number of components. The alternative is to, once created, store the form in an instance variable, the way you have been lazily initializing every component so far:

```
private SavingsAccountPanel getSavingsAccountPanel() {
```


New account

Last name: Sandu

First name: Dorin

[✓ Savings] Checking

Balance: 0.0

Interest rate: 3.2

New account

Last name: Sandu

First name: Dorin

Savings [✓ Checking]

Balance: 0.0

Last check processed: 0

Checks issued: 10

Figure 1.8: Dynamic subforms.

```
    if(savingsAccountPanel != null) return savingsAccountPanel;
    savingsAccountPanel = new SavingsAccountPanel();
    return savingsAccountPanel;
  }
  private CheckingAccountPanel getCheckingAccountPanel() {
    if(checkingAccountPanel != null) return checkingAccountPanel;
    checkingAccountPanel = new CheckingAccountPanel();
    return checkingAccountPanel;
  }
```

Can you do better than this? Consider the fact that you may have many types of account subforms in the future, each stored in an instance variable in `AccountDialog`, lazily initialized by the accessor method. The only methods invoking the accessors are the corresponding subform update methods, `updateSavingsAccount()` and `updateCheckingAccount()`:

```
  private void update() {
    ...
    updateSavingsAccount();
    updateCheckingAccount();
    ...
  }
  private void updateSavingsAccount() {
    if(getAccount().isSavingsAccount()) {
      getSavingsAccountPanel().setVisible(true);
      getSavingsAccountPanel().setAccount(getAccount());
    } else {
      getSavingsAccountPanel().setVisible(false);
    }
  }
  private void updateCheckingAccount() {
    if(getAccount().isCheckingAccount()) {
      getCheckingAccountPanel().setVisible(true);
      getCheckingAccountPanel().setAccount(getAccount());
    } else {
      getCheckingAccountPanel().setVisible(false);
    }
  }
```

For every subform, the update method decides whether the account can operate on the current account and, should that be the case, makes the subform visible and tells it to edit the account. If the subform is not suited to edit the account, it is rendered invisible. The global update method calls all the individual subform update methods with the end result that a suitable form is found and presented to the user. An interesting byproduct of this approach is that, if no subform is found to edit the account, the bank manager cannot edit the account, for example, there is no visible subform.

The update methods for all account subforms look the same and one must have great patience to type them in. While some of you may jump at the chance and rush to order that exotic ergonomic keyboard you could not justify until now, you can simplify things a great deal by storing the subforms in a collection. Therefore:

Have the parent keep track of subforms in a collection and hide all except the subform that matches the given key variation of the model aspect. Select the subform using either *Subform Match* or *Subform Mismatch* pattern.

Rather than having an instance variable and the corresponding accessor method for every subform, now you are left with only two variables, one for the collection of subforms, and one for the currently selected subform:

```
public class AccountDialog {
  ...
  private java.util.List accountPanels;
  private AccountPanel selectedAccountPanel;
  private Account account;
  ...
}
```

First thing you have to do is to initialize these variables. Variable `selectedAccountPanel` will be initialized when the dialog is created, during the update procedure. You initialize `accountPanels` to an `ArrayList` in the constructor, and add the `SavingsAccountPanel` and `CheckingAccountPanel` subforms to the collection in the method that returns the frame for the dialog:

```
public JDialog getFrame() {
  if(frame != null) return frame;
  frame = new JDialog(parent, true);
  ...
  addAccountPanel(new SavingsAccountPanel());
  addAccountPanel(new CheckingAccountPanel());
  ...
  return frame;
}
public void addAccountPanel(AccountPanel panel) {
  accountPanels.add(panel);
}
public void removeAccountPanel(AccountPanel panel) {
  accountPanels.remove(panel);
}
```

Then you have to write the update method to hide the currently selected subform, pick the appropriate subform for the given account using the two patterns we will present shortly, make it the currently selected subform, and show it to the user. Since subforms may differ in size and position, it helps to tell the parent to rearrange its components; you do this by invoking `pack()`. The last thing in the method, you have to tell the newly selected subform to edit the account:

```
private void updateAccountsPanel() {
  getSelectedAccountPanel().setVisible(false);
  ...select appropriate subform...
  getSelectedAccountPanel().setVisible(true);
  getFrame().pack();
  getSelectedAccountPanel().setAccount(getAccount());
}
```

As a result, the `AccountDialog` becomes very flexible as now it can handle an infinite number of account types, whose subforms you can register with `addAccountPanel()` and deregister with `removeAccountPanel()`. As you will see in the next two patterns, the dialog is now responsible only to manage a collection of subforms, not to match subforms to given accounts.

This pattern has been previously proposed under a different form by Bradac and Fletcher [1].

1.8 Subform Match

You are implementing a graphical user interface and have used the *Alternative Subforms* pattern to implement subforms to handle variations of some model aspect. You have also used *Subform Selection* to manage these subforms. You now have to decide how to choose the appropriate subform to display to the user when the model aspect changes. The aspect of the model changes at runtime as a result of some user action in a different part of the user interface. For example, when creating a new bank account, a bank manager may choose the type of the account from radio buttons, which should update the user interface with the subform corresponding to the chosen account. The account type radio buttons are not part of any of the account subforms.

How do you determine that a subform can operate on a given key variation of some model aspect based on changes caused by the parent?

The account subforms are selected based on the type of the `account` instance variable in `AccountDialog`. To pick the appropriate subform, you can check the type of the account and, based on that type, choose the appropriate subform. The update method for the account panels is:

```
private void updateAccountsPanel() {
  getSelectedAccountPanel().setVisible(false);
  if(getAccount().isSavingsAccount()) {
    ...select savings account subform...
  } else if(getAccount().isCheckingAccount()) {
    ...select checking account subform...
  } else {
     setSelectedAccountPanel(null);
  }
  getSelectedAccountPanel().setVisible(true);
  getFrame().pack();
  getSelectedAccountPanel().setAccount(getAccount());
}
```

If you choose the account subform this way, you cannot register or deregister subforms at runtime via the `addAccountPanel()` and `removeAccountPanel()` methods. Rather than hardwiring the selection logic into the update method, you can rely on the account to tell you which subform to use to edit it. So, for example, you would have methods in each account to return an instance of the subform that can be used to edit that particular account object:

```
...In class SavingsAccount
public AccountPanel getSubform() {
  return new SavingsAccountPanel();
}
...In class CheckingAccount
public AccountPanel getSubform() {
  return new CheckingAccountPanel();
}
```

Then you can implement the update method for `AccountDialog` like:

```
private void updateAccountsPanel() {
  getFrame().remove(getSelectedAccountPanel());
  setSelectedAccountPanel(getAccount().getSubform());
  ...layout code not shown...
  getFrame().add(getSelectedAccountPanel(), constraints);
  getFrame().pack();
  getSelectedAccountPanel().setAccount(getAccount());
}
```

This way, `AccountDialog` does not have to decide on the subform, it simply asks the account to provide it. Is this a good idea? Not really, since now you have made accounts dependent on the subforms by introducing the `getSubform()` protocol. While it makes sense for the subforms to depend on accounts, the sole purpose for subforms is to edit accounts, you should be able to use accounts independently of the subforms. Even more, this approach fails when you need to implement more than one subform to edit different views of the same account type.

So far, neither the dialog nor the accounts themselves seem appropriate to decide which subform to use on a given account. It seems you can only place this behavior in the third type of object involved, the subforms themselves. Therefore:

Implement a method for each subform that decides whether the subform can operate on the given key variation of some model aspect.

The only approach left is to allow each subform to decide whether it can operate on a given account. To do this you write a method `match()` in each account panel class that accepts an account as argument and returns a boolean indicating whether it can edit the account or not:

```
...In class SavingsAccountPanel
public boolean match(Account account) {
  return account.isSavingsAccount();
}
...In class CheckingAccountPanel
public boolean match(Account account) {
  return account.isCheckingAccount();
}
```

Then you can rewrite the update method for the dialog to traverse the list of subforms, query each one whether it matches the current account object, and select the one that returns true:

```
private void updateAccountsPanel() {
  Iterator iterator = getAccountsPanel().iterator();
  while(iterator.hasNext()) {
    AccountPanel panel = (AccountPanel)iterator.next();
    panel.setVisible(false);
    if(panel.match(getAccount())) {
      panel.setVisible(true);
      setSelectedAccountPanel(panel);
    }
  }
  getFrame().pack();
  getSelectedAccountPanel().setAccount(getAccount());
}
```

If you follow this approach, you distribute subform selection to the subforms themselves. You can add and remove subforms at runtime whithout changing the selection criteria for the existent subforms. The system is more maintainable as well since new subforms can be implemented without having to change any parts of the existing system. The only disadvantage is that, should more than one suborm match, only the first one is selected.

This pattern has been previously proposed under a different form by Bradac and Fletcher [1].

1.9 Subform Mismatch

You are implementing a graphical user interface and have used the *Alternative Subforms* pattern to implement subforms to handle variations of some model aspect. You have also used *Subform Selection* to manage these subforms. You now have to decide how to choose the appropriate subform to display to the user when the model aspect changes. The aspect of the model changes at runtime as a result of some user action in the currently selected subform. For example, when filling in the owner address for the bank account, a bank manager enters the country in the country field first, which should update the user interface with the subform corresponding to the address for that country. The country text field is part of every address subform.

How do you determine that a subform cannot operate on a given key variation of some model aspect based on changes caused by the subform?

The user interacts with a subform and can change its underlying model, thus causing the subform to be unsuitable to operate on that model. For the address example, the address type changes based on the country field text, for example from `USAddress` to `CanadianAddress`:

```
public class Address {
  private String street;
  ...
}
public class USAddress extends Address {
  private String state;
  private String zipCode;
  ...
}
public class CanadianAddress extends Address {
  private String province;
  private String postalCode;
  ...
}
```

With the help of the patterns presented so far, you implement `USAddressPanel` and `CanadianAddressPanel` subforms which are stored in a collection inside `AddressDialog`. Subform `USAddressPanel` uses text fields to edit the country, street, state, zip code and `CanadianAddress` uses text fields to edit the country, street, province, and postal code. The country field can accept and should validate strings like ’US’ and ’Canada’ and, based on this string, the subform should change to be either `USAddressPanel` or `CanadianAddressPanel`.

The subforms themselves cannot change their types and somehow have to tell their parent to choose the appropriate subform. To do this, you can invoke the update method of the parent which, in turn, will cause the appropriate subform to be selected:

```
...In class USAddressPanel
private JTextField getCountryField() {
  if(countryField != null) return countryField;
  countryField = new JTextField();
```

```
      countryField.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          changedCountry();
        }
      });
      return countryField;
    }
    private void countryChanged() {
      if(isUpdating()) return;
      String country = getCountryField.getText();
      if(!isUSAddress(country)) {
        getParent().update();
      }
    }
    private boolean isUSAddress(String country) {
      ...validate country string...
    }}
```

However, with this approach, you directly invoke the update method of the parent. That is, you assume the parent implements an update method, in other words, you assume the parent is implemented using the one of the *Complete Update* or *Multiple Updates* patterns. The best way to decouple the implementation of the subform from the implementation of its parent is to use the event notification mechanism of Java. Therefore:

Have each subform notify its parent when it can no longer operate on the given key variation of data.

The best way to notify the parent of changes is to implement *Observer*. The parent will have to provide an implementation of the `ChangeListener` protocol, which the panels use as follows:

```
public class AddressPanel {
  private ChangeListener listener;
  ...
  public ChangeListener getChangeListener() {
    return listener;
  }
  public void setChangeListener(ChangeListener listener) {
    this.listener = listener;
  }
}
```

Then, you can modify the `countryChanged()` method in `USAddressPanel` to simply invoke the `stateChanged()` method on the listener. The listener method takes as argument an instance of `ChangeEvent`, a wrapper for the source of the event, in this case, the subform:

```
private void countryChanged() {
  if(isUpdating()) return;
  String country = getCountryField.getText();
  if(!isUSAddress(country)) {
    getChangeListener().stateChanged(new ChangeEvent(this));
  }
}
```

The only thing left to do is to implement the parent to respond to changes in the subforms. You do this by subscribing to the change events generated by the subforms. When creating the subforms in `AddressDialog`, you also register the change event listeners:

```
public JDialog getFrame() {
  if(frame != null) return frame;
  frame = new JDialog(parent, true);
  AddressPanel panel;
  ...
  ChangeListener listener = new ChangeListener() {
    public void stateChanged(ChangeEvent event) {
      update();
    }
  }
  panel = new USAddressPanel();
  panel.setChangeListener(listener);
  addAddressPanel(panel);
  panel = new CanadianAddressPanel();
  panel.setChangeListener(listener);
  addAddressPanel(panel);
  ...
  return frame;
}
```

As a result, the subforms are not dependent on the implementation of the parent and yet can still notify them of changes. It is up to the parent controls to register event handlers and, upon receiving change events, to choose the appropriate subform via the update mechanism.

This pattern has been previously proposed under a different form by Bradac and Fletcher [1].

1.10 Patterns in Action

The following classes implement a possible user interface for the bank example presented in an earlier chapter. It consists of class `BankWindow` representing the main window to edit bank objects and `AccountDialog`, a dialog to create or edit bank accounts.

1.10.1 Class BankWindow

The user interface for the bank window is shown in Figure 1.9. For its implementation we used *Event Handler* and *Complete Update*.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;
```

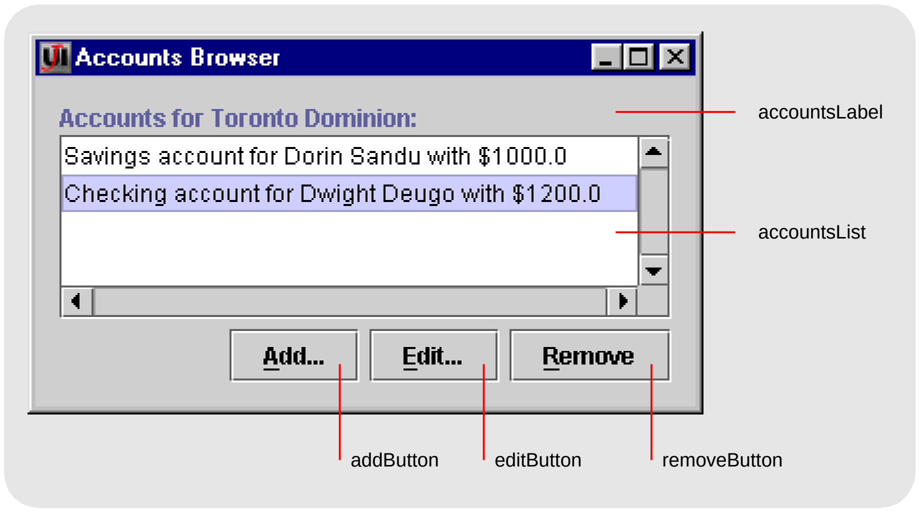

Accounts Browser
Accounts for Toronto Dominion:
Savings account for Dorin Sandu with $1000.0
Checking account for Dwight Deugo with $1200.0
Add...
Edit...
Remove
accountsLabel
accountsList
addButton
editButton
removeButton

Figure 1.9: Window for editing `Bank` objects.

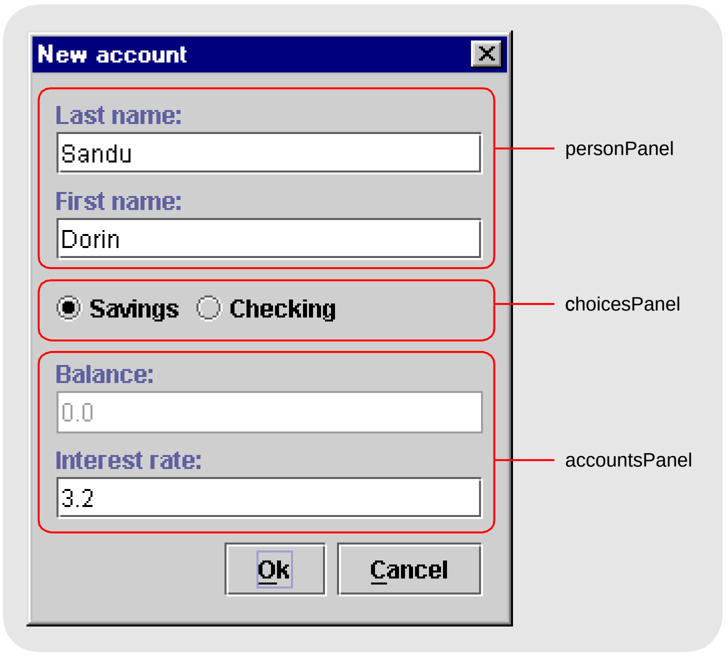
```
public class BankWindow {

  private JFrame frame;
  private JLabel accountsLabel;
  private JScrollPane accountsScrollPane;
  private JList accountsList;
  private JPanel buttonsPanel;
  private JButton addButton, editButton, removeButton;

  private Bank bank;
  private Account selectedAccount;
  private boolean updating;

  private BankWindow() {
    // do nothing
  }

  public BankWindow(Bank bank) {
    this.bank = bank;
  }

  private Bank getBank() {
    return bank;
  }

  private void setBank(Bank bank) {
    this.bank = bank;
  }

  private Account getSelectedAccount() {
    return selectedAccount;
  }
```

```
    private void setSelectedAccount(Account selectedAccount) {
      this.selectedAccount = selectedAccount;
    }

    public JFrame getFrame() {
      if(frame != null) return frame;
      frame = new JFrame();
      frame.setTitle("Accounts Browser");
      frame.setIconImage(getImage("logo16x16.gif"));
      frame.setLocation(100, 100);
      frame.setSize(400, 300);

      frame.getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints constraints;

      constraints = new GridBagConstraints();
      constraints.gridx = 0;
      constraints.gridy = 0;
      constraints.fill = GridBagConstraints.HORIZONTAL;
      constraints.weightx = 1.0;
      constraints.weighty = 0.0;
      constraints.insets = new Insets(12, 12, 0, 12);
      frame.getContentPane().add(getAccountsLabel(), constraints);

      constraints = new GridBagConstraints();
      constraints.gridx = 0;
      constraints.gridy = 1;
      constraints.fill = GridBagConstraints.BOTH;
      constraints.weightx = 1.0;
      constraints.weighty = 1.0;
      constraints.insets = new Insets(0, 12, 0, 12);
      frame.getContentPane().add(getAccountsScrollPane(), constraints);

      constraints = new GridBagConstraints();
      constraints.gridx = 0;
      constraints.gridy = 2;
      constraints.fill = GridBagConstraints.HORIZONTAL;
      constraints.weightx = 1.0;
      constraints.weighty = 0.0;
      constraints.insets = new Insets(5, 12, 12, 12);
      frame.getContentPane().add(getButtonsPanel(), constraints);

      frame.addWindowListener(new WindowListener() {
        public void windowActivated(WindowEvent event) {
          // do nothing
        }
        public void windowClosed(WindowEvent event) {
          // do nothing
        }
        public void windowClosing(WindowEvent event) {
          System.exit(0);
        }
        public void windowDeactivated(WindowEvent event) {
          // do nothing
        }
        public void windowDeiconified(WindowEvent event) {
          // do nothing
        }
        public void windowIconified(WindowEvent event) {
          // do nothing
        }
        public void windowOpened(WindowEvent event) {
          update();
        }
      });
      return frame;
    }

    private JLabel getAccountsLabel() {
      if(accountsLabel != null) return accountsLabel;
      accountsLabel = new JLabel();
      accountsLabel.setText("Accounts for " + getBank().getName() + ":");
      return accountsLabel;
    }

    private JScrollPane getAccountsScrollPane() {
      if(accountsScrollPane != null) return accountsScrollPane;
      accountsScrollPane = new JScrollPane(
        getAccountsList(),
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
      return accountsScrollPane;
    }

    private JList getAccountsList() {
      if(accountsList != null) return accountsList;
      accountsList = new JList();
      accountsList.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent event) {
          changedAccountsList();
        }
      });
      return accountsList;
    }

    private JPanel getButtonsPanel() {
      if(buttonsPanel != null) return buttonsPanel;
      buttonsPanel = new JPanel();

      buttonsPanel.setLayout(new GridBagLayout());
      GridBagConstraints constraints;

      constraints = new GridBagConstraints();
      constraints.anchor = GridBagConstraints.EAST;
      constraints.weightx = 1.0;
      constraints.weighty = 0.0;
      constraints.insets = new Insets(0, 0, 0, 0);
      buttonsPanel.add(getAddButton(), constraints);

      constraints = new GridBagConstraints();
      constraints.anchor = GridBagConstraints.EAST;
      constraints.weightx = 0.0;
      constraints.weighty = 0.0;
      constraints.insets = new Insets(0, 5, 0, 0);
    buttonsPanel.add(getEditButton(), constraints);

    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.weightx = 0.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(0, 5, 0, 0);
    buttonsPanel.add(getRemoveButton(), constraints);

    return buttonsPanel;
  }

  private JButton getAddButton() {
    if(addButton != null) return addButton;
    addButton = new JButton("Add...");
    addButton.setMnemonic('A');
    addButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        addAccount();
      }
    });
    return addButton;
  }

  private JButton getEditButton() {
    if(editButton != null) return editButton;
    editButton = new JButton("Edit...");
    editButton.setMnemonic('E');
    editButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        editAccount();
      }
    });
    return editButton;
  }

  private JButton getRemoveButton() {
    if(removeButton != null) return removeButton;
    removeButton = new JButton("Remove");
    removeButton.setMnemonic('R');
    removeButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        removeAccount();
      }
    });
    return removeButton;
  }

  private void addAccount() {
    Account account = new SavingsAccount();
    AccountDialog dialog = new AccountDialog(getFrame(), account, false);
    Account result = dialog.show();
    if(result != null) {
      bank.openAccount(result);
      setSelectedAccount(result);
      update();
    }
    }

    private void editAccount() {
      Account account = new SavingsAccount();
      try {
        account = (Account)getSelectedAccount().clone();
      } catch (CloneNotSupportedException exception) {
        System.out.println("could not create clone");
        System.exit(1);
      }
      AccountDialog dialog = new AccountDialog(getFrame(), account, true);
      Account result = dialog.show();
      if(result != null) {
        try {
          getSelectedAccount().clone(result);
        } catch (CloneNotSupportedException exception) {
          System.out.println("could not create clone");
          System.exit(1);
        }
        update();
      }
    }

    private void removeAccount() {
      bank.closeAccount(getSelectedAccount());
      setSelectedAccount(null);
      update();
    }

    private void beginUpdating() {
      updating = true;
    }

    private void endUpdating() {
      updating = false;
    }

    private boolean isUpdating() {
      return updating;
    }

    private void update() {
      beginUpdating();
      updateAccountsList();
      updateAddButton();
      updateEditButton();
      updateRemoveButton();
      endUpdating();
    }

    private void updateAccountsList() {
      getAccountsList().setListData(getBank().getAccounts().toArray());
      getAccountsList().setSelectedValue(getSelectedAccount(), true);
    }

    private void updateAddButton() {
      // always enabled
    }

    private void updateEditButton() {
      getEditButton().setEnabled(getSelectedAccount() != null);
    }

    private void updateRemoveButton() {
      getRemoveButton().setEnabled(getSelectedAccount() != null);
    }

    private void changedAccountsList() {
      if(isUpdating()) return;
      setSelectedAccount((Account)getAccountsList().getSelectedValue());
      update();
    }

    public void show() {
      getFrame().show();
    }

    private Image getImage(String theFileName) {
      Toolkit toolkit= Toolkit.getDefaultToolkit();
      try {
        java.net.URL url= getClass().getResource(theFileName);
        return toolkit.createImage((ImageProducer) url.getContent());
      } catch (Exception ex) {
        System.out.println("Failed to load resource " + theFileName);
      }
      return null;
    }

    public static void main(String args[]) {
      Bank bank = new Bank("Toronto Dominion");
      Person person;
      Account account;

      person = new Person("Dorin", "Sandu");
      account = new SavingsAccount(1000.0f, person, 1293, 3.2f);
      bank.openAccount(account);

      person = new Person("Dwight", "Deugo");
      account = new CheckingAccount(1200.0f, person, 1294, "0", 25);
      bank.openAccount(account);

      new BankWindow(bank).show();
    }
  }
```

1.10.2 Class AccountDialog

The user interface for the account dialog is shown in Figure 1.10. For its implementation we used *Event Handler*, *Complete Update*, *Subform*, *Subform Selection*, and `Subform Match`.

```
  import java.awt.*;
  import java.awt.event.*;
```



Last name: Sandu

First name: Dorin

● Savings ○ Checking

Balance: 0.0

Interest rate: 3.2

Figure 1.10: Dialog for editing `Account` objects.

```
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class AccountDialog {

  private JFrame parent;
  private JDialog frame;
  private PersonPanel personPanel;
  private JPanel choicesPanel, accountsPanel, buttonsPanel;
  private JRadioButton savingsButton, checkingButton;
  private JButton okButton, cancelButton;

  private Account account;
  private java.util.List accountPanels;
  private AccountPanel selectedAccountPanel;
  private boolean updating, editing;

  private AccountDialog() {
    // do nothing
  }

  public AccountDialog(JFrame parent, Account account, boolean editing) {
    accountPanels = new ArrayList();
    this.parent = parent;
    this.account = account;
    this.editing = editing;
  }

  private Account getAccount() {
    return account;
  }

  private void setAccount(Account account) {
    this.account = account;
  }

  private JFrame getParent() {
    return parent;
  }

  private void setParent(JFrame parent) {
    this.parent = parent;
  }

  private java.util.List getAccountPanels() {
    return accountPanels;
  }

  private void setAccountPanels(java.util.List accountPanels) {
    this.accountPanels = accountPanels;
  }

  private AccountPanel getSelectedAccountPanel() {
    return selectedAccountPanel;
  }

  private void setSelectedAccountPanel(AccountPanel selectedAccountPanel) {
    this.selectedAccountPanel = selectedAccountPanel;
  }

  public JDialog getFrame() {
    if(frame != null) return frame;
    frame = new JDialog(parent, true);
    if(isEditing()) {
      frame.setTitle("Edit account");
    } else {
      frame.setTitle("New account");
    }

    frame.getContentPane().setLayout(new GridBagLayout());
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.gridx = GridBagConstraints.REMAINDER;
    constraints.gridy = GridBagConstraints.RELATIVE;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 1.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(12, 12, 0, 12);
    constraints.anchor = GridBagConstraints.NORTH;
    constraints.ipadx = 150;
    frame.getContentPane().add(getPersonPanel(), constraints);
    if(!isEditing()) {
      constraints = new GridBagConstraints();
      constraints.gridx = GridBagConstraints.REMAINDER;
      constraints.gridy = GridBagConstraints.RELATIVE;
      constraints.fill = GridBagConstraints.HORIZONTAL;
      constraints.weightx = 0.0;
      constraints.weighty = 0.0;
      constraints.insets = new Insets(12, 12, 0, 12);
      constraints.anchor = GridBagConstraints.NORTH;
      frame.getContentPane().add(getChoicesPanel(), constraints);
    }

    constraints = new GridBagConstraints();
    constraints.gridx = GridBagConstraints.REMAINDER;
    constraints.gridy = GridBagConstraints.RELATIVE;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 0.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(12, 12, 0, 12);
    constraints.anchor = GridBagConstraints.NORTH;
    frame.getContentPane().add(getAccountsPanel(), constraints);

    constraints = new GridBagConstraints();
    constraints.gridx = GridBagConstraints.REMAINDER;
    constraints.gridy = GridBagConstraints.RELATIVE;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 0.0;
    constraints.weighty = 1.0;
    constraints.insets = new Insets(12, 12, 12, 12);
    constraints.anchor = GridBagConstraints.SOUTH;
    frame.getContentPane().add(getButtonsPanel(), constraints);

    frame.setResizable(false);
    update();
    return frame;
  }

  private PersonPanel getPersonPanel() {
    if(personPanel != null) return personPanel;
    personPanel = new PersonPanel();
    return personPanel;
  }

  private JPanel getChoicesPanel() {
    if(choicesPanel != null) return choicesPanel;
    choicesPanel = new JPanel();

    choicesPanel.setLayout(new GridBagLayout());
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.weightx = 0.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(0, 0, 0, 0);
    choicesPanel.add(getSavingsButton(), constraints);
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.weightx = 1.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(0, 0, 0, 0);
    choicesPanel.add(getCheckingButton(), constraints);

    return choicesPanel;
  }

  private JRadioButton getSavingsButton() {
    if(savingsButton != null) return savingsButton;
    savingsButton = new JRadioButton();
    savingsButton.setText("Savings");
    savingsButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        createSavingsAccount();
      }
    });
    return savingsButton;
  }

  private JRadioButton getCheckingButton() {
    if(checkingButton != null) return checkingButton;
    checkingButton = new JRadioButton();
    checkingButton.setText("Checking");
    checkingButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        createCheckingAccount();
      }
    });
    return checkingButton;
  }

  private JPanel getAccountsPanel() {
    if(accountsPanel != null) return accountsPanel;
    accountsPanel = new JPanel();

    accountsPanel.setLayout(new GridBagLayout());
    GridBagConstraints constraints;

    AccountPanel panel;

    panel = new SavingsAccountPanel();
    addAccountPanel(panel);
    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.weightx = 1.0;
    constraints.weighty = 1.0;
    constraints.anchor = GridBagConstraints.NORTH;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    accountsPanel.add(panel, constraints);

    panel = new CheckingAccountPanel();
    addAccountPanel(panel);
    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.weightx = 1.0;
    constraints.weighty = 1.0;
    constraints.anchor = GridBagConstraints.NORTH;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    accountsPanel.add(panel, constraints);

    return accountsPanel;
  }

  private JPanel getButtonsPanel() {
    if(buttonsPanel != null) return buttonsPanel;
    buttonsPanel = new JPanel();

    buttonsPanel.setLayout(new GridBagLayout());
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.weightx = 1.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(0, 0, 0, 0);
    buttonsPanel.add(getOkButton(), constraints);

    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.weightx = 0.0;
    constraints.weighty = 0.0;
    constraints.insets = new Insets(0, 5, 0, 0);
    buttonsPanel.add(getCancelButton(), constraints);

    return buttonsPanel;
  }

  private JButton getOkButton() {
    if(okButton != null) return okButton;
    okButton = new JButton("Ok");
    okButton.setMnemonic('O');
    okButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        getFrame().dispose();
      }
    });
    return okButton;
  }

  private JButton getCancelButton() {
    if(cancelButton != null) return cancelButton;
    cancelButton = new JButton("Cancel");
    cancelButton.setMnemonic('C');
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        account = null;
        getFrame().dispose();
      }
    });
    return cancelButton;
  }

  private void beginUpdating() {
    updating = true;
  }

  private void endUpdating() {
    updating = false;
  }

  private boolean isUpdating() {
    return updating;
  }

  private boolean isEditing() {
    return editing;
  }

  private void update() {
    beginUpdating();
    updatePersonPanel();
     if(!isEditing()) {
      updateSavingsButton();
      updateCheckingButton();
    }
    updateAccountsPanel();
    endUpdating();
  }

  private void updatePersonPanel() {
    getPersonPanel().setPerson(getAccount().getOwner());
  }

  private void updateSavingsButton() {
    getSavingsButton().setSelected(getAccount().isSavingsAccount());
  }

  private void updateCheckingButton() {
    getCheckingButton().setSelected(getAccount().isCheckingAccount());
  }

  private void updateAccountsPanel() {
    Iterator iterator = getAccountPanels().iterator();
    while(iterator.hasNext()) {
      AccountPanel panel = (AccountPanel)iterator.next();
      panel.setVisible(false);
      if(panel.match(getAccount())) {
        panel.setVisible(true);
        setSelectedAccountPanel(panel);
      }
    }
    getFrame().pack();
    getSelectedAccountPanel().setAccount(getAccount());
  }

  private void createSavingsAccount() {
        SavingsAccount savingsAccount = new SavingsAccount();
        if(getAccount() != null) {
          savingsAccount.setOwner(getAccount().getOwner());
        }
        setAccount(savingsAccount);
        update();
      }

      private void createCheckingAccount() {
        CheckingAccount checkingAccount = new CheckingAccount();
        if(getAccount() != null) {
          checkingAccount.setOwner(getAccount().getOwner());
        }
        setAccount(checkingAccount);
        update();
      }

      public void addAccountPanel(AccountPanel panel) {
        accountPanels.add(panel);
      }

      public void removeAccountPanel(AccountPanel panel) {
        accountPanels.remove(panel);
      }

      public Account show() {
        getFrame().show();
        return getAccount();
      }
    }
```

1.10.3 Class PersonPanel

The person panel is shown in Figure 1.11. For its implementation we used *Event Handler*, *Complete Update*, and *Subform*.

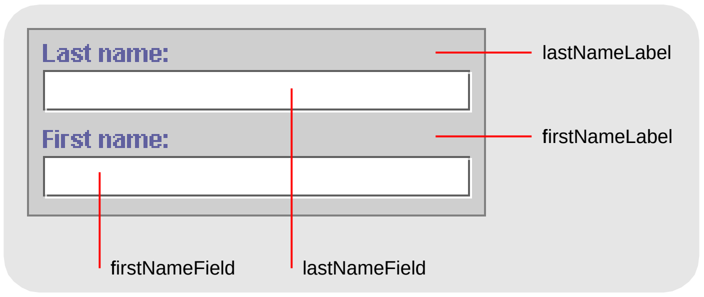


Figure 1.11: Panel for editing `Person` objects.

```
    import java.awt.*;
    import java.awt.event.*;
    import javax.swing.*;
    import javax.swing.event.*;
```

```
import java.util.*;

public class PersonPanel extends JPanel {

  private JLabel lastNameLabel, firstNameLabel;
  private JTextField lastNameField, firstNameField;

  private Person person;
  private boolean updating = false;

  public PersonPanel() {
    this(new Person());
  }

  public PersonPanel(Person person) {
    this.person = person;
    initComponents();
  }

  public Person getPerson() {
    return person;
  }

  public void setPerson(Person person) {
    this.person = person;
    update();
  }

  private void initComponents() {
    GridBagLayout layout = new GridBagLayout();
    setLayout(layout);
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 1.0;
    constraints.insets = new Insets(0, 0, 0, 0);
    add(getLastNameLabel(), constraints);

    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 1;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 1.0;
    constraints.insets = new Insets(0, 0, 0, 0);
    add(getLastNameField(), constraints);

    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 2;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.weightx = 1.0;
    constraints.insets = new Insets(5, 0, 0, 0);
    add(getFirstNameLabel(), constraints);
```

```
        constraints = new GridBagConstraints();
        constraints.gridx = 0;
        constraints.gridy = 3;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.weightx = 1.0;
        constraints.insets = new Insets(0, 0, 0, 0);
        add(getFirstNameField(), constraints);

        update();
      }

      private JLabel getLastNameLabel() {
        if(lastNameLabel != null) return lastNameLabel;
        lastNameLabel = new JLabel();
        lastNameLabel.setHorizontalAlignment(SwingConstants.LEFT);
        lastNameLabel.setText("Last name:");
        return lastNameLabel;
      }

      private JTextField getLastNameField() {
        if(lastNameField != null) return lastNameField;
        lastNameField = new JTextField();
        lastNameField.getDocument().addDocumentListener(new DocumentListener() {
          public void changedUpdate(DocumentEvent event) {
            changedLastNameField();
          }
          public void insertUpdate(DocumentEvent event) {
            changedLastNameField();
          }
          public void removeUpdate(DocumentEvent event) {
            changedLastNameField();
          }
        });
        return lastNameField;
      }

      private JLabel getFirstNameLabel() {
        if(firstNameLabel != null) return firstNameLabel;
        firstNameLabel = new JLabel();
        firstNameLabel.setHorizontalAlignment(SwingConstants.LEFT);
        firstNameLabel.setText("First name:");
        return firstNameLabel;
      }

      private JTextField getFirstNameField() {
        if(firstNameField != null) return firstNameField;
        firstNameField = new JTextField();
        firstNameField.getDocument().addDocumentListener(new DocumentLis-
tener() {
          public void changedUpdate(DocumentEvent event) {
            changedFirstNameField();
          }
          public void insertUpdate(DocumentEvent event) {
            changedFirstNameField();
          }
          public void removeUpdate(DocumentEvent event) {
            changedFirstNameField();
        }
      });
      return firstNameField;
    }

    private void beginUpdating() {
      updating = true;
    }

    private void endUpdating() {
      updating = false;
    }

    private boolean isUpdating() {
      return updating;
    }

    public void update() {
      beginUpdating();
      updateLastNameLabel();
      updateLastNameField();
      updateFirstNameLabel();
      updateFirstNameField();
      endUpdating();
    }

    private void updateLastNameLabel() {
      // do nothing
    }

    private void updateLastNameField() {
      getLastNameField().setText(getPerson().getLastName());
    }

    private void updateFirstNameLabel() {
      // do nothing
    }

    private void updateFirstNameField() {
      getFirstNameField().setText(getPerson().getFirstName());
    }

    private void changedLastNameField() {
      if(isUpdating()) return;
      if(getLastNameField().getText().length() == 0) {
        getPerson().setLastName(null);
      } else {
        getPerson().setLastName(getLastNameField().getText());
      }
      //update();
    }

    private void changedFirstNameField() {
      if(isUpdating()) return;
      if(getFirstNameField().getText().length() == 0) {
        getPerson().setFirstName(null);
      } else {
```

```
      getPerson().setFirstName(getFirstNameField().getText());
    }
    //update();
  }
}
```

1.10.4 Class AccountPanel

For the implementation of `AccountPanel`, we used *Event Handler*, *Complete Update*, and *Subform*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class AccountPanel extends JPanel {

  private JLabel balanceLabel;
  private JTextField balanceField;

  private Account account;
  private boolean updating;

  private AccountPanel() {
    // do nothing
  }

  public AccountPanel(Account account) {
    this.account = account;
    initComponents();
  }

  public Account getAccount() {
    return account;
  }

  public void setAccount(Account account) {
    this.account = account;
    update();
  }

  public boolean match(Account account) {
    return true;
  }

  protected void initComponents() {
    GridBagLayout layout = new GridBagLayout();
    setLayout(layout);
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.fill = GridBagConstraints.HORIZONTAL;
```

```
      constraints.weightx = 1.0;
      constraints.insets = new Insets(0, 0, 0, 0);
      add(getBalanceLabel(), constraints);

      constraints = new GridBagConstraints();
      constraints.gridx = 0;
      constraints.gridy = 1;
      constraints.fill = GridBagConstraints.HORIZONTAL;
      constraints.weightx = 1.0;
      constraints.insets = new Insets(0, 0, 0, 0);
      add(getBalanceField(), constraints);
    }

    private JLabel getBalanceLabel() {
      if(balanceLabel != null) return balanceLabel;
      balanceLabel = new JLabel();
      balanceLabel.setHorizontalAlignment(SwingConstants.LEFT);
      balanceLabel.setText("Balance:");
      return balanceLabel;
    }

    private JTextField getBalanceField() {
      if(balanceField != null) return balanceField;
      balanceField = new JTextField();
      balanceField.setEnabled(false);
      return balanceField;
    }

    protected void beginUpdating() {
      updating = true;
    }

    protected void endUpdating() {
      updating = false;
    }

    protected boolean isUpdating() {
      return updating;
    }

    protected void update() {
      beginUpdating();
      updateBalanceLabel();
      updateBalanceField();
      endUpdating();
    }

    private void updateBalanceLabel() {
      // do nothing
    }

    private void updateBalanceField() {
      getBalanceField().setText(String.valueOf(getAccount().getBalance()));
    }
  }
```

1.10.5 Class SavingsAccountPanel

The user interface for the savings account panel is shown in Figure 1.12. For its implementation we used *Event Handler*, *Complete Update*, *Subform*.

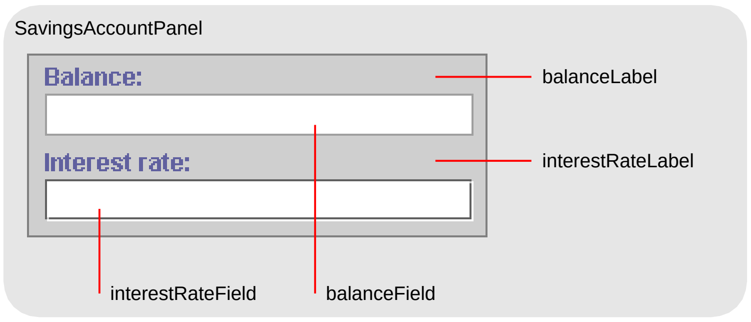


Figure 1.12: Panel for editing `SavingsAccount` objects.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class SavingsAccountPanel extends AccountPanel {

  private JLabel interestRateLabel;
  private JTextField interestRateField;

  public SavingsAccountPanel() {
    this(new SavingsAccount());
  }

  public SavingsAccountPanel(SavingsAccount account) {
    super(account);
  }

  public SavingsAccount getSavingsAccount() {
    return (SavingsAccount)super.getAccount();
  }

  public boolean match(Account account) {
    return account.isSavingsAccount();
  }

  protected void initComponents() {
    super.initComponents();
    GridBagConstraints constraints;

    constraints = new GridBagConstraints();
    constraints.gridx = 0;
    constraints.gridy = 2;
    constraints.fill = GridBagConstraints.HORIZONTAL;
```

```
        constraints.weightx = 1.0;
        constraints.insets = new Insets(5, 0, 0, 0);
        add(getInterestRateLabel(), constraints);

        constraints = new GridBagConstraints();
        constraints.gridx = 0;
        constraints.gridy = 3;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.weightx = 1.0;
        constraints.insets = new Insets(0, 0, 0, 0);
        add(getInterestRateField(), constraints);

        update();
      }

      private JLabel getInterestRateLabel() {
        if(interestRateLabel != null) return interestRateLabel;
        interestRateLabel = new JLabel();
        interestRateLabel.setHorizontalAlignment(SwingConstants.LEFT);
        interestRateLabel.setText("Interest rate:");
        return interestRateLabel;
      }

      private JTextField getInterestRateField() {
        if(interestRateField != null) return interestRateField;
        interestRateField = new JTextField();
        interestRateField.getDocument().addDocumentListener(new DocumentLis-
tener() {
          public void changedUpdate(DocumentEvent event) {
            changedInterestRateField();
          }
          public void insertUpdate(DocumentEvent event) {
            changedInterestRateField();
          }
          public void removeUpdate(DocumentEvent event) {
            changedInterestRateField();
          }
        });
        return interestRateField;
      }

      protected void update() {
        super.update();
        beginUpdating();
        updateInterestRateLabel();
        updateInterestRateField();
        endUpdating();
      }

      private void updateInterestRateLabel() {
        // do nothing
      }

      private void updateInterestRateField() {
        getInterestRateField().setText(String.valueOf(getSavingsAccount().getInterestRate()));
      }
```

```
    private void changedInterestRateField() {
      if(isUpdating()) return;
      if(getInterestRateField().getText().length() == 0) {
        getSavingsAccount().setInterestRate(0.0f);
      } else {
        Float interestRate = new Float(getInterestRateField().getText());
        getSavingsAccount().setInterestRate(interestRate.floatValue());
      }
      //update();
    }
  }
```

1.10.6 Class CheckingAccountPanel

The user interface for the checking account panel is shown in Figure 1.12. For its implementation we used *Event Handler*, *Complete Update*, *Subform*.

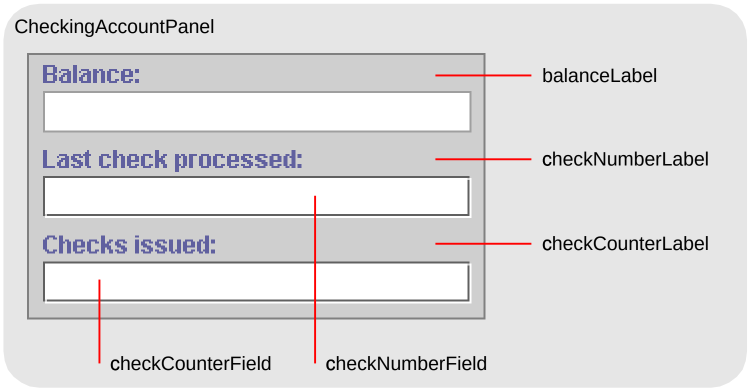


Figure 1.13: Panel for editing `CheckingAccount` objects.

```
  import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;
  import javax.swing.event.*;
  import java.util.*;

  public class CheckingAccountPanel extends AccountPanel {

    private JLabel checkNumberLabel, checkCounterLabel;
    private JTextField checkNumberField, checkCounterField;

    public CheckingAccountPanel() {
      this(new CheckingAccount());
    }

    public CheckingAccountPanel(CheckingAccount account) {
      super(account);
    }
```

```
public CheckingAccount getCheckingAccount() {
  return (CheckingAccount)super.getAccount();
}

public boolean match(Account account) {
  return account.isCheckingAccount();
}

protected void initComponents() {
  super.initComponents();
  GridBagConstraints constraints;

  constraints = new GridBagConstraints();
  constraints.gridx = 0;
  constraints.gridy = 2;
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.weightx = 1.0;
  constraints.insets = new Insets(5, 0, 0, 0);
  add(getCheckNumberLabel(), constraints);

  constraints = new GridBagConstraints();
  constraints.gridx = 0;
  constraints.gridy = 3;
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.weightx = 1.0;
  constraints.insets = new Insets(0, 0, 0, 0);
  add(getCheckNumberField(), constraints);

  constraints = new GridBagConstraints();
  constraints.gridx = 0;
  constraints.gridy = 4;
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.weightx = 1.0;
  constraints.insets = new Insets(5, 0, 0, 0);
  add(getCheckCounterLabel(), constraints);

  constraints = new GridBagConstraints();
  constraints.gridx = 0;
  constraints.gridy = 5;
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.weightx = 1.0;
  constraints.insets = new Insets(0, 0, 0, 0);
  add(getCheckCounterField(), constraints);

  update();
}

private JLabel getCheckNumberLabel() {
  if(checkNumberLabel != null) return checkNumberLabel;
  checkNumberLabel = new JLabel();
  checkNumberLabel.setHorizontalAlignment(SwingConstants.LEFT);
  checkNumberLabel.setText("Last check processed:");
  return checkNumberLabel;
}

private JTextField getCheckNumberField() {
  if(checkNumberField != null) return checkNumberField;
```

```
       checkNumberField = new JTextField();
       checkNumberField.getDocument().addDocumentListener(new DocumentLis-
tener() {
         public void changedUpdate(DocumentEvent event) {
           changedCheckNumberField();
         }
         public void insertUpdate(DocumentEvent event) {
           changedCheckNumberField();
         }
         public void removeUpdate(DocumentEvent event) {
           changedCheckNumberField();
         }
       });
       return checkNumberField;
     }

     private JLabel getCheckCounterLabel() {
       if(checkCounterLabel != null) return checkCounterLabel;
       checkCounterLabel = new JLabel();
       checkCounterLabel.setHorizontalAlignment(SwingConstants.LEFT);
       checkCounterLabel.setText("Checks issued:");
       return checkCounterLabel;
     }

     private JTextField getCheckCounterField() {
       if(checkCounterField != null) return checkCounterField;
       checkCounterField = new JTextField();
       checkCounterField.getDocument().addDocumentListener(new DocumentLis-
tener() {
         public void changedUpdate(DocumentEvent event) {
           changedCheckCounterField();
         }
         public void insertUpdate(DocumentEvent event) {
           changedCheckCounterField();
         }
         public void removeUpdate(DocumentEvent event) {
           changedCheckCounterField();
         }
       });
       return checkCounterField;
     }

     protected void update() {
       super.update();
       beginUpdating();
       updateCheckNumberLabel();
       updateCheckNumberField();
       updateCheckCounterLabel();
       updateCheckCounterField();
       endUpdating();
     }

     private void updateCheckNumberLabel() {
       // do nothing
     }

     private void updateCheckNumberField() {
      getCheckNumberField().setText(getCheckingAccount().getCheckNumber());
    }

    private void updateCheckCounterLabel() {
      // do nothing
    }

    private void updateCheckCounterField() {
      getCheckCounterField().setText(String.valueOf(getCheckingAccount().getCheckCounter()));
    }

    private void changedCheckNumberField() {
      if(isUpdating()) return;
      if(getCheckNumberField().getText().length() == 0) {
        getCheckingAccount().setCheckNumber(null);
      } else {
        getCheckingAccount().setCheckNumber(getCheckNumberField().getText());
      }
      //update();
    }

    private void changedCheckCounterField() {
      if(isUpdating()) return;
      if(getCheckCounterField().getText().length() == 0) {
        getCheckingAccount().setCheckCounter(0);
      } else {
        Integer counter = new Integer(getCheckCounterField().getText());
        getCheckingAccount().setCheckCounter(counter.intValue());
      }
      //update();
    }
  }
```

1.11 Discussion

You can apply *Event Handler*, *Complete Update*, and *Multiple Update* in two phases. The first phase changes the states of the user interface models in response to end-user events generated by the visual components, and the second phase updates the visual components to reflect the changes in the user interface model. Since the update phase immediately follows the handle phase, the user interface always reflects the latest changes.

Our patterns allow the user interface to be modified easily. In order to add a new component, it is necessary to write an update method for it, and as many handler methods as required. In a similar fashion, to remove a component, it is necessary to remove the corresponding update method and the all of the handler methods. Therefore, every component is defined by one update and many handler methods, which is consistent for all the components. Since the update and handler methods access only specific visual components, these components can be added or removed without having to modify the update and handler methods for the other components.

Our patterns form expectations in developers' minds. Any user interface contains handler and update methods and an easily identifiable interaction between the visual components and the models. If there is a problem in the handling or updating in response to a user event, the developer knows exactly what methods to examine.

If you decide to use these three patterns consistently, you can summarize them with the following step-by-step instructions:

- For every user interface, determine its state variables and associated objects.
- Write a handler method for each event a user can generate from interacting with the visual components. Allow a handler method to modify only the model objects and retrieve information from the immediate visual components but prohibit it from directly modify any other visual components. Force the last statement in each handler method to invoke the main update method for the entire user interface.
- Write a specialized update method for every visual component. Each method is responsible for updating the corresponding visual component from the model information. The method is permitted to directly load state information into the visual components or can compute the contents of the corresponding visual component on the fly.
- Write a single, main update method that invokes all the specialized update methods.

You can apply *Subform* to encapsulate parts of user interfaces in order to reuse them in other parts of the system, and to increase the readability of the code. A simple rule of thumb that has worked for us in many projects is to implement a subform to edit a given object. This subform may contain visual components to edit atomic aspects of the object such as booleans, strings, and collections and other subforms to edit complex aspects of the object.

You can use *Alternative Subforms*, *Subform Selection*, *Subform Match*, and *Subform Mismatch* to deal with highly dynamic user interfaces. To apply them use the following steps:

- Develop a subform for every specialization of the object that needs editing. Usually, the class of the object in question is part of a hierachy, for example, a specialization of some class. Implement the subforms in a similar hierarchy, rooted under `JPanel` in such a way that their construction invokes the construction of the ancestor panels in the hierarchy.
- Make the root subform in the hierarchy keep track of a `ChangeListener` and have each subform notify this listener every time it makes a change to the model object.
- Decide on how and when each subform can edit a given object. We recommend you do this in ways other than directly probing for the type of the object via `instanceof` or `getClass()`.

- For every subform, implement a `match()` public method that takes a single object as argument. The object should be the type of the root of the hierarchy of subforms, most often an abstract class. The return value has to be a boolean indicating whether the current subform can edit the given object or not, based on the tests developed in previous step.
- Store the subforms in a collection inside the main user interface. Implement appropriate methods to add, remove, query, and traverse this collection as required.
- Modify the update method in the main user interface to detect, using the `match()` method developed previously, which subform in the collection can edit a given object, make that subform visible, and hide the rest of them.

Bibliography

[1] M. Bradac and Fletcher. A Pattern Language for Developing Form Style Windows. *Pattern Languages of Program Design 3*, 1998.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[4] Mike Potel. MVP: Model-View-Presenter – The Taligent Programming Model for C++ and Java. 1996.