Patterns for Interactive Applications

William C. Wake
wwake@vnet.net

Interactive Applications

Many applications that we think of as "killer applications" on personal computers, such as word processors or spreadsheets, developed before graphical user interfaces were common. While graphical interfaces have brought several benefits to these programs (better usability, scrolling, fancy fonts, and crisper graphics), the core models of these programs haven't changed: word processors still deal with text, and spreadsheets with matrices of formulas.

The pattern language described below attempts to generalize from applications such as Emacs [1, 2], WordStar [3], dBASE III [4], and VisiCalc [5], and to capture their essence. These patterns apply fairly early in the design, after task analysis and object-oriented analysis have identified key tasks, objects, and relationships.

The patterns apply to *interactive* programs, be they terminal-based or graphical. By "interactive," we're trying to get at the difference between the `ed` and `vi` editors. The line editor `ed` works nicely on a roll of paper on a hard-copy terminal - you work one line at a time, and the file is printed only at your request. The `vi` editor works on a screen - it continuously shows the current document. You can enter text and see it in context as you type.

In the Portland Pattern Repository [6], Kent Beck has articulated two related patterns: in *Story*, he notes that while stories are presented in time, the interface must be designed in space. In *One Window Per Task*, he provides guidance on how to associate windows with tasks. The patterns below are intended to weave into that same design space, and assume Beck's patterns as a given.

This pattern language addresses three aspects of application design: appearance, behavior, and extensibility.

- For a discussion of how to design screens, see *Structuring Metaphor*.
- For a discussion of important characteristics of application behavior, see *Computer Magic*.
- For a discussion of how to make an application extensible, see *Two-Layer Architecture*.

Design of Screens

Structuring Metaphor

Problem: How can a user deal with hundreds, thousands, or more related objects?

Context: Analysis has identified critical objects and relationships.

Forces:

- There are many objects of interest.
- The user's task requires focusing on a small number of abstract, key relationships.

Solution: Give each object a visual representation based on its attributes. Let an object's placement on the screen reflect the key relationships.

Try to develop a structuring metaphor, a simple guiding principle that lets the user build a mental model of how the system works.

Examples:

- ¥ Emacs [1, 2]: the objects are characters, the key relationship is their sequence.
- ¥ Spreadsheets [5]: the objects are formulas, arranged in a table.
- ¥ SortTables [7]: the objects are records, arranged in sequence.

Related Patterns: *Zones of Similar Persistence, Space Proportional to Importance,* and *Peripheral Zones* discuss how to structure the screen. Beck's *Story* and *One Window Per Task* [6] provide guidance on preserving key relationships between objects and the user's task.

Related Ideas: Horton [8] discusses ways to arrange online documents to reflect a logical pattern.

Zones of Similar Persistence

Problem: How can we best share screen space among many objects?

Context: Screen-based interface.

Forces:

- ¥ The information of interest may be constantly changing.
- ¥ÊUsers need a consistent screen structure so they know what to expect.

Solution: Partition the window into zones. Items in a given zone should change at about the same rate. If possible, use dividing lines or other indicators to clearly delineate the zones.

Zones should have inviolate borders - they should not interfere with each other. (Popup menus and dialog boxes may be acceptable exceptions, provided they don't cover information critical to their use.)

Zones should be arranged in a way that accords with the *Structuring Metaphor.*

Example: Spreadsheets typically have a strong zone structure: the cells, their labels, and a data entry box.

Counter-Example: Some versions of the Unix editor `vi` have a status/command line, but it gets covered by the user's text when a command line is not needed. Text in that line is thus ambiguous.

Related Patterns: *One Window Per Task* [6]. Beck has alluded to a pattern *Split Into a Small Number of Zones* that is probably related. When making the zones, take *Space Proportional to Importance* and *Peripheral Zones* into account.

Related Ideas: Publishing uses the notion of a grid in layout design, providing a stable visual format throughout a publication [9].

Space Proportional to Importance

Problem: How big should the various zones be?

Context: Trying to divide a window into zones.

Forces: Many objects compete for screen space.

Solution: Make the space for a zone proportional to the importance of the objects and relations in that zone.

Example: A spreadsheet devotes most of the screen space to the user's data.

Counter-Example: At a local copy shop, the word processor is configured to show all indicators: a menu bar, a tool bar, font/size indicators, and other buttons. The screen is small, and the various indicators take up almost half the screen space, leaving little room for editing - the main task.

Related Ideas: Tufte has a rule for graphics: Maximize the data-ink ratio, within reason [10], meaning that ink should be spent providing useful information.

Peripheral Zones

Problem: What goes where on the screen?

Forces: People will tend to focus on the middle.

Solution: Put the biggest and most important zone in the middle. Put less important information on the edges surrounding it. This information is usually about less important objects, or meta-information about the central objects.

Be aware of reading habits, however. Readers of English expect to look left to right and top to bottom.

Related Patterns: *Structuring Metaphor* takes precedence over this pattern.

Related Ideas: Horton proposes a similar approach for online documents [8].

Design of Behavior

Computer Magic

Problem: How can we keep the user from having to do repetitive, tedious tasks?

Context: A computerized solution is being designed.

Forces:

- ¥ The user would like the computer to do the tedious parts of the task.
- ¥ Programming all parts of the task might be expensive or impossible.

Solution: Ensure that the computer adds value to the performance of the user's task. Look for places where the computer can go beyond being a mere storage device, and can provide summary information, computation, automatic placement, analysis, and arranging.

Examples:

- ¥ Spreadsheet [5]: automatically updating formulas.
- ¥ SortTables [7]: a self-sorting box of file cards.

Related Ideas: Smith [14] describes the "tension between literalism and magic" - how escaping the basic metaphor can yield a more effective system. (For example, we could imagine a computer spreadsheet that follows the metaphor of a paper spreadsheet *too* closely, acting as storage for a grid of numbers, but not providing any computation.)

User-Controlled Navigation

Problem: How can we keep the user from getting confused when information changes too quickly to read?

Forces:

- ¥ ItÕs disorienting to have the screen spontaneously and completely change.
- ¥ Users like to feel in charge of what they are doing and the pace of their work.

Solution: Let the user control navigation. Users should be able to move the focus of attention at their own pace. This navigation might be controlled by scrollbars and/or navigation keys (cursor keys, top/bottom, page-up/down).

Example: Most programs with graphical interfaces use scrollbars to control navigation.

Counter-Example: Some terminal emulators let text scroll past, without trying to save it. This was understandable when terminals had 4K of memory, but is ridiculous in multi-megabyte machines.

Visible Progress

Problem: How does the user know how much work is done and how much is left to do?

Forces: It's frustrating for a user not to know where they are.

Solution: Structure the visual layout to provide some indication of progress. This may take the form of progress bars, counters, scroll bars, etc.

The progress need not be shown merely at the low level - keep the user's high-level task in mind.

Example: The thumb of a scrollbar provides an indication of position as well as being a navigational widget.

Example: Many email programs put special markers on messages that haven't been read. Working through new mail can feel like checking off items on a todo list.

Counter-Example: Web users sometimes become "lost in hyperspace" - unsure of where they are relative to the whole, and unsure how much they've seen.

Related Ideas: Thimbleby [11] briefly mentions "sense of progress" as a consequence of his equal opportunity design principle.

Visible Modes

Problem: How can we help users remember what mode the computer is in?

Forces:

¥ Programs often have modes - states where input has a special meaning.

¥ Humans live in an environment the computer is unaware of: phones ring, people drop by, and so on.

¥ Humans have limited memory - they can't remember the state of the system.

Solution: Attack the problem two ways:

¥ Strive for modelessness to prevent context-dependent situations. (If there's only one mode, the user can never be in the wrong mode.)

¥ If there must be modes, provide visual indicators of the current mode.

Counter-Example: The vi editor has a command mode and a text entry mode, but no indication of which is active.

Related Patterns: Other patterns also support lapses in attention. *Zones of Similar Persistence* lets the user focus on what has changed since they last looked at the screen. *User-Controlled Navigation* restricts what will happen without user involvement.

Related Ideas: Thimbleby [11] calls this the "gone for a cup of tea problem," and discusses how it interacts with modelessness.

Design for Extensibility

Two-Layer Architecture

Problem: How can we provide several interfaces to a system? (e.g., graphical, audio, application programming interface, etc.)

Forces:

- ¥ Real systems must operate in many environments.
- ¥ We can't afford to develop completely independent versions of a system.
- ¥ Users may require different interfaces at different times.

Solution: Use a two-layer architecture: define a core set of commands that do the work, and define bindings that map user actions to those commands.

Example: Consider an electronic mail system: we might like a dial-in audio version for use on the road, and a graphical version for use at the desk.

Existing Extension Language

Problem: How to provide everything people want from the command set?

Forces:

- ¥ Once an extension language is introduced, there will be pressure to add features to it ("creeping featurism").
- ¥ Language design and support shouldn't be the focus of the application development.

Solution: When a command language (as in *Two-Layer Architecture*) reaches the point where control structures become necessary, there will be inexorable pressure to grow it into a full programming language.

So, define the core commands as primitives, and embed them in an existing language such as TCL [12] or Lisp [13].

Example: Emacs [1, 2] has Lisp [13] as an implementation and extension language.

Counter-Example: An early digital video developers' toolkit provided some simple commands. They tried to grow these into a language, but the language had strange restrictions that had nothing to do with digital video (control structures could be nested only two layers deep, limited number of variables, etc.).

Conclusions

This series of patterns has been designed as a pattern language to help in developing screen-based applications, by identifying tradeoffs and pressures in screen design, behavior, and extensibility. Most of these patterns have strong precedents in the human-computer interaction and documentation fields. We hope that the context and forces expose some design tradeoffs, and that the patterns can be used in a generative way to build highly interactive applications.

References

1. Stallman, R. "EMACS: The Extensible, Customizable Self-Documenting Display Editor." In *Proc. SIGPLAN/SIGOA Symposium on Text Manipulation,* Portland, OR, ACM, 1981.

2. Finseth, C. A. *The Craft of Text Editing: Emacs for the Modern World.* Springer-Verlag (New York), 1991.
3. Ettlin, W. A. *WordStar Made Easy.* Osborne/McGraw-Hill, 1981.
4. Weber Systems Inc., Staff. *dBASE III Users' Handbook.* Ballantine Books (New York), 1985.
5. Beil, D. H. *The VisiCalc Book (Atari Edition).* Reston Publishing Co. (Reston, VA), 1982.
6. Beck, K. "User Interface" (http://c2.com/ppr/ui.html) in the Portland Pattern Repository. *Story* was present in 1995, but no longer appears to be. *WindowPerTask* is available still.
7. Wake, W. C. and Fox, E. A. "SortTables: A Browser for a Digital Library." *Conference on Information and Knowledge Management, CIKM-95,* ACM, 1995.
8. Horton, W. "Visual Rhetoric for Online Documents." *IEEE Transactions on Professional Communication,* 33(3), Sept., 1990, pp. 108-114.
9. Swann, A. *How to Understand and Use Design and Layout.* North Light Books (Cincinnati, OH), 1987.
10. Tufte, E. R. *The Visual Display of Quantitative Information.* Graphics Press (Cheshire, CT), 1983.
11. Thimbleby, H. *User Interface Design.* ACM Press (New York), 1990.
12. Ousterhout, J. K. *Tcl and the Tk Toolkit.* Addison-Wesley (Reading, MA), 1994.
13. McCarthy, J. et al. *LISP 1.5 Programmers Manual.* MIT Press (Cambridge, MA), 1965.
14. Smith, Randall B. "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic." *IEEE Computer Graphics and Applications,* 7 (9), Sept., 1987, pp. 42-50.

Acknowledgements

Thanks to Dr. Edward A. Fox of Virginia Tech, for his encouragement and support in exploring this topic. Thanks to Robert Orenstein and Ralph Johnson for their shepherding, for pushing me to clarify my thinking, and for suggesting improved pattern names. The author is an employee of DMR, an Amdahl Corporation.