Centralized Selection Context for User Interface Data Binding and Event Handling

Stratovan Corporation
David F. Wiley
October 15, 2007

Summary

Conventional user interface paradigms have a distributed selection context having selection states spread across user interface elements (widgets). Our invention isolates the binding between widgets and data storage using a centralized selection context paradigm for all data items represented within a user interface. This selection context is fully accessible by methods that access data while considering the current selection context. The centralized selection context simplifies event-handling logic by having selection context readily available thus removing the need to query individual widgets for their state. In addition, the user interface workflow graph can be derived from the selection set definitions.

Definitions

1) **User Interface (UI)**: the human-computer interface allowing the computer user to interact with computer software.
2) **Widget**: a visual object representing some sort of data item the user can interact with on the computer monitor through the use of a keyboard, mouse, or some other means. Well known user interface widgets are: button, listbox, combobox, tree, menu, scrollbar, etc.
3) **User Interface Element**: used interchangeably with widget
4) **Data Source/Storage**: container for the data displayed by a widget usually manifesting as memory during program runtime but could reside on permanent storage.
5) **Data Binding**: encompasses the implementation of the data flow between a widget and the data source(s) it represents.
6) **Selection or Selection State**: the dynamic state of chosen a sets or subsets of data for visual depiction or upon which to perform operations.
7) **Selection Context**: the collection of multiple selection states.
8) **Workflow**: how a user traverses a user interface in terms of which widgets are interacted with and when, how often, the screen space traversed between widgets, and how this changes throughout a software session.
9) **Mediator**: a set of logic rules determining how to respond to, handle events or actions, and routing data flow between a variety of otherwise incompatible software components, such as widgets and data sources.
10) **Widget Toolkit Builder.** The provider or developer of a software development kit (SDK) supporting a generic set of widgets that an application developers can use to build software products/applications.
11) **Application developer.** A software developer responsible for designing and assembling a software product using available tools in order to create a particular workflow and dataflow in order to solve a particular problem or provide a particular service.
12) **Implement or Implementation.** This term is used primarily to describe the process of converting a concept, behavior, or feature into program source code.

Background

A user interface is a configuration of widgets conjoined with logic managing how each relates to data and interacts with other widgets for the purpose of facilitating dataflow, workflow, and ultimately program flow.

The purpose of a user interface is to provide the user with data visualization, selection capabilities, and command mechanisms. The combination of these functions and how they interrelate facilitates workflow, dataflow, and program control flow inherent to an application. User interface widgets facilitate data representation by rendering data, editing data, and/or providing lists of data items. Many widgets are selection sensitive and provide the capability to select sets of data items that can then be operated on when commands are issued to do so. These commands originate from command sensitive widgets. This patent pertains to the various aspects related to facilitating the user interface purpose and the difficulty involved with implementing such in source code.

Conventional UI Paradigm
All user interface widgets can be divided into four types:
1. **Visualization.** Typically read only, these widgets render text, graphics, or some other representation of data and generally do not allow the user to directly modify or change the displayed data.
2. **Selection based.** These widgets primarily visualize selection state, which may involve providing a visualization of data items, and may furthermore allow manipulation of the selection state.
3. **Command based.** These widgets simply inform user interface logic that the user wants to perform a specific operation on data selected in the current selection context.
4. **Combination.** These are widgets that provide some amount of each of the previous categories. For example, an edit box visualizes text, allows selection thereof, and issues commands to insert characters, delete characters, copy, paste, etc.

Every widget has some amount of visualization associate with it; for the purpose of this categorization, we are concerned with a widget's primary function. The following is a list of commonly used widgets and their primary function (this list is by no means exhaustive):

Visualization	Selection	Command	Combination
Static text	Check box	Button	Edit Control (selection and command)
Group box	Radio button		Menu (selection and command)
Picture	Combo box		Custom control
	List box		
	Scroll bar		
	Slider		
	Progress bar		

The conventional UI paradigm groups all widgets into the Command category so that each widget issues commands and/or notifications that indicate what the widget is doing. It is then the

responsibility of the application developer to manage these notifications in order to create the intended widget behavior as described by the categories above.

Additionally, the conventional paradigm includes the concept of aggregate widgets, for example, tab controls or property sheets. This is an aggregation of widget primitives and is better described as a widget configuration rather than a widget itself.

Conventional Paradigm for Selection Management
The conventional paradigm depends on having a distributed selection context for the UI where each widget internally stores and maintains the selection state relevant to that widget. This is primarily because selection information is needed during widget rendering and also when interacting with the user (having it readily available within the widget makes these tasks easier when implementing the widget).

Each widget has a well-defined purpose and can only display certain types of data resulting in a rigid interface for transferring data to/from the widget as well as manipulation of the internal selection state. The standard set of widgets generally only support primitive data types such as a string, a number, etc. or collections thereof.

A widget's primary purpose is to display data in a form understandable by a software user. Thus, each widget is connected in some way to data storage for the purpose of displaying that data. Additionally, this can involve writing to data storage as well as reading, in the case of editing data.

Data storage is a collection of data having unknown structure for which the user interface is being constructed (unknown at least to the widget toolkit builder). Since only the simplest data storage is directly compatible with conventional widget interfaces, a mediator is needed to provide the "translation" and data path between widgets and their data storage. This data flow process between widgets and data storage is known as data binding. To facilitate software workflow and dataflow, mediators perform a variety of tasks that can be divided into four groups:

1. **Data transfer.** Facilitates the data path between widgets and their data sources in addition to querying widget selection states in order to decide what data to provide or otherwise access at any moment.
2. **Widget event handling pertaining to data and/or selection context.** Encompasses managing widget selection state and also the state of widgets that depend on one another. Additionally, this includes providing new data to widgets when a change in selection state requires different data to be displayed in any widget.
3. **Widget event handling pertaining to command/control.** Encompasses the actions directly related to widgets performing operations on data selected in the current selection context. This usually involves receiving an event and then having to query widget selection states in order to determine how to best handle that event.
4. **Miscellaneous event handling and control flow.** Encompasses program flow, configuration changes, data management, layout resizing, and other miscellaneous tasks most likely related to workflow, dataflow, and control flow.

Fig. 1 shows how these duties relate to one another under the conventional paradigm. In general, selection-state management and data delivery to and from widgets is handled by the mediator. Some widget toolkits provide direct data transfer (i.e., data transfer is handled independently from the mediator), thus alleviating some data binding issues. However, direct data transfer still requires the mediator to manage widget select state for the UI.

The main difficulty regarding the UI mediator, in terms of software development, lies in having to implement the logic driving widget selection-context management and bound data. This is primarily rooted in the conventional paradigm having a distributed selection context requiring the mediator to query widget selection states for nearly every logic step.

Fig. 2 generically describes the mediator's role in the conventional data binding process and the general steps that are required. This flowchart is divided into three columns to better highlight when the mediator crosses widget and data storage boundaries. Crossing boundaries in this manner is important due to the complexity involved in implementing the crossing in source code. The flowchart for every user interface is dependent upon the application, but for the most part the application developer will follow these steps under the conventional paradigm.

Fig. 3 simplifies the process shown in Fig. 2 by representing the components responsible for the boundary crossings and their relation to where the selection context is maintained. The mediator is in the middle of the dataflow between the widget and its data storage. In this position, it must facilitate data transfer between the two in addition to managing widget selection state. However, some user interface toolkits allow the direct connection of widgets to their data storage, which results in that shown in Fig. 4. Direct data binding under the conventional paradigm alleviates some data transfer issues but still requires mediator involvement in order to manage the overall selection context so that the appropriate data can be accessed.

The following is a categorization of popular user interface toolkits and the primary data binding method they are founded on:

Conventional Data Binding (distribute selection context)	Direct Data Binding (distribute selection context)
Macintosh API, Carbon (Apple Corp.)	QT (cross platform)
Windows API, WinAPI, Win32 (Microsoft Corp.)	
Xaw, Motif, GTK+, (X Windows platform)	
FLTK (cross platform)	
wxWidgets (cross platform)	

Many additionally toolkits are available, though, they are nearly all derivatives of those listed above (i.e., MFC is based on Win32 and Cocoa is based on Carbon) and thus share the same data binding and selection paradigm. Additionally, toolkits for languages such as Java (and others) are founded on conventional data binding and distributed selection principles. Thus, the particular language of implementation is irrelevant.

It is important to note that some amount of direct data binding can be implemented in any widget toolkit; however, most are not designed to use it (QT is the exception). Additionally, it is important to note that all widget toolkits listed above utilize a distributed selection context, i.e., every widget contains its own selection internally and the mediator must explicitly query it when making decisions.

Description

Our invention modifies the conventional paradigm by centralized the user interface selection context. This allows the selection context to be fully accessible to the widgets, data storage, and mediator. Combining this with direct data binding yields a powerful, yet easy to use, data binding system, this is demonstrated in Fig. 5. These changes result in the removal of the mediator from the data-binding loop that exists in the conventional paradigm. This is only possible since the selection context can be freely queried by data storage and can be freely updated (and queried) by widgets without relying on the mediator to facilitate this.

In this situation, the widgets communicate directly with the selection context in order to update selection changes resulting from user interaction and furthermore query the selection context on their own prior to rendering to insure they render the proper state.

The data-access interface is primarily based on the required interface for each particular widget. This is because each widget can only render finite data types. When the widget needs to be filled with data, it inquires across the data-access interface to the data storage, which in turn queries the selection context in order to determine the appropriate data to give to the widget.

The mediator is still involved in creation, destruction, and other management tasks unrelated to data transfer. The mediator can still access the widgets, data storage, and central selection context if needed. However, it is not required to participate in data binding at the same level as it is in the conventional paradigm.

Fig. 6 shows how, when using a centralized selection context, the data binding steps related to selection management have been moved from the mediator into both the widgets and data storage. This removes the selection management from the mediator and places it primarily in widgets themselves. The selection state itself is maintained in a central location as opposed to distributed at each widget. Consolidation of the selection context allows the entire context to be queried more easily (this is in contrast to having to query each widget individually to obtain this information). Consolidation also allows the entire context to be easily accessed from software components that do not normally have direct access to that information. Data storage, for example, can internally query the centralized context and directly obtain selection state information needed to access the selected data items.

The mediator is simplified when using a centralized selection context by reducing complex logic associated with binding widgets to their data (it has essentially been split and moved into the widgets and data sources). Additionally, the centralized selection context only manages selection state and is simple in design. However, under this paradigm, widgets and data sources are required to provide additional functionality (to accommodate what was eliminated from the mediator). For widgets, this includes:
1. Handling its events internally.

2. Filling itself with data.
3. Updating the centralized selection context.
4. Querying the centralized selection context.

For data sources/storage, this includes:
1. Exposing an interface for transferring data to/from widgets.
2. Querying the centralized selection context in order to determine the correct data to provide to a widget.

One can argue that the increase in widget complexity is mitigated by the well-defined role of each widget. Thus, there is a finite amount of logic needed for each widget, which can be managed by the toolkit builder rather than the application developer, greatly simplifying the application development process.

The complexity of data sources slightly increases under this model since the developer must provide an interface the widget can communicate to it through. This logic previously existed in the mediator. However, the fact that data gathering (i.e., the process of preparing data for transferring to or processing received data from the widget) is located in the data source greatly simplifies the gathering process since the data source itself has full knowledge of the (potentially complex) data structure it represents. Thus, it can easily determine what and where things go depending upon the context. Whereas, in the conventional paradigm, the mediator is required to interrogate data sources in order to access their internal data structures.

Example User Interface Implementation for Selecting a Book
Fig. 7 shows an example user interface allowing the selection of a page within a book at a particular house. The conventional paradigm stores selection context for each widget inside the widget itself and when a widget changes selection state the mediator must detect this change and subsequently provide the correct data for the new context to the dependent widgets. A flowchart demonstrating this process was shown previously in Fig. 2. Under the centralized context paradigm, the mediator's data binding responsibilities are substantially reduced and the remaining responsibilities are described in pseudo code in the following to demonstrate how to implement this concept in source code.

The user interfaces that benefit most from the centralized selection context paradigm are those that represent multi-level (or nested) data structures. In contrast, conventional user interfaces are better suited for flat data structures such as:

```
class Person
{
public:
   std::string m_strFirst;
   std::string m_strMiddle;
   std::string m_strLast;
   std::string m_strStreet;
   std::string m_strCity;
   std::string m_strState;
   std::string m_strZIP;
   std::string m_strPhone;
};
```

This is because there exists only one level of selection indirection, in this case, the Person object itself. Considering the bookshelf example, multiple levels of selection must be traversed in order to arrive at the page text. Traversing selection levels is the major difficulty in implementing conventional user interfaces since there is no supporting mechanism to easily keep track of the multiple selection levels as well as providing this information to the components that need it. The centralized selection context allows this information to be readily available making this process much easier. The centralized selection context is (usually) comprised of several selection sets. Each selection set represents a group of similar data items that are displayed by the user interface. Selection sets are typically a result of the data structure being represented by the user interface.

Pseudo code for the data structure of our bookshelf example is:

```
Object Page { Text }

Object Book { Title and a Vector of Page Objects }

Object Bookshelf { Location and a Vector of Book Objects }

Object House { Homeowner and a Vector of Bookshelf Objects }

Object AvailableHouses { Vector of House Objects }
```

The same data structures in C++ are given by:
```
class Page
{
public:
   std::string m_strText;
};

class Book
{
public:
   std::string        m_strTitle;
   std::vector< Page >  m_vPages;
};
```

```
class Bookshelf
{
public:
   std::string          m_strLocation;
   std::vector< Book >  m_vBooks;
};

class House
{
public:
   std::string              m_strHomeowner;
   std::vector< Bookshelf > m_vBookShelves;
};

class AvailableHouses
{
public:
   std::vector< House >     m_vHouses;
};
```

In this case, four levels of selection are traversed to arrive at the page text: house, bookshelf, book, and the page itself.

We introduce the following terms in order to implement our centralized-selection user interface paradigm:
1. **Datum.** An individual piece of data that is represented in some way by the user interface. A unique identifier and a name identifies a datum and is referenced by: DATUM(name).
2. **Selection Set.** A set of datum handles (HDATUM) identified by a unique identifier and a name. This set can contain zero or more datum handles. A range may also be specified during initialization. A selection set is referenced by: SELECTIONSET(name).
3. **Command.** Invokes an operation on selected data items or performs some other control flow operation, is identified by a unique identifier and name, and is referenced by: COMMAND(name). (We do not used commands in this example but list it for completeness.)
4. **SelectionContext.** A collection of selection sets. This represents the centralized context and all selection information is contained within.
5. **HDATUM.** Represents an index or handle to a particular Datum and is the data type used internally within selection sets to keep track of which items are selected. An array index is a good example of this data type and how it is used.

Each of the operators DATUM, SELECTIONSET, and COMMAND are a function that converts the name into the unique identifier. We also assume that passing the same name to different operators results in different identifiers such that DATUM(name) does not equal COMMAND(name). Additionally, there exist declaration versions of each of these: DATUM_DECLARATION, SELECTIONSET_DECLARATION, and COMMAND_DECLARATION. These are used to declare the entity itself for use within the program.

The list widgets in this example simply display a list of strings from which the user selects in order to modify the selection context. The data transfer in this case can be implemented using the following interface:

```
class IDataSelectionAccess
{
public:
   virtual bool GetSelectionItems(
               const SelectionContext   &context,
               const SelectionSet        &sel,
               std::vector< std::pair< HDATUM, std::string > >
                                          &vItems)const;
};
```

In which case, upon return of the call to GetSelectionItems(), vItems is filled with the list of strings to be displayed. Additionally, a handle is associated with each string and is used to represent that particular string datum if the user selects it in the widget.

The preview widget displays text in this example and the data transfer for this widget can be implemented using the following interface:

```
class IDataAccess
{
public:
   virtual bool Get(const SelectionContext &context,
                  const Datum &datum, std::string &data)const;
   virtual bool Set(const SelectionContext &context,
                  const Datum &datum, const std::string &data);
};
```

In which case, upon return of the calls to Set() or Get(), data is either written or read to data storage. The data in this case is transferred using the parameter "data."

Subsequently, a list widget can be bound to a particular selection set and data storage in the following manner:

```
class ListWidget
{
public:
   virtual void Bind(
               const SelectionContext   &context,
               const SelectionSet        &sel,
               const IDataSelectionAccess &access)
               {
                     m_pContext   = &context;
                     m_SelSet     = sel;
                     m_pAccess    = &access;
               }
protected:
   const SelectionContext        *m_pContext;
   SelectionSet                  m_SelSet;
   const IDataSelectionAccess    *m_pAccess;
};
```

Thus, when a list widget needs to fill itself with data, it need only call GetSelectionItems() on the bound data access pointer m_pAccess. To make this call, it passes the bound selection context as well as the selection set it is querying. It can then use the resulting list to display the data.

The preview widget can be bound to a datum and data storage in the following manner:

```
class PreviewWidget
{
public:
   virtual void Bind(
                const SelectionContext    &context,
                const Datum               &datum,
                const IDataAccess         &access)
                {
                     m_pContext   = &context;
                     m_Datum      = datum;
                     m_pAccess    = &access;
                }
protected:
   const SelectionContext          *m_pContext;
   Datum                           m_Datum;
   const IDataAccess               *m_pAccess;
};
```
In this example, since the preview widget is read-only, only the Get() method (defined by the
IDataAccess interface) is used by the preview widget.

Using these constructs, we supplement the bookshelf data structures (these data structures
constitute the data storage in this example) in the following way to apply the centralized
selection paradigm to this example:
```
class Page : public IDataAccess
{
public:
   DATUM_DECLARATION(Text);       // Used to access m_strText
   virtual bool Get(...)const;
   std::string m_strText;
};

class Book : public IDataSelectionAccess, public IDataAccess
{
public:
   SELECTIONSET_DECLARATION(Page);   // Used to access m_vPages
   virtual bool GetSelectionItems(...)const;
   virtual bool Get(...)const;
   std::string           m_strTitle;
   std::vector< Page >   m_vPages;
};
```

```
class Bookshelf : public IDataSelectionAccess, public IDataAccess
{
public:
   SELECTIONSET_DECLARATION(Book);   // Used to access m_vBooks
   virtual bool GetSelectionItems(...)const;
   virtual bool Get(...)const;
   std::string        m_strLocation;
   std::vector< Book >  m_vBooks;
};

class House : public IDataSelectionAccess, public IDataAccess
{
public:
   SELECTIONSET_DECLARATION(Shelves);// Used to access m_vBookShelves
   virtual bool GetSelectionItems(...)const;
   virtual bool Get(...)const;
   std::string           m_strHomeowner;
   std::vector< Bookshelf > m_vBookShelves;
};

class AvailableHouses : public IDataSelectionAccess, public IDataAccess
{
public:
   SELECTIONSET_DECLARATION(House);  // Used to access m_vHouses
   virtual bool Get(...)const;
   virtual bool GetSelectionItems(...)const;
   std::vector< House >     m_vHouses;
};
```

In our mediator, to initialize the centralized selection context, four selection sets need to be created to represent the four selection levels in this data structure, in C++/pseudo-code this is:

```
SelectionContext context;
context.Create( SELECTIONSET(House) );
context.Create( SELECTIONSET(Bookshelf),     SELECTIONSET(House) );
context.Create( SELECTIONSET(Book),          SELECTIONSET(Bookshelf) );
context.Create( SELECTIONSET(Page),          SELECTIONSET(Book));
```

The create method creates the selection set passed as the first parameter and also establishes a dependency on the second (if there is one). This manages invalidation of dependent selection sets when a parent changes state, for example, the selected bookshelf must be invalidated when the selected house changes since the bookshelf is no longer in context.

The widgets are bound to the selection context and data storage in the following manner:

```
AvailableHouses      houses; // Data storage
ListWidget           listHouses, listBookshelves, listBooks, listPages;
PreviewWidget        preview;
ListHouses.Bind(context,       SELECTIONSET(House),      houses );
listBookshelves.Bind(context,   SELECTIONSET(Bookshelf), houses );
listBooks.Bind(context,         SELECTIONSET(Book),       houses );
listPages.Bind(context,         SELECTIONSET(Page),       houses );
preview.Bind(context,           DATUM(Text),              houses );
```

At this point, the selection set dependency has been established as well as how widgets are bound to it and the data storage. These relationships are shown in Fig. 8. The key to the centralized selection context is in how data is accessed once these relationships are established.

This only becomes apparent in the implementation of the Get(), Set(), and GetSelectionItems() methods. For example, the GetSelectionItems() method for AvailableHouses is given by the following:

```
bool AvailableHouses::GetSelectionItems(
                const SelectionContext    &context,
                const SelectionSet        &sel,
                std::vector< std::pair< HDATUM, std::string > >
                                        &vItems)const
{
    if(SELECTIONSET(Houses) == sel)
    {   // Return the list of houses
        for(int n = 0; n < m_vHouses.size(); ++n)
        {
            vItems.push_back(
                std::pair< HDATUM, std::string >(n,
                                m_vHouses[n].m_strHomeowner) );
        }
        return true;
    }
    int nHouse = context.GetSelection( SELECTIONSET(Houses) );
    if((nHouse < 0) || (nHouse >= m_vHouses.size()))
        return false;
    return m_vHouses[nHouse].GetSelectionItems(context, sel, vItems);
}
```

The first part of the method checks to see if the selection set being queried is one that this level in the data structure maintains. If so, it fills vItems with the appropriate data, in this case, homeowner names. If not, the query is passed on to the nested house data that is in context. The remaining implementations of this method for the other data objects are similar. The base case is in the Book in that if the selection set passed is not Page, then the method simply returns false (since it has nothing to pass the query on to).

Subsequently, the Get() and Set() defined by the IDataAccess interface are similarly constructed in regard to how they compare the passed datum with those that particular object handles. If the object can handle that datum, it does so. If not, it passes the query on to nested data item that is in context. This sort of data access handling allows widgets to be bound to the topmost data object and is never invasive to the data hiding mechanisms employed in most aggregate types.
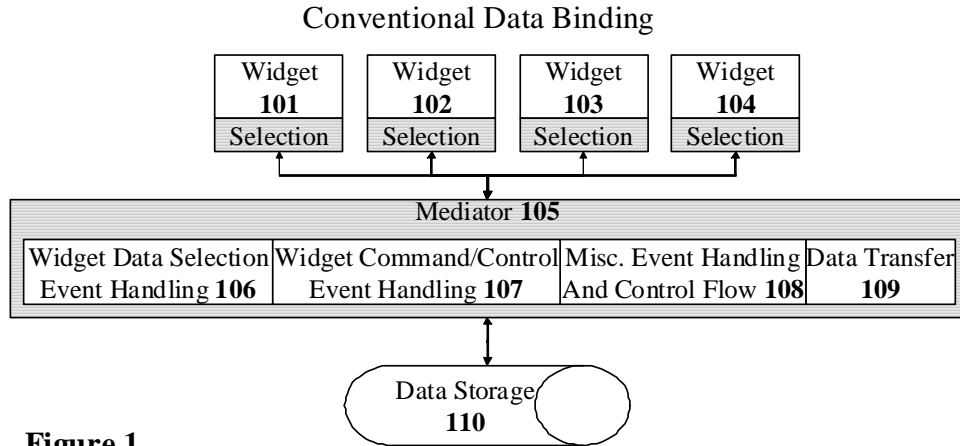
Extrapolation of User Interface Workflow
A useful byproduct of having defined selection sets is a tangible user interface workflow graph. Considering the two widget layouts for the bookshelf example shown in Figs. 7 and 9, we are concerned with how one determines which is better, easier to use, and a more intuitive user interface layout. This is conventionally done based on good design principles and ones knowledge of the art of information presentation---a qualitative measure.

Both layouts are organized and aligned but the questions remains as to how one determines that the layout shown in Fig. 9 is "poor" when compared to that of Fig. 7. This question can be answered by examining the selection set dependency of widgets. Fig. 10 numbers the widgets based on their associated level in the selection dependency hierarchy. The workflow steps are then visualized by rendering the transitions between selection set levels, since this equates to the steps a user must take in order to select data. The workflow visualizations are shown in Fig. 11.

Evaluation of the resulting workflow graph as indicated by the numbers and orange arrows clearly shows the problem with the layout shown in Fig. 9; primarily that the user must move back and forth horizontally in order to traverse the selection hierarchy; whereas, in the layout shown in Fig. 7, the user need only traverse monotonically from left to right. Thus, a workflow evaluation criterion could be how many times the workflow graph changes direction; a measureable quantity in this case

Easy to use user interface layouts follow cultural constructs that have been in place for some time. The construct that applies in this case is that of left-to-right (or right-to-left depending upon the cultural language) and usually top-to-bottom organization of information. This ordering establishes a natural progression a person follows in order to arrive at the desired result.
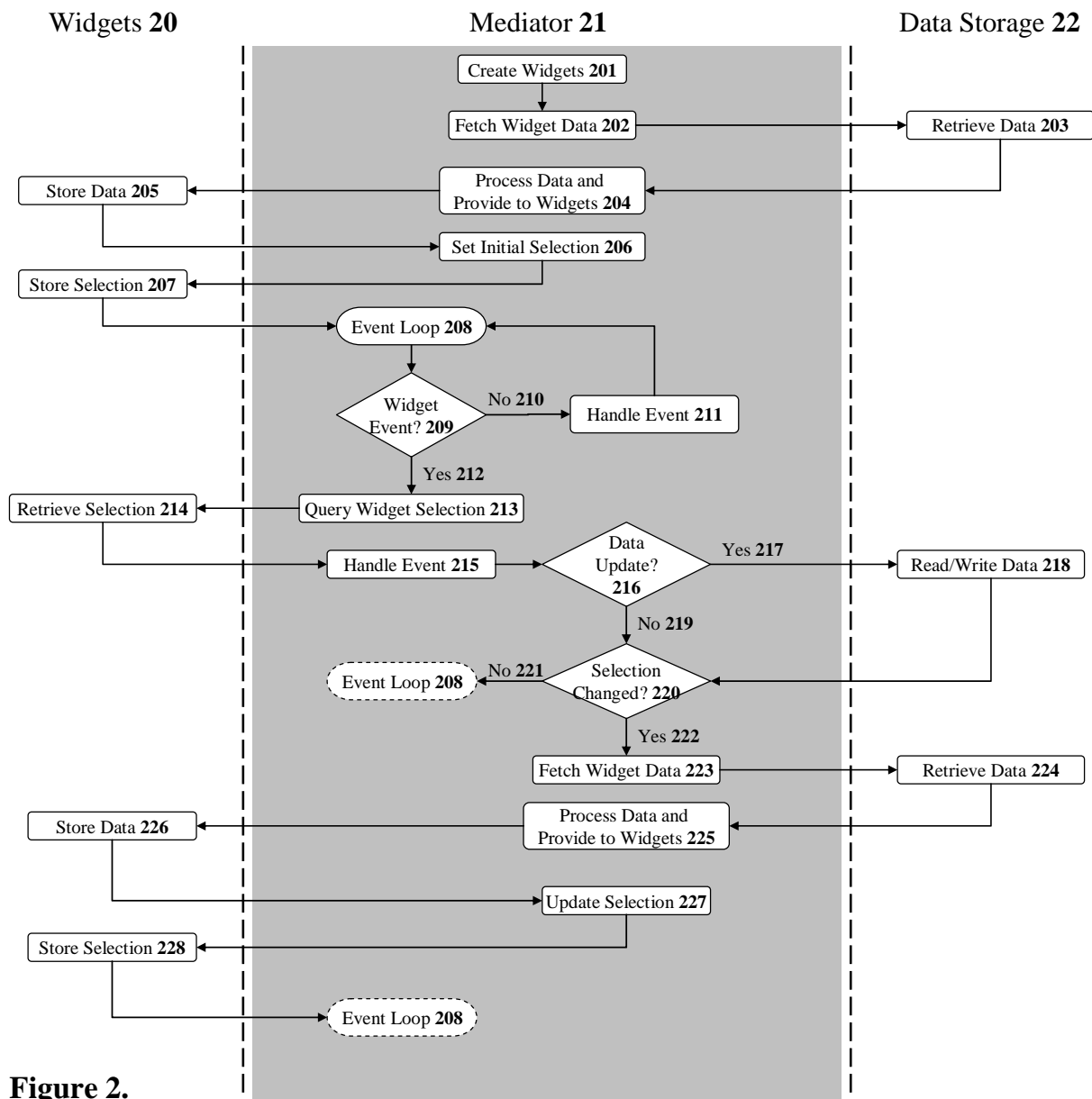
Fig. 12 shows the steps involved in obtaining the user interface workflow in tangible form with respect to widget positions within a layout. One can use this information for a variety of things such as i) automatic widget placement based on workflow; ii) layout grading (or evaluation) based on workflow complexity (as appropriate for a given culture); iii) visualization of the workflow; and iv) tracking to identify which user interface components (not necessarily widgets) a user spends their time interacting with.

Conventional Data Binding

Widget 101	Widget 102	Widget 103	Widget 104
Selection	Selection	Selection	Selection

Mediator 105			
Widget Data Selection Event Handling 106	Widget Command/Control Event Handling 107	Misc. Event Handling And Control Flow 108	Data Transfer 109

Data Storage 110

Figure 1.

Fig. 1 shows the conventional data binding of widgets (101 to 104) to the data storage 110 they represent. Widgets (101 to 104) are connected to data storage 110 through a process managed by the mediator 105. Mediator duties 106, 107, 108, and 109 facilitate dataflow, workflow, and program flow for a user interface. The user interacts with the widgets which send events to the mediator 105 that are handled depending upon the event nature. This involves managing events related to: i) selection changes 106 and how that affects widgets dependent on that change; ii) command, control, or program flow 107 operating on selected data; iii) miscellaneous event handling and program flow control 108; and iv) data transfer 109 which moves data between the widgets (101 to 104) and the data source 110 depending upon the selection context, which is distributed among the widgets.

Data Binding Steps Under the Conventional Paradigm

Widgets **20**	Mediator **21**	Data Storage **22**

Create Widgets **201**

Fetch Widget Data **202** — Retrieve Data **203**

Store Data **205** ← Process Data and Provide to Widgets **204** ←

Set Initial Selection **206**

Store Selection **207** ←

Event Loop **208**

Widget Event? **209** — No **210** → Handle Event **211**

Yes **212**

Retrieve Selection **214** ← Query Widget Selection **213**

Handle Event **215** — Data Update? **216** — Yes **217** → Read/Write Data **218**

No **219**

Event Loop **208** ← No **221** — Selection Changed? **220**

Yes **222**

Fetch Widget Data **223** → Retrieve Data **224**

Store Data **226** ← Process Data and Provide to Widgets **225** ←

Update Selection **227**

Store Selection **228** ←

Event Loop **208**

Figure 2.

Fig. 2 shows the general steps needed to implement the conventional data binding process. This process describes the interplay between the widgets 20, mediator 21, and data storage 22. The mediator 21 first creates the widgets 201 and fetches data 202 that the widgets will display. Data storage is accessed 203 to obtain the widget data that is then processed by the mediator (to make the data compatible with each widget) and subsequently provided to the widgets 204. Each widget stores a local copy of the provided data 205. The mediator then sets the initial selection for each widget 206 and the widgets subsequently store this selection internally 207. At this point, the application usually enters the main event loop 208. Then, during the remaining portion of the software session, a series of events are produced (usually by the user) and are

subsequently handled by the application in some way. In this example, we focus primarily on widget events and how the mediator 21 handles those. Thus, when a widget event occurs 212, the mediator must first query the widget selection context 213 and retrieve the selection from the widgets 214 in order to appropriately handle the event 215. To handle the event, the mediator must determine if the event modifies data storage in any way 216. If yes 217, then data is either read or written to data storage 218. If not 219, the mediator then determines if the selection context has changed 220 in such as way that dependent widgets require new data. If not 221, the application re-enters the main event loop. If widgets do need new data 222, then the mediator must again fetch data 223 from data storage 224 and subsequently process and provide the data 225 to the widgets. The widgets then store new data 226 (overwriting or modifying the existing). Finally, the mediator updates the selection context for widgets that have changed 227 and the widgets subsequently store the new selection. The application then re-enters the main event loop 208.
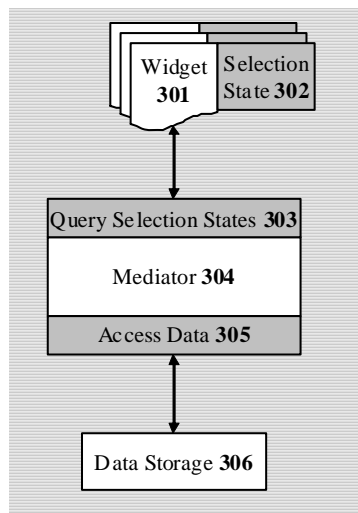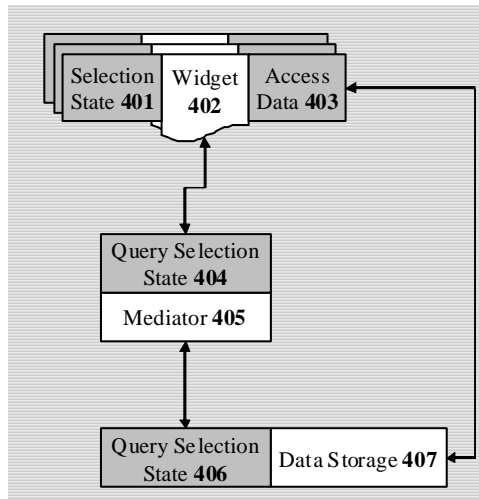
Conventional Data Binding Paradigm

Figure 3.

Fig. 3 shows a graphical representation of the process shown in Fig. 2. Each widget 301 maintains its own selection state 302. The mediator 304 must first query 303 selection states 302 through the widgets 301 in order to access the appropriate data 305 in the data storage 306. Selection state 302 can only be queried through the widget 301. When data storage 306 needs context information, the mediator must query the widgets for selection state and then provide the results to data storage 306 so it can access the appropriate data based on the context.

Conventional Paradigm with Direct Data Binding

Figure 4.

Fig. 4 shows the conventional data binding process when using direct data binding. Each widget 402 maintains its own selection state 401 and additionally has a component to access 403 the connected data storage 407 directly (for reading or writing data). However, in order to access the appropriate data (which depends on the selection context for the software) data storage 407 must query selection states through the mediator 405. The mediator 405 subsequently queries 404 selection state 401 through the widgets 402 and provides the results back to data storage 407.

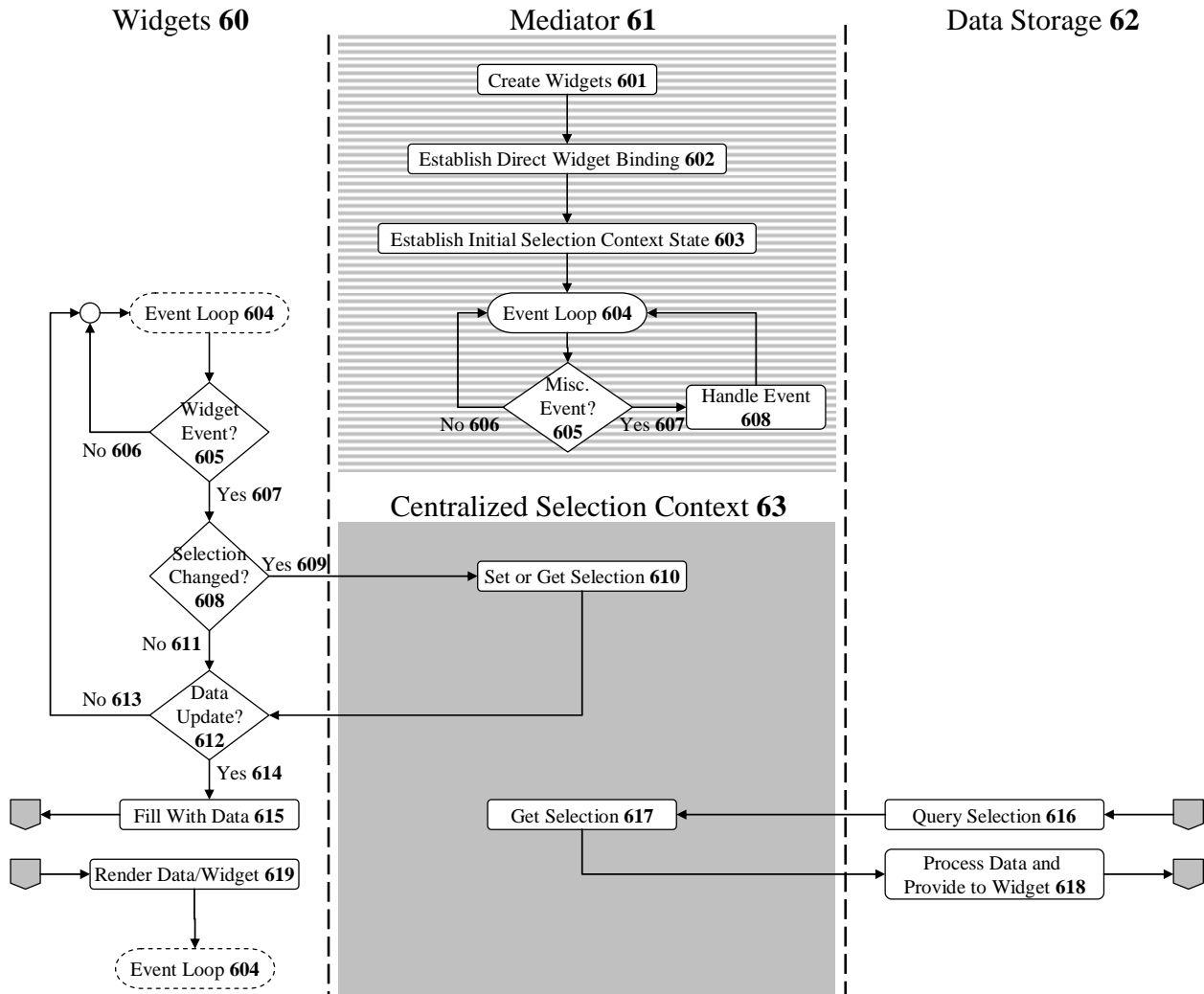Data Binding with Centralized Selection Context
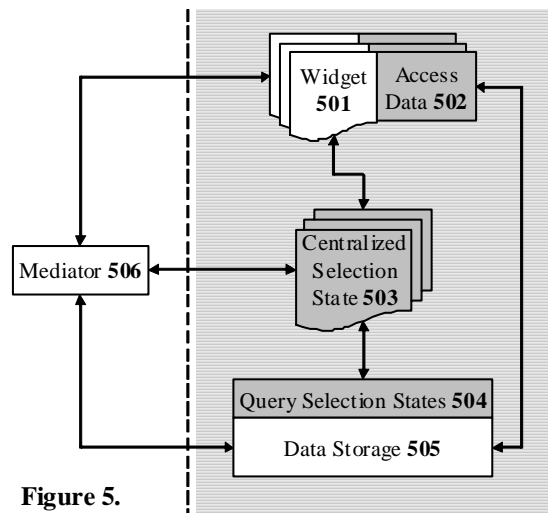
Figure 5.

Fig. 5 shows data binding with a centralized selection context 503. Each widget 501 no longer maintains the selection state relevant to that particular widget. All selection context is centralized 503 such that if a widget needs to access or modify the selection state relevant to that widget, it must access/update the centralized selection-state repository 503. When a widget 501 accesses data 502, data storage 505 can directly query 504 any needed selection context information without having to directly interact with the mediator 506 or any of the widgets 501. The mediator is still involved in miscellaneous tasks such as creation, connection, and destruction, but plays

little or no part in the data flow between widgets 501 and data storage 502 during the software session and is certainly not required to do so as it is in the conventional paradigm.

## Data Binding Steps Using a Centralized Selection Context

Widgets 60   **Mediator 61**   **Data Storage 62**

- Create Widgets **601**
- Establish Direct Widget Binding **602**
- Establish Initial Selection Context State **603**
- Event Loop **604**
- Misc. Event? **605** — No **606** / Yes **607** — Handle Event **608**

Centralized Selection Context 63

- Event Loop **604**
- Widget Event? **605** — No **606** / Yes **607**
- Selection Changed? **608** — Yes **609** — Set or Get Selection **610** / No **611**
- Data Update? **612** — No **613** / Yes **614**
- Fill With Data **615**
- Get Selection **617** — Query Selection **616**
- Render Data/Widget **619** — Process Data and Provide to Widget **618**
- Event Loop **604**

Figure 6.

Fig. 6 shows how the mediator steps related to data binding, as described in Fig. 2, change when using a centralized selection context combined with direct data binding. Initially, the mediator creates the widgets 601 and subsequently connects each widget to the data storage it represents 602. The central selection context 63 is created 603 and the initial state is set within this context. The application then enters the main event loop 604. Miscellaneous events 607 are handled by the mediator 608 and then return to main event loop 604. Under this scheme, widgets handle selection management internally as opposed to the conventional paradigm which requires the mediator to manage selection. Additional widget functionality manages selection state and data binding for that widget. When widget events occur 605, the widget first determines if this event modifies (or has modified) the selection state. If so 609, the widget queries (or updates) the
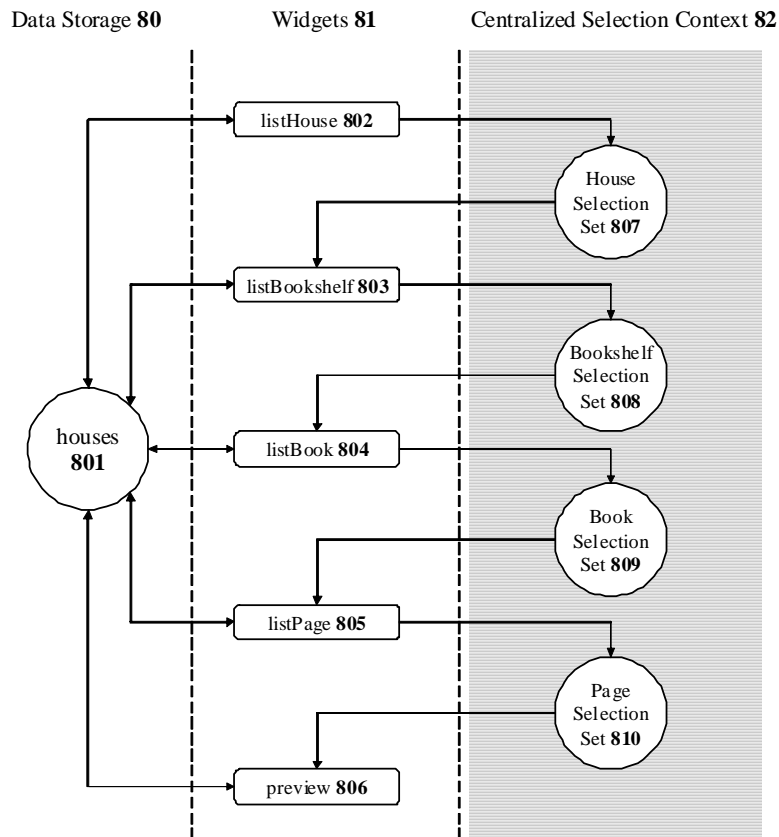
centralized selection context depending upon the event nature. If the data requires updating 612, then the widget must fill itself with data 615. This is done by querying the data access interface, which internally queries 616 the centralized selection context in order to determine the appropriate data to access. Data storage 62 then accesses the data internally and prepares 618 it for use by the widget. The widget receives the data 619 and subsequently stores or renders it prior to returning to the event loop.

Bookshelf Example
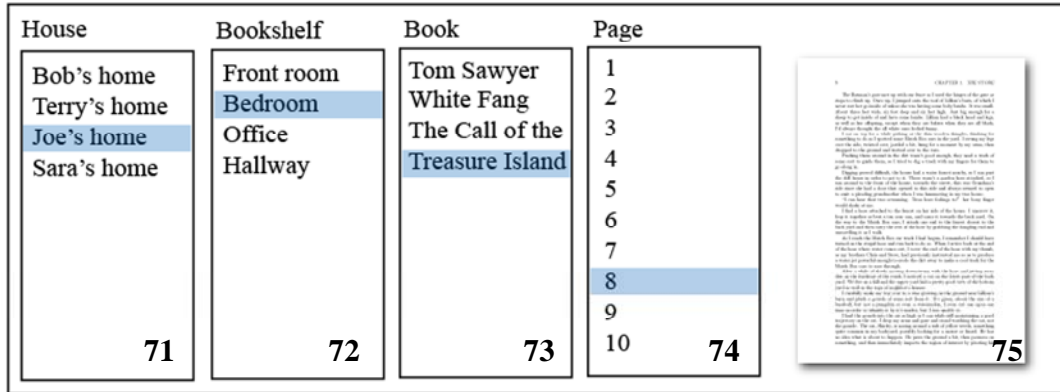
Figure 7.

Fig. 7 shows a user interface allowing the selection and subsequent display of a page in a book. The house widget 71 first selects the house in which to narrow down which book the user wants to read. The bookshelf widget 72 chooses one of the bookshelves in the house. The book widget 73 chooses a book on the shelf. The page widget 74 chooses one of the pages in the book that is subsequently displayed in the preview widget 75. Note that the items displayed in each widget are dependent upon the selection of that to its left.

Figure 8.

Fig. 8 shows the relationships of data storage 80 to widgets 81 and the centralized selection context 82. Each widget (802 through 806) is attached directly to the top-level data storage object called houses 801. Additionally, each widget is attached to the selection sets (807 through 810) that they manipulate. Arrows going from a selection set to a widget indicates that the widget's content is dependent on that selection state, for example, the list of books 804 depends on which bookshelf is currently selected 808.
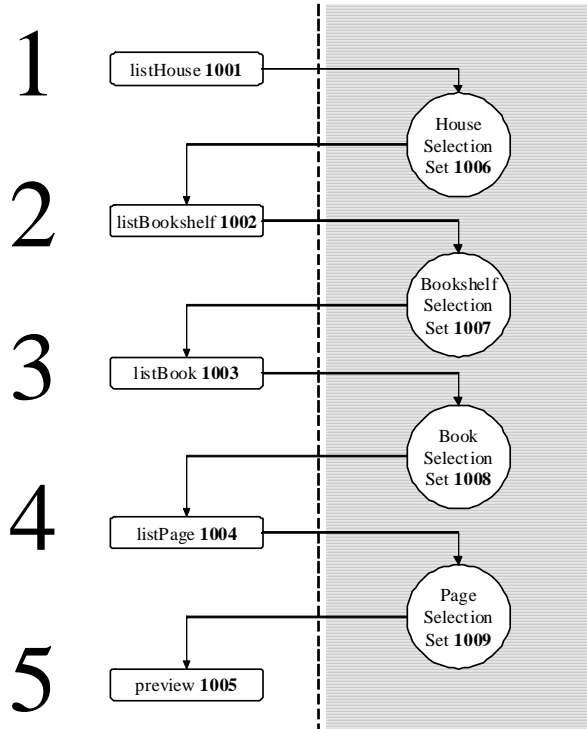
Alternative Bookshelf Layout
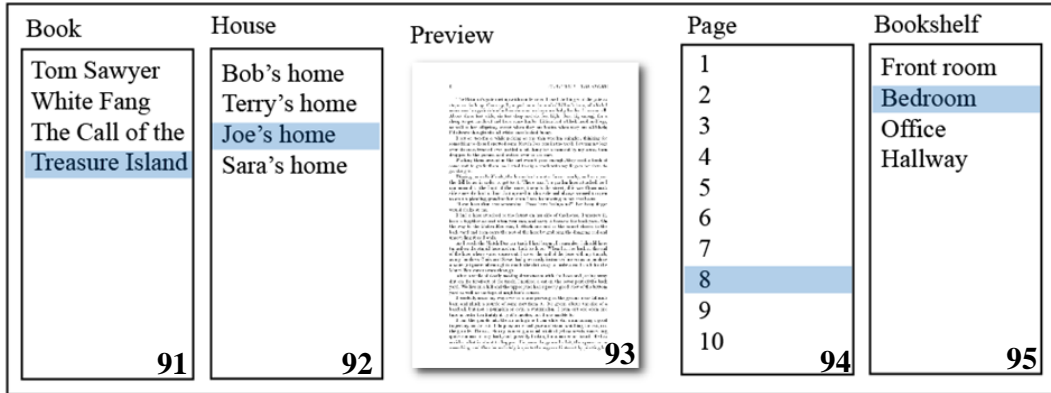
Figure 9.

Fig. 9 shows an alternative user interface layout to that shown in Fig. 7. Both layouts are organized and aligned but the questions remains as to how one determines that the layout shown in Fig. 9 is "poor" when compared to that of Fig. 7.

Figure 10.

Fig. 10 shows the same relationships between widgets 102 and the selection context 103 as that in Fig. 8 (data storage is not shown in this figure). Each widget (1001 through 1005) is then numbered 101 according to the level at which it resides in the selection dependency hierarchy.

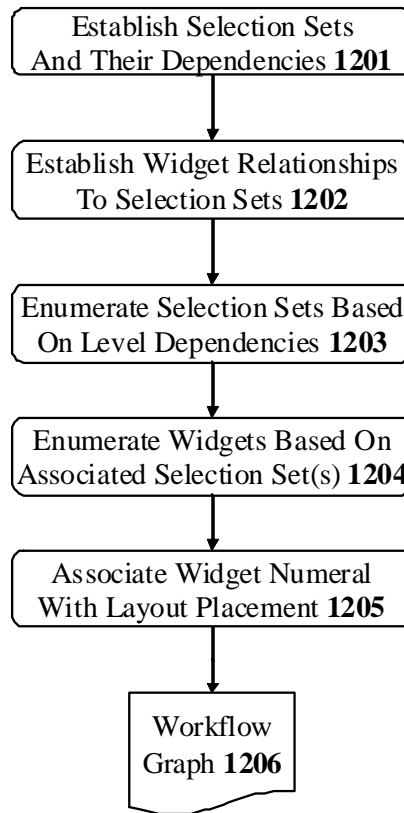## User Interface Workflow Graph

111

112
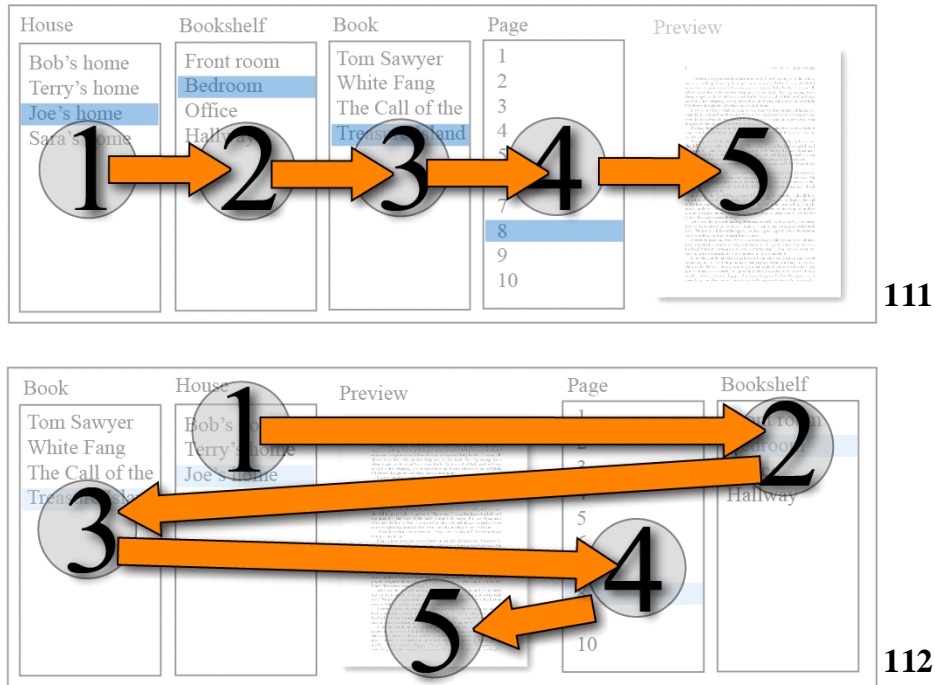
Figure 11.

Fig. 11 shows visualizations of user interface workflows for the layouts shown in Fig. 7 (111) and Fig. 9 (112). The layout on top 111 guides the user monotonically from left to right. The layout on the bottom 112 instead requires the user to navigate back and forth across the layout in order to traverse the selection dependencies.

Obtaining a User Interface Workflow Graph

```
┌─────────────────────────────┐
│    Establish Selection Sets  │
│ And Their Dependencies 1201  │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│ Establish Widget Relationships│
│   To Selection Sets 1202     │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│ Enumerate Selection Sets Based│
│  On Level Dependencies 1203  │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   Enumerate Widgets Based On │
│ Associated Selection Set(s) 1204│
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   Associate Widget Numeral   │
│  With Layout Placement 1205  │
└─────────────────────────────┘
              │
              ▼
        ┌──────────────┐
        │   Workflow   │
        │  Graph 1206  │
        └──────────────┘
```

Figure 12.

Fig. 12 shows the steps involved in obtaining a user interface workflow graph based on the relationships between selection sets. The selection sets are first created and related to one another 1201 in order to establish the dependencies upon one another. Widgets are then attached 1202 to selection sets. Enumerating selection sets 1203 (usually based on the depth from the topmost selection set) by level allows one to assign widgets 1204 to their position in the workflow sequence. Associating 1205 this sequence with the widget position within the layout makes the workflow graph 1206 so that one can visualize or otherwise manipulate the workflow.