GOF patterns for GUI Design

James Noble
MRI, School of MPCE,
Macquarie University, Sydney.
kjx@mri.mq.edu.au

June 1, 1997

Abstract

The *Design Patterns* book introduced twenty-three patterns for object oriented software design. These patterns are used widely, but only in their intended domain of software design. We describe how six of these patterns can be used for the conceptual design of graphical user interfaces. By using these patterns, designers can produce interfaces which are more consistent, make good use of screen space, and are easier to use.

Introduction

This paper presents six patterns for graphical user interface (*GUI*) design (Prototype, Singleton, Adaptor, Composite, Proxy, and Strategy), each inspired by the GOF[1] pattern of the same name. These patterns have been used widely in GUIs beginning with the early work on the Star [10] and Smalltalk [9], and we have illustrated these patterns with examples from more recent GUIs.

To apply these patterns to GUI design, we have translated the GOF patterns from the domain of OO software design to that of GUI design. Although both the GOF patterns and our GUI patterns are expressed in terms of objects, interfaces, messages, and so on, we have interpreted these terms into the GUI domain. We use *object* to mean an individually manipulable GUI component, such as an icon, a dialog box, or a window. A single object may appear in more than one way — for example, a document object may appear as an icon when it is closed, and a window when it is open. Objects have *attributes*, such as editable text boxes, radio buttons, and check buttons, and *behaviour*, which can be invoked by buttons, menus, keystrokes, or by directly manipulating the object. A group of objects with similar attributes and behaviour forms a *class*, for example, all window objects could belong to the window class. Classes can be grouped using *inheritance* to capture finer distinctions between objects, for example, all directory windows could belong to one class, all application windows to another, both of which could inherit from the window class.

This translation highlights an important difference between the GOF domain and the GUI domain: GOF patterns can address the implementation of objects, while GUI patterns necessarily address only the interfaces of GUI objects [7]. For those patterns that are mainly concerned with interfaces, such as Proxy or Prototype, this makes little difference. For patterns where the main concern is encapsulation, such as Strategy, the corresponding GUI pattern brings out a secondary aspect of the pattern, such as factoring or locating objects.

These patterns do not directly address usability, although we have found these patterns in interfaces which have been extensively designed for usability. We believe these patterns are orthogonal to other techniques for user interface design (such as walkthroughs, prototype, metrics, or visualisation), just as patterns for software development are mostly orthogonal to other software development techniques.

[1]A *GOF Pattern* is a pattern from the *Design Patterns* book [8], written by the so-called *Gang of Four* — Gamma, Helm, Johnson, Vlissides.

Similarly, in the same way that the GOF patterns address the middle ground between programming languages and methodologies, these patterns also seem to fit between detailed user interface guidelines [1, 11] (which are the specifications of user interface languages) and overarching user interface methodologies [6].

Form and Content

The bulk of this paper presents six of the GUI patterns we have identified. These patterns are organised in the same way and presented in the same order as the corresponding patterns the GOF book. We begin with the creational patterns Prototype and Singleton, then present the structural patterns Adaptor, Composite, and Proxy, and conclude with the Strategy behavioural pattern. This paper contains a pattern catalogue, rather than a system or language of patterns.

For space reasons, we have used a compressed form to present the patterns. Each pattern has a name and a statement of *intent*, derived from the corresponding sections of the GOF pattern. A short *problem* statement addresses the motivation and applicability of the pattern, and then the *forces* the pattern resolves are itemised explicitly. Aspects of the *solution*'s structure, participants, collaborations, and implementation are then combined in a single section. A picture illustrates an *example* of the pattern in use, taking the place of GOF's sample code. Finally, the positive (+) or negative (−) *consequences* of the pattern are outlined and examples of *known uses* are reviewed. Essentially, this form compresses the major narrative sections of the GOF form into two short paragraphs (problem and solution), each with an associated bullet list (forces and consequences, respectively).

Most of the forces are derived from the pattern's context, and are isomorphic to the forces in the related GOF pattern. This isomorphism is important, since it ensures the shape of the solution is similar across the domains. The patterns also refer to three forces which are specific to the GUI design domain:

- Interface consistency is an explicit force in GUI design. In an OO design, consistency resolves other forces, such as reusability.
- Screen real estate is an important and often scarce resource.
- Object identity in GUIs means that an object should only appear in one location on the screen. That is, GUIs don't have pointers.

Related Patterns

Given that the first software pattern language was about user interface design [3], it is quite surprising that so few patterns and pattern languages have been written for this domain. The tools and materials metaphor has been described in a pattern language, although the language concerns implementation as much as design [16]. Pattern languages have also been written for designing essay based web sites [14] and form style windows [4].

Envoi

We believe that many GOF patterns capture deep general properties of object oriented software design. This paper is an experiment to determine whether (and how well) the patterns appear in the related, but significantly different, domain of GUI design.

GUI Prototype

Object Creational

Intent Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

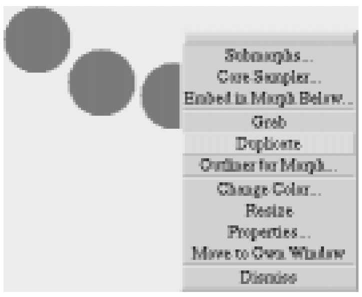
Problem GUIs support many different kinds of objects which have different capabilities. Users need to be able to create the right kind of object to hold their data. For example, desktops support a number of different document types, such as processed words, spreadsheets, and databases. *How can the user create new objects?*

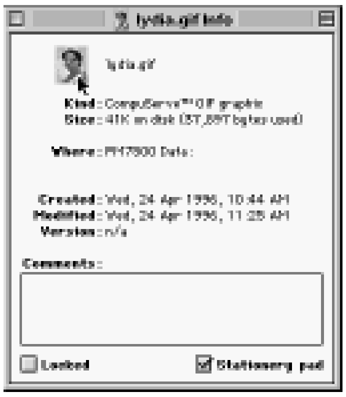
Forces The GUI Prototype pattern resolves the following forces:

- The user needs to create various classes of documents.
- You cannot anticipate all the classes in the system.
- Objects should be created in a consistent way.

Solution Make a *prototype* object for each class. Let the user create new objects by copying the appropriate prototype. Each class should understand a "copy" message, inherited from the class of all user objects.

Example In the Self UI all objects are created by duplicating existing prototypes — there is no "create" operation. Macintosh System 7 allows documents to be marked as "stationery pads" — when the user opens stationery pad, the pad is copied and the copy is opened.




lydia.gif Info

lydia.gif

Kind: CompuServe™ GIF graphic
Size: 41K on disk (37,897 bytes used)

Where: PM7600 Data:

Created: Wed, 24 Apr 1996, 10:44 AM
Modified: Wed, 24 Apr 1996, 11:25 AM
Version: n/a

Comments:

☐ Locked ☑ Stationery pad

Consequences The GUI Prototype pattern has the following benefits and liabilities:

+ Users can create objects of all copyable classes in the same way.
+ New classes can be added by supplying new prototypes.
+ Users can make their own classes, by copying objects they've initialised.
− Users may need to empty old data out of newly copied objects.
− Users may forget to copy objects and accidently edit the prototype directly.

Known Uses Prototypes were first used in Sketchpad [19]. They are used explicitly in the Self UI [18], the MoDE Composer [17] and the Macintosh [2]. We have suggested many users use prototypes instinctively [13].

GUI Singleton — Object Creational

Intent Ensure a class only has one instance, and provide a global point of access to it.

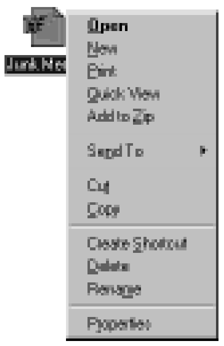
Problem Some classes should always have exactly one instance. For example, objects representing real hardware resources, like disks, printers, or networks, or special system objects like trash cans, should only appear once in any GUI. *How should you manage these unique objects?*

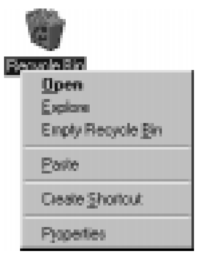
Forces The GUI Singleton pattern resolves the following forces:

- There should be only one instance of these objects.
- The single instance should be widely available.
- The user interface should be consistent.
- Screen real estate is limited.

Solution Make a singleton object, and design the singleton's class so that the user can't copy or delete the singleton. Create the singleton when the GUI starts, and place it on the screen (typically on the desktop) so it is always available.

Example The Recycle Bin in Windows95 is a singleton — it does not include the deletion and renaming commands provided by most objects.





Consequences The GUI Singleton pattern has the following benefits and liabilities:

+ Only one instance of a singleton can be created.
+ The singleton is readily available on the desktop.
+ The singleton behaves mostly like other objects.
– The GUI is less consistent, because singletons can't be copied or deleted,
– A singleton is always present, occupying screen real estate.

Known Uses The Macintosh and Windows95 GUIs use this pattern for several desktop icons. The VisualWorks Launcher [15] is a variant of singleton, since although it is created on startup, it can be deleted.

See Also GUI Proxy can allow a Singleton object to appear to be in more than one place.

GUI Adaptor

Object Structural

Intent Convert the interface of a class into another interface users expect. Adaptor lets classes work together that couldn't otherwise because of incompatible interfaces.

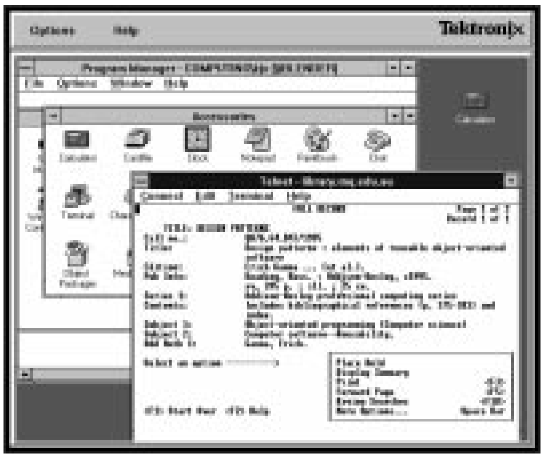
Problem Interfaces are worlds unto themselves, each with their own culture, language, and style. Users often need to inhabit several worlds, for example, when using legacy mainframe applications from GUI desktops, or using one desktop from another. *How can the user access one interface from another incompatible interface?*

Forces The GUI Adaptor pattern resolves the following forces:

- The user needs to use objects which have incompatible interfaces.
- You don't want to modify either interface.
- The user interface should be consistent.

Solution Make an adaptor object which hosts one interface inside the other. An adaptor is a normal object in the host interface, graphically containing the hosted interface. Let the user interact with the hosted interface via the adaptor. Adaptors often provide out-of-band operations for manipulating the connection between the two interfaces.

Example The Windows NT interface can be adapted to run inside the X Window system (the Tektronix menu bar manipulates the adaptor). Windows NT can host textual applications via terminal emulators.



Consequences The GUI Adaptor pattern has the following benefits and liabilities:

+ The hosted interface can be used from the host interface.
+ Neither interface needs to be modified.
+ A *generic* adaptor can adapt sets of interfaces, such all ASCII text applications.
− The resulting user interface may be inconsistent if the host interface's conventions are not used within the adaptor.

Known Uses Most GUIs include terminal emulators which give access to textual applications. Windows and Macintosh interfaces can be adapted to run under X, and vice versa. Web browsers adapt the WWW user interface to wide range of host interfaces.

See Also A Legacy Wrapper is a more general form of this pattern [12].

GUI Composite

Object Structural

Intent Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

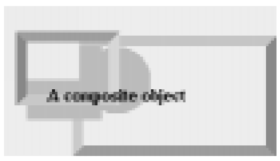
Problem Many GUI objects are made up recursively of other GUI objects. For example, disks contain directories, and directories contain other directories and files. *How can the user manipulate composite objects?*

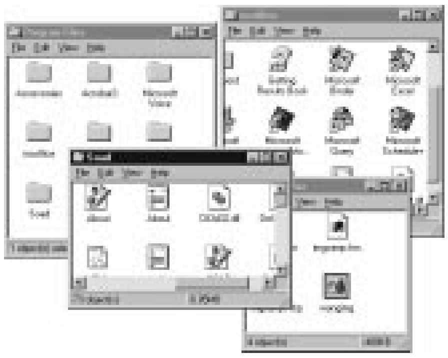
Forces The GUI Composite pattern resolves the following forces:

- The user needs to manipulate the whole composite object.
- The user needs to manipulate the individual parts of the composite.
- The user interface should be consistent.

Solution Make composite objects which can recursively contain other objects, both composites and primitives. Make a common class for all the operations shared by primitive and composite objects. Particular objects should inherit from this class, and extend it to provide operations applicable to them.

Example In the Self UI, any graphical objects can be combined (embedded) to build up composite structures. Windows NT represents directory structures as composite objects.





Consequences The GUI Composite pattern has the following benefits and liabilities:

+ The user can manipulate both the whole object and the individual parts.
+ The interface for common operations is consistent.
− It can be hard to isolate a particular individual part inside the whole.

Known Uses The Macintosh and Windows95 interfaces use composite objects to represent directory structures. MacDraw and many other drawing editors let the user construct composite pictures by explicitly grouping and ungrouping graphical objects.

GUI Proxy

Object Structural

Intent Provide a surrogate or placeholder for another object to provide access to it.

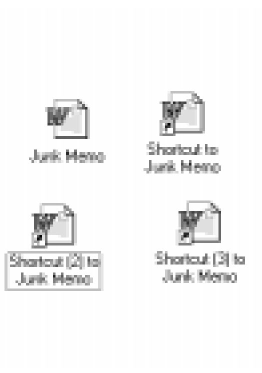
Problem Some objects are never where the user wants them to be. For example, the user might want to store a file deep within a directory hierarchy, but keep it easily accessible. Or the user would like to download a web page, ignoring any in-line images but keeping the document's structure intact. *How can an object be in two places at once?*

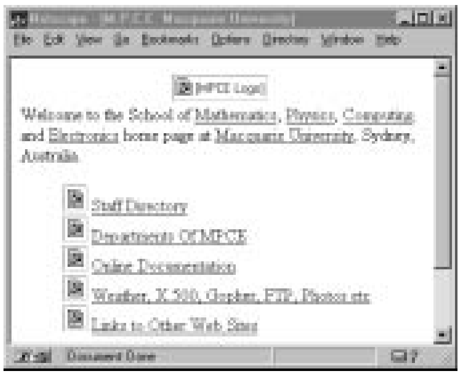
Forces The GUI Proxy pattern resolves the following forces:

- The user wants an object in two places at once.
- GUI object identity requires that an object can only be in one place.
- You don't want to modify or move the original object.
- The original object may be expensive or difficult to retrieve.
- The user interface should be consistent.

Solution Make a proxy object to stand in for remote or expensive objects. Put the proxy where you'd like to put the original object, but can't. Let the proxy behave as if it were the original object, but visually distinguish the proxy from the original.

Example Windows95 shortcuts act as proxies so that an object can appear in multiple places on the desktop. Netscape uses icons as proxies for images which haven't been downloaded.





Welcome to the School of Mathematics, Physics, Computing and Electronics home page at Macquarie University, Sydney, Australia.

Staff Directory
Departments Of MPCE
Online Documentation
Weather, X.500, Gopher, FTP, Photos etc
Links to Other Web Sites

Consequences The GUI Proxy pattern has the following benefits and liabilities:

+ The original object appears to be in two places at once.
+ The original object doesn't need to be changed.
+ Access to the original object via the proxy is transparent.
+ The user can distinguish between the original object and the proxy.
− If the original object becomes unavailable, the proxy will be unusable.

Known Uses Macintosh aliases and Windows95 shortcuts act as *remote proxies* for objects in other places. Netscape uses icons as *virtual proxies* for images which haven't been downloaded. Many web pages use small image thumbnails as proxies for larger images.

See Also *Pattern-Oriented Software Architecture* also describes the Proxy pattern [5].

GUI Strategy

Object Behavioural

Intent Define a family of algorithms, and make them interchangeable.

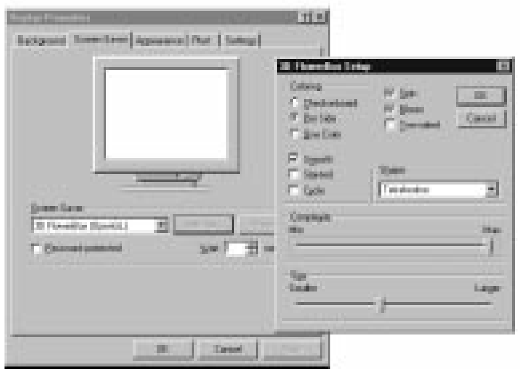
Problem An object uses several algorithms, each with its own interface and customisable parameters. The user needs to set the parameters for the algorithm they have chosen. For example, a screen saver may provide a number of different display algorithms (text, 2D graphics, 3D graphics) each with its own parameters (the text to display, or the colours, textures, and 3D objects to draw). *How can the user deal with the different algorithms?*

Forces The GUI Strategy pattern resolves the following forces:

- An object needs different algorithms which require different parameters.
- Users should choose the algorithm and its parameters.
- The user interface should be consistent.
- Screen real estate is limited.

Solution Make a separate strategy object to represent each algorithm, and make the algorithm's parameters into the attributes of the strategy object. Make the strategy object belong to the object which uses the algorithms. The user can choose an algorithm via the main object, then interact with the strategy object to set the algorithm's parameters.

Example Windows NT uses strategy objects to set the parameters for its screen saver.



Consequences The GUI Strategy pattern has the following benefits and liabilities:

+ The strategy objects explicitly represent the different algorithms.
+ The user can select an algorithm and set its parameters.
+ Presenting one strategy object at a time preserves real estate.
− Changing strategy objects may make the interface less consistent.
− The interface includes an increased number of objects.

Known Uses Windows NT uses strategy objects to set parameters for its printer drivers, as well as its screen saver. Paint Shop Pro and XV use strategy objects to configure their file conversion algorithms.

Acknowledgements

Thanks are due to Kent Beck, the EuroPLOP'97 shepherd for this paper, to Michael Richmond, Jonathon Tidswell, Geoff Outhred, and Larry Constantine for comments on drafts of this paper, and for help with Macs and PCs.

References

[1] Apple Computer, Inc. *Human Interface Guidelines: The Apple Desktop Interface.* Addison-Wesley, 1987.

[2] Apple Computer, Inc. *Inside Macintosh*, volume VI. Addison-Wesley, 1991.

[3] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical report, Tektronix, Inc., 1987. Presented at the OOPSLA-87 Workshop on Specification and Design for Object-Oriented Programming.

[4] Mark Bradac and Becky Fletcher. Developing form style windows. In *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1997.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[6] Dave Collins. *Designing Object Oriented User Interfaces.* Benjamin/Cummings, 1995.

[7] Larry L. Constantine. Getting the message. *Object Magazine*, September 1996.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1983.

[10] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. Irby, M. Beard, and K. Mackey. The Xerox Star: A retrospective. *IEEE Computer*, 22(9), 1989.

[11] Microsoft Inc. *The Windows Interface Guidelines for Software Design.* Microsoft Press, 1995.

[12] Diane E. Mularz. Pattern-based integration architectures. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.

[13] James Noble. Prototype based user interfaces. Technical report, MRI, School of MPCE, Macquarie University, Sydney, 1996. Presented at the COTAR'96 Workshop, Melbourne, 1996.

[14] Robert Orenstein. A pattern language for an essay-based web site. In *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[15] ParcPlace Systems. *VisualWorks Smalltalk User's Guide*, 2.0 edition, 1994.

[16] Dirk Riehle and Heinz Züllighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.

[17] Yen-Ping Shan. MoDE: A UIMS for Smalltalk. In *OOPSLA Proceedings*, October 1990.

[18] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *OOPSLA Proceedings*, 1995.

[19] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, Detroit, Michigan, May 1963.