Interaction Widget

Mads Ingstrup
ingstrup@daimi.au.dk

March 2004

This pattern describes the idea of making a user interface of discrete, reusable entities---here called interaction widgets. The idea behind widgets is described using two perspectives, that of the user and that of the developer. It is the forces from these two perspectives that are balanced in the pattern. The intended audience of the pattern is developers and researchers within the field of human computer interaction.

Problem

The user and the system [developer] need to exchange information, yet they do not speak the same language.

When a user looks at an application, he or she sees the user interface. When a developer looks at an application, the user is mainly seen as a source and destination of information. The user interface should thus be made sense of in different ways from different perspectives.

Consider the user interface of DOS. It provides no visual guidance for the user. Essentially, everything the user does has to be entered as an array of strings, a form of input that is rather close to the developer's way of seeing a user action (shells are still used a lot ---by developers). Many programs made during the period where DOS was prevalent provided a better user interface than DOS, some did little beyond providing a better UI to DOS (e.g. Norton Commander, Shell). But in order to make these interfaces, the developers needed to build each interface from scratch. This meant drawing the interface as arrays of ASCII characters on the screen, and making the code more complex so that it could deal with registration of user actions and updating the interface.

Building an interface from the ground up is a complex task that requires a lot of work, and even when it is complete and working its appearance and programmatic interface is still not standard and cannot be reused by others. A developer, even one familiar with building user interfaces, would have a difficult time understanding the interface code written by another developer. So for developers of DOS applications, the main problems were lack of a way to reuse code and the difficulty in understanding the code written by others.

In the days of DOS, standardization of the user interface could not be relied upon, which was at the expense of the users, who had to work differently from

application to application. That each interface had to be built from scratch also meant that their quality varied in terms of ease of use and functionality. So it may be fair to say that both the users and the developers suffered as a result of a lack of bridging between their two perspectives---the users because of bad interfaces, the developers because they were entangled with rethinking the details of the interface for each application (while perhaps not the overall approach).

From the perspective of the user, the following *forces* are at play:

- The users desire a standardized interface through which interactions take place. This enables them to familiarize themselves with the interface, not just for a single application but across different applications.
- Interactions should be understandable for the users and the interface should guide them through the use of the application. This is especially important for new users. Part of making the user interface understandable is to enable the user to interact with it using user interface elements that he or she can understand or is familiar with. These include such artifacts as: mice, double-clicking, arrow-keys and respective behaviours.

From the perspective of the developer, the following *forces* are at play:

- Designing interactions and interfaces is difficult, so developers should be able to reuse proven ways of channeling input and output between the user and the system. The developers should be insulated from having to deal with the complexities of low-level interactions like capturing events and updating the screen.
- Interactions should be understandable for the developer in that it should be clear what information in what format is exchanged. The developer should be able to access and handle this interaction in a way that he or she is familiar with; a standardized programmatic interface. These means include concepts such as method invocations, use of operators and data types. For instance, in the Java Api this is done by defining interfaces such as java.awt.event.ActionListener and requiring these to be implemented by classes whose instances should be notified of events in widgets like java.swing.JButton. Another principle used in Java Api is the convention of prefixing method names for reading an objects value with *get*, e.g. *getPercentComplete()* in java.swing.JProgressBar. Though the get idiom is not universally well-liked it is an example of an approach which has some acceptance and is thus better than nothing.

Solution

Bridge the gap between these perspectives by providing reusable mediators (widgets) for each characteristic kind of shared information; let each widget have two interfaces, one tailored for each perspective.

Thus, bridging the gap between these two perspectives is accomplished by using widgets in the creation of a user interface. A widget is an element of the user interface that the user can interact with—such as a button, scroll bar, canvas, and window. A widget encapsulates the behaviour of that user interface element. It hides the user's perspective from the developer, and the developer's perspective from the user.

What a widget does is to capture the essence of a part of an interaction and present different abstractions of this part to the user and the developer. To the developer the abstraction hides the concrete mechanics and complexity of the implementation (e.g. how and when the screen is updated) and it obeys a standardized interface in order for it to be reusable, c.f. the last force in the developer's perspective above. For the user each widget must capture the nature of the information it should mediate by preventing wrong information from being entered or visualized. For instance, the way radio buttons prevent more than one option from being selected.

A widget consists of:

- a *mechanism* that enables capturing or presentation of information. (e.g. a button can be clicked on).
- a *name* that is descriptive of the widget's behavior and characteristics. (e.g. "button" is descriptive of the behavior and characteristics of physical buttons, as is "scroll bar".).
- a *representation* that matches and helps shape the user's mental model of that widget and its behavior. (e.g. a button widget in a GUI uses a stylized pictorial representation of a button, an icon for it). The representation of the widget should convey its affordances. An affordance, a concept introduced by the psychologist Gibson [6], is not a property of the widget per se, but more a relationship between the widget and the user (or user + mouse combination). For instance a button affords being pointed at and clicked on, but this only makes sense in relation to the user+mouse combination that is able to point and click. Using metaphors or analogies is a good way of conveying the affordances of a widget to the user. A good analogy or metaphor is one that gives the user an accurate understanding of what can and cannot be done with the widget. The metaphor or analogy needs to be understood by users with many different backgrounds and ages. See [5] for a further discussion of such issues.

- a *standardized interface* through which the functionality of the widget can be accessed and used by a developer. (e.g. a class java.swing.JButton with methods and attributes accessible to the developer).

From the perspective of the user and the developer, an important thing is held in common. This is the understanding of the information and actions that are exchanged in the interactions. For instance, a button named "Save" will be associated with the meaning that something is stored permanently, and this meaning is shared by the user and the developer. The two perspectives and their overlap is illustrated in the figure below.

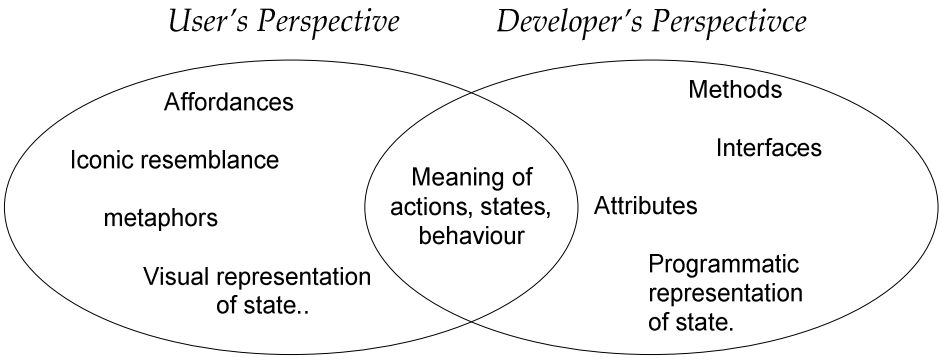

Figure 1. Some overlaps as well as differences in the concepts from the two perspectives

EXAMPLE

The buttons that are used in GUIs are examples of widgets. They mediate information both to and from the system. Firstly, a button presents an action to the user through its label, e.g. "Open". The meaning of the action is also conveyed through the position of the widget in relation to other widgets, for instance in the image below, it is clear that it is the file by the name appearing in the textfield that is opened.

Secondly, it allows the user to perform an action since it through being clickable allows the user to express his or her intent to perform this action. From the user's perspective, the button resembles a physical button. This is significant because it helps convey the button's affordance of being clickable by establishing a connotation to a physical button whose affordances the user is familiar with. Note the use of standardization here, where the underlined "O" in Open means that the action can also be performed with the shortcut »Alt+O«.

From the developer's perspective, the button is an object that can fire notifications to an observer. This functionality is hidden behind an interface and

accessed by methods such as addActionListener(…), in the case of Java buttons. This means that the developer need not worry about the button's implementation and the conformance with a clearly defined interface enables reuse of the button. What is shared by the user and the developer is the meaning of the information being mediated, in this case the meaning of the action that the button lets the user perform.

Consequences

Use of the pattern has the following benefits:

- By looking at and interacting with a widget, a user can derive its affordances. This makes it easier to understand in what way the widget should be used. For instance, the idea behind a scroll bar may be comprehended by trying to use the mouse on the parts of it that resemble buttons. The widget can also preclude the entry of information in the wrong format, structuring the user's access (e.g. radio buttons do not allow more than one selected option).
- Because widgets are standardized and encapsulate their behaviour, a developer can reuse the widgets and save time in developing ways of interaction with the system.
- Reuse of widgets strengthens consistency across applications.
- The specifics of the interaction are hidden from the developer, thus insulating the application: as noted by [3], "Whether the user points and clicks with a mouse ... or uses keyboard shortcuts doesn't require changes to the application." So the widget provides the developer with abstracted information (effectively "this action was performed" rather than "the user used the mouse to click at coordinate (x,y)".

When applying the pattern, be aware of the following:

- If widgets are not designed carefully, they might be difficult to use, especially for new users. The lesser the degree to which a widget conveys its affordances through its appearance, the more important standardization becomes (by the same argument that non-blue non-underlined web links reduces usability). For instance, many widgets found in GUI toolkits can be disabled, often shown by the widget being 'greyed out'. The use of 'greying out' is an example of a standard screen metaphor that is used for indicating disabled widgets to the user.
- Also, from the developer's perspective, if a widget is not designed in a predictable way, it may be difficult to use. Specifically, once a developer has found the right widget for some purpose, it may not be easily apparent

how to extract information (text, numbers, selections) from it---especially if the programmatic interface for it does not follow any conventions or otherwise is non-intuitive. For instance, a developer will almost certainly know that a String object can be obtained from a text field widget, but may be in doubt as to how this is done. So care must be taken to obey the standard style and idioms used in formatting an interface (for instance, in Java classes, methods for getting a value of a field should be called get<FieldName>())

- Depending on how an interface is designed, widgets may slow down the user's interaction with the system. For instance, a highly GUI-based, mouse-driven travel reservation system may slow down a booking agent, who may be used to text-based, keyboard-intensive entry (a historical artifact of early systems). Thus, the type of widgets that are used for an interface should take into account the types of users who will be the target audience of the application.
- In many ways the description of widgets given here points towards the principle of direct manipulation. This is treated more extensively in [5], and.
- Finally, widgets are not silver bullets. Widgets offer advantages, but a developer will still need to use their functionality and combine them in a way that is usable and useful to the user. For instance, a change in the state of a program should be visible to the user. The selection of what actions are possible at any given time should also be considered carefully. The widgets should be spatially grouped in a sensible way (e.g. all buttons for selecting alignment of paragraphs of text in a word processor should appear near each other, see [4]). In other words, the usability design of a user interface is not automatically resolved by this pattern, but interaction widgets may provide a basis for reasoning about some of these issues, like the one with spatial grouping of widgets.

Known Uses

Widgets are used widely in GUI toolkits, taking such forms as buttons, radio buttons, check-boxes, windows etc.

The emerging discipline of Pervasive and Ubiquitous computing introduces a new, physical dimension compared to more traditional systems. This does not preclude use of the pattern. To ease the construction of user interfaces for ubiquitous systems, the Context Toolkit [3] defines a number of interaction widgets that mediate between the environment (including users) and the application in the same way as graphical widgets mediate between the user and the application. An example of an interaction widget from the toolkit is the "Activity

Sensor", designed to register when there is activity in a given room. It consists of physical sensors, e.g. an activity detector, and a software interface/class. Here the pattern extends into the physical dimension, but the forces remain the same.

Another example of a currently emerging class of widgets is in the area of biometric identification [2]. One widget may scan the fingerprint of a user; another may scan the iris of the user's eye, but both deliver the same abstract information to an application, namely the verified identity of the person in front of the widget.

Related Patterns

The Model-View-Controller architectural pattern [1] shows how an application can be divided into Model, View and Controller components. This pattern is different in the following respects:

- The perspectives differ. MVC is an architectural pattern focusing on how to build an application from the perspective of the developer; it is a pattern for a whole application. This pattern employs both the user and developer perspective and seeks to resolve forces perceived from both of these perspectives; it only focuses on the interface part of the application.
- Both patterns argue building the interface of blocks (views and widgets respectively), but there is a difference in scale. Widgets are more fine-grained such that a view may be made of several widgets.
- The patterns complement each other with Interaction Widget describing the part of building views that is left out in MVC. Interaction Widget may be used equally well together with the PAC architectural pattern [1]. This is because both PAC and MVC are concerned with the internal structure of the system, but advocates that the interface is kept separate. The use of interaction widget, pertains only to the interface. And, the issues it addresses are not addressed in either MVC or PAC, and will therefore not interfere with or contradict anything prescribed by these two. One might say that doing as prescribed by Interaction Widget does not cross any of the boundaries set up in the division of a system prescribed by these two.

Acknowledgements

Thanks to EuroPLoP 2003 shepherd Daniel May, and to the members of the writers workshop.

References

1. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: A system of Pattern*s. John Wiley & Sons Ldt., Baffins Lane, Chichester, West Sussex PO19 1UD England, 1996.
2. Anil Jain, Lin Hong, and Sharath Pankanti. Biometric identification. *Communications of the AC*M, 43(2):90-98, 2000. Available from WWW: < http:// doi.acm.org/10.1145/328236.328110> , accessed december 19, 2002.
3. Daniel Salber, Anind K. Dey, and Gregory D. Abowd. *The context toolkit: Aiding the devel-opment of context-enabled applications.* In Proceedings of CHI'99, Pittsburgh, PA, May 1999. ACM Press.
4. Martijn van Welie. *Interaction design patterns.* Available from WWW: < http://www.welie.com/patterns/> , accessed february 27, 2003.
5. Schneiderman, B. *Designing the user interface.* Addison-Wesley, 1997. 3rd edition. Chapter 6.
6. Gibson, J.J. *The ecological approach to visual perception.* Lawrence Erlbaum Associates. 1979.