A Pattern Language for Developing Graphical User Interfaces

Dwight Deugo and Dorin Sandu

School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada, K1S 5B6

Email: deugo@scs.carleton.ca, sandu@turing.scs.carleton.ca
Telephone: (613) 520-4333
Fax: (613) 520-4334

Abstract

A graphical user interface (GUI) is composed of three entities: a container object such as a window or form, other visual components such as buttons and selection lists, and objects modeling the domain. One goal for GUI development is that application windows are easy to build, modify and maintain. An obvious way to meet this goal is to have all developers use the same approach. Patterns and pattern languages are excellent vehicles for expressing common design and architectural approaches. The Observer pattern, the Model-View-Controller pattern and the pattern language for developing form style windows provide a partial set of patterns for building GUIs. After building GUIs for years, we found that more details were required for properly handling the event notification from the visual components and for updating them. We have incorporated these decisions into new patterns without changing the above-mentioned patterns, forming an ever-growing pattern language for GUI development.

1 Introduction

A graphical user interface (GUI) is composed of three entities: a container object such as a window or form, other visual components such as buttons and selection lists, and objects modeling the domain. The function of the container object is to act as the master of the visual components, coordinating their assembly and the activity between the user, the visual components and the models. Some visual components, such as a button, enable an application's user to initiate actions on the models in addition to displaying information about them. Except for a window, containers usually coordinate the display and actions of a single model rather than those of an entire application. In this paper, we collective call all visual components, including containers, views. Therefore, it is possible for views to be recursively defined.

An important goal to keep in mind when constructing a view is that it is easy to build, modify and maintain. Although many developers would claim their views meet this goal, and in addition are works of art, beauty is in the eye of the beholder. If a developer has a unique view implementation style, only he understands the result. It may be easy for him to build, modify and maintain his view. However, we require an approach enabling anyone, not just the original developer, to work on a view without spending weeks understanding the previous implementation.

An obvious way to meet this goal is to have all developers use the same approach. Patterns and pattern languages are excellent vehicles for expressing common approaches to designs and architectures. The Model-View-Controller pattern [3], the Observer Pattern [4] and the pattern language for developing form style windows [2] provide a partial set of patterns for building views.

Braduc and Fletecher's pattern language addresses how to use forms to manage the construction of views and their reuse while minimizing the number of discrete views required. The language consisted of five patterns: Subform, Alternative Subforms, Subform Selection, Subform Match and Subform Mismatch. The patterns present a systematic approach to the construction and decomposition of an application's views by factoring visual components into a number of reusable components called subforms and then using them to construct multiple application windows.

The Model-View-Controller (MVC) pattern divides an application into three parts: a model, a view (Window, Form, or some other visual container) and a controller. The model contains the information and functionality required by the application. The view displays the information in the model to the user, and the controller handles the user input. This architectural pattern describes the relationships between the model, view and controller and provides a change-propagation mechanism to maintain the consistency between the models and the views.

The Observer pattern, used by the MVC pattern, manages the notification and updating between dependent objects called observers and subjects. The pattern describes how to notify observer objects when a subject changes state, without forcing the subject to know the specific classes of the observers.

After building views for years, we found there were design decisions not dealt with in the previously mentioned patterns and pattern language that were required to make our views maintainable. This situation is common. As our understand of each pattern grows, so too do the ways in which we can extend and refine them, enabling us to provide more concrete and comprehensive advice. In particular, we found that more details were required to handle the event notification from the visual components and for updating the views. We incorporated these decisions into new patterns without changing the above-mentioned patterns, forming an ever-growing pattern language for GUI development.

1.1 Contributions of this Paper

This paper presents a pattern language for building views. Specific contributions include the following:

- ***We describe three new patterns for handling event notification in views and their subsequent updating***.

- ***We link the MVC pattern and the patterns in the pattern language for developing form style windows to our new patterns, forming a more comprehensive pattern language for building views.*** The new patterns provide concrete advice for building views not covered in the original patterns.

- ***We include sample implementations with our patterns, providing developers with concrete examples of how to apply them to their views.***

The pattern language proposed here attempts to resolve the following forces:

- ***A view's functionality changes often.*** As an application evolves from early prototypes to a shipped product, practical experience suggests that its overall views 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.
- ***A significant period may elapse between the time a developer last worked on a view 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. Under no circumstances should a developer consider changes in a view as a task of recreating it from scratch.
- ***The original developer of a* view *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.
- ***Tightly coupled* view *components are difficult to merge or split.*** It should be possible to merge or split views 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 is significantly reduced if parts or entire views can be merged or split, i.e., one should be able to add and remove visual components with minimum disturbance to the existing ones or user interface.
- ***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 views. The best place to save development time is to streamline the part of development that takes the most time.
- ***Unique view designs are more difficult to support than consistent ones.*** Views 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.
- ***Since a view is a user's entry point into an application, it must be user-friendly.*** An application's view that is either complicated to use or slow to react will frustrate a user. Users expect intuitive views that operate in real-time.

1.2 Pattern Summary

Table 1 summarizes the problems addressed by the patterns in our pattern language and their solutions. In the following sections, we describe the new patterns we have added to the pattern language. None of them stands on their own. Rather they form dependencies with others in the language, as shown in figure 1, with single-headed arrows pointing to the preceding patterns and double-headed arrows indicating two patterns in tension with each other.

The MVC pattern represents the core of the pattern language. It provides the overall architecture for connecting models, views and controllers. The Subform patterns help describe how to structure the views. Our new patterns help describe how to handle and process user generated events in the MVC pattern and in the Subforms patterns. Sections 2's Handle Visual Component Events pattern describes how a view should handle a user event such that its update process is independent from the view generating the event and from the models that have changed state. Section 3's Single View Update Pattern describes how to structure a view's update code such that you or any future developer can quickly and easily understand the logic of the update. Section 4's Multiple View Update pattern describes how to manage the update of visual components arranged in multiple subforms. In section 5, we summarize.

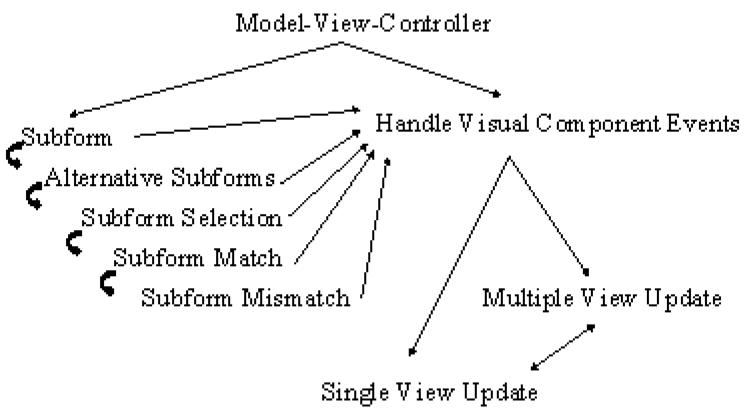


Figure 1. Pattern Language Dependencies

Pattern Name	**Problem**	**Solution**
Model-View-Controller *<existing>*	How do you architect an interactive application?	Divide interactive application into three parts: model, view and controller.
Subform *<existing>*	How do you minimize the development and maintenance efforts when designing form style windows?	Divide a form into subforms.
Alternative Subform *<existing>*	How do you design a window where different sets of widgets are needed based on application state data?	Create a subform for each variation of widget sets.
Subform Selection *<existing>*	How do you choose from a collection of subforms that become active/inactive based on application state data?	Have the parent form maintain and coordinate the subforms.
Subform Match *<existing>*	How does a parent subform select the appropriate subform based on application state data?	Have the parent form maintain and poll the subforms for a match.
Suform Mismatch *<existing>*	How does a parent subform determine when a child subform no longer applies based on child state data?	Have the subform notify parent when it no longer matches.
Handle Visual Component Events <*new*>	**How should a view, such as a form, handle an event notification message from an observed visual object, such as a Button?**	**Write a handler method for the reaction that only pulls information from the visual components and then calls update.**
Single View Update <*new*>	**How do you structure a view's update behavior such that any developer can quickly and easily understand the logic of the update?**	**Assume the entire view is out-of-date with the models and update everything.**
Multiple View Update <*new*>	**How does one subform force the update of visual components in other subforms?**	**The subform invokes the update behavior of its parent and child subforms**

Table 1. Pattern Problems and Solutions for Developing Graphical User Interfaces

2 Handle Visual Component Events Pattern

2.1 Problem

How should a view[1] you construct handle event notifications from its observable visual objects[2] such that the view can be updated independently of them?

2.2 Context

You have incorporated the Observer Pattern [4] into your view's design. This may have been a deliberate decision or the result of using another pattern, such as MVC, that uses the Observer pattern as part of its structure. Your view is interested in receiving event messages from its observable visual objects when specific events occur in them. Your view 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 have decided that rather than have the models, which are independent of one another, notify the visual components directly that they have changed, it is advisable to have a single coordinated update method in response to each event message.

2.3 Forces

- Many observable visual objects can generate identical event messages. Therefore, your view should not make any assumptions about which one sent the message.
- Using case statements to determine which observable visual object sent the event message or how to process it is inefficient and can make it difficult to decide how to add, remove or change the way you handle the events.
- Setting flags when handling event messages in order to decide which observable visual objects to update and when is complicated for you and for the future developers to understand.
- If the handling of a single event message results in many changes to your view's model, the MVC pattern forces the model to send notification messages to its corresponding visual objects several times. This can be inefficient when the visual components needs only to update once after the model has completed all changes.

2.4 Solution

Before your view handles an event message from a visual component, first disable your view from receiving any other events from its models or its visual objects. Next, your view's handler method should pull information from the appropriate visual components and then invoke the appropriate services in the models. Since the models are not generating event (update) messages anymore, make sure to update the corresponding visual objects with the latest states of the models when finished with the models. Your handler method should not do any updating of the visual components itself. Your handler should use a single update method that is responsible for all updating. When done, restore your view's observer relationship with the models and visual components in order to keep receiving events.

2.5 Resulting Context

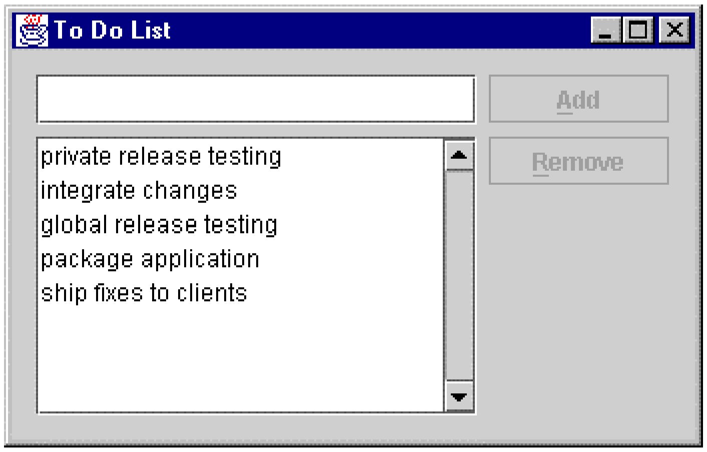
After receiving an event message, your view takes the required information from the appropriate visual components and invokes the correct service requests in the models. These actions change the states of the models, and, consequently, the view is now in need of an update to reflect the models' latest states. 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 process, change the corresponding handler method. One of the important features of this pattern is that 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.

2.6 Example

Consider the following Java implementation of a To Do List Window, as depicted in figure 2. The business model consists of a collection of items, where each item is a String. The user can add new items to the list by typing the new item 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.

[1] E.g., widgets subclassed from window, form, pane, applet, subform or subcanvas.

[2] E.g., widgets such as buttons and selection lists.



private release testing
integrate changes
global release testing
package application
ship fixes to clients

Figure 2. To Do List

The view's models are represented by instance variables associated to the **items** collection, the **selectedItem**, and the **newItem** to be added. In addition, the view keeps track of each visual component along with their corresponding listeners:

```
public class ToDoListFrame extends JFrame {
   private JTextField    newItemField;
   private JList  itemsList;
   private JButton       addButton;
   private JButton       removeButton;

   private String newItem;
   private Vector items;
   private String selectedItem;

   private EditableEntryListener newItemFieldListener;
   private EntriesListener       itemsListListener;
   private AddButtonListener     addButtonListener;
   private RemoveButtonListener  removeButtonListener;
   ...
}
```

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 objects (code not shown in the interest of brevity) in order for the view to receive event notification messages from them. In this example, the ***DocumentListener*** for the text field sends the ToDoListFrame the message ***newItemChanged()*** every time the user types in the text field. The ***newItemChanged()*** handler method reads the contents of the text field into the ***newItem*** instance variable.

```
private void newItemChanged() {
   if(newItemField.getText().length() == 0) {
          newItem = null;}
   else {
          newItem = newItemField.getText();}
   update();}
```

The ***ListSelectionListener*** for the list sends the ToDoListFrame the message ***selectedItemChanged()*** when the user selects an item in the list. The handler method reads the selected item from the list into the instance variables ***selectedItem*** and ***newItem.***

```
private void selectedItemChanged() {
          selectedItem = (String)itemsList.getSelectedValue();
          newItem =  selectedItem;
          update();}
```

The Add and Remove Buttons use an ***ActionListener*** to send the ToDoListFrame the messages ***addTodoEntry()*** and ***removeTodoEntry()*** when the user clicks on them, respectively. The ***addTodoEntry()*** handler method adds the new item to the collection of existing items, whereas the ***removeTodoEntry()*** handler method removes the currently selected item from the same collection.

```
private void addTodoEntry() {
   items.add(newItem);
   selectedItem = newItem;
   update();}


private void removeTodoEntry() {
   items.remove(selectedItem);
```

```
    selectedItem = null;
    newItem = null;
    update();}
```

All handler methods invoke a single update method just before they complete. This call to update **must** be consistent when using the *Handle Visual Component Events pattern.* The view models' states are modified as a result of handling user events, after which the states are loaded back into the visual components, during updating, to force the view to reflect the changes to the states.

3 Single View Update Pattern

3.1 Problem

How do you structure a view's update method such that you or any future developer can quickly and easily understand and change the logic it?

3.2 Context

Due to receiving and handling an event message, your view's models have changed and its visual components are out of date. In addition, you used the Handle Visual Components Events pattern to develop your view.

3.3 Forces

- One or more of the models may have changed. This may be a result of your view's direct interactions with the models, but it may also be due to external interactions between the models and other objects your view does not collaborate with directly. However, you must handle both cases.
- User created flags indicating which visual components need updating in a view are complicated to understand for both the immediate and future developers. Flags used in a systematic and documented way by all views makes them easier to understand and to use by developers.
- The speed of computer processors makes it possible to update the entire view without users noticing or even slowing them down.
- You are not always the one who will maintain your views so their logic must be easily understood.
- You can often improve performance buy using tricky implementation techniques. However, these tricks often complicate your code by making it less easy to read and comprehend.
- You do not always know what has changed in your view's models or which visual components need updating.

3.4 Solution

Assume the entire view is out-of-date and update everything. The speed of processors today allows this luxury and eliminates the need for developers to create their own special purpose flags to remember what to update. Write a specialized ***update*** method for each logically arranged part of your view. 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.

3.5 Resulting Context

The view's update method ensures that the view reflects the current state of the models. An important feature of this pattern is that to add, remove or change the way visual components are updated, there is one and only one corresponding update method to add, remove or change. In the case of adding or deleting visual components, there is also only one main update method to modify.

3.6 Example

Continuing with the previous example, update methods need to be written for all the components in the To Do List. The update method for the text field sets the contents of text field to the String in ***newItem*** variable. The update method for the list sets the contents of the list to the collection of ***items*** and the selection of the list to the ***selectedItem*** String:

```
 private void updateNewItemField() {
    newItemField.setText(newItem);}

 private void updateItemsList() {
    itemsList.setListData(items);
    itemsList.setSelectedValue(selectedItem, true);}
```

The update methods for the Add Button and the Remove Button only take care of enabling or disabling the buttons based 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() {
   addButton.setEnabled(newItem != null);}

private void updateRemoveButton() {
   removeButton.setEnabled(selectedItem != null);}
```

The main update method invokes the specialized ones. This approach can cause infinite loops in some windowing systems. This situation happens when the update method triggers a new event message in one of the visual components, causing the subsequent execution of a handler method, which results in another invocation of the update method, and so on.[3] To prevent this situation from occurring, you can disable the process that generates the events. 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 result is that we update all visual components while the listeners are disabled.

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

3.7 Variants

3.7.1 Partial Updates

As the view stabilizes, and refactoring becomes less and less frequent, one can improve the approach's performance by implementing partial updates. In this case, 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 views on slower machines.

```
private void updateNewItemField() {
   if(!newItemField.getText().equals(newItem)) {
          newItemField.setText(newItem);}}

private void updateItemsList() {
   itemsList.setListData(items);
   itemsList.setSelectedValue(selectedItem, true);}

private void updateAddButton() {
   boolean enableState = newItem != null;
   if (addButton.isEnabled() != enableState )
          addButton.setEnabled(enableState);}

private void updateRemoveButton() {
   boolean enableState = newItem != null;
   if (addButton.isEnabled() != enableState )
          addButton.setEnabled(enableState);}
```

3.7.2 Stopping Recursive Updates Without Disabling Listeners

As mentioned before, the updating of visual components by a view 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 the listener objects that generate the new event messages, which may be computationally expensive, another solution is to avoid processing any new event messages while already processing one. To include this ability, add a new private instance variable to your view called **reacting** and initialize it to false when creating your view. In our example, the class definition for the ToDoListFrame is modified as follows:

```
public class ToDoListFrame extends JFrame {
   private boolean        reacting  = false;

   private JTextField     newItemField;
   private JList  itemsList;
   private JButton        addButton;
```

[3] In some other frameworks, like the ones in VisualSmalltalk and VisualWorks Smalltalk, this never occurs because when writing a value into a component the component checks if the new value is equal to the currently stored value; if they are equal, nothing happens; if not, a changed event is triggered.

```
   private JButton       removeButton;

   private String newItem;
   private Vector items;
   private String selectedItem;

   private EditableEntryListener newItemFieldListener;
   private EntriesListener       itemsListListener;
   private AddButtonListener     addButtonListener;
   private RemoveButtonListener  removeButtonListener;
   ...
}
```

Also provide the following methods:

```
private boolean notReacting() {
   return !reacting;}

private void stopReacting() {
   reacting = false;}

private void startReacting() {
   reacting = true;}
```

Next, change all handler methods to the following format.

```
private void removeTodoEntry() {
   if (notReacting ()) {
          startReacting();
          items.remove(selectedItem);
          selectedItem = null;
          newItem = null;
          update();
          stopReacting ();}
```

Finally, remove the enable and disableListener methods for the main update method.

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

Now, no handler methods can fully 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.

4 Multiple View Update Pattern

4.1 Problem

Since a view's update methods are only responsible for updating their 'owned' visual components, how do you update the visual components in other views if required?

4.2 Context

You are using the Handle Visual Component Events pattern, the Single View Update pattern and one or more of the subform patterns in the pattern language for developing form style windows. In each one of your subforms, which you are treating just like any view, you have implemented a number of handler and update methods to keep the visual components in the subform up-to-date with the corresponding models. Either your subforms share models, or events generated and processed in one subform require others to be updated. The immediate subform is related to the others requiring updates in one of three ways:

1. They may be one of its parent subforms.
2. They may be one of its child subforms .
3. They may be one of its sibling subforms.

Since a subform is just a view and can therefore contain other visual components, a subform can contain other subforms. The root form of a view is usually a window, or in Java it can also be an Applet. This root form can contain additional subforms, and they in turn contain others. Although an update may start in any subform, any or all others may need to go through their own update.

4.3 Forces

- It is not scaleable for every subform to be informed when every other subform changes in order to perform updates. One could use the Observer pattern with subforms playing the parts of observer and observables, allowing subforms to be notified when another one changes. However, this would create dependencies between all subforms when the dependencies should be between the subforms and the models. In large views, the complexity of the implementation and the potential for cyclical dependencies makes this approach difficult to understand and manage.
- Subforms are often used as plug and play components. Therefore, they must be easy to install or remove from a view. For example, if you have an address subform that works with a Canadian address, you may need to change this to a US address subform when working with a US address object. The switching between subforms should not demand a large coding effort.
- Subforms often change at runtime. In some applications, often involving notebooks or tab controls, subforms are swapped dynamically. This requires that subforms and their update mechanisms must work for all update situations, not just situations involving specific subforms.
- You are not always the one who will maintain your code so the update logic must be easily understood by anyone.
- No subform magic flags should be used to indicate what needs to be updated because it is complicated to understand for both the immediate and future developers.

4.4 Solution

The first part of the solution is to insure that an updating subform has a reference to its parent subform and references to all of its child subforms. The updating subform can establish these references when initialized The second part of the solutions has the updating subform send both its parent and child subforms the update message whenever it is requested to update. Since both the parent and child subforms will do the same, all subforms must ignore other update messages when currently processing one. This will prevent any infinite updating loops from occurring. Since it is user events generated by visual components, such as a button, that begin the updating processing, and not the models, there is a single point of control. This insures that no simultaneous updates can occur within one view.

4.5 Resulting Context

From the single initiation point, update requests traverse the subform component hierarchy. Eventually, all subforms execute their updates and the views will reflect an accurate display of the underlying models. No subform magic flags are required and subforms can be changed, added or removed without having to make major modifications to existing subforms. 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 subforms

4.6 Example

Consider the following Java implementation of a Multiple To Do List Window, as depicted in figure 3. For this example, the corresponding Buttons, List and TextField are assembled in a ToDoListPanel (Java's version of a subform). Therefore, the example consists of a single Window with two child Panels. The window and subforms share common models: a collection of **items** (each item a String), a **selectedItem**, and the **newItem** to be added. The functionality of this window is the same as in the original To Do List. However, now what happens in one of the Panels will also change what is displayed in the other. For example, if an item is removed from one Panel's list, it will also be removed from the other.

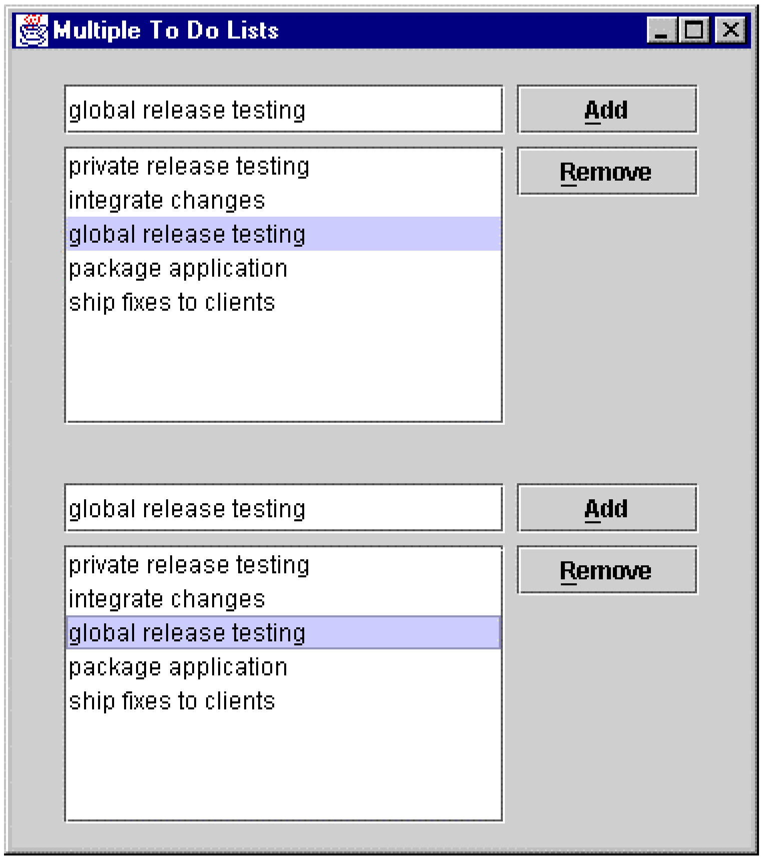


Figure 3. Multiple To Do List

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, we must handle the fact that multiple panels may require updating. Therefore, an update method invokes the update methods of its parent Window or subform and any of its child subforms, which in this is only the MultipleToDoListWindow. We also wrap the update method in a guard like we did for reacting to ensure that any new requests to update will not be done until the current one completes. The following update method is for the ToDoListPanel.

```
public void update() {
   if (notUpdating()) {
           startUpdating();
           disableListeners();
           updateNewItemField();
           updateItemsList();
           updateAddButton();
           updateRemoveButton();
           parent.update();
           enableListeners();
           stopUpdating();}}
```

The MultipleToDoListWindow's update method follows the same convention, but needs only to invoke the update of its subforms since the window has no parent.

```
public void update() {
   if (notUpdating()) {
           startUpdating();
           panel1.update();
           panel2.update();
           stopUpdating();}}
```

In some situations, updating may also generate new event messages. These events should be ignored while updating is taking place, so a similar react guard is applied to all handler methods, as shown for the Panels *addTodoEntry* method:

```
private void addTodoEntry() {
   if (notUpdating() && notReacting()) {
           startReacting();
           items.add(newItem);
           selectedItem = newItem;
           update();
           stopReacting();}
```

However, if your panels do not share models you do not need to guard the handler methods since updating in other panels will not generate any event messages for yours.

5 Summary

Our new patterns work in two phases. The ***handle*** phase changes the states of a view's models in response to end-user events, and the ***update*** phase updates the visual components to reflect the models' new states. Since the update phase immediately follows the handle phase, the view always reflects the latest changes.

Our patterns allow the views 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 view contains handler and update methods and an easily identifiable interaction between the views and the models. If there is a problem in the handling or updating in response to a user event, the developer knows where to look!

In order to successfully use these patterns, apply the following steps:

1. Force your developers to use the **same** design and implementation approach for **all** views.
2. Determine each view's state variables and associated objects.
3. Write a ***handler*** method for each event a user can generate from interacting with the view's visual components. Allow a handler method to modify **only** the view's model (state) 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 view's main update method.
4. Write a specialized ***update*** method for each visual component. Each method is responsible for updating the corresponding visual component from the view's state information. The method is permitted to directly load state information into the visual components or can compute the components' contents on the fly.
5. Write a single, main update method that invokes all the specialized update methods.

5.1 Known Uses

Carleton University has taught these patterns for many years to its first year students. Employees of The Object People, Inc. have successfully used them during large consulting projects. The patterns have been used with Digitalk's Smalltalk/V (now SmalltalkExpress) and VisualSmalltalk, ObjectShare's VisualWorks Smalltalk, IBM's VisualAge Smalltalk, and recently, with Sun's Java AWT and Swing frameworks.

5.2 Related Patterns

The Model-View Controller (MVC) pattern [3] and the Observer pattern [4] rely on an update mechanism similar to our patterns. However, the MVC pattern does not specify when information can be retrieved or placed in the visual components. Without this specification, one often requires either several user defined flags or spaghetti code to get the updating sequence correct. Our patterns make the assumption that the user interface is completely out-of-date after handling a request. This may seem a bit extreme. However, with today's computer speeds, this simplification does not cause **any** performance degradation and results in a much easier and extensible design than one that must figure out what and when to update.

References

[1] K Auer, K Beck. Lazy Optimization: Patterns for Efficient Smalltalk Programming, *In Pattern Languages of Program Design 2*, eds. Vlissides, Coplien and Kerth, pages 19-42, Addison Wesley, 1995.

[2] M. Bradac and Fletcher, A Pattern Language for Developing Form Style Windows, *In Pattern Languages of Program Design 3,* eds. Martin, Fiehle and Buschmann, h, pages 347-357, Addison Wesley, 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, R. Sommerlad, M. Stal. Model-View-Controller. *In Pattern-Oriented Software Architectures: A System of Patterns*, pages 125-143, Wiley, 1996.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Observer, In Design Patterns: Elements of Reusable Object-Oriented Software, pages 293-303, Addison Wesley,1995.